Role Types and their Dependencies as Components of Natural Types

Daniel Chernuchin and **Gisbert Dittrich**
University of Dortmund

Abstract

This paper presents a canonical extension of object-oriented development for roles and context-dependent behavior. We syntactically distinguish between classes as natural types and roles as role types. Thereby a class contains roles and their dependencies. The visibility and the access to an object depend on its current role. We develop powerful facilities to organize the dependencies of roles: equality and functional dependencies between attributes, equality dependencies between methods and complex dependencies called inheritance and class dependencies. Our approach supports information hiding as well as encapsulation.

Introduction

Object-oriented development is standard today. One of the 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. This ubiquitous paradigm of real world won recognition in many research areas. In object-oriented development roles are visible properties of an object, a subset of all properties (Riehle 2000). An object may play different roles simultaneously. It can be viewed from different perspectives so that different properties appear. Attributes and methods of an object may be overloaded on a by-role basis, different roles may have the same features (Steimann 2000). A client accesses only some roles of an object and views only the relevant information.

An important property of roles is dependency because roles may influence each other. The state and the behavior of a role may affect the state and the behavior of another one because roles are properties of the same entity. Roles can have shared, dependent and independent parts. In spite of extensive research in roles we suppose that up to now the problem of dependency is not adequately solved (Chernuchin & Dittrich 2005). In addition to implementing dependencies, advanced information hiding and encapsulation in statically typed object-oriented languages are our main goals.

In this paper we extend the standard object model with a new level of roles. In particular we explore the dependencies between roles. The design of a programming language must correspond as much as possible to the conceptual model. Therefore we not only develop a model for our approach, but we suggest programming constructs.

In our approach we usually call natural types *classes* and role types *roles* (Sowa 1984). An object is an instance of a class and it holds all roles of its classes. Classes are instantiable, but roles are not. Hierarchies of classes as well as roles are possible. Thereby inheritance of roles is similar to the inheritance of classes in the standard object modell.

Statically typed languages differentiate between the static and the dynamic type of a variable. The static type is the declared type of the variable. The dynamic type is the actual type of the referred object. We denote by dynamic type the class of the object which does not change during the lifecycle of the object. The dynamic type may be one of the subtypes of the static type. In the following assignment `A` is the static type of the variable `a`. `b` is an expression which returns the reference of the object with the dynamic type e.g. `B`. `b` is usually a variable or a method call:

```
A a = b;
```

Because `B` is a subtype of `A`, the variable `a` accesses only a subset of properties in `B`. We have information hiding because the programmer who uses the variable `a` does not need to know its dynamic type. The static type specifies the role sufficiently except the cases described in section *Further Aspects of Inheritance*. The dynamic type is the class of an object. If an object is referred of several variables with different static types it plays several roles.

The next section discusses the related work in the object-oriented area. The following section presents a versatile example to show some kinds of dependencies of roles and their practical use. The section *Roles and the Dependencies of Attributes and Methods* introduces basic dependencies. Then in the section *Dependencies by Using Inheritance of Roles* we construct complex dependencies based on these of the previous section. The next section describes inheritance of classes and further aspects of the role inheritance. The section *Implementation* demonstrates implementation aspects of future tools for the role concept. The last section summarizes the results and outlines future work.

Related Work

As described in section *Introduction* we need to define several static types to one dynamic type, that is several supertypes of the dynamic type. One way to achieve this are interfaces as they occur in Java (Steimann 2001). A class can implement interfaces which are considered as roles. The drawback of this approach is the complexity of the subclass itself because it implements all methods defined in the interfaces. It is desirable to outsource parts of implementation to the supertypes. This is possible by means of multiple inheritance where superclasses are viewed as roles (see figure 1) (Chernuchin, Lazar, & Dittrich 2005). The drawbacks of multiple inheritance are name collisions and the diamond problem. Our approach is similar to multiple inheritance but it solves name collisions and the diamond problem and introduces dependencies between roles. Moreover, we syntactically distinguish between role and natural types; this distinction promotes comprehensibility.

A further established approach is the role object pattern (Bäumer *et al.* 2000; Fowler 1997). It is very flexible because roles can be attached and removed which is a significant benefit in contrast to our approach. Many frameworks base on this pattern (Riehle 2000) because it allows to implement versatile role issues. But for designs with only few roles this pattern is too complex. Furthermore, the role and the core object do not share their identity. We think that dependencies of roles are more flexible than the core object as the common part of roles. Different roles of the same class can have a different amount of dependent attributes and methods. Additionally to role frameworks, there also exist class libraries developed for Smalltalk and Java to handle roles (Gottlob, Schrefl, & Roeck 1996; Schrefl & Thalhammer 2002).

B. B. Kristensen and K. Østerbye describe a conceptual model for roles (Kristensen & Østerbye 1996). Their distinction of intrinsic and extrinsic properties differ from our model which composes roles to classes. They assume that all roles of one entity have the same common part. Our opinion is that common parts of different roles vary. The dependencies allow a more granular description of dependent parts as intrinsic properties.

Further approaches for role modelling base on aspect-oriented programming which facilitates 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, (Hanenberg & Unland 2002) make the conclusion 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 2002), Chameleon (Graversen & Beyer 2002) and EpsilonJ (Tamai 2005) are aspect-oriented approaches which syntactically define roles.

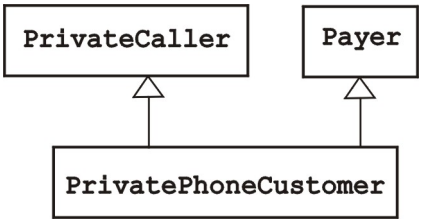

Figure 1: Roles by means of multiple inheritance

Subject-oriented programming is a universal theory (Kristensen 2001). Role-dependent behavior is a part of this theory. In this context our approach is a contribution to the subject-oriented programming. The initial goal of subject-oriented programming was to make possible that several teams work independently on a software project (Ossher *et al.* 1995). Information hiding is given because a team does not have the knowledge about foreign contexts. After that, the contexts are matched. One of the main projects of this community is implemented with the aspect-oriented programming. This is the open source project Concern Manipulation Environment (Harrison, Ossher, & Tarr 2004).

Further role approaches are variational-oriented programming (Mezini 1998) and templates (VanHilst & Notkin 1996). But their role definitions differ much from our definition because information hiding is not their focus. M. Mezini handles incremental behavior variations in dynamic typed languages. M. VanHilst and D. Notkin work on the composition of roles by means of inheritance and templates. The solution of the diamond problem is given by the inheritance order. The result is similar to priorities of roles (see *Further Aspects of Inheritance*). The information hiding is not given because a role can access properties of other roles.

A comparison of several role approaches is described in (Chernuchin, Lazar, & Dittrich 2005; **?**).

Example

We use a phone carrier software as an example to illustrate the following scenarios. The division *connection* is responsible for the physical connections between customers. On the *website* private customers monitor their invoices and change their contract details. The *administration* is in charge of handling of customers' payments. All these different contexts have equal parts: The connection-division and the website interact with customers. Both, the customers via website and the administration, need to access to the invoices. All these different cases are put together to only one complex application.

Figure 2 displays the example in an UML-like notation. If a class has several roles e.g. `PrivatePhoneCustomer` 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. `Person` match the standard UML notation. In this case the name of the role is concordant with the name of the class. Dependencies are marked with horizontal lines e.g. between `designation` and `name` in `PrivatePhoneCustomer` or between the two inheritance arrows pointing toward `Invoice`. Inheritance arrows and associations are drawn between roles and not between classes.

The role `PointOfConnection` has a designation, a phone number, a connection sort and a method which creates a phone connection between two points. The role `Person`

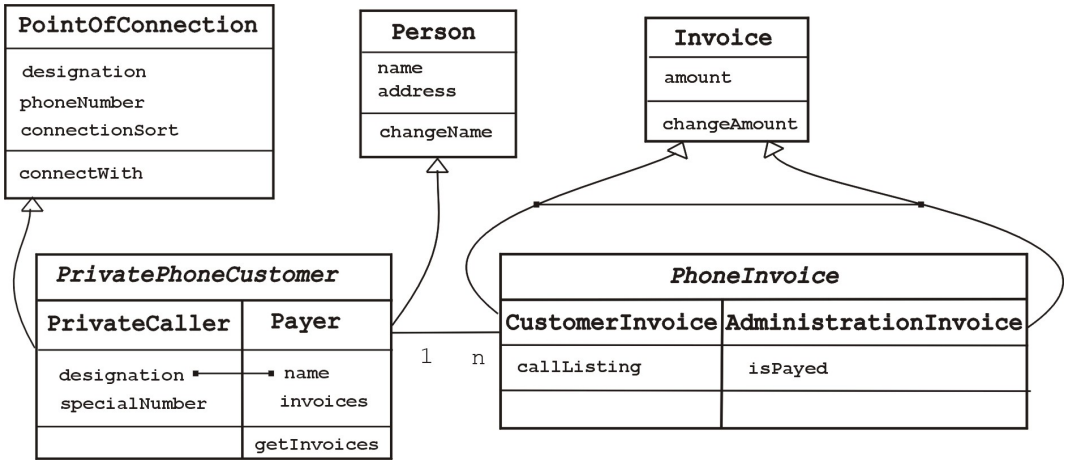

Figure 2: Class diagram about the phone customer

has a `name`, an address and a method to change the `name`. The class `PrivatePhoneCustomer` contains two roles: `PrivateCaller` and `Payer`. `PrivateCaller` is used in the context of the *connection* and `Payer` in the context of the *website*. `PrivateCaller` inherits from the role `PointOfConnection` because it is a special kind of the connection, business connection could be another one. `Payer` inherits from `Person` and has attributes and methods used for invoices. The attribute `designation` in `PrivateCaller` should be identical with the `name` of `Payer`. If the `name` changes, `designation` has to be changed too and vice versa. The roles `CustomerInvoice` and `AdministrationInvoice` inherit from `Invoice`. They build the class `PhoneInvoice`. `CustomerInvoice` provides an overview of single phone calls for the customer and `AdministrationInvoice` informs the administration whether the invoice is payed or not. In both roles the attribute `amount` which is inherited from `Invoice` always has the same value, the inherited method `changeAmount` is semantically equal.

The role `Payer` has an association to the role `CustomerInvoice`, it only views the attributes `amount` and `callListing`, but not `isPayed`. If the administration changes `amount` in `AdministrationInvoice`, the appropriate object of the class `PrivatePhoneCustomer` realizes that.

Roles and the Dependencies of Attributes and Methods

In this paper we concentrate on the static case. That is all roles of an entity are always active. It is not possible to append and remove roles. We take over all fundamentals from the object model except for classes and roles.

Definition 1. A *role* is a tuple $R = \langle N_R, A_R, M_R \rangle$ where N_R is the name of the role, A_R the set of attributes and M_R the set of methods.

We denote attributes of `R` by a_R and methods by m_R.

Now we introduce dependencies. Roles are dependent if a dependency among their attributes or methods exists. It is allowed to define dependencies between each pair of roles. This is impossible in other approaches; most of them assume that roles of the same entity have the same common part.

Definition 2. Two attributes of different roles are *equality-dependent* if and only if these attributes always have the same value.

Equality-dependent attributes are described in (Bækdal & Kristensen 2000) as shared properties. Let `a` and `b` be equality-dependent attributes of roles `R` and `S`. We denote it by $a_R \sim b_S$.

Definition 3. Two methods of different roles are *equality-dependent* if and only if these methods are semantically equal.

If `m` and `n` are methods of roles `R` and `S` we denote the equality dependency of methods by $m_R \sim n_S$. Two semantically equal methods are not necessarily syntactically equal. E.g. let the method `m` use the attribute `a`. Let `b` be an equality-dependent attribute on `a`. We generate the method `n` by copying the contents of `m` and then by replacing every occurrence of `a` with `b`. In this way we get a new method `n` which is semantically but not syntactically equal to `m`. Actually we are interested in the subset of semantically equal methods which can be automatically adjusted if one of the equality-dependent methods changes.

Definition 4. A *class* is a tuple $C = \langle N_C, R_C, D_C \rangle$ where N_C is the name of the class, R_C the ordered set of roles and D_C the dependencies between these roles.

The reason for using *ordered* is explained in the section *Further Aspects of Inheritance*. $R >_{pr} S$ means that role `R` is before the role `S` in the ordered set of roles. That is the role `R` has higher priority than `S`.

Roles build a new level between classes and attributes and methods. Roles roles correspond to classes in the standard object model. But roles are not instantiable. Here several roles belong to the same class: `PrivateCaller` and `Payer` are considered as aspects of the same person.

Classes in the former object model can be interpreted as classes with only one role, e.g. `Person`. Hence, the role concept is a canonical extension of the object model. In this way existing software can be integrated without changes.

The restriction, that equality-dependent attributes and methods have to belong to different roles, makes sense because dependencies inside a role must be implemented inside this role and not at the class level.

The following code fragment with Java-like syntax clarifies the issue. The class `PhoneInvoice` has two roles `CustomerInvoice` and `AdministrationInvoice`. The attributes named `amount` and the methods named `changeAmount` are equality-dependent in both roles. The priority order is given by the syntactical order of roles: `CustomerInvoice`$\geq_{pr}$`AdministrationInvoice`. For now we neglect that the roles `CustomerInvoice` and `AdministrationInvoice` inherit from `Invoice`.

```
role CustomerInvoice {
  double amount;
  String[] callListing;
  public void changeAmount(double amount) {
    this.amount = amount;
  }
}
role AdministrationInvoice {
  double amount;
  boolean isPayed;
  public void changeAmount(double amount) {
    this.amount = amount;
  }
}
class PhoneInvoice {
  include CustomerInvoice;
  include AdministrationInvoice;
  equal(CustomerInvoice:amount,
      AdministrationInvoice:amount);
  equal(CustomerInvoice:changeAmount(double),
      AdministrationInvoice:
          changeAmount(double));
}
```

If a class has more than two roles, it is often the case that attributes or methods of all roles are equality-dependent. But this is not necessary. There are many examples where there is need for such flexible dependencies (Chernuchin, Lazar, & Dittrich 2005). In contradiction to this example attributes or methods of roles can have different names but the same values or meanings, respectively. It is also possible that attributes or methods have equal names but different values or meanings. Moreover, different classes can include the same role.

The *function dependency* generalizes equality dependency if there is a function f with $a = f(b)$. Thereby `a` and `b` are attributes of roles which belong to the same class. The definition of function dependency in (Chernuchin & Dittrich 2005) differ much from the presented above.

The following example gives the idea of the function dependency. The function has to be implemented as a method of the class or of its role:

```
role EuropeanObject {
  int size;
  //...
}
role AmericanObject {
  double size;
  //...
}
class InternationalObject {
  include EuropeanObject;
  include AmericanObject;
  int inchToCentimeter(double inch) {
    return inch * 2.54;
  }
  double centimeterToInch(int cm) {
    return cm / 2.54;
  }
  dependent(EuropeanObject:size,
     AmericanObject:size, centimeterToInch);
  dependent(AmericanObject:size,
     EuropeanObject:size, inchToCentimeter);
}
```

This example demonstrates two function dependencies. There is one representation of size measured in inch and one measured in centimeter. The representations are adjusted with the class methods `inchToCentimeter` and `centimeterToInch`. If one of the function-dependent attributes changes, the corresponding method changes the other attribute. These methods at the class level are similar to static methods in the object model. But they can also be located in a role.

An object has the whole complexity of its class. The role concept supports information hiding because a client accesses only some roles, e.g. a variable of type `CustomerInvoice` occurs only in the context of *website* and a variable of type `AdministrationInvoice` occurs only in the context of *administration*. We access both roles via references: two variables refer to the same object, but they have different static types:

```
CustomerInvoice invoiceInPortal = new
   PhoneInvoice();
AdministrationInvoice administrationInvoice
   =(AdministrationInvoice) invoiceInPortal;
```

The role corresponds to the static type and the class to the dynamic type. A class can not occur as a declaration type.

The constructor is at the class level. Values are transmitted to the roles. One possible constructor of `PhoneInvoice` is

```
public PhoneInvoice(double amount,
    boolean isPayed, String[] callListing){
  AdministrationInvoice:amount = amount;
  AdministrationInvoice:isPayed = isPayed;
  CustomerInvoice:callListing =
     callListing;
}
```

The constructor has three arguments `amount`, `isPayed` and `callListing`. The attribute `amount` of the role `AdministrationInvoice` gets the value of the argument `amount`. Because of the equality dependency the attribute `amount` of `CustomerInvoice` gets the same value. Then the attribute `isPayed` is transmitted to the role `AdministrationInvoice`, etc. Furthermore, the class constructor can invoke local role constructors. An alternative implementation of the constructor above is:

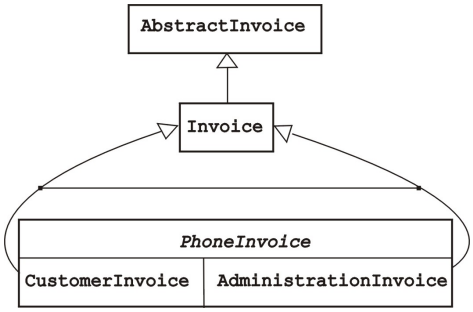

Figure 3: Same superrole

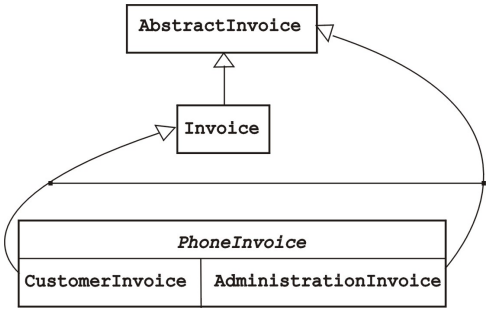

Figure 4: Different superroles

```
public PhoneInvoice(double amount,
    boolean isPayed, String[] callListing){
  AdministrationInvoice(amount, isPayed);
  CustomerInvoice(amount, callListing);
}
```

The dependency of attributes is of different kind from the dependency of methods. The dependency of attributes is a runtime dependency. If during the runtime the value of one attribute changes, the value of the other changes, too. This is similar to two variables referring to the same object. The dependency of methods is a development dependency. Usually methods do not change their contents during the runtime. But during the development a programmer has to change only one of the methods because the other changes automatically.

Dependencies by Using Inheritance of Roles

In this section we present the inheritance of roles. With $R < S$ we denote that role `R` inherits from role `S`. $R \leq S$ means that `R` inherits from `S` or that `R` and `S` are the same role. The latter is due to the fact that different classes may have the same role. We allow only single and interface inheritance of roles. In our example the role `Payer` inherits from `Person`. The rules of inheritance are taken over from the object model. Aspects of multiple inheritance can be expressed because an object contains all roles of its class (see section *Related Work*)).

A special case of inheritance of roles is given, if different roles of one class have the same ancestor. In our example `Invoice` is such an ancestor. If there is no dependency declared, attributes and methods of subroles are independent.

Definition 5. Two roles are *inheritance-dependent* if and only if a common ancestor of both roles exists and each inherited attribute and not overwritten method is equality-dependent on its counterpart of the other role.

Let R, S and T be roles and T be an ancestor of R and S. More formally we can describe the situation of definition as follows:
$R \sim_T S \Leftrightarrow$

1. For any $a_T \in A_T$ and its inherited counterparts $a_R \in A_R$ and $a_S \in A_S \quad a_R \sim a_S$ holds.
2. For any $m_T \in M_T$ and its inherited not overwritten counterparts $m_R \in M_R$ and $m_S \in M_S \quad m_R \sim m_S$ holds.

We always have to refer to the ancestor, because two roles can have more than one common ancestor (see figure 3). Furthermore, the ancestor does not need to be the direct superrole (see figure 4). The methods should not be overwritten because the overwriting usually changes their semantics.

The following code fragment is a simplification of the code example from previous by means of inheritance dependency.

```
role CustomerInvoice extends Invoice {
  String[] callListing;
}
role AdministrationInvoice extends Invoice {
  boolean isPayed;
}
class PhoneInvoice {
  include CustomerInvoice;
  include AdministrationInvoice;
  inheritanceDependent(
      CustomerInvoice:Invoice,
          AdministrationInvoice:Invoice);
}
```

Proposition 1. *Let* `R`, `S`, `T` *and* `U` *be roles. Let* `R` *and* `S` *are inheritance-dependent via* `T`. *If* `U` *is an ancestor of* `T` *then* `R` *and* `S` *are inheritance-dependent via* `U`*:*

$$R \sim_T S \wedge T \leq U \Rightarrow R \sim_U S$$

Proof. Since all attributes of `T` are equality-dependent in `R` and `S`, each subset of these attributes is equality-dependent. Because of the inheritance attributes of `U` are a subset of attributes of `T`:

$$T < U \Rightarrow A_U \subseteq A_T$$

Hence, all attributes of `U` are equality-dependent in `R` and `S`. Analogously, one shows that all non-overwritten methods of `U` are equality-dependent in `R` and `S`. Consequently, `R` and `S` are inheritance-dependent via `U`. □

If, in extension to our example, `Invoice` inherits from `AbstractInvoice` then `CustomerInvoice` and `AdministrationInvoice` are inheritance-dependent via `AbstractInvoice`, too.

We have another special case of inheritance if several roles of one class inherit from several roles of another class. Without class dependency (see definition 6) we have to declare all dependencies between these roles

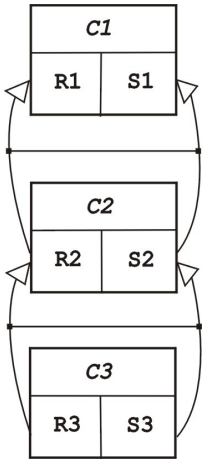

Figure 5: Transitivity of the class dependency

twice. By means of class dependencies we can easily add a new class e.g. `ExtendedPhoneInvoice` with the roles `ExtendedCustomerInvoice` and `ExtendedAdministrationInvoice` without declaring dependencies which are already between roles `CustomerInvoice` and `AdministrationInvoice`.

Definition 6. Let `R1` be an ancestor of role `R2` and `S1` an ancestor of role `S2`. Let the roles `R1` and `S1` belong to the class `C1` and the roles `R2` and `S2` to the class `C2`. The roles `R2` and `S2` are *class-dependent* via `R1`, `S1` and `C1` if and only if all dependencies of inherited attributes and not overwritten methods of roles `R1` and `S1` in `C2` are taken over to roles `R2` and `S2`.

The following proposition shows that class dependency is transitive. This is possible to inherit dependencies through several levels.

Proposition 2. *Let* `R1` *and* `S1` *be roles in class* `C1`*,* `R2` *and* `S2` *roles in class* `C2` *and* `R3` *and* `S3` *roles in class* `C3`*. Let* `R1` *be an ancestor of* `R2` *and* `R2` *be an ancestor of* `R3`*. Let* `S1` *be an ancestor of* `S2` *and* `S2` *be an ancestor of* `S3`*. Further let* `R2` *and* `S2` *be class-dependent via* `C1`*,* `R1` *and* `S1`*;* `R3` *and* `S3` *be class-dependent via* `C2`*,* `R2` *and* `S2`*. Then* `R3` *and* `S3` *are class-dependent via* `C1`*,* `R1` *and* `S1`*. Formally:*

$$\begin{aligned} R1, S1 \in R_{C1} \wedge R2, S2 \in R_{C2} \wedge R3, S3 \in R_{C3} \\ \wedge\, R3 \leq R2 \leq R1 \wedge S3 \leq S2 \leq S1 \\ \wedge\, R2 \sim_{C1,R1,S1} S2 \wedge R3 \sim_{C2,R2,S2} S3 \\ \Rightarrow R3 \sim_{C1,R1,S1} S3 \end{aligned}$$

Proof. Figure 5 describes the situation. Inheritance is transitive, hence `R1` is ancestor of `R3` and `S1` is ancestor of `S3`.

$$R3 \leq R2 \leq R1 \Rightarrow R3 \leq R1$$
$$S3 \leq S2 \leq S1 \Rightarrow S3 \leq S1$$

Because of the class dependency the set of dependent attributes of `R2` and `S2` is a subset of dependent attributes of `R3` and `S3`. Analogously, the set of dependent attributes of `R1` and `S1` is a subset of dependent attributes of `R2` and `S2`

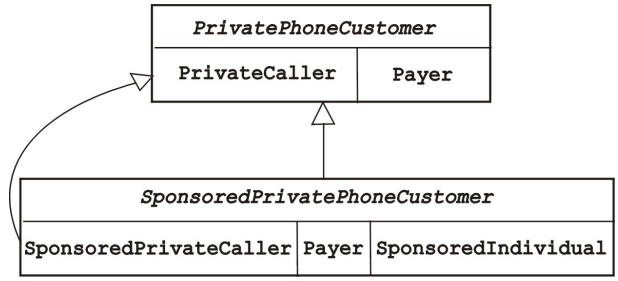

Figure 6: Class inheritance

Hence, all dependent attributes of `A1` and `B1` are in `A3` and `B3` dependent, too. The argumentation about dependent methods is analogous. Hence, `A3` and `B3` are class-dependent via `C1`, `A1` and `B1`. □

Further Aspects of Inheritance

In addition to the access levels public, protected and private we introduce the new level *inner*. It resides between protected and public. Private properties are only accessible within a role, protected properties within the inheritance hierarchy of roles. By means of inner, additional access to properties of other roles inside the same class is allowed, by means of public, to properties of all roles of all classes. The access level inner is default.

Accordingly to the real world we introduce the inheritance of natural types or classes, respectively.

Definition 7. Let
$C = \langle N_C, \{R_1, \ldots, R_n, S_1, \ldots, S_m\}, D_C \rangle$ and
$D = \langle N_D, \{R_1, \ldots, R_n, T_1, \ldots, T_m, U_1, \ldots, U_k\}, D_D \rangle$
be classes. D inherits from C if and only if the following conditions are fulfilled:

$$\begin{aligned} T_1 < S_1, \ldots, T_m < S_m \\ \forall i, j \in \{1, \ldots, m\} \text{ with } i \neq j : T_i \sim_{C,S_i,S_j} T_j \\ \forall i \in \{1, \ldots, n\}, j \in \{1, \ldots, m\} \ : R_i \sim_{C,R_i,S_j} T_j \end{aligned}$$

This means that roles of the superclass can be overtaken or replaced with their subroles in the subclass. The dependencies are overtaken from the superclass or extended. Finally, new roles can be attached in the subclass.

The following code fragment demonstrates inheritance of classes in our approach. The new class `SponsoredPrivatePhoneCustomer` inherits from the class `PrivatePhoneCustomer` (see figure 6). The new class has three roles. The role `Payer` is taken over without changes. The role `PrivateCaller` is replaced with its subrole `SponsoredPrivateCaller`. The third role `SponsoredIndividual` is new. The order of roles does not change, the new role is as usual at the end of the priority order.

```
role SponsoredPrivateCaller extends
    PrivateCaller {
  String secondSpecialNumber;
}
role SponsoredIndividual {
```

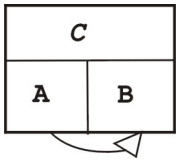

Figure 7: Inheritance of roles in the same class

```
  //...
}
class SponsoredPrivatePhoneCustomer extends
    PrivatePhoneCustomer {
  replace PrivateCaller,
      SponsoredPrivateCaller;
  include SponsoredIndividual;
}
```

In the following, we demonstrate why the *set* of roles inside a class does not suffice and why we need the *ordered set* and priorities, respectively. We use a role or its superrole as static type. If e.g. `CustomerInvoice` and `AdministrationInvoice` are not inheritance-dependent, the attribute `amount` will have different values in both roles. A possible reason is that the administration calculates with taxes. Let the variable `phoneInvoice` have the dynamic type `PhoneInvoice`. The assignment

```
CustomerInvoice invoice = phoneInvoice;
System.out.println(invoice.getAmount());
```

exactly specifies the role and thus the value of `amount`. But if we use the superrole of `CustomerInvoice` as the static type

```
Invoice invoice = phoneInvoice;
System.out.println(invoice.getAmount());
```

the compiler cannot distinguish between `amount` of `AdministrationInvoice` and `amount` of `CustomerInvoice`. In this case the ordered set gives us a role with the highest priority that is the role which is syntactically the first in the class definition and is a subrole of `Invoice`. In this context we distinguish between the *static role* `Invoice` and the *actual role* `CustomerInvoice`.

The change in the order of roles by class inheritance does not disturb the is-property of inheritance. The subclass can replace its superclass. We demonstrate this situation by means of an example: Let `C` be a class with roles `R` and `S`. Let `R` have higher priority than `S`: $R >_{pr} S$. Class `D` inherits from `C` and has the same roles `R` and `S` but in the inverse order $R <_{pr} S$. We consider two assignments

```
R r = value;
S s = value;
```

Let `value` have the dynamic type `C`. If `S` inherits from `R` $S < R$, the variable `r` has the actual role `R` because of its higher priority. In the second assignment `s` has role `S` because `R` with higher priority is not specific enough. Now let `value` have the dynamic type `D`. In the first assignment `r` has role `S` because `S` has the highest priority and every `S` is always `R`, too. In the second assignment `s` has the role `S`.

Moreover, this example demonstrates the problem with roles in one class which inherit from each other (see figure 7). If $A < B$ and $A >_{pr} B$ there is no way for an object to have the actual role `B`.

Nevertheless, priorities solve the most problems in this context because they always has an unambiguous solution. But this solution does not conform to the real world paradigm. There, the choice of the appropriate role depends on the context and the priority may change. Our favorite solution deals with contexts (Chernuchin & Dittrich 2005). Another possible way is the specification of the current role with a cast, similar to the as notation in (Carré & Geib 1990) e.g.:

```
((CustomerInvoice) invoice).getAmount();
```

A desirable property by modelling roles is that a class can have the same role for several times. This is necessary if e.g. class `FlyingTicketWithChanging` has two independent roles `Fly`. For this situation we have the rename-construct `as`.

```
class FlyingTicketWithChanging {
  include Fly as FirstFly;
  include Fly as SecondFly;
}
```

We can unambiguously specify the current role with casts:

```
  SecondFly fly = //...;
  System.out.println(fly.getDestination());
```

Implementation

The idea is to implement a development environment for our role concept. At the moment we are developing an Eclipse plugin. While changing an equality-dependent method in one role, the plugin automatically modifies corresponding methods in other roles. On the one hand, it will facilitate displaying the roles of a class at the same time. On the other hand, it will allow to handle dependencies with help of a precompiler. The precompiler converts the role code to legal Java code and passes it to the Java compiler.

In Java we have at least two possibilities to implement roles: role object pattern (Bäumer *et al.* 2000) and interface inheritance (Steimann 2001). By using role object pattern the equality dependency of attributes is implemented via references. There are constant references to the same object.

```
privateCaller.setDesignation
    (currentDesignation);
payer.setName(currentDesignation);
```

But references do not suffice to express equal dependencies between attributes. It is one of the points of our current research.

By using interfaces all functionality of the roles is in the same class. All methods work on the same attribute:

```
// overwrites the method inherited from
// the interface PrivateCaller
public String getDesignation() {
  return designation;
}
// overwrites the method inherited from
// the interface Payer
public String getName() {
  return designation;
}
```

The implementation is in a very early stage. Until now there are neither tools nor documentation.

Summary and Future Work

We are developing a new concept which allows a client to view an object from several perspectives in the statically typed object-oriented area. The object can have different roles in different contexts. The complexity is reduced by hiding irrelevant information. If objects have a lot of attributes and methods, it is reasonable to restrict diversity via projections or roles, respectively.

The role concept is a canonical extension of the object model. We introduce a new layer of roles. A class contains roles and their dependencies. Roles can have shared, dependent and independent parts. We allow inheritance of roles as well as classes.

To evaluate our approach we take some criteria from (Chernuchin, Lazar, & Dittrich 2005; Steimann 2000). Following criteria are supported: information hiding, encapsulation, dependency, hierarchy of natural and role types, identity sharing. Up to now only dynamicity is not supported, i.e. the quality to attach and remove roles dynamically. The most important advantages of our approach are its comprehensibility, information hiding and versatile dependencies between roles.

The most important future task is the development of tools to work with roles. We are working on an Eclipse plugin for this purpose. Particularly, the implementation of equality-dependent attributes is interesting. Until now all roles must always be activated. We have to enlarge this concept to be able to have dynamic roles which can be activated and deactivated. Furthermore, we have to improve our solution for the diamond problem. Finally, we intend to export the dependencies, in particular the equality dependencies between attributes, outside of the role concept to the object-oriented area.

References

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.

Carré, B., and Geib, J.-M. 1990. The Point of View Notion for Multiple Inheritance. *ACM SIGPLAN Notices* 25(10):312–321.

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

Chernuchin, D.; Lazar, O.; and Dittrich, G. 2005. Comparison of Object-Oriented Approaches for Roles in Programming Languages. In *2005 AAAI Fall Symposium: Roles, an interdisciplinary perspective*.

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

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. 2002. 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.

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

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

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.

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*.

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.