Application Of Role Modeling In Designing Component

***Preeti Gupta**

Abstract

Components are collection of cooperating entities. New abstraction and techniques are required for designing software components. In this paper, i use role models to represent component interaction and collaboration. I adopt role models because of its strong support for many of criteria, rules and principles that form the basis of modularity. As role models can be employed for analysis, design and implementations, they also provide a direct mapping to applications that can be traceable throughput a components lifecycles.

1. Introduction: Role modelling is relatively new in object-oriented software development. It was introduced to complement object modelling [1, 5, 12, 14, 20, 21]. There are two related role modelling approaches. One treats roles as evolving aspects of objects that can be attached or removed from objects [5,12]. This approach makes use of existing object modeling abstractions and object-oriented programming languages. Another approach uses a new abstraction called role models to capture patterns of interaction [1, 10, 20]. This approach is often used in object analysis. Roles and role models can be implemented using some design patterns [2, 22]. This paper summarizes my work on component design that is based on the role model approach. Sections 2 and 3 provide background information, discussing the problems involved in attempting to design reusable software components with object modelling. Section 4 gives background information on role models. Section 5 illustrates our approach to component design. Finally, in section 6, I discuss about the future work.

2. Background : More than two decades ago, Yourdon and Constantine defined a software *module* as "a lexically contiguous sequence of program statements, bounded by boundary elements, having an aggregate identifier" [29]. They proposed two key techniques -- *coupling* and *cohesion* -- for evaluating and measuring the connections and dependencies between modules. Meyer argues that the traditional definition of modularity is informal and does not address the benefits of extensibility and reusability in object technology [16,17]. He proposes a set of complementary properties, which he suggests cover the most important requirements for designing reusable and extensible modules (Figure 1) [16,17].As shown in Figure 1, some of Meyer's proposal is based on Yourdon and Constantine's classic definition. For example, *The Linguistic Modular Units Principle* states that modules must correspond to syntactic units in the language used. The *Few Interface Rule* states that every module should communicate with as few others as possible, whereas *Small Interface* requires that two modules should exchange as little information as possible. These rules are examples of weak coupling. The *Single Choice* principle is in accordance with high cohesion. In addition, Meyer proposes that a module should be autonomous and self-organising. His *Composability, Decomposability,* and *Open-Closed* principles address these goals.

*Lecturer SoCA

FIVE CRITERIA	FIVE RULES	FIVE PRINCIPLES
Decomposability	Direct Mapping	Linguistic-Modular Units
Composability	Few Interface	Self-Documentation
Understandability	Small Interface	Uniform Access
Continuity	Explicit Interface	Open-Closed
Protection	Information Hiding	Single Choice

Figure 1. Modularity criteria, rules, and principles

According to Meyer, “classes should be the only modules [17].” However, i argue that the true benefit of modules is their capability of packaging multiple objects and other heterogeneous entities, such as routines and procedures. A similar argument applies to the term component. Some people use component and object interchangeably. Again, we accept that an object can be regarded as a component, but we prefer to use it in a broad sense: a component is a composition of entities, which collaborate to fulfil a specific function. An entity in a component can be an object, a procedure, or another component. Components are in fact centers or noticeable, recursive structures in component software.

“A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be
deployed independently and is subject to composition by third parties.”

According to the above definition, a component has four major characteristics:

- A component has contractually specified interfaces.
- A component has explicit context dependencies.
- A component is a unit of composition by third parties.
- A component can be deployed independently.

I have observed that these four component characteristics are consistent with Meyer’s 15 modular properties. I can use Meyer’s properties as common requirements for both modular and component design, as summarised below.

i. Composability and Decomposability. A module should have composability so that it can be combined with other modules to produce a system. Composability and the Open-Closed principle are closely related; a module must be organisationally closed and structurally open so that it can maintain its internal stable form and at the same time be open for extension. Composability is also central to a component: “Components are for composition [26].” Decomposability is the inverse of composability, and a module or component should have this duality.

ii. Small and Few Interface. Modules are connected by interfaces, and so are components.“Interfaces are the means by which components are connected [26].” These two properties enforce lower coupling between modules. Related properties are Continuity and Protection. Continuity means that a small change to a system should be localized within only one or a very few components. A module that satisfies the protection criterion will reduce the propagation of side effects from an abnormality that occurs at run time. Besides, Information Hiding and Single Choice also lead to modular continuity and protection.

iii. Understandability and Semantics. To facilitate the maintenance process and composition, both modules and components should be semantically understandable. Being a Linguistic Modular Unit, with an Explicit Interface, and providing Self-Documentation are essential to modular and component organization. For a component to provide an explicit context, it should provide both syntactic and semantic information so that it can be easily understood by third parties.The above discussion shows that there are some common requirements for both modules and components. Modular development and component development are both aimed at achieving software reusability and extensibility.

3. Designing Reusable Software Components with Object Modelling: Component-based software is usually designed and developed with object technology. In this section I discuss four major problems with the use of object modelling for components. I attempt to show why other abstractions and techniques are needed for component modelling.

3.1 Interactions and Collaborations: In component development, interactions and collaborations are of paramount importance. However, as pointed out in [19], object-oriented programming "too often concentrates on individual objects, instead of whole collections of objects. Focusing on individual objects is misleading and often results in software which cannot be used as components." In an application where objects are the only structuring facility or the only unit of abstraction, it becomes very difficult to extract and package a suitable subset or subsystem [19]. Therefore, the primary problem in building reusable software components seems to be the need for a shift of focus from the level of individual objects to the level of subsets of interacting, collaborating entities.

3.2 Interface Translation: Another major obstacle in building reusable software components is the lack of standards. Components developed independently cannot be readily integrated into an application. Although some interoperability standards have recently become available which define mappings from a client component to a server component [18], such standards do not provide any means for specifying the interfaces between the client and server components [24]. One solution to overcome this problem is to reproduce the server components to conform to the interface requirements of the client components. This suggests that the server components cannot be reused.

3.3 Reuse through Inheritance: Object-orientation has made many claims regarding software reuse. However, while an abject can be regarded as a basic module, or component, it is not inherently reusable. One day argue that class libraries and code inheritance can achieve reuse. Yet this form of reuse violates many of the properties in Figure 1. First, reuse of class libraries means that the developer has to know the details of the source code. This violates the *Protection* criteria and the *Information Hiding* rule. Second, reuse via inheritance is similar to copy and paste [25]. This makes it difficult for the developer to decide what to inherit and what to override. It also violates the *Decomposability* criteria because inheritance means that a subclass obtains all of the superclass' features and behaviour as a monolithic block.A reusable component should therefore hide design and implementation details from clients, and highlight the interface properties. Further, a reusable component should be readily integrated into an application and composed with other components. A reusable component therefore goes beyond object inheritance.

3.4 Reuse through Delegation: The second common argument for object reuse is delegation. With this approach, a container object delegates behaviour to an object that is inside or within it. This has been advocated as a major avenue to reuse, as new objects can be placed inside the container object, providing new behaviour. However, due to the way that object interfaces work, a designer who chooses delegation is faced with two options that are less than desirable. First, they can make the contained object public so it can be messaged directly. Alternatively, they can reproduce the contained object's interface in the container so a message can be passed from a client object, via the container, to the contained object. Reuse through delegation therefore either violates the Information Hiding rule or leads to a complicated chain of communication (contrary to Few, Small, Explicit Interfaces).

4. Role Modelling Techniques: Role modelling techniques can be used to address some of the problems discussed in sections 2 and 3. In particular, role models address composability, decomposability, understandability, and semantics or context. Role models also concentrate on interactions and collaborations (section 3.1)

4.1 Role and Role Model: The central activity of role modelling is role model construction. A role model is an abstraction that describes patterns of interactions between a set of entities. The entities play certain roles in a given context; the context is captured by the role model. A role model depicts frequently occurring but transient relationships between entities or objects that are working together to perform a certain task or accomplish a certain goal. As an example, we consider a high level view of process management in manufacturing.

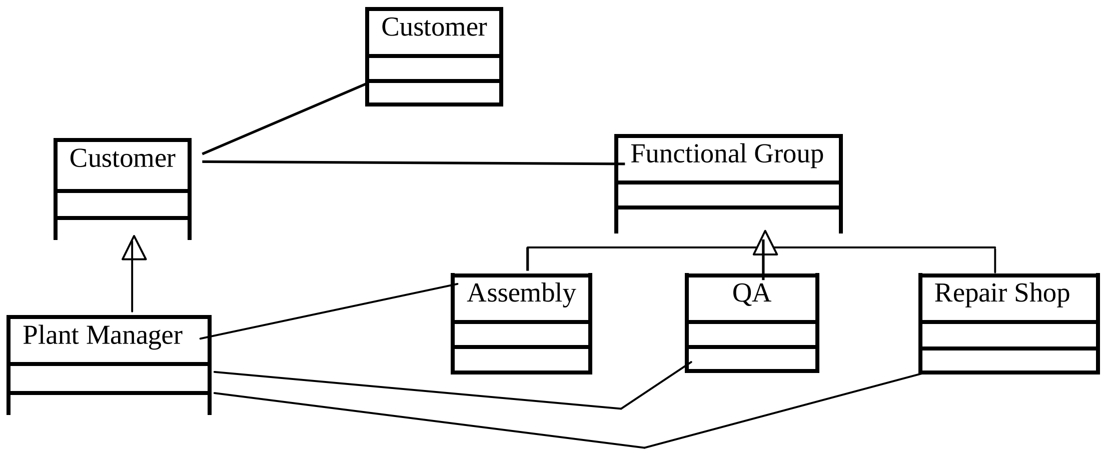

Figure 2. A Class diagram of manufacturing process management

Figure 2 (a UML class diagram) shows static relationships between customers, managers, and various functional groups (assembly, quality assurance, and a repair shop). An instance collaboration diagram for this same application is depicted in Figure 3. Figure 3 shows object interactions in a particular collaboration -- a customer requests a new product. In the figure, the message sequence is numbered and the direction of messaging is shown as an arrow. When a customer makes a request for a new product, the request propagates through messages 2 to 5, where the plant manager delegates work to assembly and then to QA. The product is then delivered to the customer in message 6.

The collaboration depicted in Figure 3 is a pattern that is characteristic of centralized communication or the *Mediator* pattern [4]. i can capture and abstract this pattern of interaction explicitly in a role model (Figure 4), where there are three roles: *Client*, *Mediator*, and *Colleague*. A role is denoted as a rounded box and the solid arrows indicate collaboration paths between the roles. The direction of the arrow represents the direction of messaging, and the solid circle on the link from the *Mediator* to the *Colleague* indicates that there is more than one *Colleague*. Figure 4 also shows an example of role assignment: the objects in Figure 3 (shown as rectangles below the horizontal line in Figure 4) play the various roles, as indicated by the dashed arrows.

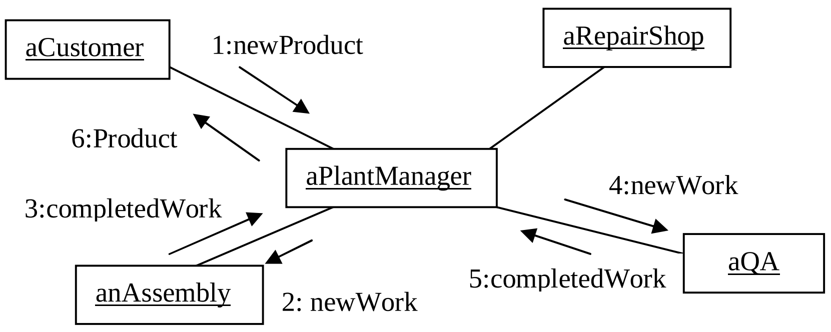

Figure 3.Object interactions in an instance collaboration diagram

The important distinction between the collaboration diagram in Figure 3 and the role model in Figure 4 is that the role model is an abstraction; the object collaboration diagram is an instance of it. Additional role model views, notation, and semantics are detailed in [1].

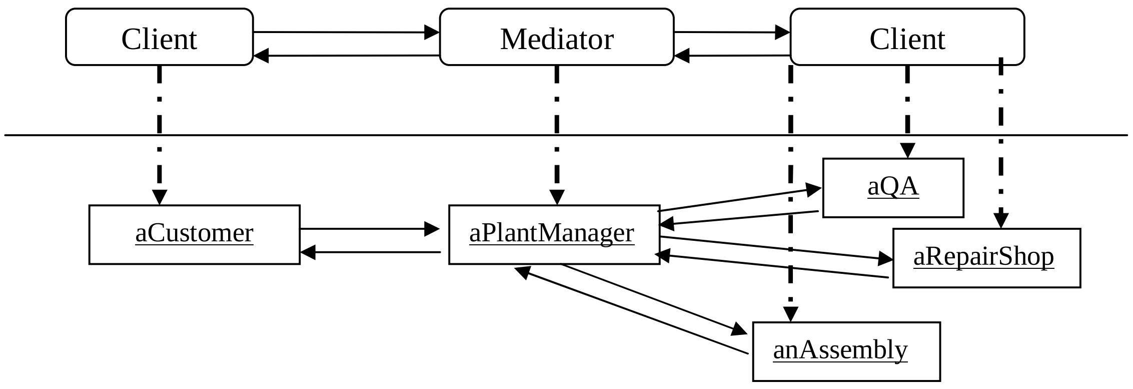

Figure 4. A role model for the object interaction in Figure 3

4.2 Other Examples: In order to illustrate how role models can be used to design components, we introduce two more examples here: *Bureaucracy* and *Supply Chain*.**The Bureaucracy** In a multilevel hierarchical organisation, the *Mediator* pattern is in fact a role model that can be aggregated within larger role models. One such multilevel role model is called the *Bureaucracy* pattern [22], as in Figure 5, where a *Mediator* has now become a *Manager* anda *Colleague* has become a *Subordinate*. There are six roles involved in Bureaucracy: *Director*, *Director Client*, *Manager*, *Subordinate*, *Clerk*, and *Clerk Client*. In the role model, a manager and a subordinate must also be a clerk (indicated by the triangle for refinement),and a director must also be a manager. A client is free to interact with any part of the hierarchy. However, a director has additional responsibilities for error handling and managing the entire hierarchy.

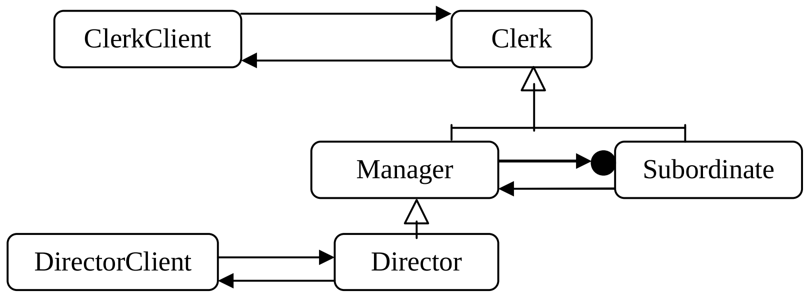

Figure 5. Role model of the Bureaucracy pattern

5. The Supply Chain: A Supply Chain is a common pattern of collaboration [8, 10] (Figure 6); it often appears in agent systems, manufacturing, and other enterprises. It is similar to the Chain of Responsibility pattern [4], except that each link in the chain is required to deliver a product or perform a service for its predecessor. A Supply Chain (SC) is comprised of suppliers and consumers. A consumer can have many suppliers, but a supplier usually only has oneconsumer in any given supply chain. At the highest level, a supply chain is made up of SC Predecessors and SC Successors. A predecessor can have many successors. As shown in Figure 6, a SC Participant is both a predecessor and a successor, while a SC Head is a specialization of a predecessor, and a SC Tail refines a successor.

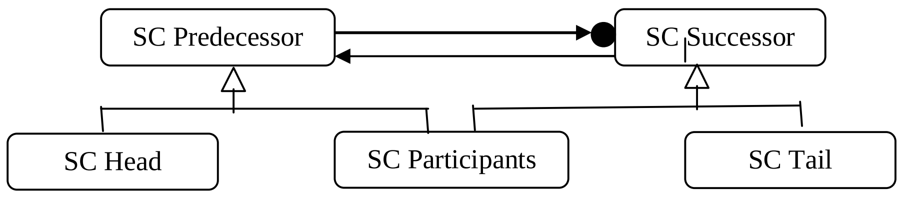

Figure 6. Supply Chain: top level role model

We have introduced *Role Responsibility Collaboration (RRC)* cards [8,10] as a simple way to document the responsibilities and collaborations of a role in a given role model. Sample RRC Cards for the SC Predecessor, SC Successor, and SC Participant roles are provided in Figure 7. As shown in the cards, the responsibilities can be viewed to belong to lower level roles, such as Customer, User, Provider, and Operator. Lower level, interior or aggregate roles are more detailed views of a given role.

Role : Supply Chain (SC) Predecessor	
Responsiblities :	**Collabrators:**
Initiate and complete supply negotiation (Customer)	SC Successor
Receive supplies (User)	SC Successor
Role : Supply Chain (SC) Successor	
Responsiblities :	**Collabrators:**
complete supply negotiation (Provider)	SC Predecessor
Produce supplies (Operator)	
deliver supplies (Operator)	SC Predecessor

Role : Supply Chain (SC) Participant	
Responsiblities :	**Collabrators:**
complete supply negotiation (Provider- Participant)	SC Predecessor
Initiate and complete supply negotiation (Customer - Participant)	SC Successor
Receive supplies (User - Participant)	SC Successor
Produce supplies (Operator - Participant)	
deliver supplies (Operator)	SC Predecessor

Figure 7: Role responsibility cards for Supply Chain role model

6. Component Design as Role Composition:

6.1 Overview: As discussed in section 3, interactions and cooperation are of paramount importance for components. We propose that role models are an excellent vehicle for capturing, abstracting, and assembling component collaborations. However, my emphasis is on the fact that components interact by playing roles. In a given application, a component may play one or more roles; a role may also be played by one or more components. This statement is in contrast to Pfister and Szyperski [19], who seem to be stipulating that a component is mapped onto one or more roles, but not vice versa. Components should be composable and decomposable, and so should their roles. The primary task in component design is therefore identifying and composing the roles played by a given component. A component is designed to meet the criteria of the roles it must play. A component is also the result of role composition because it may appear in many role models, playing various roles. A similar approach has been applied to the design of frameworks [15,21] and agents [9]. my approach to component design therefore consists of the following steps.

i. Identify all the role models in a subsystem or an entire application. Each role model accomplishes a particular task or performs a specific function (Figures 4-6).
ii. Specify all the roles in these models. Each role is assigned responsibilities and collaborators (Figure 7).
iii. Assign role(s) in a given role model to a component or components (Figures 4 and 8).
iv. Carry out steps 2 and 3 for all of the role models in a given application.
v. Compose roles and role models to form a component (Figure 9).
vi. Refine the role composition to remove any conflicts, overlap, or redundancies. Ensure that the component's interface is not overly large.

6.2 Illustration: As an illustration to my approach, in Agent Enhanced Workflow (AEW) and flexible manufacturing [8, 10], an agent represents an individual, organization, or machine that can do work. An agent is responsible for assigning and scheduling work for the entity that it represents, and agents depend on each other to deliver products and/or work. In other words, some agents supply work or products, while others consume it.Some may be managers, while others are subordinates. For example, three agents may represent an end customer and two enterprises, respectively. The customer deals directly only with Enterprise 1. Enterprise 1 depends on Enterprise 2 for supplies or work,. At the highest level, the application (Figure 8) is an instantiation of the Supply Chain role model (Figure 8). The Customer is the SC Head, Enterprise 1 is a SC Participant, and Enterprise 2 is a SC Tail. Enterprise 1 is a SC Successor to the Customer, but it is a SC Predecessor to Enterprise 2. In Figure 8, the relevant role models appear in the top half of the diagram, while the entities in the application that play the roles appear in the bottom half. As in Figure 4, dashed lines indicate role assignments.

However, each enterprise in the supply chain can be made up of several entities. For example, Enterprise 1 may be a manufacturing company with a hierarchical structure and agents to represent each domain. In this case, both the Bureaucracy (Figure 5) and Supply Chain role models appear. This is captured through the Manager and Subordinate roles in Figure 8. It is the responsibility of the Plant Manager (a manager) to be the SC Successor to the Customer, but it is the Assembly functional group (a subordinate) that requires input from Enterprise 2, so it is the SC Predecessor in that context.

The Plant Manager must play all of the lower level roles found in a Supply Chain Successor. (In a more detailed view of Figure 8, these consist of Negotiator, Producer, and Supplier). In addition, the Plant Manager must be able to play the role of a Manager in a Bureaucracy. Likewise, the Assembly group must be a Supply Chain Predecessor in addition to satisfying the responsibilities of a Subordinate. Both entities must appropriately address context switching as they go from role to role.

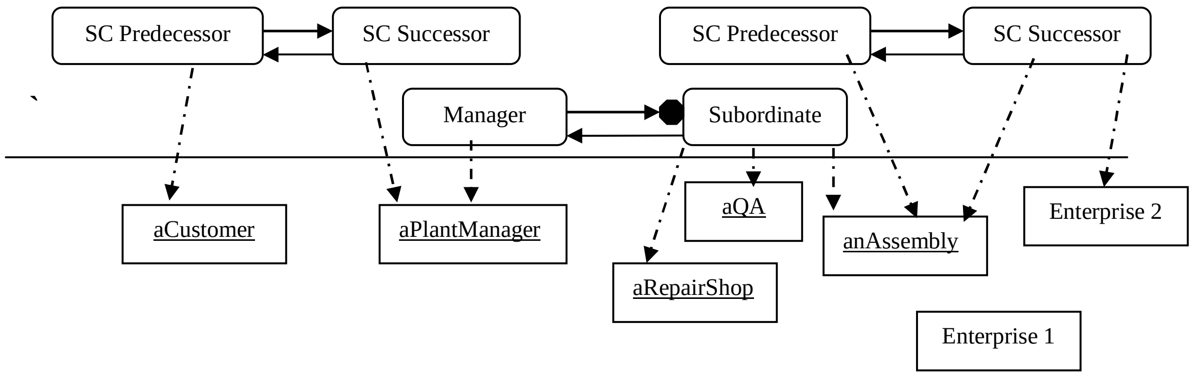

Figure 8. Bureaucracy with Supply Chain for agent enhanced workflow application

The consequence of role model composition is role composition. If an entity is going to play more than one role at a time, these roles have to be composed. Role model composition occurs during application analysis, as depicted in Figure 8. Role composition, on the other hand, occurs during design. As mentioned in Section 4, an individual role model focuses on a single context. When a role from one role models is assigned to a particular object, the object plays that role only in the given role model. A specific role is relevant only to a given context. When different role models are composed, it is important to indicate the context in which a role exists. As an example, Figure 9 illustrates the roles that an agent component plays in agent enhanced workflow (AEW). Because workflow or manufacturing formations can vary, each agent must be capable of being a Supply Chain Participant; this in fact means that it must be able to play the four lower level roles in Figure 7 (Customer, User, Provider, and Operator). Additionally, agent hierarchies will be variable, so an AEW agent should be able to be a Manager, a Subordinate, or a Client of a Bureaucracy. As shown in Figure 9, each role is represented as a role-context pair. Thus, an agent component plays a customer role in a Supply Chain and plays a manager role in a Bureaucracy, and so on.

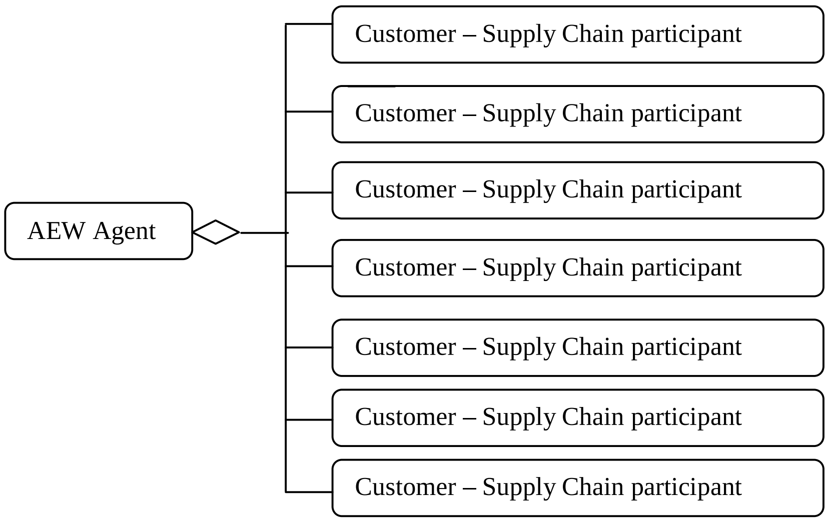
AEW Agent
Customer – Supply Chain participant
Customer – Supply Chain participant
Customer – Supply Chain participant
Customer – Supply Chain participant
Customer – Supply Chain participant
Customer – Supply Chain participant
Customer – Supply Chain participant

Figure 9. Role composition for an AEW agent component

6.3 Discussion: The approach described in section 5.1 and illustrated in section 5.2 achieved composability and decomposibility because individual components are composed to form a Supply Chain, a Bureaucracy, or the subsystem depicted in Figure 8. If a component plays many roles, the approach illustrated in section 5.2 may violate the small and few interfaces rules from section 2. In this case, the role composition must be refined to remove any conflicts, overlap, or redundancies. For example, in Figure 9, the Customer role from Supply Chain and the Client role from Bureaucracy may have some of the same behavior. In this case, redundant behavior can be removed, and the composed interface can be simplified.

Role composition addresses design. Object-oriented design patterns, such as the Role Object pattern [2] can be followed for subsequent implementation. Alternatively, more dynamic approaches [5], aspect-oriented programing [9] or subject-oriented programming [6] can be utilised.

7. Conclusions: I have proposed role models as abstractions and representations for components, based on the premise that components *collaborate* with each other in a specific *context.* i have illustrated that role modelling is pertinent to component design in the following ways:

- *A role model is context-specific.* A role model captures and provides a context in which we can describe how each component plays a given role.
- *A role model captures a pattern of object interaction.* Each role model is an abstraction that can be instantiated by specific applications. Such an abstraction can be used to represent both internal and external collaboration of components.
- *A role model provides richer semantics that go beyond an interface specification in a class.* Within a role model, roles have specific responsibilities.

- *A role model is dynamic.* A given component can play roles in different role models.
- *A role model is independent of implementation.* A role model and the roles within it provide a specification without any restriction on how it may be implemented.

The application of role modelling techniques to component design is still a new area. I suggest the following areas for future research:

- A formal design procedure for component role modeling.
- A formalisation of mappings between component roles and components.
- A syntactic and semantic specification of role and context composition.
- A formal specification of contracts between role interactions.

8. References:

1. E. P. Andersen. (1997) Conceptual Modeling of Objects: A Role Modeling Approach, Ph.D Thesis, University of Oslo.
2. D. Bäumer, D. Riehle, W. Siberski, and M. Wulf.(1997) "The Role Object Pattern." In Proceedings of 4 The Conference on Pattern Languages of Programs.
3. J.O. Coplien. (1997) "On the Nature of The Nature of Order," www.bell-labs.com/cope.
4. E. Gamma (1995) et al. Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley.
5. G. Gottlob, M. Schrefl, and B. Rock. (1996) "Extending Object-Oriented Systems with Roles". ACM Transactions on Information Systems, 14(3), 268-296.
6. W. Harrison and H. Osher, (1993) "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, pp. 411 - 428.
7. R. Helm, I. M. Holland, and D. Gangopadhyay, (1990) "Contracts: Specifying Behavioral Compositions in Object- Oriented Systems," Object Oriented Programming, Systems and Lanugages, ECOOP/ OOPSLA '90
Proceedings, October, pp. 169 - 180.
8. E.A. Kendall. (1998) "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.
9. E.A. Kendall.(1999) "Role Model Designs and Implementations with Aspect Oriented Programming," OOPSLA'99, Denver, November, 1999.
10. E.A. Kendall, (1999) "Role Modelling for Agent System Analysis, Design, and Implementation," International Conference on Agent Systems and Applications/ Mobile Agents (ASA/ MA'99),Palm Springs, October, 1999.(Submitted)

11 G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. - M. Loingtier, and J. Irwin, (1997) "Aspect Oriented Programming," Xerox Corporation, 1997. www.parc.xerox.com/spl/projects/aop/

12. B.B. Kristensen. (1996) "Object-Oriented Modelling with Roles", OOIS'95, Proceedings of the 2nd International Conference on Object-Oriented Information Systems, Dublin, Ireland.
13. B.B. Kristensen and D. C. M. May. (1996) "Component Composition and Interaction." Proceedings of International Conference on Technology of Object-Oriented Languages and Systems (TOOLS PACIFIC 96), Melbourne, Australia.

14. B.B. Kristensen and Osterbye, K., (1996)“Roles: Conceptual Abstraction Theory and Practical language Issues”, Special Issue of Theory and Practice of Object Systems (TAPOS) on Subjectivity in Object-Oriented Systems.

15. E.C. Lupu and Sloman, M., (1996) "Towards a Role Based Framework for Distributed Systems Management," Journal of Network and Systems Management.

16. B. Meyer. (1988) Object-Oriented Software Construction. Prentice Hall. New Jersey.

17. B. Meyer. (1998)Object-Oriented Software Construction. 2nd Ed. Prentice Hall. New Jersey.

18. Object Management Group (OMG). (1997) The Common Object Request Broker: Architecture and Specification. Version 2.1. August, 1997.

19. C. Pfister and C. Szyperski.(1996) "Why Objects Are Not Enough." Component Users Conference, Munich, Germany, 1996

20. T. Reenskaug, P. Wold, and O.A. Lehne. (1996), Working with Objects, The OOram Software Engineering Method, Manning Publications Co, Greenwich.

21. D. Riehle and T. Gross, (1998) "Role Model Based Framework Design and Integration," OOPSLA'98, Proceedings of the 1998 Conference on Object Oriented Programming Systems, Languages and Applications, ACM Press.

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

23. J. Skansholm. Ada From the Beginning, Addison Wesley, 1995.

24. G. Smith, J. Gough, and C. Szyperski. (1998),"Conciliation: The Adaptation of Independently Developed Components", Second International Conference on Parallel and Distributed Computing and Networks (PDCN '98), pp. 31-38. Brisbane. 14-16 Dec.

25. C. Szyperski. (1995) "Component-Oriented Programming -- A Refined Variation on Object-Oriented Programming", The Oberon Tribune, Vol. 1 (2).

26. C. Szyperski. (1998), Component Software - Beyond Object-Oriented Programming, Addison- Wesley / ACM Press, 1998 .

27. C. Szyperski and C. Pfister. (1997) “Workshop on Component-Oriented Programming, Summary.” In M. Muhlhauser (Ed.) Special Issues in Object-Oriented Programming –ECOOP96 Workshop Reader. dpunkt- Verlag, Heidelberg, 1997.

28. C. Szyperski and R. Vernik, (1998) "Establishing System-Wide Properties of Component-Based Systems: A Case for Tiered Component Frameworks," Position Statement to OMG-DARPA-MCC Workshop on Compositional Software Architectures, Monterey, January, 1998.

29. E. Yourdon and L. Constantine. (1978), Structured Design, Prentice Hall. New Jersey.

30 L. Zhao and T. Foster. (1999) "Modelling Roles with Cascade," IEEE Software, Sep/Oct.