How to Solve the Inefficiencies of Object Oriented Programming : A Survey Biased on Role-Based Programming

Yunus Emre Selcuk
Computer Engineering Department, Istanbul Technical University
Ayazaga, 34390, Istanbul, TURKEY

and

Nadia Erdogan
Computer Engineering Department, Istanbul Technical University
Ayazaga, 34390, Istanbul, TURKEY

ABSTRACT

This survey discusses the inefficiency of the OOP paradigm in modeling real world objects that change with time and focuses on the role concept as a solution. The most representative work towards role support in OOP languages are presented, followed by an evaluation and comparison according to various criteria.

Keywords: object oriented programming, role based programming, design.

1. INTRODUCTION

In the class-based object-oriented programming (OOP) paradigm, the relationship between an object and its respective class is persistent, static and exclusive. This makes OOP suitable for modelling only real world objects that can be divided into distinct classes and never change their classes. However, the real world mainly consists of objects which constantly change and evolve. The best example is a human being. More examples can be given in domains such as production mechanisms and classification of technical materials.

Modeling objects that change in time with the class hierarchies of OOP is difficult. An object should be re-classified each time it evolves. It is even more difficult to model an object which can have different roles independent of each other. To classify such objects, seperate classes must be defined for each possible combination of roles. These classes are defined as intersection classes and usually created by multiple inheritance.

Behavioural classification can be modeled with inheritance but many OOP languages bind the identity of an object to only one type permanently and invariably. This is not a mere inconvenience but a major problem because it dictates to model the dynamically changing real world objects with static classes. If dynamic entities are modeled with a series of static objects, the relation between the conceptual entity and these objects can be lost or overlooked.

Specialization at the instance level is a better approach than specialization at the class level when modeling evolving entities. An entity can be represented with multiple objects, each executing a different role that it is required to perform. These objects do not simply change from one into other : An entity can have multiple unrelated types. If multiple inheritance is used to model all types, an exponantially growing tree of the class hierarcy will be sparsely populated with necesary objects. Moreover, name conflicts can occur as OOP provides only one behavioral context for any object.

2. DETAILED DISCUSSION OF THE PROBLEM

With the traditional class hierarchy, where derived sub classes inherit the variables and methods of the super class, only the sharing of attributes and behaviour can be modeled. Remaining properties stated below are hard to implement.

2.1. It is Hard to Model Evolving Entities :

If an object gains a new role or looses one, it needs to be re-classified. This seemingly easy task consists of the following steps :

- A new object of the new class is created
- Necessary members and methods of the old class are copied to the new class.
- References of the old type should be converted to the new type in the entire system (the biggest burden in large scale applications).
- The old object is removed.

2.2. Class Hierarchy Grows Exponantially Where Entities Can Assume Multiple Roles

The previously defined intersection classes, usually implemented with multiple inheritance, grow exponentially with each role. The work of embedding necessary functionality into the intersection classes will be hard in some OOP languages which do not support multiple inheritance.

Naming conflicts are inevitable in multiple inheritance. Even though each language supporting multiple inheritance has a way to prevent or resolve these naming conflicts, the task of using these tools while maintaining both syntactic and semantic correctness is still a big burden for programmers.

2.3. Class Hierarchies do not Support Context Based Behaviour

When an object is accessed from a specific view point representing one of its roles, it should act as this particular role suggests. This is not possible in class hierarchies of class-based inheritance.

2.4. Class Hierarchies are not Flexible in Restriction of Access

All members and methods of a class are always accessed in the same way. Members defined as public or private are always accessible to objects having the right while they are inaccessible by others. However, context based behavior inherent in the real world suggests context based access.

2.5. Class Hierarchies cannot Model the Representation of an Entity with Multiple Objects

A real world entity is usually represented by only one object belonging to a particular class. If multiple objects are involved, the fact that all these objects represent the same entity is lost. Programmers need to implement additional mechanisms to keep that information, such as a member for labeling purposes.

3. THE ROLE CONCEPT AS A SOLUTION

Although OOP is a widely accepted and powerful modeling tool, its inherent inefficiencies have motivated researchers to come up with new techniques. Among those, we've observed that role based programming (RBP) approach is an elegant way to overcome the inefficiencies of OOP. Other techniques include schema evolution and object migration.

The role concept comes from the theoretical definition where it is the part of a play that is played by an actor on stage. Roles are different types of behavior that different types of entities can perform. An entity can play some of these roles simultaneously while acquiring or losing roles during its lifetime. Therefore, an entity is defined with all of the roles it is interested in.

The role concept with the above context was first proposed by Bachman and Daya in 1977 [1]. The foundation of the approach was built and examples were given in a COBOL-like language. Work that applied these concepts to OOP started in late 80's and still continues.

3.1. Characteristics of Roles

The following requirements should be satisfied in order to model the dynamic nature of real world entities:

- Some roles of an object can share commont structure and behavior. For example, student and research assistant roles of a person share name information. OOP can fully satisfy only this requirement.
- Objects should be able to acquire and loose roles dynamically. For example, an employee can be promoted to leadership of a project but can later be dismissed if he proves incapable.
- Roles can be won or lost independently of each other. For example, a person can find a part-time job independent of being a student.
- Entities behave just like their roles dictate. For example, different answers are received if one is asked his phone number, either home or work, according to his current role. Polymorphism mechanism of OOP gives only limited satisfaction to this requirement.
- An entity can have more than one instance of a role. For example, an employee can be a group leader in more than one project.
- An entity could switch between its roles any time it wishes. This means that a programmer should be able to access any of the roles of an object when he has a reference to only one of its roles.
- Different roles should be allowed to have members with same names without conflicts.
- There should be some sort of priority among roles because of the previous property. It is also implied that previous behavior, which is hidden by acquiring a new role, can reappear by losing that role.
- Some dependency rules may be needed when acquiring or loosing a role.
- Some restrictions may be needed among the primary characteristics of roles. For example, some roles should be defined as mutually exclusive.

4. ROLE BASED APPROACHES

This section scrutinizes the work, which aims to solve the inefficiencies of OOP by implementing the role concept. They all focus on the relational approach for roles and have a hierarchy of roles, which stems from the object-level inheritance relations among roles. However, we examine the role hierarchy approach in a separate section as it differs from the others by providing role support simply by a few special classes and using no other mechanism but the ones supported in Smalltalk.

4.1. Object Interfaces (Aspects)

Skarra and Zdonik [3] suggest the use of object interfaces as a mechanism to solve the problems of OOP. The original term used is *aspects*, which are augmented class interfaces similar to the ones found in Java language. In order to avoid confusion with aspect oriented programming paradigm and Java interfaces, the term *object face* – or simply *face* – is used instead of *aspect*. The face concept suggested in this work has not been implemented in a programming language. Object faces support the following properties while a Java interface does not.

- Dynamically defined during runtime.
- A face can implement other faces.
- Different kinds of equity-checking operators are defined.
- Only permitted classes can implement a face.
- An object can have multiple instances of a face.

A face gives an object additional state and behavior, while not changing its identity. An object may acquire a new face or drop one of its faces. An object which has more than one face belongs to all classes from which it has taken its faces. The behavior of an object changes according to its face, through which it is being accessed .

4.1.2. Object Model, Faces and Usage

The proposed model is inspired by the Emerald language [4]. Abstract data types and face implementations are separated: An abstract type is a set of method signatures and defines an interface; while an object is defined as an implementation, which realizes the codes for the methods stated in the interface, together with the additional presentation of itself. A face is a set of method signatures and an object should have all the methods defined in the face with the same signatures. A face is created by augmenting an implementation with the base object, which is to be extended. Each face of an object is in fact a role of that object.

Abstract types are related to each other and to implementations by the *rule of conformity*. Consider the abstract types of A and B. If A provides all the method signatures (or maybe more), than it is said that B conforms to A. B still conforms to A if B is a class instead. As long as this rule of conformity is satisfied, it is also said that B implements A. Abstract types can implement each other.

Rule of conformity allows to declare a general type and to create different specialized types sharing the common behavior. Specialized objects conform to the general one and they can be used in any context where the general object is expected. There is not a class-based inheritance; instead, this defines an object based inheritance mechanism. Moreover, additional super types can be implemented without changing the definitions of existing subtypes.

With the combination of abstract types, implementations and rule of conformity; a mechanism for supporting roles is proposed. A face can add new state and behavior as well as it can import or redefine desired parts of the base object's interface. Non-redefined methods do not exist by default, unlike class-level inheritance. The need for importing or redefining all methods of the object's interface for satisfying the role of conformity still exists. This adds data and behavior hiding properties but can also be perceived as extra work. The face can access the base object's members and methods by using a special reference named `base`. `this` is another operator that points to the face whose method is called. Redefined or imported methods do not raise naming conflicts. However, `this` or `base` object's members can be renamed to avoid semantic confusion.

A base object can have more than one instance of the same interface. The distinction of faces is achieved by supplying different values to their constructors.

To demonstrate the definition and usage of faces, we have used a language representation similar to Java in Figures 1 - 3.

```
type Person /* abstract type... */ {
String name();
String phone(); };
implementation personlmpl
/* ...and its implementation */ {
String myName, myPhone;
public:
String name() { return my Name; };
String phone() { return my Phone; };
joeP personlmpl( String n, String p )
{ myName = n; my Phone = p; }};
```

Figure 1 : An abstract type and its implementation.

```
/* an employee aspect definition */
implementation emplmpl //class name
with type Employee //aspect name
extends Person  //base object type {
int myEid;
String myDept, myPhone;
public:
emplmpl( int e, String d, String p )
{ myEid = e, myDept = d, my Phone = p; };
String phone()
{ if( my Phone != nil )return myPhone;
else return base.phone(); };
/* imported from Person */
Person::name; /* Code is same with Person's as
not redefined here. */
Person::phone as homePhone; };
```

Figure 2 : An employee face of a Person object.

```
type Person /* abstract type... */ {
String name();
String phone(); };
implementation personlmpl
/* ...and its implementation */ {
String myName, myPhone;
public:
String name() { return myName; };
String phone() { return myPhone; };
joeP personlmpl( String n, String p )
{ myName = n; my Phone = p; }};
```

Figure 3: Creating and using an Employee face

4.2. Fibonacci

Fibonacci [5] is a strongly typed OODB programming language derived from Galileo [6] by Albano et al. Fibonacci has its own data model which supports inheritance and persistency. It is also an interactive language, e.g. all given commands or definitions give immediate response.

4.2.1. Object Mechanism and Usage

An object in Fibonacci can not directly be modified unless it is done via one of its methods, e.g. roles. This rule can be described as "Messages are sent to roles". An object's answer to a received message completely depends on the role which has received the message.

The dominating role in Fibonacci is always the mostly evolved one, that is, the deepest role in the hierarchy. Therefore delegation is done from bottom to top. If sibling roles are able to answer a message, the more recently acquired role is used. Fibonacci can also handle delegation in the opposite order.

Unused role objects are removed by the built-in garbage collector, hence, explicit removal is not allowed. This presents a black-box view of real world objects where they are only accessed via the ports of a black-box. Methods of roles constitute the connection points and a dispatcher module is responsible from delegation. Fibonacci objects are composite objects as seen in Figure 4. The whole object is the realization of a real world object while rightmost cells are implementation of roles. R1 to R4 are the methods of the composite object which are implemented in role objects.

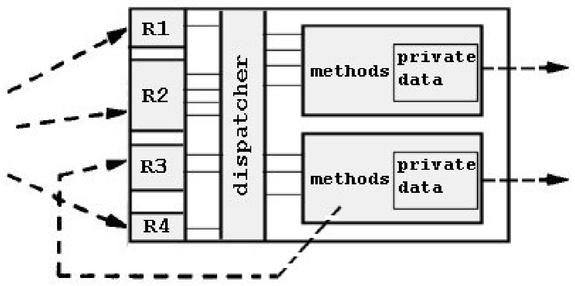

Figure 4. Internal Structure of an Object in Fibonacci

Fibonacci supports class-based inheritance between roles, too. A subtype inherits all members and methods but they can be overridden. Naming conflicts which may occur with multiple inheritance is prevented according to the "most recent role is the most dominant" rule.

An example of creating a person role and assigning it to an object which utilizes this shorthand is given in Figure 5. Immediate responses of the system is denoted by " >>>".

Role switching in Fibonacci is possible by role casting and there are operators for comparison of objects. Equality of two objects without considering their role types is checked with the equality operator. The `isAlso` statement checks whether an object has a particular role and `isExactly` statement checks if two role objects are from the same type.

```
Let Person = IsA PersonObject With
   const Name : String;
   const BirthYear : Int;
   Introduce : String;
End;
>>>Let Person <: PersonObject = <Role>
Let john = role Person
   private
      let Name = "John Daniels";
      let BirthYear = 1967;
   methods
Introduce = "My name's " & Name & " and I was
born in " & intToString( BirthYear );
end;
>>>Let john : PersonObject = <role>
john!Introduce:
>>> "My name is John Daniels and I was born in
1967" : String
```

Figure 5 : Creating a person role.

```
let johnAsStudent = ext john to Student
   private
      let Faculty = var "Science";
   methods
      Introduce = (me as Person) ! Introduce &
". I am a Science student";
end;
>>> let johnAsStudent : Student = <role>
```

Figure 6 : Acquiring a role.

```
john = johnAsStudent;
>>> true : Bool
john.Introduce;
>>> "My name is John Daniels and I was born in
1967. I am a Science student" : String
johnAsStudent.Introduce;
>>> "My name is John Daniels and I was born in
1967. I am a Science student" : String
(johnAsStudent as Person)!Introduce:
>>> "My name is John Daniels and I was born in
1967" : String
```

Figure 7 : Changing behaviour of an object with roles.

An object can acquire new roles as shown in Figure 6. After this, the object's behavior changes as it is supposed to. This is proven in Figure 7.

4.3. INADA

INADA [7] is an ODMG standards compliant object oriented language with role support and it is still being improved by a team in Kyushu University. As INADA is built on a distributed object storage server, WAKASHI [8], it supports persistent objects. Every INADA type is equal to a role.

4.3.1. INADA Extensions

A persistent object of type T can be declared as `d_Ref<T>` *`object_name`*. The `new` operator for persistent objects takes two additional parameters which come before the constructor: First one is a reference to an ODMG database and the second one is the name of the class. Figure 8 shows creation of an object `p` of class `Person` to model a person.

```
d_Database dbobj;
d_Database *database = &dbobj;
main(){
   database -> open("dbfile");
   d_Ref<Person> p = new(database,
     "Person")Person("John", 30, 1976, ...);
   //1976 : John's home phone }
```

Figure 8: Creating a persistent real world object

```
d_Ref<Employee> e = new(database, "Employee")
    Employee ("sale", 3300)
    transforms p; //3300 : John's work phone
```

Figure 9: Giving a role to an object.

```
return e->TelNo( ); //returns 1976
return e as Person->TelNo( ); //returns 3300
```

Figure 10: Role switching in INADA.

Objects with multiple roles are supported with two statements: `transforms` and `as`. Adding and removing types as well as reaching one of the roles of a persistent object is possible with these commands. When a new role is to be added to a persistent object, the `transforms` statement is used with the new operator. A role is given to an object by adding a new type to this object. An example is given in Figure 9. The pointer e of this example and the pointer p of the previous example are equivalent, pointing at the same persistent object. However, they don't act in the same manner because they represent different roles of the object.

A persistent object can be reached via any of its roles and each role of an object acts differently. The `as` operator is used for role switching as given in Figure 10.

4.4. Extending OOP with Role Hierarchies

The work by Gottlob, Schrefl ve Röck [10] shows how role support with object-level inheritance can be implemented in current OOP languages. The proposed model does not prohibit class-based inheritance but suggests using it wherever necessary. The extended OOP language is Smalltalk.

4.4.1. Role Hierarchies

A role hierarchy consists of a tree having objects from a special class modeling roles as its nodes. The root of the tree is an object from another special class modeling real world entities : It represents the invariant properties of this entity and determines how the entity can evolve. A new node is created and added to the tree when a new role is acquired and it is destroyed when this role is abandoned. An entire role hierarchy represents a real world entity.

Role hierarchies extend OOP concepts in a natural way. Unlike class hierarchies, a sub role does not inherit its super roles' methods and members. However, a method unknown to a sub role is delegated to its super roles.

Adding a new role to the tree does not affect other roles. However, children of a role should also be deleted when abandoning this role. This is semantically correct as these children depend on this role to be meaningful.

Role hierarchies bring the following flexibilities:

- Sharing of knowledge between different roles via mutual parents
- Ease of modeling by ease of tracking evolving objects by gaining and loosing roles

- Modeling independent roles without an exponentially growing number of intersection classes
- Enabling differentiation of behavior according to the role currently being used
- Limiting access to the object to only the current role
- Support for multiple roles of the same type

4.4.2. Role Definition in Smalltalk and Their Usage

Role hierarchies are implemented by three classes. An object which can have roles and to become root of a hierarchy should belong to a subclass of `ObjectWithRoles`. The role objects are similarly modeled by inheriting from `RoleType`. Multiple roles from the same type are named as qualified roles and they need to be further specialized. They are implemented with the class `QualifiedRoleType`, which is created by regular inheritance from `RoleType`. Information about these classes are given in Table 1.

Table 1: Information about classes that give role support.

Methods of ObjectWithRoles and RoleType :	
root	Returns the root of hierarchy
roleOf	Returns the parent of this role in the hierarchy
as("aRoleType")	Switch to the aRoleType role, if any
existsAs("aRoleType")	Checks if aRoleType exists in this object
entityEquiv(anObject)	Checks if anObject and this represent the same entity.
Additional method for RoleType :	
Abandon	Leave this role
Methods of ObjectWithRoles and RoleType for qualified roles :	
as(aQualifiedRoleType, qualifyingObj)	Switch to the aQualifiedRoleType which is qualified by the qualifyingObj.
existsAs(aQualifiedRoleType, qualifyingObj)	True if object has the aQualifiedRoleType role, which is qualified by the qualifyingObj.
Additional method for QualifiedRoleType :	
qualifier( )	Return the qualifying object

To define a new role type, `defRoleType` method of class `RoleType` is used. This method's parameters include the super role and class structure (members and methods) of the role to be created. To abandon a role, its `abandon` method is called.

Role switching is possible in the following three ways:

1. `roleOf` method of a role object returns the parent role
2. `root` method of a role object returns the root of the role hierarchy
3. Role casting : accessing a desired role from a particular one. If we need to access the `aRoleType` role of the role object `anObject`, we use `anObject as: aRoleType`.

Using roles with these mechanisms are demonstrated in Figure 11 in a Java-like language.

```
//Create root of hierarchy :
personSmith = new Person( );
personSmith.setName( "Smith, Anne" );
personSmith.setPhoneNo( "312-4534" );

//Define an employee role :
empSmith = new Employee( );
empSmith.newRoleOf( personSmith );
empSmith.setPhoneNo( "304-5601" );

//define a student role :
studentSmith = new Student( );
studentSmith.newRoleOf( personSmith );

//checking for equivalency :
anEmployee = empSmith;
aStudent = studentSmith;
if( aStudent == studentSmith ) { /*TRUE*/ }
if( aStudent == anEmployee ) { /*FALSE*/ }
if( aStudent.entityEquiv( studentSmith ) )
{ /*TRUE*/ }
if( aStudent.entityEquiv( anEmployee ) )
{ /*TRUE*/ }
//role switching :
System.out.println( empSmith.getPhoneNo() );
    //prints 304-5601
System.out.println(empSmith.roleOf().
    getPhoneNo()); //prints 312-4534
personSmith = new Person(studentSmith.root());
System.out.println(depMgSmith.root().
    getPhoneNo());//prints 312-4534
if(personSmith.existsAs("Employee")) /*TRUE*/
    System.out.println(personSmith.as("Employee
    ").getPhoneNo());//prints 304-5601
System.out.println( empSmith.getName( ) );
//prints Smith, Anne

//When Smith graduates :
studentSmith.abandon( );
```

Figure 11: Examples for usage of roles.

Qualified roles are taken into special consideration by defining a separate class for their implementation. This is necessary since simply creating several instances and attaching them to a parent role does not tell anything about the semantic connection among the qualified roles and the parent. Moreover, any information used for separating these qualified roles from each other can not be stored this way. We need to know what qualifies these roles and how.

Both class-level inheritance and object-level inheritance are allowed in role hierarchies. Roles may have sub roles via object-level inheritance or sub classes via class-level inheritance. Leaves of a class hierarchy can be the roots of separate role hierarchies.

5. EVALUATION

We have identified four different approaches that provide relational role support: Multiple typed objects, role hierarchies, object faces and composite objects. These are represented by the work done in INADA [7], role hierarchies [10], aspects [3] and Fibonacci [5], respectively. Other works in literature can be placed into one of these groups.

[10] uses a few special classes for role support and a hierarchical approach while [3] uses a language which has not been implemented. [7] is a C++ - like language built on top of an OODB engine. [5] has a unique structure which favors a black-box approach.

The concept of having multiple faces in [3] is similar to the concept of having multiple types in [7]. In both of them, a sub type imports only the needed methods and members of its super type and throws out the rest. This causes the advantages and disadvantages of both approaches to be more or less the same, so [3] is chosen for further discussion in this section.

The approach presented in [5] can be implemented more easily by using composite objects. Fibonacci is a language derived from Galileo and we think that this decreases its portability into other systems. However, its fundamental idea of using a dispatcher may be implementable. The same argument is true for INADA which is based on a propriety persistent object storage module.

The acquisition of a role is called object-level inheritance instead of the traditional class-level inheritance. However, these two different inheritance mechanisms do not completely obsolete each other as both have non-overlapping strengths. Therefore, it is better to use both of them in any role based system. [10] has the flexibility of using sub roles which inherit from a super role either in object-level or class-level and extends the traditional concepts of OOP in a natural way.

The relationship between a real world entity and its roles are considered to be hierarchical in explicit terms in [10]. A hierarchy is not imposed in [3] although the multiple types of an object are assumed to be modeled as a linked list. This can be viewed as a hierarchy of depth one. However, if it is possible to give an object face as an argument to the *extends* statement, role hierarchies can also be constructed in [3].

Having more than one role objects of the same class, a fundamental characteristic, is only allowed in [10] and [3].

Table 2 summarizes this discussion through a comparison of the four approaches.

6. CONCLUSION

In this paper, we have presented the inefficiency of the OOP paradigm in modeling real world objects that change with time and have focused on the role concept as a solution. We have also presented the most representative work towards role support in OOP languages. Considering their properties pointed out in the previous section, we conclude that [10] and [3] possess the most elegant, flexible and natural means for implementing role support to a new system. As future work, we consider to focus further on the role concept and extend the Java language with role capabilities.

REFERENCES

[1] C.W. Bachman, M. Daya "The Role Concept in Data Models". **Proc. Intl. Conf. Very Large Databases,** 1977 (VLDB'77) p.464-476.

[2] Lieberman, Henry, "Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems". **Proc. ACM OOPSLA Conf., New York, USA (OOPSLA '86).** p. 214-223.

[3] A.H. Skarra, S. B. Zdonik. "Aspects: Extending Objects to Support Multiple, Independent Roles". **Proc. ACM OOPSLA Conf., 1986, New York, USA (OOPSLA '86).** p483–495.

[4] B.A. Hutchinson, et al. "Object Structure in the Emerald System". **Proc. ACM OOPSLA Conf., 1986, New York, USA (OOPSLA '86).** p.76-86

[5] A. Albano, et. al. "An Object Data Model with Roles". **Proc. Intl. Conf. Very Large Databases, 1993, Calif., USA (VLDB '93),** p39–51.

[6] A Albano, et. Al. "Galileo: A Strongly Typed, Interactive Conceptual Language". **ACM Trans. Database Systems,** Vol. 10, No. 2, 1985. p. 230-260.

[7] M. Aritsugi, A. Makinouchi. "Multiple-Type Objects in an Enhanced C++ Persistent Programming Language". **Software-Practice And Experience** Vol. 30 No. 2 Feb 2000 p151–174

[8] G. Yu, et. al. "Transaction management for a distributed object storage system WAKASHI - design, implementation and performance". **Intl. Conf. Data Engineering,** p. 460-468, New Orleans, 1996

[9] J. Eliot, B. Moss "Working with Persistent Objects : to Swizzle or not to Swizzle". **IEEE Trans. Software Eng.**, Vol.18 No. 8 Aug. 1992

[10] George Gottlob, et al. "Extending object-oriented systems with roles". **ACM Trans. on Information Systems** Vol. 14, No. 3, July 1996, p268–296

Table 2: Comparison of Role Based Approaches

	INADA [7]	**Role Hierarchies [10]**	**Object Faces [3]**	**Fibonacci [5]**
Implementation	INADA language with WAKASHI (OODBMS)	In any OOP language by using a few classes	In a theoretical language.	A new language descendant of Galileo
Relation between roles	Adding/removing types, set-fashion	Role hierarchy, tree based.	Adding/removing faces, set-fashion	Role hierarchy linked by a dispatcher.
Support for hierarchy	No	Naturally	Possible	Yes
Application to an existent OOP language	Possible but costly	Possible with less cost	Possible but costly	Possible but costly
Inheritance for roles	Only object-level	Object and class-level	Only object-level	Object and class-level
Qualified roles	No	Yes	Yes	No
Data hiding	Yes	Yes	Yes	Yes, best.
Delegation	Yes, random	Yes, governed by rules	Manual casting if conformity applies	Yes, governed by rules