INTERNATIONAL JOURNAL OF INTELLIGENT CONTROL AND SYSTEMS
VOL. 13, NO. 2, JUNE 2008, 97-108

New Java Mechanism for Role-Based Information System Development

Haibin ZHU and MengChu ZHOU

Abstract- **Object evolution and separation of concerns are two important issues in building a complex information system using object-oriented design and programming technology. The two issues are closely related to role playing activities in an organization. Roles have been discussed in object modeling for a long time. It is well-accepted that supporting roles in programming languages is required. This paper clarifies the role concept in the sense of object evolution and claims that a role should be taken as a fundamental component in object evolution. It then discusses the mentioned issues in software design and their resolutions. It proposes a principle that keeps the object identity and structure unchanged when passing messages to relevant roles. Next, it introduces the role play regulations, hierarchies and history recording mechanism. A package of classes for role play in Java is developed. Finally, it proposes a method to visualize the object evolution and role hierarchy. The work offers a conceptually solid and practically simple way to have objects in Java play roles. With the proposed methods, it is easy to express the evolution and separation of concerns of objects. The advantages over the previous methods are also discussed. The work assumes fundamental significance in building new information systems.**

***Index Terms*— Role, Object Evolution, Separation of Concerns, Information System, Software Development, Modeling and Simulation**

1. INTRODUCTION

People, goals and roles are the three essential components of an organization. Playing roles is a common activity in it. An information system imposes its own logic on a company's organization, strategy and culture [6]. Information systems are tools that help people perform their tasks better and more efficiently. Their goal is to help companies regulate their business processes [7]. Human roles are an important aspect of a system model [21]. It is important to introduce roles in modeling methodologies and role-based methodologies are emerging in different applications [22-25, 27, 28]. Roles have been widely used in organizations [3], project managements, software engineering and object modeling as they are good artifacts to deal with some aspects of these tasks and model natural phenomena more accurately. A common understanding is that when an object plays a role, it accepts messages and provides services related to its role. The introduction of roles to object modeling is to support object evolution and the separation of concerns [2, 5, 8-9, 12-14, 17-28]. They are closely related and always used to express the requirements of the role mechanisms in the modeling areas:

Received on February 2, 2007; revised on October 12, 2007. H. Zhu is with Department of Computer Science and Mathematics, Nipissing University, 100 College Drive, North Bay, Ontario, P1B 8L7, Canada, email: haibinz@nipissingu.ca. M. C. Zhou is with Electrical and Computer Engineering Department, New Jersey Institute of Technology, University Heights, Newark, NJ 07102, USA, email: zhou@njit.edu.

1) Object evolution states that an entity evolves and its status changes. A person plays a role of *student* in a year and may play role *professor* in a later year. It is required to express the person's status at different times.

2) Separation of concerns states that each concern of a given entity should be modeled by a single component in a system. A problem should be decomposed into entities such that each entity has one concern only. People may play roles *consultant* and *professor* during the same period of time. If you ask them "Could you teach a course?" or "Could you develop a system?" they should be able to properly respond to these requests within their roles.

As a tool to describe natural structures and phenomena, an ideal programming language provides direct modeling mechanisms to simplify the work of modelers. Therefore, providing role mechanisms in a programming language is a required task to support system modelers to deal with objects and the separation of concerns. Java is a popular general-purpose modeling and programming language widely used in many fields, especially information system development. Providing available, efficient and easy-to-use role mechanisms in Java is highly demanded by the designers, programmers and modelers of information systems.

Simulating their role playing is important in understanding the behaviors of people in organizations. Although there is some practice in supporting roles with object-oriented programming languages [5, 8, 19], many aspects are left for improvement, e.g., role definition, role playing, and expressing dynamic and traceable states and activities of objects. With the new approach proposed in this paper, it is expected to be easier for system modelers to simulate objects' collaboration activities, especially with roles. The proposed Java mechanisms and package can be directly used by system modelers in their applications requiring the use of roles.

This paper intends to help simulate objects' activities with roles more effectively, i.e., to express easily the separation of concerns of objects and object evolution in Java. A role is emphasized as an entity to express an object's current status in its evolving history. With this emphasis, we can extract many common properties of roles and evolving objects. The first beneficial application of the proposed mechanism is complex software system design. The second application is simulation of complex organizational systems. For example, we can build a role hierarchy of all the possible roles in a company and create role players to play these roles in a system simulation program. By simulation, we can review the past, view the

current and predict the future status of the company. Such capabilities should allow managers and strategists to identify problems in a company regarding roles and role players, i.e., people. The proposed methodology and implemented packages can be used in systems such as enterprise resource planning (ERP) systems [15] and agent-mediated systems [10].

The major contributions of this work are as follows:

- It reviews the work of supporting evolving objects in information systems.
- It proposes an approach to keeping the object structure and object identity unchanged when passing messages to relevant roles.
- It proposes role playing regulations and a mechanism to follow them;
- It proposes mechanisms to trace past roles and check whether an object can play particular roles;
- It develops an easy-to-use class package for role playing in Java; and
- It presents role playing graphs and role hierarchy graphs to visualize the evolution of objects.

The rest of this paper is organized as follows: Section 2 reviews previous contributions; Section 3 discusses the concepts and principles of roles and role players in object systems; Section 4 discusses the different aspects of expressing objects with roles; Section 5 describes the basic implementation of the proposed Java role package; Section 6 illustrates how to simulate role playing processes with the proposed mechanism and package; Section 7 demonstrates a simulation case with the implemented package; and Section 8 concludes this paper by indicating further research topics.

2. LITERATURE REVIEW

Despite different definitions of roles, it is beneficial to build practical role facilities in a programming language. The most significant contributions are made by Pernici [17], Albano et al. [2], Gotlob et al. [8], Schrefl and Thalhammer [19], and Dahchour et al. [5]. Respectively, they propose a model [17], a new language Fibonacci [2], an extension to Smalltalk [8], an extension to Java [19] and an extension to VODAK modeling language (VML) [5] to support the roles played by objects.

Pernici uses the role concept to model the separate behaviors of an object [17]. The work is motivated by the role concepts applied in office automation systems. An object can play different roles at different times, and may also play more than one role at the same time. Hence, partitioning messages for an object in terms of different roles has the advantage, i.e., allowing programmers to concentrate on one aspect at a time. A class with roles as its components is thus defined:

- Class ::= $\{N_c, R_0, R_1, \ldots, R_n\}$
- Role ::= $<N_r, P, S, M, U>$

In this definition, N_c is an identification of a class, N_r is an identification of a role, P is a set of properties, S is a set of states, M is a set of messages, U is a set of rules, and $R_{0\text{-}n}$ are roles. This definition actually divides all the methods and properties of a class into different groups; i.e., a role is a special group of properties and methods or services of a class. An object identity and the role identity must be explicitly pointed out when a message is sent. Consequently, this method is too complex to be practical. For example, one full page fails to describe a class Car with all the roles for a car [17]. To describe a class, one must clearly specify all the roles that might be played by the objects of this class, i.e., Pernici's method is essentially a static object instance space method to be discussed later.

Albano, et al. extend an object-oriented data model with the notion of objects with roles [2]. They implement an object-oriented programming language, Fibonacci, with roles as common components. The roles are designed as classes. An object can have several roles and the messages are sent to roles. The behavior of an object depends on the objects with roles. The answer of an object to a given message depends on the role that receives it. The roles express the different states of an object. An object may change its identification to express the different roles it plays. Fibonacci can extend an object to a new one with the same identity by assigning the extended data structure to the original identity. Therefore, a message is sent to the same object identity but actually to a new instance relevant to a role. An object is copied in part to form a new instance with the new role class. The new instance and the original instance are combined to form the state of the object and therefore to express object evolution. With the help of roles, objects may acquire new types and new behaviors, retain their identity and preserve the encapsulation. This method is the same as the dynamic object instance space method to be discussed later.

The necessity to introduce roles into object systems is deeply discussed [5, 8, 19]:

- Traditional object systems provide classes to support the static properties of entities in applications. Once a system is built, it is difficult to change the behaviors of the objects in the system. In other words, representing and maintaining objects using class hierarchies that require an object to be classified into a single class is a tough task [8].
- The relationship between an instance and a class is static and steady [19]. If one really wants to express continuously evolving objects, one needs to create subclasses and rebuild the system. Despite dynamically loaded class mechanisms in Java, it is difficult to express the evolution of objects originally existing in the system.
- To express object evolution in class-based programming languages is possible and tedious.

It is difficult for traditional object-oriented programming languages to express evolving objects because they apply specialization and inheritance at the class level [5, 8, 19]. Therefore, one solution is a role hierarchy in Smalltalk [8]

and Java [19] by applying specialization and inheritance at the instance level. The difference between role hierarchies and class hierarchies is that a subtype in a role hierarchy does not inherit the definitions of instance variables and instance methods from its supertype. Another solution [5] is a new role model and an extension to VML with role hierarchies and role control predicates implemented with metaclasses. These solutions emphasize the rules and methods for how to express the roles of objects, such as assigning a role, deleting a role, checking the roles played by an object, and establishing role relationships.

The major motivation to specify roles is to support object evolution in order to overcome the difficulties of representing and maintaining objects with class hierarchies [8, 19]. The roles are thus designed as special classes that support specific object properties and services. Therefore, roles support specific behavior that is different from that of the relevant classes. As a result, a class-based object-oriented system can be extended with roles and role hierarchies can enhance basic object-oriented principles such as classification, object identity, specialization, polymorphism and behavioral contexts. The object structures used in their methods are similar to the role instance space methods to be discussed later.

One may have several occurrences of a particular role type. The following example is cited from [8, 19]. Mr. Smith, who manages two projects "CAD/CAM" and "E-Business', is represented by two instances of ProjectManager. Each instance models Mr. Smith as project manager of either project. Mr. Smith has different responsibilities in each of them. He is responsible for "reuse management" in the former, and "quality assurance" in the latter. The qualifiers such as responsibility as "reuse management" and "quality assurance" are introduced to differentiate different role occurrences, i.e., if a real-world entity has several occurrences of that role, the role is specified as a qualified role. We deal with this situation with role subclasses and role instances because these occurrences are distinguished by the methods and states of the role instances. In fact, to express the several occurrences of a particular role type is similar to the requirement of active roles to be discussed later.

The existing message processing methods [8, 19] are mainly based on language mechanisms. Messages are searched within a class hierarchy. A message may implicitly make an object switch its roles in Smalltalk [8]. Roles must be switched explicitly in Java [19].

A metaclass extension to VML is proposed to support roles [5]. After all the contributions discussed above and other relevant publications prior 2004 are reviewed, the work [5] analyzes the properties and semantics of roles and role relationships in object systems. In its implementation, roles can be used to support the dynamic change of classes, multiple instantiation of the same class, and context-dependent access. Its major contribution lies in the proposed model and method that can support role relationships between a class of objects and a class of roles with metaclasses.

Three major problems are revealed in the past work:

1) It proposes no role-related regulations.

Such regulations can tell if an object is qualified to play a role based on its historical behavior. Without them, it would be very difficult to completely express object evolution. This work proposes a mechanism for modelers or programmers to specify role hierarchy and a set of algorithms to check if an object is able to play a role or not.

2) The message processing methods are mainly based on the mechanisms in programming languages.

By applying original language mechanisms, object status must be processed as separated instance spaces, thereby leading to impractical information system designs. This work solves this problem by using an integrated object state with dynamic spaces. An integrated object status reflects the real-world accurately. That is to say, in reality, one has many roles and all the messages to these roles are processed according to one's state directly, but not on one's each separate role's state.

3) The Java mechanism uses explicit role switching only [19].

Explicit role switching greatly affects the benefits of role playing mechanisms in object-oriented programming, because modelers or programmers are forced to express role switches in a program-leading to a complex program. It is almost similar to sending a message to a newly created object. Our proposed solution is providing a mechanism to switch roles implicitly by searching the roles to respond to the incoming message.

3. ROLES AND ROLE PLAYERS IN OBJECT SYSTEMS

In common sense, the term "role" is derived from the theater and refers to the part played by an actor. A role represents a specific status that possesses certain rights and accompanying responsibilities. It can be defined as a set of regulations defining what the behavior of a position member should be. A role defines a set of responsibilities and capabilities needed to perform the activities relevant to these responsibilities [1]. A role can also be defined as the prescribed pattern of behavior expected of a person in a given situation by virtue of the person's position in that situation. It is simply defined as a position in a social structure [3]. A position means a more or less institutionalized or commonly expected and understood designation in a given social structure such as an accountant, mother, or church member [3]. A role is a set of expectations about behavior for a particular position within a work system. Generally speaking, a role is a position occupied by a person in a social relationship. Occupying this position, one possesses special rights and takes special responsibilities.

Roles are a fundamental concept in modeling. They are entities that temporarily confine the behavior of objects. A role is considered as an abstraction and decomposition

mechanism related to objects. Objects may play roles. When an object plays a role, it accepts messages and provides services related to its role. A role constitutes a part of an object's behavior that is obtained by considering only the interactions of that role and hiding all other interactions. Roles are defined as a concept that is founded but not semantically rigid [9]. "Founded" means that a concept can exist essentially independently. "Semantically rigid" means that a concept contributes to the identity of its instances. This definition can help determine if a concept is a role. Objects that depend on other objects for their existence should not be roles. Roles allow not only for the representation of multiple views of the same phenomenon, but also for the representation of changes in time. Roles are also the bridges between different levels of detail in an ontology structure. The role analysis technique [14] is proposed to analyze dynamic programs. Its role concept reflects two important aspects of roles: the separation of concerns and role transition. The working behavior of an object represents the specific context in which it is defined, together with other objects. All the actions in working behavior belong to one or more of its roles.

There are many different discussions about the concepts of roles. Some aspects of roles seem contradictory. The comprehensive studies on roles as fundamental concepts and mechanisms [5, 12-13, 18, 20] are good guidelines for understanding roles in object systems. To support role playing with object-oriented programming languages, we need to confine our role concepts in the sense of data modeling and programming. Based on the basic view of object-orientation and the system modeling requirement, we can constitute the fundamental principles relevant to object evolution and separation of concerns based on roles. We clarify and extend object-oriented programming principles to the role playing principles as follows [11, 23-28]:

1) An object, or role player, or simply a player, evolves and has separate concerns in the system.
2) A role is an entity or data structure that expresses an aspect of an object.
3) A role can be created, changed and destroyed.
4) An object playing a role means that it can respond to the messages specified by the role.
5) A player can play many roles over a period of time but play one role at a time.
6) A role is independent of its players and can be defined independently.
7) A role in object systems is mainly concerned about its services. Its rights are implicitly expressed in the implementation of the services.

4. CLARIFYING OBJECT BEHAVIOR WITH ROLES

An object in a system may experience evolution in its life. However, traditional object-oriented programming languages can only express static behaviors of objects with classes. To express objects in behavior and status, roles are required. Object-oriented programming methodologies provide a very high-level abstraction to express concepts and ideas. Therefore, it is possible to express roles in such languages and extend them. Considering roles as a fundamental concept in a system, each object has a set of roles. An object can be manipulated directly, by sending messages to it, or indirectly by sending messages though a special mechanism to its roles only. To support object evolution completely, we need to consider many different aspects, such as the accommodation of separate concerns, message processing for roles, role hierarchies and accommodation of evolving status. This paper uses objects and role players or simply players interchangeably.

4.1 Separate concerns

People can change their status by playing different roles. For example, they can obtain school credits by playing the role of *student* and collect working experiences in a company by playing the role of *programmer*. The number of received credits is stored in a part of their memory while the collected experience in another. To support this requirement in information systems, we have the following solutions.

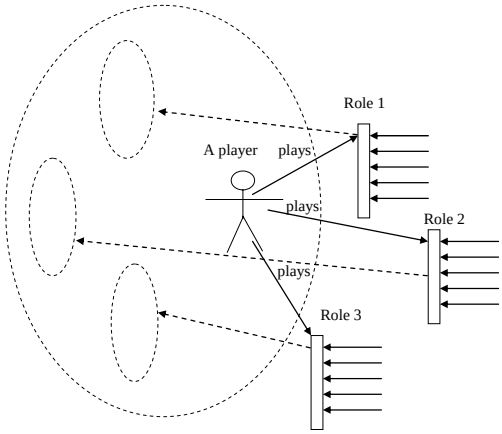


Fig. 1. The state of an object with the static object instance space

A. Static object instance space

A large enough memory space is created for an object as the case of a human being. People have a memory that is large enough to accommodate everything new throughout their life. When they play a new role, they allocate a part of their memory to remember the new information. In object-oriented programming, one can design a class with a large enough memory space for objects. When an object plays a role, it locates and uses a space (Fig. 1). Unfortunately, this is often impractical because of limited memory space and efficiency to accommodate all the different roles an object may play. Such problem is also encountered when a base class is specified [4]. The method in [17] is in fact similar to this one.

B. Dynamic object instance space

The space is expanded to accommodate a new space relevant to a new role. That is to say, only when an object plays a new role, does it allocate a new space corresponding to it. It is similar to that depicted in Fig. 1 and the difference is that the spaces relevant to roles are dynamically allocated instead of statically allocated. The language Fibonacci [2] actually adopts this method.

C. Role instance space

An object is allocated with the minimum required space statically when created. To evolve, role instances are created. The original object does not need to change. A new role instance is created and the new value is put in it. In this way, the status of an evolving object is composed of both the original object instance and all of the role instances (Fig. 2).

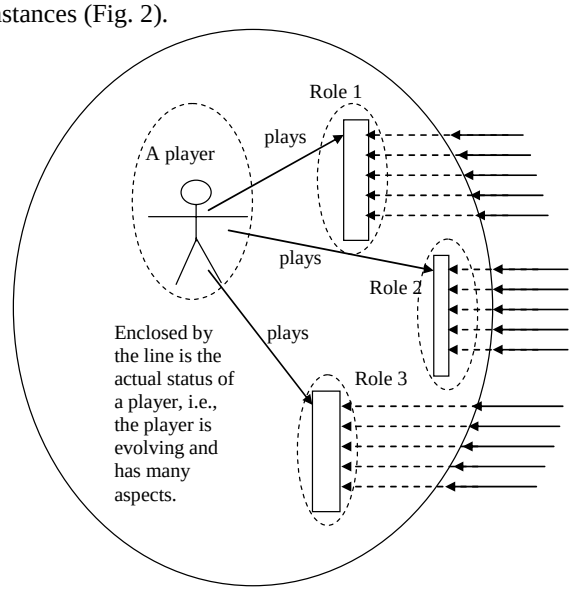


Fig. 2. The state of an object is composed of both an object instance and role instances.

For the previous example, a credit variable in the role instance can change when the role of *student* is played. The current status of a person is thus composed of both a person and his/her role instances. This method is practical because it is easy to create an instance of a role (Fig. 2). This method is used in the extensions to Smalltalk [8] and Java [19].

4.2 Message processing

In daily life, people accept and process a message based on the roles they play. In object-oriented software systems, an object accepts a message and its class determines how to process it. To have an object accept messages for the roles it is playing, we need to develop a specific mechanism. Usually, data structures determine the algorithms related to them [24]. Dealing with message passing for objects is up to the structure that accommodates the separate concerns.

If a static object instance space method is used, an object should be able to respond to messages relevant to the roles. The player must allocate some variables and have methods to record or modify its states in response to the requests to a role. The most difficult task is that the methods for a role must be able to modify the variables of the object. For example, to play the role of *student*, it is required to have a variable for recording credits and a method that *student* manipulates the variable. This makes the object and its roles tightly-coupled. It means that at the beginning of the object creation, we should have all the relevant methods of roles prepared for all the state space of the object. This method becomes infeasible if we do not know at the beginning what roles the player may play in the future. The modelers or programmers need to remember all the roles a player is currently playing in order to send messages that can be processed by the current roles. Therefore, it is impractical to implement all the relevant methods to all the relevant roles.

With the dynamic object instance space method, the messages are processed by the newly-playing roles and finally modify the newly allocated space. The player can respond to only messages relevant to the new role. It is difficult to respond to the messages relevant to the other roles that are not current (Fig. 1), however.

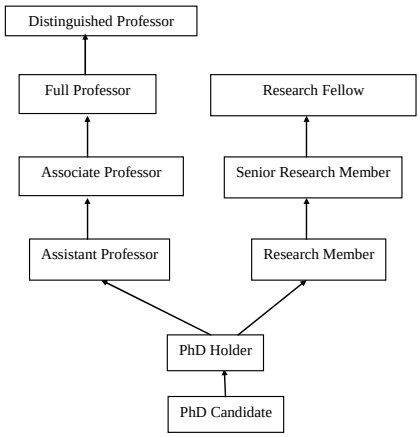
With a role instance, a message sent to an object is transferred to a role and is finally processed by the role's class. It is the role instance not the player instance that is processed by the method relevant to a message. In Fig. 2, messages are accepted by an object and then transferred to its relevant roles. To differentiate messages sent to an object directly and those sent to its roles, we need to provide a way to make such a difference and to have the role player transfer the messages to a relevant role. That is to say, a role player should be able to check its available roles and dispatch a message to a role instance. It saves the effort of programmers to consider such message matching. In other words, they do not need to consider what roles an object is currently playing. This method is therefore the best one.

4.3 Role hierarchy and role playing regulations

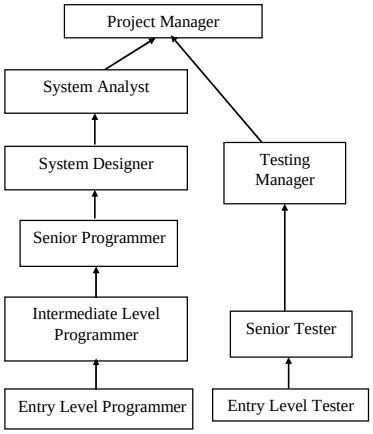
Objects play roles in a regulated way and they cannot play roles at will and at any time. Roles have relationships among themselves. That is to say, all the roles together should build a hierarchy. Nyanchama and Osborn compose a role graph model based on the requirements of role-based access control [16]. They discuss the relationships among roles and demonstrate five kinds of conflicts relevant to roles.

To support the simulation of real activities of societies and avoid conflicts among roles, some regulations in a programming tool are required. Based on the behavior of an organization and characteristics of management, the proposed following regulations should help programmers easily simulate the role playing activities and object evolution [1, 3, 24]:

1) Roles are defined as classes and abide by the class inheritance and classification relationships as they are designed.
2) Role players play their own role instances. Role instances follow hierarchies specified by designers.



(a)



(b)

Fig. 3 Examples of role hierarchies for (a) a researcher and (b) a member in a software team.

3) Each role instance may have super role instances and sub role instances. A bottom role instance is a role that has no sub role instances. A top role instance is a role that has no super roles. For example, in Fig. 3, *PhD Candidate*, *Entry Level Programmer* and *Tester* are bottom ones; *Distinguished Professor*, *Research Fellow*, and *Project Manager* are top ones. *Testing Manager* is a direct super role of *Senior Tester;* and vice versa, *Senior Tester* is a direct sub role of *Testing Manager*. To check if an object is able to play a new role, we enforce the following restrictions:
   a) It can play a direct super role of the active roles;
   b) It can play a bottom role;
   c) It can play a sub role of the active roles;
   d) It may play a role that is a past role; and
   e) It cannot play a role that conflicts with the player's active roles.
4) A role instance is unique for a player. This would relieve the extent of conflicts. In reality, if two people play a role of *Assistant Professor,* they actually create two instances of this role. A conflict between them occurs when they are accessing the shared resources of the two instances, such as a book and computer. To describe this situation by role instances, we can use static variables in the class of the role instances to express their shared resources.

These regulations minimize the problems such as role transitions and role conflicts [22]. By 2), we can avoid the user-user conflicts [16] because no role instance is played by two objects. By 3), we can avoid the role-role and user-role assignment conflicts by setting a role conflict policy before playing roles.

4.4 Evolving states

One may play many roles. For example, a person is a professor, technical consultant and project manager in the same year. To express this situation and be consistent with the principle "a player plays only one role at a time", we introduce the concepts of active roles and current roles as follows:

- By "active", we mean that the player responds to the messages relevant to these roles. Sometimes, an active role should be transferred to the current one to respond to the messages.
- By "current", we mean the object is currently playing this role, i.e., it directly responds to the messages relevant to this role.

These concepts are used to express the current state of an object. Active roles are ones that a player is holding. They are ready to respond to messages. The current role directly responds to messages coming to it. That is to say, a role player can hold many active roles at the same time but only one current role at a time. By holding only one current role, we can avoid role-role conflicts [16]. Active roles can also be used to express the meaningfulness of role transition, i.e., changing the current role from one active role to another.

To express evolution, we cannot avoid time. To know completely an evolving object, we need to recognize its past, know current, and predict future states. When an object evolves, we need to track its evolution status, i.e., its history. It is required to answer such questions: What roles did it play in the past? What are its active roles? What is its current role? To express the past of an evolving object, we need to introduce the concept of past roles:

- By "past roles", we mean the roles played by an object in the past and this role must be cancelled before.

Fig. 4 shows an example of the roles and their times played by an object.

Suppose that one plays a student role again after playing a project manager role, should we create a new role instance or replay the original role instance? This is left to the actual implementation. If it is really a different one, say, a graduate student, we need to create a new role instance. If the same student role instance as before is played, the original one can be used.

Are the three types of roles enough to express evolving objects? The answer is yes. We can use Fig. 4 to show this point. The horizontal axis stands for time and the vertical one for roles. To view an object's evolution with the above concepts, we can concentrate on the role transition of the object along the time axis horizontally. In Fig. 4, the current roles are expressed by the right most bold line segment and the active roles by the left most thin line segments. Other bold line segments express past roles. Suppose that the current time is t_c. The past roles of the object in Fig. 4 were R_1-R_3, the active ones are R_1, R_4 and R_n, and current one is R_n.

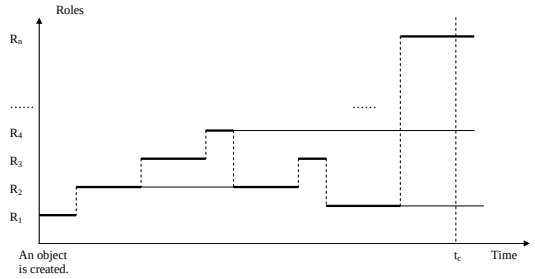


Fig. 4. The role play graph showing the roles an object plays.

In fact, separation of concerns is the other side of an evolving object. When we view an object along the time axis, we concentrate on its evolution. If we view an object at a definite time vertically, we see the different aspects of an object or the different roles it is playing, i.e., the separation of concerns.

5. IMPLEMENTATION

Everything in the world can be viewed as an object and every object has its class. From this point of view, to express new ideas with classes seems to be a panacea in object-oriented design. With the above discussion of evolving objects, we design a Java class package to support role play in Java. Only two major classes Role and RolePlayer are required to support roles and role players. Class Role describes the common properties of all roles. Class RolePlayer describes the common properties of all objects with roles. They are created and displayed at http://www.nipissingu.ca/faculty/haibinz/RolePlaying.

To define a role, one needs to define only a class that extends class Role. To simulate an object to play roles, one needs to define only a class that extends class RolePlayer. With class Role, we can easily express the relationships among roles, i.e., the role hierarchy. When an object plays a role, it actually attaches an instance of a role class that is a subclass of Role. With classes RolePlayer and Role, we can easily express the relationships between an object and a set of roles. In the main class, we can simulate object evolution, role transition and role play.

The most significant methods in class *RolePlayer* include *canPlay(), playAt(),*and *accept()*:

- The *canPlay(Role aNewRole)* method checks if the player can play a new role specified by its parameter by incorporating the regulations discussed in Section 3.3.
- The *playAt(Role aRole, Date aTime)* method modifies the trace records of past roles, active roles and the current role with a designated time. The current role trace is actually a record to express transitions among roles.
- The *accept()* method has overloaded methods such as *accept (String aMessage)* and *accept (String aMessage, Obejct [] args).* The first one is used to let programmers send a message without arguments to an object and the second one a message with arguments. These two methods implement automatic strategies to transfer messages from an object to its role and programmers do not need to care what roles the object is playing. They also support implicit role switching that is emphasized as an important improvement for role mechanisms in Java. The implementation of *accept()* methods includes application of the reflection mechanism of Java and a searching algorithm.

The most significant property of class *Role* is to specify the promotion relationships among roles. With *Vector* variables *subRoles*, *superRoles* and *conflictRoles*, we can express the hierarchy and conflict relationships of roles. With the methods of class Role, we can compare roles with respect to the role hierarchy. The *canPlay(Role aRole)* method checks if a role player with this role is qualified to play a role specified by the parameter and it helps the player to check if it can play a role via the *canPlay()* method in class *RolePlayer*.

There is an abstract method *instanceVariables()* in both classes *Role* and *RolePlayer*. This shows an important aspect of objects, i.e., the current state of an object includes values of the inherent object instance variables and its relevant role instance variables.

To express the history of evolving objects, we record time and introduce a variable *roleHistory* into the class Player which is a map to hold a role and a time to start and stop it. With this history record, we can express the object evolution shown in Fig. 4.

To support normal objects to play roles without changing their original class hierarchy and instantiations, we provide a class *ProxyPlayer* that is a subclass of *RolePlayer*. The proxy object is actually a wrapper of a normal object and has the same behavior as a role player.

To support the visualization of evolving objects, we implement a supplement but important class *RoleGraph*. In this class, we implement two static methods:

RoleGraph.drawHierarchy(ARole, height, width) and *RoleGraph.draw(aRolePlayer, height, width).* In both methods, we create a role panel and we can draw on this panel to show the object's role information, such as, role hierarchies, past roles, active roles and current role. To facilitate different simulation and graph drawing requirements, we differentiate six scales to draw a graph, that is, years, months, days, hours, minutes and seconds, which are used to determine the width of a panel. Another important point to design class *RoleGraph* is to arrange the roles in the panel. In our implementation, we allow the duplication of role names, i.e., if a role is in both the past and active role records, it is shown on the panel twice.

The Timer class is used for simulating the durations of role playing.

In our implementation, we have to answer "where should we implement methods." Two ways are considered:

a) In the role class: all the methods are implemented in the role class and operate on the role instances.
b) In the player class: all the methods in the role class only transfer the message back to a player and the method of the player class operates on the instance of the player.

The first way is practical, easy-to-understand and efficient with object-oriented paradigms. The second one is impractical because the messages are required to be transferred between the player class and the role class. It is clumsy and wasteful that the message is transferred back and forth between a player and its role.

In our implementation, we keep the message passing tradition of object-oriented programming, i.e., a message is sent to an object. To differentiate the messages to objects from those to roles, we use a special message transferring method *aPlayer.accept ("aMessage")* to mean the messages relevant to the roles of an object.

According to the sequential programming idea, an object can only play one role at one time. One role instance can be played by one player only. It should be beneficial to model players in this sequential way.

6. PLAYING ROLES IN JAVA

To play roles, one often faces such challenges as who specifies roles, who plays roles, how to specify roles, how to play roles, how to transfer roles and how to respond to messages. These questions can easily be answered with the help of our proposed and implemented ideas. Here, a designer or programmer specifies roles and role players using classes. Societal phenomena can be simulated by performing the following processes:

1) Specify role players: every player is an instance of a subclass of the class *RolePlayer*. One needs to define a subclass of *RolePlayer* to specify a role player class and must implement the method instanceVariables(), because it is defined as an abstract method in the *RolePlayer* class.
2) Specify roles: every role is an instance of a subclass of the class Role. One needs to define a subclass of Role to specify a role class and must implement the method *instanceVariables*(), because it is defined as an abstract method in the *Role* class.
3) Instantiate roles: every role is an instance of a subclass of the class *Role*. One must instantiate a role in order to play it.
4) Specify the role hierarchy: there might be super or sub relationships among roles. One needs to specify these relationships by setting the *superRole* and *subRole* variables of a role instance.
5) Specify the role conflict policy: every role may have conflicting roles. One needs to set conflictRoles for a role instance.
6) Play roles: an instance of player may play a role instance by *anPlayer.play(aRole)* or *anPlayer.playAt(aRole, aTime),* where *aPlayer* is an instance of a subclass of class *RolePlayer*, *aTime* is the assumed time to play the role. The class *RolePlayer* specifies some restrictions for a player to play a role.
7) Arrange messages: an object responds to messages relevant to roles by *accept(String aMessage), accept(String aMessage, Obejct [] args), accept(Role aRole, String aMessage)* and *accept(Role aRole, String aMessage, Obejct [] args),* where *aMessage* is the message name, args is the argument vector for the methods relevant to the message, and *aRole* is a role instance. A player can accept all the messages relevant to the active roles. If many roles have methods with the same message name and the role is not specified, the message will be processed by the current role or the first active role identified in the active role vector. A player does not respond to the messages relevant to past roles.
8) Have objects play roles: this is to have an object play roles. One needs to instantiate an instance of the class *ProxyPlayer* with the object as the instantiating parameter. The proxy player plays roles in the same style as role player.

The following tasks are accomplished by the proposed package:

1) Transfer roles: our method supports two kinds of role transitions: implicit or explicit. By "explicit", we mean that a role player transfers the current role to an active role explicitly by a message *aPlayer.transfer(aRole).* By "implicit", we mean that if a role player accepts a message that cannot be resolved by the current role, it automatically searches for an active one that can resolve this message and the current role transfers to this role.
2) Resolve the separation of concerns issue: when a player plays a role, it only accepts the messages relevant to the current role and active roles. Each active role considers one concern. Hence,

concerns are separated.

3) Resolve the evolution of a player: a player has a vector of roles it played in the past. The element is added to the vector chronically. It explicitly expresses the history of a player. A player can cancel an active role. A canceled role is put into the past role vectors.

7. SIMULATION OF ROLE PLAY

We use a common simulation problem to demonstrate the proposed extension to Java is practical. This problem is to simulate a person, Mr. Smith, who plays different roles at different time segments and provides different services.

1) He first plays a role of Student as his current role and accepts messages such as "study" and "takeExam".
2) He plays a role of Programmer as his current role and provides services such as "writeCCode" and "writeJavaCode".
3) His current role is transferred to Student, because he must respond to messages such as "study" and "takeExam".
4) His current role is transferred back to Programmer, because he must respond to a message "writeJavaCodeWithComments".
5) He rejects the messages such as "test" and "writeTestPlan", because his active roles Student and Programmer are not relevant to them.
6) He rejects to play the role of Tester, because the role Tester is in conflict with the role Programmer (Fig. 6).
7) He rejects the messages "test" and "writeTestPlan" relevant to the role of Tester.
8) He plays the role of Designer and accepts the messages "designASystem" and "writeDesginDoc".
9) His role of Programmer is canceled.
10) He can play the role of Tester and he can accept the messages "test" and "writeTestPlan".
11) He plays the role of Project Manager and accepts messages such as "writeAPlan" and "negotiateAContract".
12) His role of Project Manager is canceled and his current role is transferred to an active role of Student.
13) He replays the role of Project Manager.

To simulate this process, five roles are implemented as subclasses of Role: Student, Programmer, Designer, ProjectManager and Tester. Five role instances s, p, d, pm and ts are instantiated. An instance of Person (a subclass of RolePlayer) is created as smith.

Fig. 5 shows the object smith, where the red (bold) line segments express the traces of current roles; blue (thin) line segments express the past roles and the green (grey) line segments are active roles. From this graph, we know that Student, Programmer, Designer and ProjectManager are the active roles played by smith; the current role is ProjectManager and the past roles are Programmer and ProjectManager; and the Person starts to work at second 1 ("S" in the graph) that can be scaled to minutes, days, months, and years. To simulate this process, we only need the following simple statements to compose the whole simulation. Note, after "//" are comments.

```
s.superRoles.add(p);   //Adding super roles
pm.subRoles.add(d);    //Adding sub roles to form the
                       //hierarchy of roles
Timer.passSeconds(1); //Simulate the time period for a
                       //player to play a role
smith.play(s);         //Play a role instance s
smith.transfer(d);     //Transfer the current role to an
                       //active role d
RoleGraph.draw(smith, 5, 1);  //Draw the graph for the
                              //evolving object smith.
```

In the above simulation, we can find that the characteristics of the proposed methods:

- The identity and data structure of a role player are kept unchanged;
- Messages are sent to players in the same style as that in object-oriented programming languages;
- Specifying roles is the same as specifying classes;
- Java programmers can easily play roles with the implemented package; and
- To have old objects play roles, we only need to create a proxy player with class ProxyPlayer.

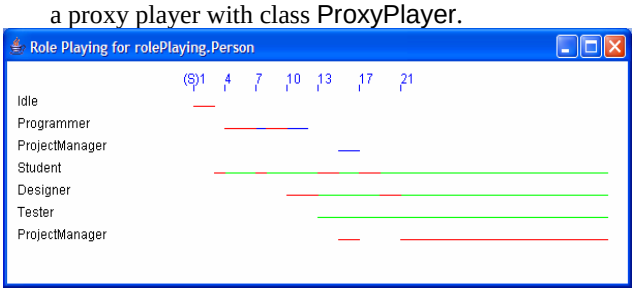


Fig. 5 The graph drawn by the simulation program

Fig. 6 shows the hierarchy and Fig. 7 shows the text result of the simulation. In this simulation, we use real time and the time scale is in seconds. The whole simulation program is only about 250 lines of simple Java source code. To simulate with assumed time, we can use the following statements.

```
Date t = new Date(100,1,2,0,1,0);
        //t is 00:01:00 of the 100th day in the current year

RolePlayer smith= new Person("Smith", t);
        //It means smith is created at the time specified by t
t = new Date(100,1,4,1,3,1);
        //t is 01:03:01 of the 100th day in the current year
smith.playAt(s, t);      //Smith starts to play role s at t
```

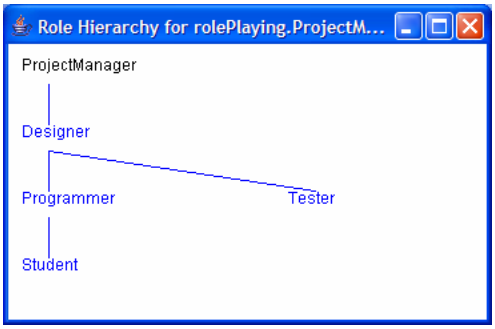


Fig. 6. The role hierarchy in the simulation.

```
A Student Role instance is created.
A Programmer Role instance is created.
A Designer Role instance is created.
A Project Manager Role instance is created.
A Tester Role instance is created.
Simulation Begins:
null
rolePlaying.Student
I am Studying.
I am taking an exam.
My Salary is $0.
rolePlaying.Programmer
I am writing C codes.
I am writing Java Code.
I am Studying.
I am taking an exam.
rolePlaying.Student
I am writing Java Code with comments: Be careful
My Salary is $60,000.
rolePlaying.Programmer
There is no an active role to respond such a message.
There is no an active role to respond such a message.
My Salary is $60,000.
rolePlaying.Programmer
You can not play this role instance.
rolePlaying.Programmer
There is no an active role to respond such a message.
There is no an active role to respond such a message.
rolePlaying.Designer
I am designing a system.
I am writing a design document.
My Salary is $80,000.
Programmer role canceled.
rolePlaying.Tester
I am testing a program.
I am writing a test plan.
rolePlaying.Student
rolePlaying.ProjectManager
I am writing a plan.
I am negotiating a contract.
My Salary is $100,000.
Project Manager role canceled.
rolePlaying.Student
My Salary is $0.
There is no an active role to respond such a message.
There is no an active role to respond such a message.
rolePlaying.Designer
rolePlaying.ProjectManager
Smith
Smith
My salary is 60000.
My project is a data base system.
My credit is 6.
My experience is a designer.
I am testing a data base system.
My project is a data base system.
Simulation ends.
```

The object smith plays the role of Student and accepts the messages relevant to it.

The object smith plays the role of Programmer and accepts the messages relevant to it.

The object smith transfers to the role of Student because the messages are relevant to it.

The object smith transfers back to the role of Programmer because the messages are relevant to it.

The object smith rejects the messages not relevant to its active roles (Student and Programmer) accepts the message “salary” to Programmer.

The object smith rejects to play the role of programmer because it is conflict with the active role Programmer.

The object smith plays the role of Designer and accepts the messages.

The role of programmer is canceled and smith can play the role of Tester and accepts the messages.

Smith plays the role of Student again and then plays the role of Project Manager.

Project manager is canceled and the current role is set to student The messages to project manager are not accepted.

The designer is transferred.
The project manager is replayed.

This is all the relevant instance variables of smith and relevant roles.

Fig. 7. The text output of the simulation.

8. CONCLUSION

Role playing is an important aspect for management. Simulating a role playing process can help managers understand the performance of an organization to avoid risks significantly in resource management.

We have presented a new mechanism to express objects with roles in Java. It significantly improves the existing methods [5, 8, 19] by:

- Keeping the object structure and object identity unchanged when passing messages to relevant roles;

- Introducing the role playing regulations, past roles and hierarchies in the proposed role playing mechanism;
- Implementing an easy-to-use class package for role playing in Java; and
- Visualizing the object evolution via a role playing graph and role hierarchy graph.

Table 1. A comparison among the contributions of different authors to role playing mechanisms

Ref.	Platform	Separate concerns	Role Hierarchy	Object Structure	IRS	RPR	Past roles	V
[17]	An abstract model	Yes	No	Static	No	No	No	No
[2]	Fibonacci	Yes	Role types	Dynamic	No	No	No	No
[5]	VML	Yes	Role types	Role instance	No	No	No	No
[8]	Smalltalk	Yes	Role types	Role instance	Yes	No	No	No
[19]	Java	Yes	Role types	Role instance	No	No	No	No
This work	Java	Yes	Role types and role instances	Role instance	Yes	Yes	Yes	Yes

Keys: IRS-Implicit Role Switching; RPR- Role Playing Regulations; and V- Visualization

Table 1 shows the comparison among the relevant contributions on roles to support objects based on the discussion of the fundamental requirement of separation of concerns of objects (Section 3). Most previous work did not emphasize the role instance hierarchy, role playing regulations, expression of past roles, and visualization of evolving objects. Most role mechanisms used in [5, 8, 19] and this paper are role instance method to model evolving objects. The reason is that it facilitates the message distribution of an object [8]. This work improves the method of [19] by allowing a player to switch current roles when its current role cannot respond to an incoming message.

Readers who are interested in applying the package to their information system development projects can contact the authors. Some future improvements can be obtained by:

- Simplifying the procedure of defining roles and role playing procedures with graphics user interfaces (GUI), i.e., modelers simulate roles and role playing without using Java;
- Extending the package in order to support multi-player simulation to simulate the collaboration among objects; and
- Applying and extending this package into multi-object systems to encourage objects to contribute to the systems [24].

ACKNOWLEDGMENTS

This research is funded in part by National Sciences and Engineering Research Council, Canada (NSERC, No. 262075) and IBM Eclipse Innovation Grant Funding.

REFERENCES

[1] Acuna, S.T. and Juristo, N., "Assigning people to roles in software projects," *Software-Practice and Experience*, vol. 34, 2004, pp. 675-696.

[2] Albano, A., Bergamini, R., Ghelli, G. and Orsini. R., "An object data model with roles," In *Proc. of the International Conference on Very Large Databases*, 1993, pages 39-52.

[3] Ashforth, B. E., *Role Transitions in Organizational Life: An Identity-based Perspective*, Lawrence Erlbaum Associates, Inc., 2001.

[4] Budd, T., *An Introduction to Object-Oriented Programming (3rd Ed.)*, Addison-Wesley, 2002

[5] Dahchour, M., Pirotte, A., Zimányi, E., "A role model and its metaclass implementation," Information Systems, vol. 29, no. 3, May 2004, pp. 235 – 270.

[6] Davenport, T. H., "Putting the Enterprise into the Enterprise System", *Harvard Business Review*, 1998, vol. 76, no. 4, pp. 121-131.

[7] Fan, M., Stallaert, J., Whinston, A.B., "The adoption and design methodologies of component-based enterprise systems", *European Journal of Information Systems*, vol. 9, no. 1, March 2000, pp. 25-35.

[8] Gottlob, G., Schrefl, M. and Röck, B., "Extending Object-Oriented Systems with Roles," *ACM Transactions on Information Systems (TOIS)*, vol. 14, no. 3, July 1996 , pp. 268-296.

[9] Guarino, N., "Concepts, Attributes and Arbitrary Relations: Some Linguistic and Ontological Criteria for Structuring Knowledge Bases," *Data & Knowledge Engineering*, vol. 8, 1992, pp. 249-261.

[10] Guttman, R.H., Moukas, A.G., Maes, P., "Agent-mediated electronic commerce: a survey", *The Knowledge Engineering Review*, vol. 13, 1998, pp. 147-159.

[11] Kay, A. C., "The Early History of Smalltalk," *The Second ACM SIGPLAN History of Programming Languages Conference*, ACM SIGPLAN Notice, vol. 28, no. 3, March 1993, pp. 69-75.

[12] Kendall, E. A., "Role Model Designs and Implementations with Aspect Oriented Programming," in *Proc. of ACM 1999 Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99)*, CO, USA, Nov. 1999, pp. 353-369.

[13] Kristensen, B. B. and Østerbye, K., "Roles: Conceptual Abstraction Theory & Practical Language Issues," Special Issue on Subjectivity in Object-Oriented Systems, *Theory and Practice of Object Systems*, vol. 2, no. 3, 1996, pp. 143-160.

[14] Kuncak, V., Lam, P. and Richard, M., "Role Analysis," *The 29th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL'02)* Portland, OR, USA, Jan. 2002, pp. 17-31.

[15] Nah, F.F.H., Lau, J.L.S., Kuang, J., "Critical factors for successful implementation of enterprise systems", *Business Process Management Journal*, vol. 7, no. 3, Aug 2001, pp. 285-296.

[16] Nyanchama, M. and Osborn, S., "The Role Graph Model and Conflict of Interest", *ACM Transactions on Information and System Security*, vol. 2, no. 1, 1999, pp. 3-33.

[17] Pernici, B., "Objects with Roles", *ACM SIGOIS Bulletin, Proceedings of the Conference on Office Information Systems*, March 1990, vol. 11, no. 2-3, pp. 205-215.

[18] Riehle, D., Brudermann, R., Gross, T. and Mätzel, K. U., "Pattern Density and Role Modeling of an Object Transport Service", *ACM Computing Surveys (CSUR)*, vol. 32, no. 1, March 2000, pp. 1-6.

[19] Schrefl, M. and Thalhammer, T., "Using Roles in Java", *Software - Practice and Experience*, vol. 34, 2004, pp. 449-464.

[20] Steimann, F., "On the representation of roles in object-oriented and conceptual modeling", *Data & Knowledge Engineering*, vol. 35, 2000, pp. 83–106.

[21] Williams, T.J., "The Purdue Enterprise Reference Architecture and Methodology (PERA)", In Molina, *A., Sanchez, J.M., Kusiak, A. (Ed), Handbook of Life Cycle Engineering: Concepts, Tools and Techniques*, Chapman & Hall, London, 1998, avail: http://iies.www.ecn.purdue.edu/IIES/PLAIC/Enterprise-Handbook_PERA.pdf .

[22] Zhu, H. "Conflict Resolution with Roles in a Collaborative System", *International Journal of Intelligent Control and Systems*, vol. 10, no. 1, 2005, pp. 10-19.

[23] Zhu, H., "A Role Agent Model for Collaborative Systems", *Int'l Conference on Information and Knowledge Engineering*, USA, June 2003, pp. 438-444.

[24] Zhu, H., "Role Mechanisms in Collaborative Systems", *International Journal of Production Research*, vol. 44, no. 1, Jan. 2006, 181-193.

[25] Zhu, H. and Zhou, M.C., "Role-Based Collaboration and its Kernel Mechanisms", *IEEE Trans. on Systems, Man and Cybernetics, Part C*, vol. 36, no. 4, July 2006, pp. 578-589.

[26] Zhu, H. and Zhou, M.C., "Methodology First and Language Second: A Way to Teach Object-Oriented Programming", *Companion of the 2003 Object-Oriented Programming, Systems, Languages and Applications*, CA, USA, Oct., 2003, pp. 140-147.

[27] Zhu, H. and Zhou, M.C., "Roles in Information Systems: A Survey", *IEEE Trans. on Systems, Man and Cybernetics, Part C*, vol. 38, no. 3, May 2008, pp. 377-396.

[28] Zhu, H., Zhou, M.C. and Seguin, P., "Supporting Software Development with Roles", *IEEE Trans. on Systems, Man and Cybernetics, Part A*, vol. 36, no. 6, Nov. 2006, pp. 1110-1123.

Haibin Zhu is an Associate Professor of the Department of Computer Science and Mathematics, Nipissing University, Canada. He received B.S. degree in computer engineering from Institute of Engineering and Technology, China (1983), and M.S. (1988) and Ph.D. (1997) degrees in computer science from the National University of Defense Technology (NUDT), China. He was a visiting professor and a special lecturer in the College of Computing Sciences, New Jersey Institute of Technology, USA (1999-2002) and a lecturer, an associate professor and a full professor at NUDT (1988-2000). He has published 70+ research papers, four books and one book chapter on object-oriented programming, distributed systems, collaborative systems and computer architecture. He is serving and served as co-chair of the technical committee of Distributed Intelligent Systems of IEEE SMC Society, editor for the Int'l J. of Intelligent Control and Systems, member of the Domain Experts Board of International Journal of Patterns, member of the editorial board of International Journal of Software Science and Computational Intelligence, guest editor for the special issue of "Collaboration Support Systems" for IEEE Trans. on SMC(A), guest associate editor for a special issue for the Int'l Journal of Pervasive Computing and Communications, and program committee member for more than 20 international conferences. Dr. Zhu is a senior member of IEEE, a member of ACM, and a life member of the Chinese Association for Science and Technology, USA.

MengChu Zhou received his B.S. degree in Electrical Engineering from Nanjing University of Science and Technology, China in 1983, M.S. degree in Automatic Control from Beijing Institute of Technology, China in 1986, and Ph. D. degree in Computer and Systems Engineering from Rensselaer Polytechnic Institute, Troy, NY in 1990. He joined New Jersey Institute of Technology, Newark, NJ in 1990, and is currently a Professor of Electrical and Computer Engineering and the Director of Discrete-Event Systems Laboratory. His research interests are in Petri nets, computer-integrated systems, and semiconductor manufacturing. He has about 300 publications including 6 books, 120+ journal papers, and 16 book-chapters. From Scopus database, in the area of Petri nets, Dr. Zhou ranks the top one in terms of the number of publications; and top two in terms of the number of citations. He ranks the top one in the area of automated manufacturing systems in both indices. He is Associate Editor (AE) of IEEE Trans. on Robotics and Automation and IEEE Trans. on Automation Science and Engineering, and currently is Managing Editor of IEEE Trans. on Systems, Man and Cybernetics: Part C, AE of IEEE Trans. on Systems, Man and Cybernetics: Part A and IEEE Trans. on Industrial Informatics.