FI MU

Faculty of Informatics
Masaryk University

Object with Roles and VREcko system

by

Lubomír Markovič

FI MU Report Series **FIMU-RS-2003-06**

 September 2003

Object with Roles and VREcko system

Lubomír Markovič

Abstract

A model based on objects with roles concept is presented. It provides both theoretical and practical framework for construction of objects. In this model the objects with roles can dynamically change the interfaces they support. Some advantages of creating applications using this concept are discussed on an example of a system of virtual reality.
Keywords: object model, role, dynamic applications, virtual reality

1 Motivation

Object oriented technology brings many advantages when used to design systems with static structures. Requirement analysis methods are tailored to capture the traditional "is-a" and "part-of" structures in class hierarchy models and to derive the necessary associations for inter-object communication. The serious drawback of traditional techniques is the impossibility to model the situations where an application has to reflect the dynamical changes in object responsibilities and behavior.

The need to support dynamically changing objects can be found in many application domains. Frequently mentioned example is an object representing a person, playing the different roles over the time (child, student, parent, employee, ...). Many others objects, e.g. documents, products passing production processes, may also serve as the examples. We will discuss another interesting domain here – virtual reality systems.

In [14] and [15] the flexible object model based on roles was proposed. A slightly modified model extending expressiveness of mutual exclusiveness between roles is presented in this report.

2 Existing Techniques

The advocacy of *objects with roles* is done in many papers ([8][18][16][2] [14][21][12]...) Models dealing with evolving objects were published e.g. in [6][4][7][11][17]. The main techniques (ideas) that stay behind these models are shown in Figure 1.

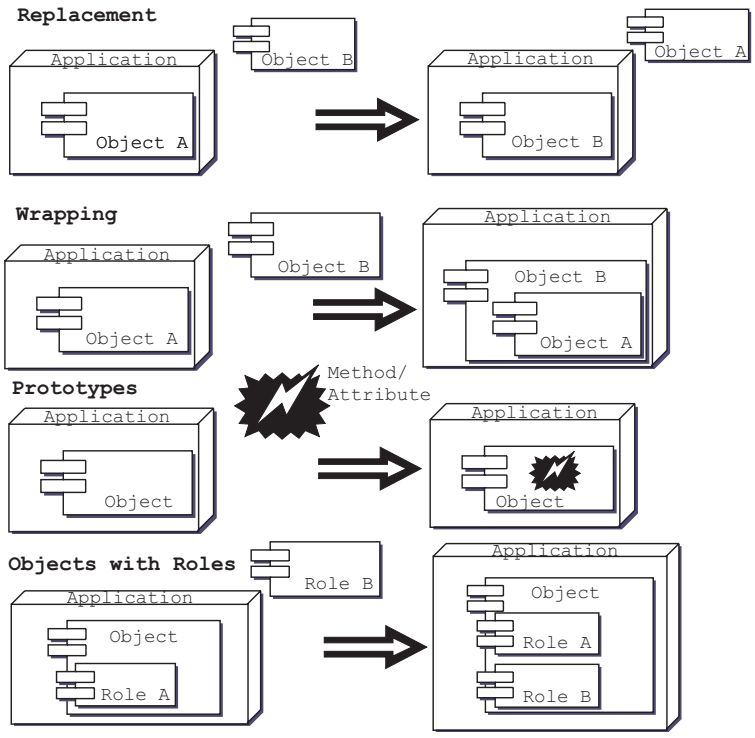

Figure 1: Existing Techniques

Replacement – The change of the behavior of an object in an application is accomplished by replacement of the original object with another one.

Wrapping – This technique is very similar to previous one. The main difference is that the original object remains in the application to be internally used by a new object (using delegation or consultation [11]). The wrapping object needs to implement only the changes to wrapped object.

In the case of replacement or wrapping the object's interfaces are static, they cannot be extended or modified later. The advantage of both techniques is that if the type (interface) of a new object is the same as or a subtype of, a type of a previous object, the object modifications can be done type-safely. The disadvantage is that a new object, representing the same real entity, is introduced in an application.

Prototypes – Objects formed with prototyped based OOP techniques [3] [20] [22] are not restricted by any static interface. Every object can have its own interface that can be changed dynamically. New methods or attributes can be added to any prototyped object directly. The disadvantage of these techniques is their problematic type control, as the objects can change their interfaces unpredictably.

Objects with Roles – Objects consist of "smaller objects" called role instances (or just roles). Object interface is changed dynamically by adding new, or removing existing roles.

The definition of the *role* appropriate for our model is taken from [13]:

> A role of an object is a set of properties which are important for an object to be able to behave in a certain way expected by a set of other objects.

3 Previous Work – Objects with Roles

In this section we will give an overview of previous work dealing with objects with roles. In some works the roles are called differently (aspects, views, subjects ...) but semantically they describe the same idea and try to solve the same problems.

Aspects

This article [18] from Richardson and Schwarz propose a strongly-typed model based on interfaces (types) and implementations. The Melampus Data Model enabling dynamic acquisition of new interfaces is presented. Objects may have multiple aspects (in the sense of roles) which has still the same OID of the whole object. There is no restriction on aspect acquisition. An object can be manipulated only through one of its aspects. There is strict separation between interfaces and implementations. No inheritance relation between interfaces is supported, the structural subtyping (conformity) is used instead thus an aspect extending an object must explicitly duplicate the object's interface in its definition. An object can't support one aspect more than once in this approach.

Fibonacci

Albano et al. have defined [2] a strongly typed object-oriented database programming language called Fibonaci. An object has only an identity and contains an acyclic graph of roles. An object is not manipulated directly, but always through one of its roles (messages are sent to roles). Objects can change their set of roles without affecting their identity. There is strict separation between interfaces (role types) and implementations (roles). There exist two orthogonal type hierarchies, an *object type hierarchy* and a *role type hierarchy*. A role type can extend an object type. A role type hierarchy may be dynamically extended by defining a new role type as subtype of one or several existing role types. Multiple roles are not supported. The importance of mutually exclusive roles is advocated on the example of a person that can't be employed and unemployed at the same time.

Objects with Roles

Pernici described an formal model of objects with roles [16]. The advocacy and semantics of base role, suspension of roles and the need for some constrains of objects extensions by roles is provided. Multiple roles (as defined in previous sections) are not supported.

Extending Object-Oriented Systems with Roles

In [8] a model of objects with roles based on classical class-based system (SmallTalk here) is described. Two orthogonal hierarchies are considered – classes and roles. Both can be organized in inheritance relation and objects (instances of classes) can be dynamically extended by instances of roles (that can be extended by another instances of roles as well). Extension is limited by "roleOf" relation. Objects are manipulated through one of its role instance or object itself (it's a base roles in fact). Advocacy of multiple roles (called ,,qualified roles" here) is provided.

Delegation

Very detailed description of delegation technique is done in [11]. In [10] is provided a comparison of delegation and some techniques of objects with roles. Support for multiple roles is depicted as useless because it can be simulated by association between more objects. But the purpose of technique of objects with roles is mainly to eliminate associations between objects describing the same entity. Delegation in general has a serious problems with support of type control.

4 Formal Model of Objects with Roles

In the following sections we introduce the new model of objects with roles. It's features: the support of replacement and wrapping techniques on the level of role instances, support of dependency and of role exclusion, are described.

4.1 Basic Terms

We need to establish a basic terminology more precisely at first.

Role is a type determining interface. In many papers it's called *role type*.

Role Class is a concrete class that implements some role, it's obvious that there can be many different role classes implementing the same role. A role class can implement only one role, but this role can inherit from many other roles.

Role Instance is an instance of some role class.

Let $\mathcal{R}$ be a set of all roles meaningful in application context.

Let $\mathcal{C}$ be a set of all role classes defined in application context.

Let $\mathcal{C}_R$ be a set of all role classes implementing $R \in \mathcal{R}$ as the most specific role.

$$\mathcal{C} = \bigcup_{R \in \mathcal{R}} \mathcal{C}_R$$

Similarly $\mathcal{RI}$ denotes the set of all role instances.

The instance $RI \in \mathcal{RI}$ supporting $R \in \mathcal{R}$ as the most specific role is denoted as $RI :!R$.

The instance $RI :!R$ of the role class $C \in \mathcal{C}_R$ is denoted as $RI!C$.

Objects are distinguishable by their unique global identifier OID (in some implementations their memory address can be used for this purpose).

The role instances are distinguishable by their unique global identifier GID (in some implementations their memory address can be used for this purpose).

The role instances are distinguishable *in an object* (page 8) by their object–unique role instance's identifier RID. Role instance that is included in some object has set their RID property and also has set its property called OID which refers back to the object. If a role instance is not included in any object than its RID has a value $NoRID$ and a value of OID is *null*.

Role instance $RI \in \mathcal{RI}$ is said to be *bounded*, if it is included in an object.

The set of all bounded role instances is $\mathcal{RI}_{\mathcal{B}ound}$ ($\mathcal{RI}_{\mathcal{B}ound} \subseteq \mathcal{RI}$).

With respect to role types [14] the set of all roles $\mathcal{R}$ splits in two disjoint subsets $\mathcal{R}_{\mathcal{S}}$ and $\mathcal{R}_{\mathcal{M}}$. Roles from the set $\mathcal{R}_{\mathcal{S}}$ are *s-roles* (single roles), roles from the set $\mathcal{R}_{\mathcal{M}}$ are *m-roles* (multiple roles).

$$\mathcal{R}_{\mathcal{S}} \cup \mathcal{R}_{\mathcal{M}} \quad = \quad \mathcal{R}$$

$$\mathcal{R}_{\mathcal{S}} \cap \mathcal{R}_{\mathcal{M}} \quad = \quad \emptyset$$

Every object includes at most one instance of some s-role (exception to this rule is described in 4.6.1) and zero-to-many instances of any m-role.

4.2 Inheritance Relation

(Multiple) Inheritance Relation $\leq$ is defined as relation $\leq \subseteq \mathcal{R} \times \mathcal{R}$ that is reflexive, antisymmetric, transitive and meets condition
(I1) $(R_1 \leq R_2) \wedge (R_2 \in \mathcal{R}_{\mathcal{S}}) \Longrightarrow (R_1 \in \mathcal{R}_{\mathcal{S}})$
$R_1 \leq R_2$ means that role R_1 is a *descendant* of role R_2, and vice versa R_2 is the *ancestor* of R_1.

The condition (C1) states that a child of s-role must be also an s-role. The next condition can restrict the model to single inheritance only. But we will consider multiple inheritance in the following. If single inheritance would be consider, nothing needs to change but some expressions could be more simple.

Single Inheritance Relation $\leq$ is defined as inheritance relation which meets the condition
(I2) $(R_1 \leq R_2) \Longrightarrow \nexists R_3 : (R_1 \leq R_3) \wedge (R_2 \not\leq R_3) \wedge (R_3 \not\leq R_2)$

This rule expresses that any role can have at most one parent.

The inheritance relation defines an oriented acyclic graph usually organized as a tree.

Let's define some abbreviations that helps to simplify next expressions.

- $\star$ $(RI :: R)$ abbreviates $\exists (R_1 \in \mathcal{R}) : (RI :! R_1) \wedge (R_1 \leq R)$
- $\star$ $R_1 < R_2$ abbreviates $(R_1 \leq R_2) \wedge (R_1 \neq R_2)$
- $\star$ $R_1 <_! R_2$ abbreviates $(R_1 < R_2) \wedge \nexists R_3 (R_3 < R_2) \wedge (R_1 < R_3)$
- $\star$ Function $SubRoles(R \in \mathcal{R}) : 2^{\mathcal{R}}$ returns all $R_1 : (R \leq R_1)$ (all sub-roles of role R).

4.3 Dependency Relation of Roles

Relation $\hookrightarrow \subseteq \mathcal{R} \times \mathcal{R}$, that meets (D1-D5) is *dependency relation of roles.* $R_1 \hookrightarrow R_2$ means that role R_1 depends on role R_2 (or RI_1 is a role of RI_2 e.g.[8]).

$\hookrightarrow^*$ is a transitive closure.

(D1) $(R_1 \leq R_2) \Longrightarrow (R_1 \hookrightarrow R_2)$
Roles depend on self and their ancestors. The rule is called ***wrapping rule***.
(D2) $(R_1 \leq R_2) \wedge (R_2 \hookrightarrow R_3) \Longrightarrow (R_1 \hookrightarrow R_3)$
If an ancestor of a role R_1 depends on another role, then R_1 depends on it too. Dependency is hereditary.
(D3) $(R_1 \hookrightarrow R_2) \Longrightarrow (R_2 \not\leq R_1)$
Roles cannot depend on their children.
(D4) $(R_1 \hookrightarrow^* R_2) \wedge (R_3 \leq R_2) \wedge (R_1 \not\leq R_2) \wedge (R_3 \not\leq R_1) \Longrightarrow (R_3 \not\hookrightarrow^* R_1)$

Role cannot depend on roles that depend on it's ancestors (cyclic dependency is not allowed).

Relation $\hookrightarrow$ defines an oriented acyclic graph on the set of roles $\mathcal{R}$ which is in some sense orthogonal to inheritance relation.

4.4 Relation of Mutually Exclusiveness

Semantics of some roles forces them to be mutually exclusive. It means there can't be role instances of both of them in one object. Let's consider role hierarchy of roles of "Man" and "Woman". There is an s-role Person and its two children: Man and Woman. From a semantics of an s-role it follows that Man and Woman are mutually exclusive since both are Persons which is an s-role. This mutually exclusiveness is implied by their common s-role's ancestor.

Not every mutually exclusiveness can be expressed in the way of semantic of s-roles. Example of such a situation can be described on the example of the roles "Businessman" and "Politician". No politician can be a businessman usually. One person can be a politician or businessman more than once (this roles are multiple) so there is not any their common single ancestor (we don't need to define any their common ancestor in fact). But we need to express their mutually exclusiveness somehow, so we will define *relation of mutually exclusiveness*

$$\ominus \subseteq \mathcal{R} \times \mathcal{R}$$

$\ominus$ relation must meet next conditions:

(M1) $R_1 \ominus R_2 \Longrightarrow R_2 \ominus R_1$

Relation is symmetric.

(M2) $(R_1 < R_2) \wedge (R_3 < R_2) \wedge (R_2 \in \mathcal{R}_\mathcal{S}) \wedge (R_1 \not\leq R_3) \wedge (R_3 \not\leq R_1) \Longrightarrow R_1 \ominus R_3$

Mutual exclusiveness implied by semantics of s-roles is implicitly included.

(M3) $R_1 \leq R_2 \Longrightarrow R_1 \not\ominus R_2$

No role can be mutually exclusive with its ancestor.

(M4) $(R_1 \ominus R_2) \wedge (R_3 \leq R_1) \Longrightarrow R_3 \ominus R_2$

Mutually exclusiveness is hereditary.

(M5)

$(R_1 \hookrightarrow^* R_2) \wedge (R_3 \hookrightarrow^* R_4) \wedge (R_2 \ominus R_4) \Longrightarrow R_1 \ominus R_3$

Roles depending on *mutually exclusive roles* are mutually exlusive as well. No role may depend on mutually exclusive roles at the same time. Some examples of situations where this rule is violated are shown in the figure 2. In all examples it's not possible to add role R_1 into the existing hierarchy.

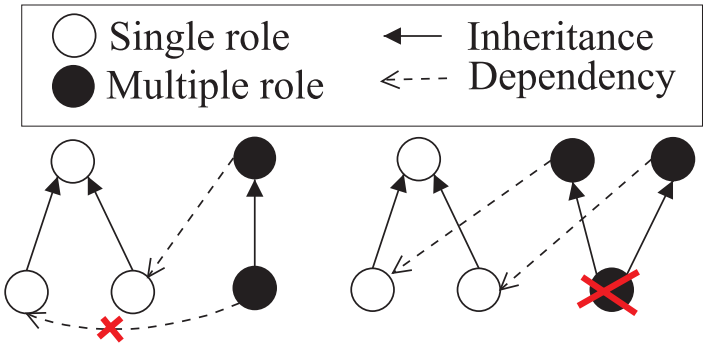

Figure 2: Examples of wrong roles' hierarchy

4.5 Object with roles

An object encapsulates the nonempty list of role instances it includes, containing its base role instance at least. Object has proper interface and functionality to maintain the dependency relations between its role instances.

The role R is a *base role* (b-role) iff $\nexists(R_1 \in \mathcal{R}) : (R \hookrightarrow R_1) \wedge (R \not\leq R_1)$.

The base role is every role that does not depend on any other role but on its children.

$\mathcal{R}_{\mathcal{B}} \subseteq \mathcal{R}$ is the set of all b-roles.

Object with roles o is defined as a tuple

$$o = (OID, RI_B, G)$$

where
– OID is a global unique object identifier
– $G = (V, E)$ is oriented acyclic graph of dependencies between role instances in object
– $V \subseteq \mathcal{RI}$ is a set of graph nodes, each node for one role instance included in an object
– $E \subseteq \mathcal{RI} \times \mathcal{RI}$ is a set of oriented edges storing the dependency relations. Dependency relation of role instances is described on page 10. – $RI_B \in o.V$ is the base role instance of an object

$\mathcal{O}$ is a set of all objects with roles.

$o!R$ means that an object $o \in \mathcal{O}$ supports a role R.

$$o!R \Longleftrightarrow \exists RI \in o.V : (RI :: R)$$

In the following text the next set elements are used:
$RI, RI_? \in \mathcal{RI}$; $R, R_? \in \mathcal{R}$; $o \in \mathcal{O}$

4.6 Wrapping Rule and Replacement

4.6.1 Wrapping Rule

The most significant change against the model described in [14] is the rule (D1) named as *wrapping rule*. This rule says that roles always depends on self and all its ancestors.

Wrapping rule allows to substitute role instance supporting role R_1 by another one supporting role $R_2 \leq R_1$, in such a way that original role instance remains in object to be internally used by wrapping role instance in performing its tasks.

The intended semantics of a wrapping rule is that a role instance that wraps another one takes over a task of a wrapped role. Especially wrapping role instance takes over the RID of the wrapped role instance (it has assigned another one consecutively).

As it makes no sense to wrap any role instance more than once, this possibility is forbidden by rules ((DI2),(DI3) on page 10). However, the wrapping role instance can be wrapped again, so we may get linear graph of wrapping role instances.

Every role instance has an attribute called $acceptedWrappedRole$. When role instance is not intended to wrap any role, this attribute has a value $null$, but if a role instance is intended to wrap some role, than this attribute determines this role. For example, if for some role instance $RI.acceptedWrappedRole = R$ then RI can wrap any role instance $(RI_1 :: R) : Role(RI) \leq Role(RI_1)$.

Let's define a set of all role instances that are intended to wrap another role instance:

$$\mathcal{RI}_{\mathcal{W}} = \{RI \in \mathcal{RI} : RI.acceptedWrappedRole \neq null\}$$

$$\mathcal{RI}_{\mathcal{W}} \subset \mathcal{RI}$$

4.6.2 Replacement

Wrapping is useful when implementation of wrapping role instance does not differ from wrapped role instance very much. If we want to change the behavior of a role more radically it is better to replace the original role instance by another one. Replacement technique allows to take some role instances out of the object and put another role instance to their place. The new role instance has the same RID as replaced one. Wrapped role instance cannot be replaced. The replacing role instance must be of the same role or a sub-role of the role of the replaced role instance.

We can simulate this replacement RI_1 with RI_2 by removing RI_1 from an object an putting RI_2. But with removing RI_1 all role instances that

(recursively) depends on RI_1 must be removed too so we must put them again into the object after RI_2 was added. The operation of replacement simplify this process and assures that all dependencies to an from RI_1 will be correctly remapped to RI_2.

A simplified schema of an object is shown in figure 3. A base role instance is shaded. Wrapped role instances are not accessible from outside of the object, they are hidden inside their wrapping roles. Dependencies between role instances are marked by dashed arrows.

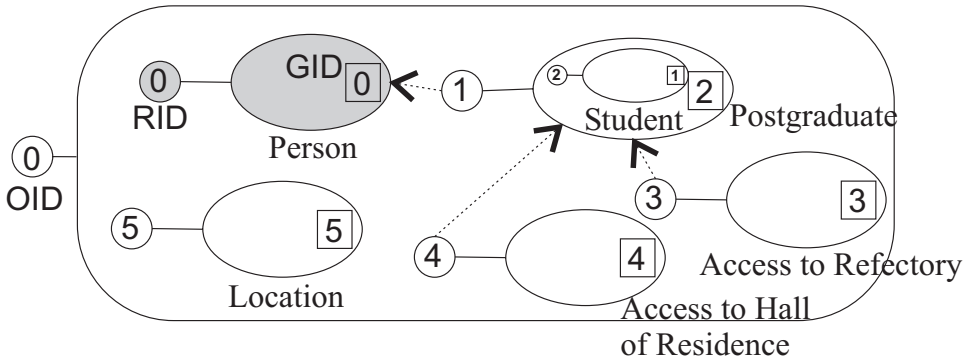

Figure 3: Simplified schema of an object

4.7 Dependency Relation of Role Instances of an Object

Relation $\hookrightarrow \subseteq \mathcal{RI} \times \mathcal{RI}$, that meets conditions (RI1-RI7) is *dependency relation of role instances*. $RI_1 \hookrightarrow R_2$ means that role instance RI_1 depends on role instance RI_2. The relation is defined only between role instances that are in the same object and this relation is reflected by the objects' graph $o.G$.

$(RI_1, RI_2) \in o.E \iff (RI_1 \hookrightarrow RI_2)$.

$\hookrightarrow^*$ is a transitive closure of this relation.

The dependency relation of roles determines many properties of the relation between instances of roles. The dependency relation of role instances is defined in a such way, that it emulates the relation of roles (rules (D1),(D5)), but restricts dependency between role instances using the wrapping rule and adds some other restricting rules.

We use the same symbol for dependency between role instances as for dependency between roles. Since both relations are defined on different sets, it will not cause a misunderstanding.

Some ancillary functions and abbreviations[1] are defined in the following list (all role instances are considered to be included in an object o):

[1]some abbreviations have changed against [15], especially $\overset{C}{\hookrightarrow}$ and $\overset{!}{\hookrightarrow}$ changed their meaning. The reason for this solution is better readability of the text.

- ⋆ Expression $RI_1 \not\rightarrow RI_2$ abbreviates $(RI_1, RI_2) \notin o.E$.
- ⋆ Function $Role(RI) : \mathcal{R}$ returns role $R : (RI :!R)$.
- ⋆ Expression $R_1 \overset{C}{\hookrightarrow} R_2$ abbreviates $(R_1 \hookrightarrow R_2) \wedge (R_1 \not\leq R_2)$, we say that R_1 cleanly depends on R_2 or R_1 depends on R_2 not using wrapping rule.
- ⋆ Expression $RI_1 \overset{C}{\hookrightarrow} RI_2$ abbreviates $(RI_1 \hookrightarrow RI_2) \wedge (Role(RI_1) \overset{C}{\hookrightarrow} Role(RI_2))$, we say that RI_1 cleanly depends on RI_2 or RI_1 depends on RI_2 not using wrapping rule.
- ⋆ Condition $RI_1 \overset{!}{\hookrightarrow} RI_2$ abbreviates $\exists(R \in SubRoles(Role(RI_2))) : (Role(RI_1) \hookrightarrow R)$, with semantics RI_1 can depend on RI_2.
- ⋆ Expression $RI_1 \overset{W}{\hookrightarrow} RI_2$ abbreviates $(RI_1 \hookrightarrow RI_2) \wedge (Role(RI_1) \leq Role(RI_2))$, we say that RI_1 wraps RI_2 or that RI_1 depends on RI_2 using wrapping rule.
  $RI_1 \overset{W}{\hookrightarrow}^* RI_2$ is a transitive closure.
- ⋆ Expression $RI_1 \overset{!W}{\hookrightarrow} RI_2$ abbreviates $(Role(RI_1) \leq Role(RI_2)) \wedge (Role(RI_2) \leq RI_1.acceptedWrappedRole)$, we say that RI_1 can wrap RI_2.
- ⋆ Set $o.\mathcal{W}rapped = \{RI \in o.V : \exists RI_2 \in o.V : RI_2 \overset{W}{\hookrightarrow} RI\}$ is a set of all role instances in o that are wrapped.
- ⋆ Set $o.\mathcal{N}onWrapped = \{RI \in o.V : \nexists RI_2 \in o.V : RI_2 \overset{W}{\hookrightarrow} RI\}$ is a set of all role instances in o that are not wrapped. — $o.\mathcal{N}onWrapped \cup o.\mathcal{W}rapped = o.V$
- ⋆ Condition $RI_1 \overset{!C}{\hookrightarrow} RI_2$ abbreviates $(RI_1 \overset{!}{\hookrightarrow} RI_2) \wedge (RI_1 \not\leq RI_2)$, with semantics RI_1 can cleanly depend on RI_2 (not using wrapping rule).
- ⋆ Function $GenerateUID() : GUID$ returns global unique identificator. This function is used for generating new unique OID for objects and GID for role instances.
- ⋆ Function $o.GenerateRID() : RID$ as a method of an object o that returns unique role instance identificator. This function is used for generating new unique RID for role instances. No two roles has the same RID during the o's lifetime.

Dependencies between *role instances in one object* must meet next conditions:

(DI1) $(RI_1 \hookrightarrow RI_2) \Longrightarrow (RI_1 \overset{!}{\hookrightarrow} RI_2)$

When a role instance RI_1 depends on another role instance RI_2, than the role of RI_1 must depend on some sub-role of RI_2.

This rule determines that all dependencies between role instances in an object are prescribed by a dependency relation between their roles.

(DI2) $(RI_1 \overset{W}{\hookrightarrow} RI_2) \Longrightarrow \nexists RI_3 : (RI_1 \neq RI_3) \wedge (RI_3 \hookrightarrow RI_2)$

Every role instance can be wrapped by only one another role instance, and if it is wrapped nothing can depend on it.

(DI3) $(RI_1 \overset{W}{\hookrightarrow} RI_2) \Longrightarrow \nexists RI_3 : (RI_2 \neq RI_3) \wedge (RI_1 \overset{W}{\hookrightarrow} RI_3)$
Every role instance can wrap at most one another role instance. This rule together with (C10) says, that dependencies between role instances based on wrapping rules can create only linear graphs.

(DI4) $(R_1 \overset{C}{\hookrightarrow} R_2) \wedge \exists(RI_1 :!R_1) \Longrightarrow \exists(RI_2 :: R_2) : (RI_1 \hookrightarrow RI_2)$
From dependency between roles $R_1 \overset{C}{\hookrightarrow} R_2$ it follows that every instance $RI_1 :!R_1$ depends on at least one instance $RI_2 :: R_2$ in the same object.

(DI5) $RI_1, RI_2 \in o.V \Longrightarrow Role(RI_1) \notin Role(RI_2)$
There can't be mutually exclusive roles in an object.

(DI6) $(R \in \mathcal{R}_\mathcal{S}) \wedge \exists(RI_1 :!R) \Longrightarrow \forall(RI_2 :: R) : (RI_1 = RI_2) \vee (RI_1 \overset{W^*}{\hookrightarrow} RI_2) \vee (RI_2 \overset{W^*}{\hookrightarrow} RI_1)$
There can be only one instance of one single role in an object (except instances of the same single role that wrap themselves).

(DI7) $(RI_1 \hookrightarrow^* RI_2) \Longrightarrow (RI_2 \not\hookrightarrow^* RI_1)$
No dependency cycles are allowed.

4.8 Consistent object

Object $o \in \mathcal{O}$ is said *consistent* if it matches next conditions:

- $\mid o.V \mid \geq 1$
  an object has at least one role instance — base role instance (RI_B)
- $(RI \in o.V) \Longrightarrow \nexists(o_2 \in \mathcal{O}) : (o \neq o_2) \wedge (RI \in o_2.V)$
  role instance can be included at most in one object (this is assured also by the next rules)
- $\nexists(RI \in o.V) : (o.RI_B \overset{C}{\hookrightarrow} RI)$
  base role instance can't depend on any other role instances (it's an instance of a base role)

auxiliary information (that helps to improve implementation of some algorithms ??ref??)must be in consistent state:

- $\forall RI \in \mathcal{RI} : (RI.RID \neq NoRID) \Longleftrightarrow (RI.OID \neq null)$
- $\forall RI \in \mathcal{RI} : RI \in \mathcal{RI}_\mathcal{B}ound \Longleftrightarrow RI.RID \neq NoRID$
- $\forall RI \in o.V : (RI.acceptedWrappedRole = null) \vee \exists RI_1 \in o.V : (RI \overset{W}{\hookrightarrow} RI_1) \wedge (Role(RI_1) \leq RI.acceptedWrappedRole)$
- for all role instances of an object and dependencies between them are valid rules (DI1) – (DI7)

5 VREcko System

*V*REcko is a system of virtual environment being developed in HCI (Human–Computer Interaction) Laboratory at Faculty of Informatics at Masaryk University. The purpose of this system is to verify applicability of the model of objects with roles on one side and to provide robust environment for research in the area of human–computer interactions on the second side. We will focus on dynamic features that are provided by using an implementation of objects with roles model in this report.

The whole VREcko is written in C++ language. The first implementation of objects with roles model was written in the Java language that has better environment for writing things like that, especially thanks to the reflection API. But because many libraries used in VREcko are written in C++ or C language and because Java has serious performance problems in the area of VRE systems, the whole implementation of objects with role model was rewritten into C++.

5.1 Dynamic Aspects of VREcko

A screenshot in figure 4 gives a basic overview of VREcko's user interface. It's a 3D scene that contains objects that can interact each other and can be also manipulated using external input devices as a pinch gloves, Phantom, keyboard, ...

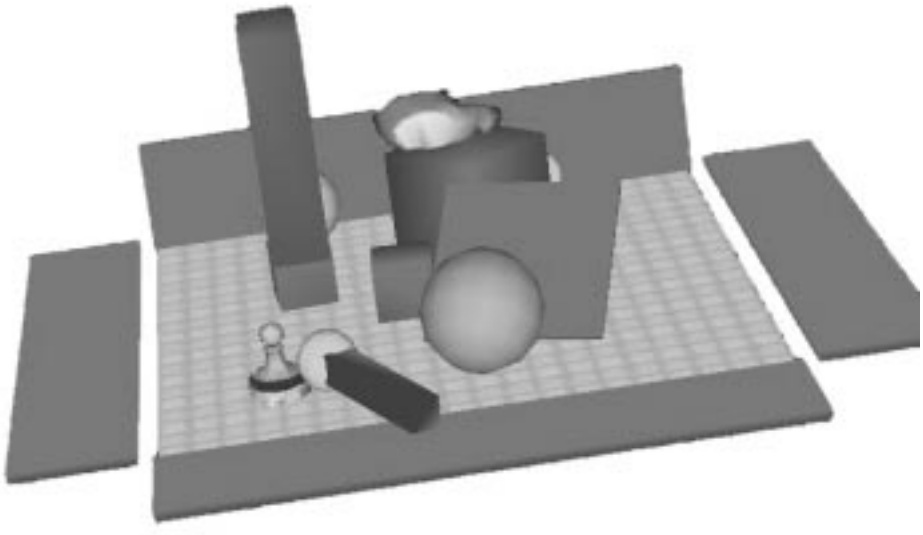

Figure 4: Screenshot

Each "environment object" (in a sense of "visual") in the scene is implemented by one object with roles. The whole scene – it's a container of all environment objects – is realized also as an object with roles. It enables

to dynamically change behavior and state structure of both environment objects and the whole scene.

Typical scenarios of dynamic behaviour follows:

- role instance connecting an environment object with some input device (keyboard, pinch gloves, ...) can be added to an object. As a result some attributes (typically the position) can be modified by this device. The role instance can be removed again of course.
- role instance extending capabilities of scene can be added to the scene object. For example role instance generating clock ticks, role instance implementing physical behaviour of the scene, ...
- role instances connecting to the new role instances of the scene object can be added to the particular environment objects. For example role instances reacting to the clock ticks, role instances defining physical properties of the environment object (weight,...), ...
- role instances reacting on interaction between environment objects. For example a role instance playing some sound when something "touches" its environment object.
- non-properly working role instance can be replaced/wrapped to repair the broken behavior without the need to stop the application and without the need to build the internal structure of role instances in the object from the beginning.

Object with roles approach has some significant advantages over traditional ones. All properties (roles) of environment objects an the scene are handled in one place describing different features/roles/points of view/aspects of the whole entity they describe. In traditional approaches some more complicated framework should be established to provide the same functionality. All dynamic features of individual environment objects as well as of the whole scene should be collected somewhere to work with them but all the associations between them are on the object level and most likely without any control of type consistency. This more free organization should also complicate implementation of features like replacing and wrapping as described in objects with roles model.

6 Conclusion and Future Work

An objects with roles model was presented and an example of an application where it's suitable to use such a model was described. The VREcko system is already partially implemented at present. Since the implementation of objects with roles model in C++ wasn't already prepared in the time of starting the VREcko project, there are two versions of VREcko now. The

first uses ideas of object with roles model but it doesn't use it already this version is being developed by Jan Flasar at HCI Laboratory. The second one uses it but has less features mainly in the sense of supporting external input devices for now.

The future plans are directed to merge this two projects into one and to support more input devices and features. One of the most required features to be implemented is support for distributed environment where some input devices installed in one computer in the net could be adopted by a VREcko system running on different computer in the net. All input devices have to be installed in the same computer as a VREcko system is for now.

7 Acknowledgements

I acknowledge the help that Jan Flasar gives me in introducing problems of VRE systems in general and particularly in discovering features of concrete graphical libraries included in VREcko system.

References

[1] Abadi, M., Cardeli L.: A Theory of Objects. Springer-Verlag, New York 1996.

[2] Albano, A., Bergamini, R., Ghelli, G., Orsini, R.: An Object Data Model with Roles. In Proceedings of the International Conference on Very Large Data Bases, pp. 39–51, 1993

[3] Blashek, G.: Object Oriented Programming with Prototypes. Springer-Verlag, 1994

[4] Bardou, D., Dony, Ch.: Split Objects: a Discipline Use of Delegation within Objects, OOPSLA'96, in ACM– SIGPLAN Notices Vol. 31, pp. 122-137, 1996

[5] Booch, G.: Object Oriented Analysis and Design with Applications. 2nd Ed., Benjamin Cummings, Redwood City, CA, 1994

[6] Büchi, M., Weck, W.: Generic Wrappers, ECOOP 2000, LNCS 1850, pp. 201–225

[7] Clifton, C., Leavens, G.T., Chambers, C., Millstein, T.: MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java, OOPSLA 2000, Minneapolis

[8] Gottlob, G., Schrefl, M., Röck B.: Extending Object-Oriented Systems with Roles. ACM Transactions on Information Systems, Vol. 14, No. 3, pp. 268–296, 1996

[9] Hjálmtýsson, G., Gray, R.: Dynamic C++ Classes A Lightweight mechanism to update code in a running program, USENIX, Annual Technical Conference, 1998

[10] Kniesel, G.: Object do not migrate! Perspectives on Objects with Roles. Report IAI–TR–96–11. Universität Bonn, Institut für Informatik III, April 1996

[11] Kniesel, G.: Dynamic Object–Based Inheritance with Subtyping. Dissertation work. Universität Bonn, Institut für Informatik III, 2000

[12] Kristensen, B.,B.: Object–Oriented Modeling with Roles. Proceedings of the 2nd International Conference on Object–Oriented Information Systems (OOIS'95), Dublin, Ireland, 1995

[13] Kristensen, B.,B., Österbye, K.: Roles: Conceptual Abstraction Theory & Practical Language Issues. Special Issue of Theory and Practise of Object Systems (TAPOS) on Subjectivity in Object–Oriented Systems, 1996

[14] Markovič, L., Sochor, J.: Objects with Changeable Roles. International Symposium on Distributed Objects and Applications — Short Papers, Rome, 2001

[15] Markovič, L., Sochor, J.: Object Model Unifying Wrapping, Replacement and Roled-Objects Techniques. In: Workshop #09 - Unanticipated Software Evolution (USE), 16th European Conference on Object-Oriented Programming (ECOOP 2002) University of Málaga, Spain, June 2002

[16] Pernici, B.: Objects with Roles. ACM. pp. 205–215, 1990

[17] Plášil, F., Bálek, D., Janeček, R.: SOFA/DCUP: Architecture for Component Trading and Dynamic Updating, Proceedings of ICCDS'98, Annapolis, 1998

[18] Richardson, J., Schwarz P.: Aspects: Extending Objects to Support Multiple, Independent Roles. ACM. pp. 298–307, 1991

[19] Siegel, J. et al.: CORBA Fundamentals and Programming. John Willey & Sons, 1996

[20] Ungar, D., Smith, R.: Self: The Power of Simplicity. Proc. of OOPSLA'87, ACM Sigplan Notices, 1987

[21] Wieringa, R., de Jonge, W., Spruit, P.: Roles and dynamic subclasses: a modal logic approach. In ECOOP 94 Proceedings, Springer–Verlag, LNCS 821, 1994

[22] http://research.sun.com/research/self

[23] http://www.omg.org

[24] http://www.fi.muni.cz/∼markovic/USE02.html

Publications in the FI MU Report Series are in general accessible via WWW and anonymous FTP:

`http://www.fi.muni.cz/informatics/reports/`
`ftp ftp.fi.muni.cz (cd pub/reports)`

Copies may be also obtained by contacting:

**Faculty of Informatics
Masaryk University
Botanická 68a
602 00 Brno
Czech Republic**