Comparison of Object-Oriented Approaches for Roles in Programming Languages

Daniel Chernuchin, Oliver S. Lazar and **Gisbert Dittrich**
University of Dortmund

Abstract

This paper presents, explores and compares object-oriented approaches for roles in statically typed programming languages. We choose five solutions which support information hiding. On the one hand, we investigate the established object-oriented possibilities multiple and interface inheritance and the role object pattern, on the other hand, we examine the language extensions Object Teams and the syntactical extension of classes with roles. We discuss all approaches in turn, using a versatile example.

The investigation shows that standard approaches interface inheritance and role object pattern are more appropriate for enterprise projects. New approaches are not developed far enough to be used widespread. But they are all promising proceedings, particularly the approach of roles as components of classes.

Motivation

Object-oriented development is standard today. One of the most important reasons is the comprehensibility of models and codes. This comprehensibility stems from transferring aspects of the real world to programming languages. In this way classes, objects, methods, inheritance and templates are originated. An aspect of the real world is the possibility that an object appears in different roles. In object-oriented development roles are visible properties of an object (Riehle 2000). An object may play different roles simultaneously. It can be viewed from different perspectives, so that different properties appear.

Advantages of object-oriented languages are information hiding and encapsulation. Functionality and properties are organized in classes on different abstraction levels and clients can only access the information on the permitted level. A role concept should improve information hiding and encapsulation.

The research of roles in the object-oriented area is extensive. But the definitions, modelling ways, examples and targets are often different (Steimann 2000). We pick out five approaches which all support a high level of information hiding. These approaches allow the programmer only to access relevant role properties. In our opinion, it is one of the most important features by dealing with roles. We explore the techniques in the area of the statically typed languages like Java and C++. None of the approaches has until now become standard.

This paper is a compendium of (Lazar 2005).

In the next section we discuss the related work in the object-oriented area. Then we introduce criteria of comparison of approaches and present an example of an application for role investigations. In the following sections we consider the approaches multiple and interface inheritance, role object pattern, Object Teams and roles as components of classes. The last section presents results and the conclusion of our comparison.

Related Work

The definitions of the term *role* are various. They vary from the *observable* (Steimann 2001), *perspective-dependent behavior* of objects (Mezini 1998), to a certain part of *subjective behavior* (Kristensen 2001). In this paper we consider *natural types* which relate to the essence of the objects. Roles, which appear in specific perspectives, are called *role types* (Sowa 1984). A *context* is an expedient collaboration of several roles and classes.

An extensive object in a computational system often completes its tasks while crosscutting a certain amount of various contexts. Such an object can be reclassified in subunits corresponding to contexts. Hence, in each context an object provides a specific role.

In this article we can not explore all approaches for roles, we have chosen only five representatives which support information hiding well. Now we shortly discuss some other related work.

B. B. Kristensen and K. Østerbye describe a conceptual model for roles (Kristensen & Østerbye 1996). They distinguish between intrinsic and extrinsic properties of objects. Another approach is Object-Oriented Role Analysis and Modelling (Andersen 1997). In an OOram role model, patterns of interacting objects are abstracted into a corresponding pattern of interacting roles (Reenskaug, Wold, & Lehne 1995). There are also class libraries developed for Smalltalk and Java to handle with roles (Gottlob, Schrefl, & Roeck 1996; Schrefl & Thalhammer 2002).

Aspect-oriented approaches for role modelling facilitate further modularisation. Aspect-oriented programming has been controversially discussed in this context. On the one

hand, it is described as a promising approach for role models (Kendall 1999), on the other hand, in (Hanenberg & Unland 2002) the conclusion is drawn that roles and aspects are too different. (Hanenberg, Stein, & Unland 2005) even suggest that role programming is a special case of the aspect-oriented programming. Object Teams (Herrmann 2002b), Chameleon (Graversen & Beyer 2002) and EpsilonJ (Tamai 2005) are aspect-oriented approaches which syntactically define roles. In our comparison we take Object Teams as a representative of the aspect-oriented approaches.

One goal of using roles is to make possible that several teams of developers work independently from each other on a software project. This idea was suggested by (Ossher *et al.* 1995; 1996) as subject-oriented programming. With subject-oriented programming the denotation of a method invocation for a given object is said to be subjective if different method interpretations can result (Kristensen 2001). Those interpretations depend on the current role of an object. That is whenever we speak about programming of roles we speak about subject-oriented programming. The five presented approaches deal with subjective views. The main projects of the subject-oriented community deal with aspect oriented programming. There are Hyper/J (Thang & Katayama 2002) and Concern Manipulation Environment (Harrison, Ossher, & Tarr 2004).

Further role approaches are variational-oriented programming (Mezini 1998) and Templates (VanHilst & Notkin 1996). The information hiding is not their main focus. M. Mezini handles with incremental behavior variations in dynamic typed languages. M. VanHilst and D. Notkin work on the composition of roles to classes by means of inheritance and templates. The solution of the diamond problem is given by the inheritance order. The information hiding is not given, because a role can access properties of other roles.

An overview of several role approaches can be find in (Baumgart 2003; Steimann 2000).

Criteria of comparison

We need criteria for the comparison of the approaches. We distinguish between features and general properties of the approaches. The properties of roles in (Kristensen 1996; Steimann 2000) form the basis for our research.

We consider the following feature criteria:

1. **Encapsulation:** This is a distribution of information so that relevant information to one subject stands together in pieces of code of appropriate size separated from non-relevant information.
2. **Dependency:** A role can influence another one. Hence, a role-based approach has to provide a possibility to express dependencies of attributes of different roles. Attributes can have the same values or their values depend from each other. This is important that different roles of the same entity can have different dependent parts. When all roles have the same equal part, this is a restricted form of dependency.
3. **Dynamicity:** An entity may acquire and abandon roles dynamically. Restricted forms of dynamicity are activation and deactivation of roles in a predefined set of potential roles and the substitution of method contents depending on the current context.
4. **Identity sharing:** The object and its roles are seen and can be manipulated as one entity.
5. **Several same roles:** An object may play the same role several times.

A not explicitly named criterion is information hiding. When being addressed in a certain role, features of other roles of the object remain invisible. All investigated approaches support information hiding.

Another not explicitly named criterion is hierarchy of role and natural types. As information hiding it is fulfilled in every investigated approach.

The following list shows general properties:

1. **Closeness to the object model:** This criterion evaluates how far an approach is geared to the object-oriented paradigm.
2. **Maintainability and extensibility:** This criterion examines how far the modelling and implementation of an application or a module is maintainable and extensible.
3. **Comprehensibility:** This criterion judges the comprehensibility of models and codes. Furthermore, the closeness of modelling roles and the corresponding code will be evaluated. From our point of view this criterion is the most important.
4. **Documentation:** This criterion evaluates the documentation of an approach concerning roles.
5. **Progress of development:** This criterion evaluates the quality and the amount of supporting tools and how far the progress of development is sophisticated. Moreover, the support for graphical modelling is considered e.g. UML.

There are correlations between the criteria. For example comprehensibility strongly depends on the closeness to the object model.

Example

We illustrate all approaches using a versatile example developed for this purpose. The following example is a part of a *hospital information system* (HIS). As there is no standard method for graphically modelling roles in a computational system (Steimann 2001), we suggest using a modified UML-classdiagram as shown in figure 1. For a better general survey all get- and set-methods have been masked out. Mapping roles into discrete classes is not a compulsory presetting for the approaches. This is just one possible alternative which we have chosen to visually illustrate the example. The roles of `Patient` are marked with gray background colors. These roles appear in contexts which are diagrammed as rectangles with round corners. Depending on a role inheritance path a context can be nested into another one, e.g. contexts *ward* and *medicine*. Besides the requested relations between classes and roles shown in figure 1 the attributes `name` of the role `SickPatient` and `id` of the role `StudyPatient` always contain the same value. These attributes are called equality-dependent. Furthermore the attributes `heightInch` of the subrole `AmbulantPatient`

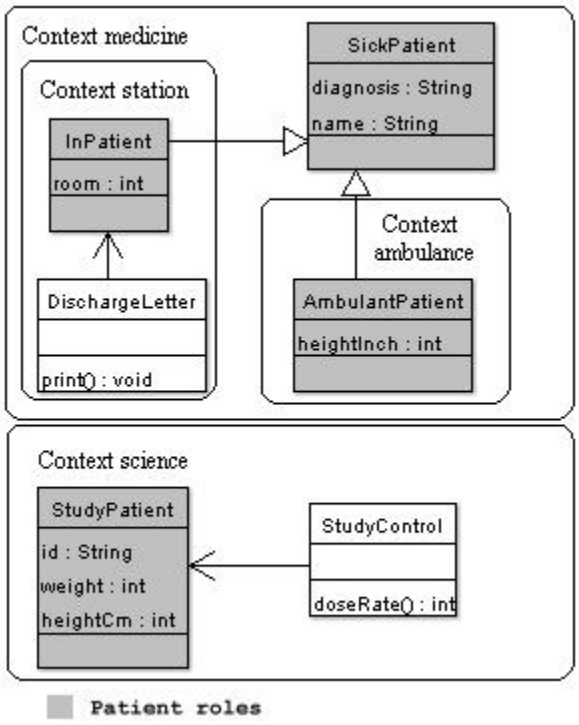

Figure 1: Modified UML-classdiagram with contexts and roles

and `heightCm` of the role `StudyPatient` are function-dependent. The conversion factor for these attributes is 2.54 (1 inch = 2.54 cm).

Those classes of the example with white background color are to be considered as classes with exactly one role.

Multiple Inheritance

Multiple inheritance means that a class has multiple superclasses. Every role gets its own class. Figure 2 depicts an example as an UML-classdiagram denoted in short-form for multiple inheritance as a fraction of the HIS-example. The class `Patient` inherits from the roles `InPatient` and `AmbulantPatient` at one time. Furthermore, `DischargeLetter` is associated with the role `InPatient`. The following C++ codefragment according to figure 2 shows how variables with types of the superclasses get a reference on their subclass. These superclasses come up to the role types, the subclass comes up to the natural type. Such an upcast is always possible and typesafe.

```
Patient* patient = new Patient;
DischargeLetter* dischargeLetter = new
    DischargeLetter;
InPatient* inPatient = patient;
dischargeLetter->setPatient(inPatient);
dischargeLetter->print();
```

Every role type variable of the patient, e.g. `inPatient` holds a reference to the same object with the same identity. While using variables of the type of the superclass, clients have a constricted view on the object. All the properties of the natural type that do not belong to the specific role

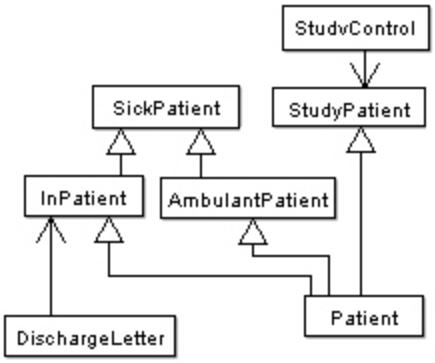

Figure 2: Multiple inheritance HIS-example

type are masked out. Hence, those restricted views are in accordance with the roles.

It is easy to handle roles using upcasts. An important disadvantage is the lack of possibility to realize dynamicity. Inheritance is a compile time feature. Thus, one cannot change the roles of an entity during the run time. A further drawback is that it is not possible to directly implement dependencies between attributes offhand. One could explicitly use references and modified set-methods to solve that problem (Lazar 2005).

The pros and cons of multiple inheritance have been discussed for a long time (Mezini 1998). In the meantime most programmers have accepted that the disadvantages are preponderant. Hence, many modern programming languages only provide single and interface inheritance (see section *Interface Inheritance*). The most prominent disadvantages of multiple inheritance are name conflicts and diamond inheritance. Name conflicts occur when different superclasses of one subclass provide attributes with the same name or methods with the same signature. As soon as the inheriting subclass uses such an attribute or method, the compiler will put out an error because there is no definite allocation. When using appropriate upcasts, the upcast leads to an explicit allocation. Alternative approaches to avoid name conflicts can be found in (Mezini 1998). Diamond inheritance occurs when a subclass inherits from the same superclass on different inheritance paths multiple times. In figure 2 diamond inheritance is shown with the classes `Patient`, `InPatient`, `AmbulantPatient` and `SickPatient`. Multiple inheritance provides most of the postulated criteria, except dynamicity, implicitly defining dependencies between attributes and playing the same role coevally multiple times. It supports information hiding, encapsulation and hierarchy of role and natural types and identity sharing.

Multiple inheritance is not supported by every object oriented programming language. Especially modern languages, like Java and C#, only support interface inheritance. Multiple inheritance is a comprehensible approach although there is no syntactical difference between role and natural types. Moreover, it is conform to the object oriented paradigm. Multiple inheritance has already been existing for a long time, development is sophisticated and lots of docu-

mentation are available. However, there is less documentation concerning roles. Furthermore, standard UML can be used for graphical modelling.

Interface Inheritance

Interface inheritance is similar to multiple inheritance (see section *Multiple Inheritance*). In contrast to multiple inheritance, a class can inherit from only one class but it can implement several interfaces. Interfaces declare only method signatures and constants. These methods have to be implemented in subclasses. Nevertheless, interfaces are legal types, in this case role types. Some classes of figure 2 become interfaces. Only one of the superclasses of `Patient` can stay a class, the others turn into interfaces. Class `Patient` in this approach is more complex than in *Multiple Inheritance* because it has additional functionality of e.g. `AmbulantPatient`, `StudyPatient` and `PayingPatient`. `SickPatient` is an interface, too because interfaces only inherit interfaces.

A client which accesses an object does not take notice of the difference between interface and multiple inheritance. It views only the role type. Steimann defines interfaces as role types (Steimann 2001).

Thanks to the interfaces, the diamond problem and the name conflicts of multiple inheritance are solved because the methods in interfaces do not have contents. The drawback of multiple inheritance is that the functionality cannot be branched out into superclasses. Hence, the classes tend to become complex.

The progress of development of interface inheritance is very advanced. All advantages of multiple inheritance for the client are available with the disadvantage that classes become more complex. Thus, encapsulation is poorly supported. Dependencies can be solved because the whole functionality is in one class. Methods can basically use the same attributes and call each other.

Interface inheritance is supported by many modern programming languages e.g. Java and C#. Documentation about roles is good (Steimann 2001). Like multiple inheritance, interface inheritance is a component of the object model. Comprehensibility loses a little versus multiple inheritance because of complex classes. All unmentioned criteria of comparison remain equal to the criteria of multiple inheritance.

Role Object Pattern

The role object pattern is a specific design pattern. Design patterns provide successful and practically approved solutions for recurring design problems (Gamma *et al.* 1997).

The role object pattern models context-specific views on a so called *core object* while using *role objects*. Role objects can dynamically be appended to or removed from their core object (Bäumer *et al.* 2000; Fowler 1997). As described in (Fowler 1997) the role object pattern is similar to the state design pattern. Figure 3 depicts the basic structure of the role object pattern showing a part of the HIS-example. In this example the core class is the class `Patient`. Clients, in this case `DischargeLetter`, have to contact the core object of the type `Patient` and to ask for the required role object. Thereby, all role classes are subclasses of the class `PatientRole` so that the returned reference always is of the type `PatientRole`. This role

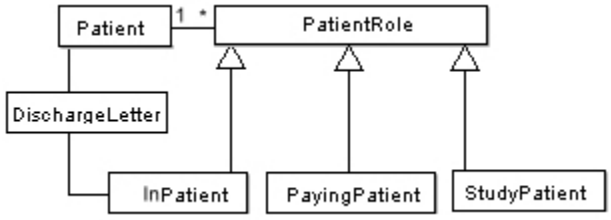

Figure 3: The role object pattern

object then needs to be downcasted to the required role object type. Furthermore, the client only works on the role object, not long on the core object. That is why in figure 3 the client `DischargeLetter` provides associations to the core class `Patient` and to the role class `InPatient` as well.

```
InPatient roleObject = new InPatient();
patient.addRole(roleObject);
// ...
InPatient inPatient = patient.roleOf(
    InPatient.class);
dischargeLetter.setPatient(inPatient);
```

For the implementation of the role object pattern the core object of the type `Patient` has a container as an attribute which stores the corresponding role objects. Further it contains methods to return a reference on the required role object, to dynamically append and remove role objects at runtime.

The role object pattern allows an object to play the same role multiple times with different characteristics, respectively (Lazar 2005). This implies that all role objects and the core object have different object identities. Furthermore, this approach is dynamic. It is easy to append and remove roles which is an important property by modelling the real world. Information hiding, encapsulation, hierarchy of role and natural types are supported. Implicitly defining dependencies between attributes is not possible. However, we can place shared parts of all roles in the core object. M. Fowler recommends not to use this pattern, if the potential roles have strong interdependencies (Fowler 1997).

This pattern is fully developed and many frameworks based on it (Riehle 2000) are developed. Despite the fact that it is conform with the object-oriented paradigm, it is difficult to apply and to comprehend because of its complexity. Maintainability and extensibility is well supported. An adequate documentation and examples can be found in (Fowler 1997) and (Bäumer *et al.* 2000). Moreover, standard UML can be used for graphical modelling.

Object Teams

Aspect-oriented programming is an enhancement of object-oriented, procedural and functional software development (Kiczales *et al.* 1997). We only consider the object-oriented variant. Aspect-oriented programming facilitates a further

dimension of modularisation. A concern is a specific functionality within the whole system. Such a functionality is usually spread over different classes, which is called crosscutting concerns or aspects. Aspect-oriented programming encapsulates code fragments, representing crosscutting concerns.

Role modelling and implementation with aspect-oriented programming have been controversially discussed (see section *Related Work*). In this paper we investigate the programming model Object Teams. Object Teams introduces new programming language constructs which support the modularisation of crosscutting concerns by defining *teams* and *roles*. A team is a collaboration of classes which interact for a common purpose. Classes inside of a team are called roles which are played by their *base classes*. Such a base class comes up to the natural type. The role and the base class are syntactically connected with the `playedBy` keyword. The roles ensure information hiding because they can restrict the access to the base classes. Thereby we have to distinguish between *callout* and *callin* method bindings. Callout means that a base class forwards the methods to its roles. Vice versa a callin weaves role method code into a base method which causes a change of the behavior of the base class. For a role method which is called by callin binding there are three alternatives. A role method can be executed `before` or `after` the base class method or the base class method can be `replaced`.

```
team class ContextWard {
  class InPatient playedBy Patient {
    private int room;
    int getRoom() {
      return room;
    }
    String getDiagnosis() -> String
        getDiagnosis();
  }
  class DischargeLetter {
    private InPatient inPatient;
    void setPatient(InPatient p) {
      inPatient = p;
    }
    void print() { ... }
  }
}
```

This code fragment demonstrates a simplified version of the team `ContextWard` in the HIS-example. This team contains two roles `InPatient` and `DischargeLetter`. The class `Patient`, which is played by its role, is declared outside the team. `->` means a callin: The method `getDiagnosis` is imported to the role `InPatient` from the base class `Patient`.

Changing and extending an existing application is an excellent property of Object Teams. The approach supports information hiding because roles are used as types above all inside a team. Some dynamicity is given because the content of methods can be replaced depending on the current active context. In (Kristensen & Østerbye 1996) this property is described as a method role. Ever, the possibility to activate and deactivate contexts is an advantage of Object Teams. This allows to define an additional level for role interaction. But the realization of dynamicity and information hiding simultaneously requires very advanced skills. Inheritance of contexts and base classes is allowed. Roles can inherit from other roles in the parent context or from regular classes. Single and interface inheritance are possible.

By virtue of the lifting and lowering mechanisms, a role and its base appear as almost the same object. Callins and callouts are self-calls within this compound object. There is planned to add a method `roleEQ` which would yield true for each pair of objects based on the same natural object.

Dependency interactions are possible between the base class and the roles by means of callout to field. But roles themselves cannot be dependent on each other.

The largest drawback of the approach is the need to rethink and to learn new programming constructs. Several features as role creation require complex implementation. Furthermore, the criterion dependency is not fulfilled.

This approach supports encapsulation, maintainability and extensibility by outsourcing functionality into teams and roles. Playing of several same roles is also possible because several teams instances of the same type can coexist. Dynamicity of role types is restricted to replacing methods by activating and deactivating teams.

The progress of development of Object Teams is quite advanced. A well-engineered tool in form of an Eclipse plugin is developed. The current release is 0.8.8.

Object teams has lost some closeness to the object-oriented paradigm. It provides new models and code constructs making it difficult to learn and to apply. The developers of Object Teams have written a language definition including helpful examples. But because of the not extensive distribution of the tool, the documentation amount is not so huge as by some other approaches. With UML for aspects (Herrmann 2002a) there is a concept for graphical modelling but the tool support for that is poor.

Roles as Components of Classes

The approach described in (Chernuchin & Dittrich 2005a; 2005b) is especially developed for roles. It syntactically distinguishes between role and natural types. Thereby a class contains roles and their dependencies. The visibility and the access to an object depend on its current role.

Classes in the standard OOP are interpreted as classes with only one role, e.g. `DischargeLetter`. Hence, this role concept is a canonical extension of the object model. That is existing software can be integrated without changes.

An object has the whole complexity of its class. The approach supports information hiding because a client accesses only one role, e.g. variable of type `InPatient` only occurs in context `ambulance` and variable of type `AmbulantPatient` occurs only in context `ward`. Both roles are accessed via references. Two variables refer the same object but they have different static types:

```
InPatient inPatient = new Patient();
AmbulantPatient ambulantPatient =
    (AmbulantPatient) inPatient;
```

Thereby only roles can occur as static types and only classes as dynamic types. This approach is similar to multiple inheritance (see section *Multiple Inheritance*)

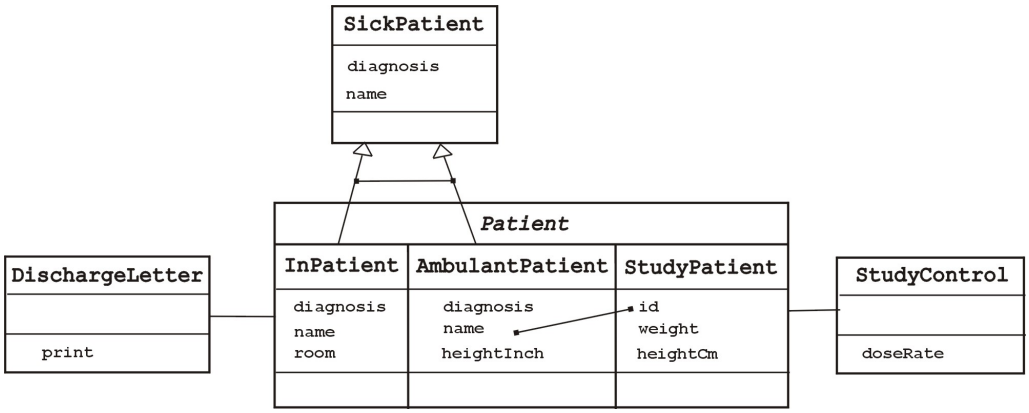

Figure 4: Roles as components of classes in HIS-example

The syntax of roles correlates to the syntax of classes in the standard object model. Classes of this approach compose roles to natural types with help of dependencies.

The HIS-example is graphically presented as an UML-like diagram in figure 4. If a class has several roles e.g. `Patient` it is shown as a big rectangle with the class name above. Underneath there are role rectangles with their names, attributes and methods. Classes with a single role e.g. `DischargeLetter` match the standard UML notation.

Particularly, this approach realizes explicit notion of dependencies between roles. Roles can have shared, dependent and independent parts. *Equality dependencies* on attributes and on methods of different roles can be declared. Bækdal and Kirstensen call equality-dependent attributes shared properties (Bækdal & Kristensen 2000). In our example, `name` of `AmbulantPatient` and `id` of `StudyPatient` are equality-dependent which is denoted in figure 4 with a line between these attributes. In this approach function dependencies are possible, too.

Role types as well as natural types can inherit from each other. In the HIS-example the roles `InPatient` and `AmbulantPatient` inherit from the role `SickPatient`

Chernuchin and Dittrich developed further dependencies. Roles `InPatient` and `AmbulantPatient` are inheritance-dependent via `SickPatient`. This means that all attributes and methods which are inherited from `SickPatient` are equality-dependent in `InPatient` and `AmbulantPatient`. That is `SickPatient` is the shared part of `InPatient` and `AmbulantPatient`. In figure 4 this dependency is denoted by a line which connects the inheritance arrows. The following Java-like code example demonstrates inheritance-dependency.

```
role InPatient extends SickPatient {
  int room;
}
role AmbulantPatient extends SickPatient {
  int heightInch;
}
class Patient {
  include InPatient;
  include AmbulantPatient;
  inheritanceDependent(InPatient,
      AmbulantPatient, SickPatient)
}
```

Class dependency allows to take over dependencies of roles from one class to another. This is helpful when two classes have at least two roles and the roles of one class inherit from the roles of the other class. In this way class dependency help to avoid redundancy of dependency declarations.

If natural types inherit from each other the subclass inherits all roles and their dependencies from the superclass.

The approach supports encapsulation thanks to development environment which presents roles in an adequate way. This approach fulfills the criteria of dependency better than all the others which are presented in this paper. With the exception of dynamicity all feature criteria are fulfilled.

Roles as components of classes are comprehensible. It does not conform to the object model, but it is its canonical extension. The biggest disadvantage is that the approach roles as components of classes is only a theoretical construct so far. There are neither supporting tools nor satisfactory documentation. But the researchers are working on it.

Conclusion

We compare five approaches for roles in the object-oriented area which all support information hiding. Tables 1 and 2 sum up our results of the comparison of object-oriented approaches for roles in programming languages. Thereby, criteria, which are fulfilled in all approaches, are faded out.

We first investigate the established approaches multiple inheritance, interface inheritance and role object pattern because of their high progress of development. These approaches have in common that they are close to the object-oriented paradigm. The most important criteria is comprehensibility. Multiple inheritance and interface inheritance are very comprehensible. The most prominent disadvantage of interface inheritance is the lack of support for maintainability and extensibility because complex subclasses have to be changed. The role object pattern is more complicated because of the big amount of classes. It is very dynamic, it can coevally play several same roles and complex role scenarios

	Encapsulation	Dependencies of roles	Dynamicity	Identity sharing	several same roles
Multiple inheritance	+	−	−	+	−
Interface inheritance	−	+	−	+	−
Role object pattern	+	−	+	−	+
Object Teams	+	∘	∘	∘	+
Roles as components of classes	+	+	−	+	+

Table 1: Comparison of features

	Closeness to the object model	Maintainability and extensibility	Comprehensibility	Documentation concerning roles	Progress of development
Multiple inheritance	+	∘	+	∘	+
Interface inheritance	+	−	+	+	+
Role object pattern	+	+	−	+	+
Object Teams	∘	+	−	∘	∘
Roles as components of classes	∘	∘	∘	−	−

Table 2: Comparison of higher level criteria

can be modelled and implemented. On the one hand, the role object pattern complicates an application, on the other hand, it ensures flexibility for an extensive role composition. All in all, in this category, multiple inheritance would come off best, if there were no disadvantages like diamond inheritance, name conflicts and a constricted support of programming languages. We recommend using interface inheritance if there are only a few roles or roles have many dependencies. Otherwise we recommend role object pattern.

Aspect-oriented programming becomes popular also in role modelling. We chose as tool representative Object Teams. This is an eclipse plugin. The role handling is unusual in comparison to established approaches. Thus, it is not very comprehensible. New code constructs have to be learned and applied. Nevertheless, this approach is powerful and innovative. Thanks to the introduction of teams new possibilities are open. Separation of concerns is also well suited in this approach. The supporting Eplipse plugin has a quite advanced progress of development. But we think that this is too early to use it widespread in enterprise projects. Many software developers must rethink before this is possible. This tool will be developed further but today we recommend to use standard approaches as interface inheritance and role object pattern.

The syntactical extension, roles as components of classes, is exclusively developed for roles and is suited well for the requirements of roles with the exception of dynamicity. It is comprehensible because of a syntactic separation of role and natural types. Above all the support of dependencies of roles is extensive. The disadvantage is that this approach is in an early stage of development. There is neither any documentation nor any supporting tool.

Established approaches which are close to the object model are better in this comparison. But other newer approaches are also very promising and have a big chance in the future.

References

Andersen, E. P. 1997. *Conceptual Modeling of Objects: A Role Modeling Approach*. Ph.D. Dissertation, Department of Informatics, Faculty of Mathematics and Natural Sciences, University of Oslo.

Bækdal, L. K., and Kristensen, B. B. 2000. Perspectives and Complex Aggregates. In *Proceedings of the 6th International Conference on Object-Oriented Information Systems*. England: Springer.

Bäumer, D.; Riehle, D.; Siberski, W.; and Wulf, M. 2000. Role Object. In *Pattern Languages of Program Design*, 15–32. Addison-Wesley.

Baumgart, J. 2003. *Analyse, Entwurf und Generierung von Rollen- und Variantenmodellen*. Ph.D. Dissertation, University of Darmstadt.

Chernuchin, D., and Dittrich, G. 2005a. Dependencies of Roles. In *Views, Aspects and Roles at ECOOP*.

Chernuchin, D., and Dittrich, G. 2005b. Role Types and their Dependencies as Components of Natural Types. In *2005 AAAI Fall Symposium: Roles, an interdisciplinary perspective*.

Fowler, M. 1997. Dealing with Roles. In *The 4th Pattern Languages of Programming Conference*.

Gamma, E.; Helm, R.; Johnson, R.; and Vlissides, J. 1997. *Design Patterns*. Addison-Wesley.

Gottlob, G.; Schrefl, M.; and Roeck, B. 1996. Extending Object-Oriented Systems with Roles. *ACM Press* 14(3):268–296.

Graversen, K. B., and Beyer, J. 2002. Chameleon. Master's thesis, IT-University of Copenhagen.

Hanenberg, S., and Unland, R. 2002. Roles and aspects: Similarities, differences, and synergetic potential. In *8th International Conference on Object-Oriented Information Systems*.

Hanenberg, S.; Stein, D.; and Unland, R. 2005. Roles From an Aspect-Oriented Perspective. In *Views, Aspects and Roles at ECOOP*.

Harrison, D.; Ossher, H.; and Tarr, P. 2004. Concepts for Describing Composition of Software Artifacts. IBM Research Report, IBM.

Herrmann, S. 2002a. Composable designs with ufa. In *Workshop on Aspect-Oriented Modeling with UML at 1st Intl. Conference on Aspect Oriented Software Development*.

Herrmann, S. 2002b. Object teams: Improving modularity for crosscutting collaborations. Technical University of Berlin.

Kendall, A. 1999. Role model designs and implementations with aspect-oriented programming. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 353–369. USA: ACM Press.

Kiczales, G.; Lamping, J.; Menhdhekar, A.; Maeda, C.; Lopes, C.; Loingtier, J.; and Irwin, J. 1997. Aspect-oriented programming. In *ECOOP '97*, volume 1241, 220–242. Springer-Verlag.

Kristensen, B. B., and Østerbye, K. 1996. Roles: Conceptual Abstraction Theory and Practical Language Issues. *TAPOS* 143–160.

Kristensen, B. B. 1996. Object-Oriented Modelling with Roles. In *Proceedings of the 2nd International Conference on Object-oriented Information Systems*.

Kristensen, B. B. 2001. Subjective Behavior. *International Journal of Computer Systems Science and Engineering* 16(1):13–24.

Lazar, O. 2005. Vergleich objektorientierter Ansätze für Rollen. Master's thesis, University of Dortmund.

Mezini, M. 1998. *Variational Object-Oriented Programming Beyond Classes and Inheritance*. Kluwer.

Ossher, H.; Kaplan, M.; Harrison, W.; Katz, A.; and Kruskal, V. 1995. Subject-oriented composition rules. *ACM SIGPLAN Notices* 30(10):235–250.

Ossher, H.; Kaplan, M.; Katz, A.; Harrison, W.; and Vincent, K. 1996. Specifying Subject-Oriented Composition. In *Theory and Practice of Object Systems*, volume 2, 179–202. Wiley & Sons.

Reenskaug, T.; Wold, P.; and Lehne, O. A. 1995. *Working with Objects : The OOram Software Engineering Method*. Prentice-Hall.

Riehle, D. 2000. *Framework Design: A Role Modeling Approach*. Ph.D. Dissertation, ETH Zürich.

Schrefl, M., and Thalhammer, T. 2002. Using Roles in Java. *Journal Software - Practice and Experience* 34(5):449–464.

Sowa, J. F. 1984. Conceptual structures: information processing in mind and machine. In *Addison-Wesley Systems Programming Series*. Addison-Wesley.

Steimann, F. 2000. On the representation of roles in object-oriented and conceptual modelling. *Data & Knowledge Engineering* 35(1):83–106.

Steimann, F. 2001. Role = Interface: A Merger of Concepts. *Journal of Object-Oriented Programming* 14(4):23–32.

Tamai, T. 2005. Conquering the Eight-Tailed Dragon - An Attempt to Deal with Structural and Behavioral Complexities. In *ICECCS2005*.

Thang, N., and Katayama, T. 2002. Collaboration-based evolvable software implementations: Java and Hyper/J vs. C++-templates composition. In *Proceedings of the international workshop on Principles of software evolution* , 29–33.

VanHilst, M., and Notkin, D. 1996. Using C++ Templates to Implement Role Based Designs. In *Proceedings of the International Symposium on Object Technologies for Advanced Software*, 22–37.