

## Concepts and Concept-Oriented Programming

**Alexandr Savinov**

1. Department of Computer Science III, University of Bonn
2. Institute of Mathematics and Computer Science, Acad. Sci. Moldova

### Abstract

In the paper we introduce a new programming language construct, called concept, which is defined as a pair of two classes: one reference class and one object class. Instances of the reference class are passed-by-value and are intended to indirectly represent objects. Instances of the object class are passed-by-reference. Each concept has a parent concept specified by means of the concept inclusion relation. This approach where concepts are used instead of classes is referred to as concept-oriented programming (CoP). CoP is intended to generalize object-oriented programming (OOP). Particularly, concepts generalize conventional classes and concept inclusion generalizes class inheritance in OOP. This approach allows the programmer to describe not only objects but also references which are made integral and completely legal part of the program. Program objects at run-time exist within a virtual hierarchical address space and CoP provides means to effectively design such a space for each concrete problem domain.

## 1 INTRODUCTION

Let us assume that a variable stores a reference to a bank account object. This variable is then used to access this account balance using its method, for example: `account.getBalance()`. In OOP we are completely unaware of the reference format and the operations used to access the represented object. The compiler provides primitive (native) references and the programmer has an illusion of *instant access* to objects. However, it is only an illusion and something always happens behind the scenes during object access. In other words, any object access results in a potentially complex sequence of hidden operations. For example, if the bank account is represented by a Java reference then for each access it needs to be resolved into a memory handle by JVM, which in turn needs to be resolved into an address in memory by OS and then this address is processed by CPU and other hardware. If the target object is represented by some kind of remote reference then again one method call results in a sequence of operations executed at different levels of the distributed system organisational structure. Here we see that object representation and access (ORA) functions, even if they are hidden, may account for a great deal of the overall system complexity and hence we need appropriate means for their description.

Above we provided two examples where references have some internal structure and associated functions which are activated implicitly during object access at the level of run-time environment. In these and many other cases the reference format and access functions are provided in the form of a standard library or middleware. One problem with this conventional approach is that frequently we need to define our own *custom references* with their own format and associated access procedures. In this case using universal standard references with built-in functionality might be a limiting factor. For example, creating and deleting many tiny objects is known to be a very inefficient procedure. Dealing with remote objects requires special references encoding information on their location and special access procedures based on some network protocol. Using persistent objects is also based on specific requirements to their identifiers and access rules. In all these cases it would be natural to develop a special memory manager with a dedicated container which takes into account specific features of its objects.

Of course, all these and many other problems can be solved using special features of operating system (like local heap), middleware (like CORBA or RMI), libraries (like Hibernate) or programming patterns (like proxy). Yet the problem remains: the standard approaches can be applied to only standard situations they are designed for while we would like to find a method for an *arbitrary* case. In other words, we would like to find a method for *modelling references* having any format and any behaviour. It is assumed that it is not known what references will be used for and what kind of objects they will represent. The programmer might need many types of references for different purposes which depend on each concrete program. In addition, ORA functions of references and functions of objects have a cross-cutting nature and cannot be easily separated and should be modelled together. Thus the idea is that the functionality that is normally part of hardware, operating system, middleware or a library can be described in the program itself using the same code as for any other functions.

In this paper we propose a solution which is based on extending an object-oriented programming language by introducing new language constructs and mechanisms. The programmer then does not depend on the available environment with its standard ORA mechanisms. Rather, using the proposed approach it is possible to create internal custom containers with a virtual address system where all the objects will live. In particular, the functionality provided by middleware, OS and CPU can be successfully modelled using the programming language. If we see a statement `account.getBalance()` then we cannot assume anything on the bank account location and how it will be accessed. Its reference might contain the bank account number while the object itself might really reside in a database on a remote computer. Getting the account balance might mean using some special protocol involving also operations with some special database developed for this and only this bank. Thus references are interpreted as *virtual addresses* providing at the same time a possibility to bind them to real locations.

In the proposed approach it is assumed that objects are represented and accessed *indirectly* using custom references described in the same language as used for the rest of the program. So references are completely legalized and play the same role as objects. They may have arbitrary structure and arbitrary behaviour. We also assume that developing references is as important as developing objects. Since objects are



---

represented indirectly by custom references, any access like method call or message triggers a sequence of intermediate actions such as reference resolution, security checks, transactional operations etc. However, in contrast to the existing technologies, the proposed approach allows us to model these hidden operations within the program as its integral part. Custom references are used to create a level of indirection and unbinding object identifiers from the available software and hardware environment.

The paper is organized as follows. In section 2 we introduce a new programming construct, called concept, which generalizes classes and underlies other mechanisms described in the paper. In section 3 we describe inclusion relation between concepts, which generalizes conventional class inheritance. Section 4 describes other important mechanisms such as dual methods, polymorphism and life-cycle management. Section 5 gives a short overview of related work and finally in section 6 we make concluding remarks.

## 2 CONCEPT DEFINITION

### Object class and reference class

Since references are supposed to have an arbitrary structure and functions they can be modelled by the same means as objects by using classes, which are called *reference classes*. Thus in CoP there are two kinds of classes: object classes for describing objects and reference classes for describing references. For example, if we would like to identify bank accounts by their numbers then the reference could be defined as follows:

```
reference AccountReference {
    String accNo; // Identifying field
    ... // Other members of the reference class
}
class Account {
    double balance;
    ... // Other members of the object class
}
```

Here we use keyword ‘reference’ instead of ‘class’ to mark this class as a reference class. We might also add other members to this class, say, opening date field and a method for getting balance. The most important thing is that instances of a reference class, called *references*, are passed-by-value only, i.e., they do not have their own references and are intended to represent objects. On the other hand, instances of an object class are passed-by-reference, i.e., there is always some reference which represents this object. An important consequence of introducing reference classes is that objects can be represented by custom references rather than only primitive references. A program is then split into two kinds of things – objects and references – both having their own structure and functions.

Using separately reference classes and object classes is possible but is not very convenient because references and objects do not exist separately but rather are two sides of one and the same thing. To model these two main elements of any program in

their inseparable unity we propose to use a new programming construct, called *concept*, which is a pair consisting of one reference class and one object class. Following this approach, instead of defining separately one account class and one account reference class we need to define one concept which contains the both:

```
concept Account // One name for the pair of classes
  reference { // Reference class of the concept
    String accNo;
    ... // Other members of the reference class
  }
  class { // Object class of the concept
    double balance;
    ... // Other members of the object class
  }
```

Notice that object classes and reference classes cannot be used separately anymore. Instead, we have to use them in pairs using concept names. Such an approach where concepts are used instead of classes is referred to as concept-oriented programming (CoP). Thus concepts in CoP are used where classes are used in OOP when declaring a type of variables, fields, parameters, return values etc. For example, in the following code all types are concepts:

```
Account account = getAccount("Alexandr Savinov");
Person person = account.getOwner();
Address addr = person.getAddress();
```

From this fragment it is actually not possible to determine if it is OOP program or CoP program. If the types used in it (Account, Person, Address) are defined as conventional classes then it is an OOP program. If they are defined as concepts then it is a CoP program. The main difference of CoP from OOP is that variables store custom references in the format defined in the reference class of their concepts. Say, variable `account` stores account number and variable `person` might store passport number and birth date which identify the owner of the account. In contrast, in OOP all these variables would store primitive references provided by the compiler.

An interesting and important property of concepts is that they may provide two definitions for one method in the reference class and the object class, which are referred to as *dual methods*. For example, method `getBalance` can be defined in both the reference class and the object class of concept `Account`. Then the question arises: what definition to use for each method invocation? In order to resolve this ambiguity we use the following principle:

*reference methods of a concept have precedence over its object methods*

In other words, applying a method to a reference means applying the reference method rather than the object method (Fig. 1). For example, the statement `account.getBalance()` will use the definition provided in the reference class of concept `Account`. Once a reference method got control it may decide how to proceed. In particular, it may call the dual object method. The moment where control is passed from the reference to the object is referred to as a *meta-transition*.

This principle means that references intercept any access to the represented object. It is quite natural because we simply are not able to execute the object method



due to the absence of its primitive reference. It is the reference that knows where the object resides and how to access it, particularly, how to call its methods. If one of dual methods is absent (not defined by the programmer) then it is assumed that there is some default implementation. Theoretically it is more convenient to assume that both definitions are always available.

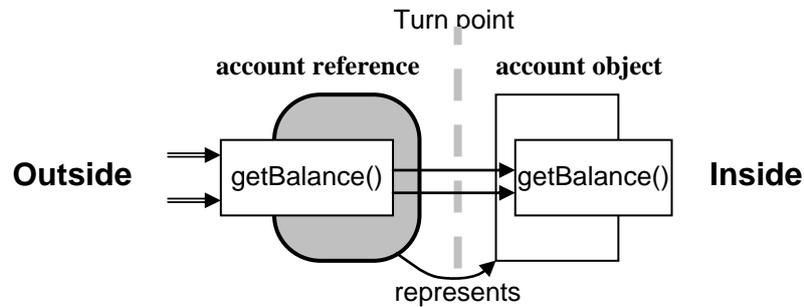


Figure 1. Reference and object.

## Reference resolution

Program objects represented by custom references can be manipulated as if they were normal directly accessible objects in OOP, i.e., in CoP we retain the complete illusion of direct instant access on custom references. However, the question is then how concretely an indirectly represented object can be accessed if its primitive reference is not available? Indeed, if variable `account` stores account number then method `getBalance()` cannot be directly executed because the object primitive reference is not known and theoretically the account may reside anywhere in the world because its reference is a virtual address. Thus any reference needs to be resolved before the represented object can be really accessed.

The task of reference resolution is implemented by a special *continuation method* of the reference class. This method is called automatically for this reference when the represented object is about to be accessed. When this reference is resolved the sequence of access continues. In particular, the method called by the programmer can be applied to the resolved object. Instead of returning the resolved reference, the continuation method marks the point where the reference is resolved and then the compiler knows where the access can be continued. This allows us to perform necessary actions before and after access. It is assumed that the continuation point is where the next (nested) continuation method is called.

Listing 1 shows an example of the continuation method which converts this reference account number into the corresponding primitive reference. It loads the value of the primitive reference from some storage (line 5). Then it proceeds by applying the continuation method to the obtained reference (line 6). And finally it stores the state of the object back in the storage (line 7). If we apply some method to an account object then it will be actually called at line 6 because it is the point where we found the real location of the object. Thus each access is implicitly wrapped into the reference continuation method.

```

01 concept Account
02   reference {
03     String accNo;
04     void continue() {
05       Object o = load(this.accNo);
06       o.continue(); // Proceed
07       save(this.accNo, o);
08     }
09     ...
10   }
11   class {
12     ...
13   }

```

Listing 1. Continuation method.

### 3 CONCEPT INCLUSION

#### Complex references

Concepts do not exist in isolation and each concept has a *parent* concept specified in its definition using an *inclusion relation* formally denoted by ‘<’ (this notation is used in the theory of ordered sets, including formal concept analysis and ‘less than’ sign means the number of elements in extension). If concept *B* is included in *A* then it is written as  $B < A$  (*B* is less than *A*) where *A* is called a parent or super-concept and *B* is called a child or sub-concept. The root of the concept inclusion hierarchy is denoted as *Root*:  $\forall C, C < Root$ . In code, concept inclusion will be specified using keyword ‘in’ followed by the parent concept name, for example:

```

concept A in Root ... // By default
concept B in A ... // B < A
concept C in B ... // C < B

```

Let us assume that a concept within an inclusion hierarchy is used as a type of some variable. Then by default this variable will contain a sequence of references of all concepts starting from the root and ending with this concept. This sequence is referred to as a *complex reference* while each its part is called a *reference segment*. An object represented by one reference segment is referred to as an *object segment*. For example, if concept *C* is included in *B* which is included in *A* then a reference to *C* will consist of three reference segments with the format defined by the reference classes of concepts *A*, *B* and *C* (Fig. 2).

Complex references are analogous of structured or layered addresses in a hierarchical space like postal addresses. Each next reference segment is a local identifier relative to the previous segment. The root concept represents the outer most



space (the global space where all objects live) while each new sub-concept describes an internal subspace within its parent concept. Each object in this hierarchy has a base object, called also its *context*. In code, the current reference segment is denoted by keyword ‘this’. The previous (higher) segment is denoted by keyword ‘super’. And the next (lower) segment is denoted by keyword ‘sub’.

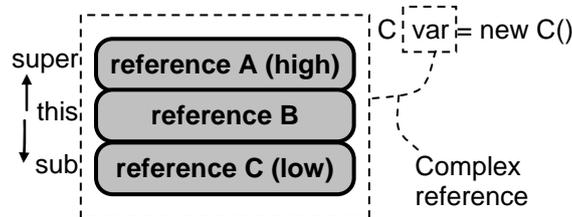


Figure 2. Structure of complex reference.

### Sequence of resolution

If an object is represented by one reference segment then it is accessed by means of this reference continuation method. If the object is represented by a complex reference consisting of many segments then each of them can be individually resolved by its continuation method. An approach where individual reference segments are resolved each time the represented object segment needs to be accessed is referred to as *resolution on demand*. The main problem of this method consists in multiple repeated resolutions of the same reference segments because object parts are normally accessed many times. In particular, each access to the parent object from its child will result in the resolution of one and the same parent reference segment. If there are 100 calls of base methods then there will be 100 resolutions of the base reference segment.

In order to overcome this problem we propose to use the mechanism called *resolution in advance*. This approach is based on resolving reference segments *before* real access happens. The resolution sequence starts from the first (high) segment and then proceeds to the next segments ending with the last (low) segment. The result of each resolution is stored in a special data structure, called *context stack* (Fig. 3). It grows as each next segment is resolved. Initially it is empty. When the first segment A is resolved it contains a direct (primitive) reference to the first context (the first object segment). The result of resolving the second segment B is pushed on top of the context stack, which now contains two direct references and so on till the last segment. Finally, the number of elements on it is equal to the number of segments in the complex reference being resolved (3 in this example). The top of the context stack is a direct reference to the target object of concept C.

The most important property of this mechanism is that parent objects are directly accessible from their child objects and need not to be resolved. So each occurrence of keyword ‘super’ in code means *direct* access using the primitive reference from the context stack. Since normally concept functionality is based on using parent concepts, this mechanism leads to significant performance increase because it guarantees that any reference segment is resolved only once for each use of the complex reference.

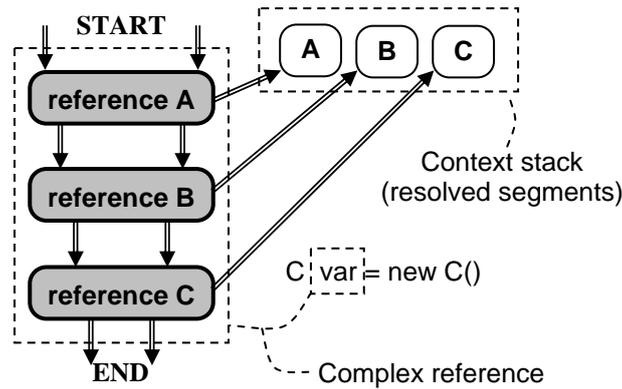


Figure 3. Reference resolution and context stack.

## 4 OTHER MECHANISMS

### Dual methods

Let us now assume that a reference has many segments each implementing one method. The question is what definition will be used if we call this method for this complex reference? For example, if concepts A, B and C implement method `doSomething` and then we call this method for a reference to an object of concept C then what method has to be really executed? Earlier we assumed that reference methods will intercept object methods but in this case there are many reference methods defined in different segments. In order to resolve this ambiguity we use the following principle (Fig. 4, left):

*parent reference methods have precedence over (override) child reference methods*

In other words, parent reference methods of higher segments intercept all accesses to the child reference methods of lower segments. In our example, `doSomething` of reference A will be called first and only after that it is possible to call `doSomething` of reference B and C. Child reference methods are called using keyword 'sub'.

An important application of this principle is the mechanism of method overriding. However, the direction of such overriding is opposite to the conventional one (as used in OOP), which means that base reference methods override child reference methods. If we define a method of the reference class then it can be overridden in the base reference class. This rule is quite natural and simply follows from the necessity to intercept all incoming requests by external space before it reaches any internal space. Thus parent reference methods protect child reference method from direct use from outside.

For object methods this principle has the conventional direction as used in OOP (Fig. 4, right):

*child object methods have precedence over (override) parent object methods*

This means that if we call some object method which is implemented by all object classes in the hierarchy then the compiler will use the definition provided by this object class (we say, that this method overrides its parent methods). After that this



method can continue by calling its parent methods using keyword 'super'. Thus child object methods protect parent object methods from direct use from inside.

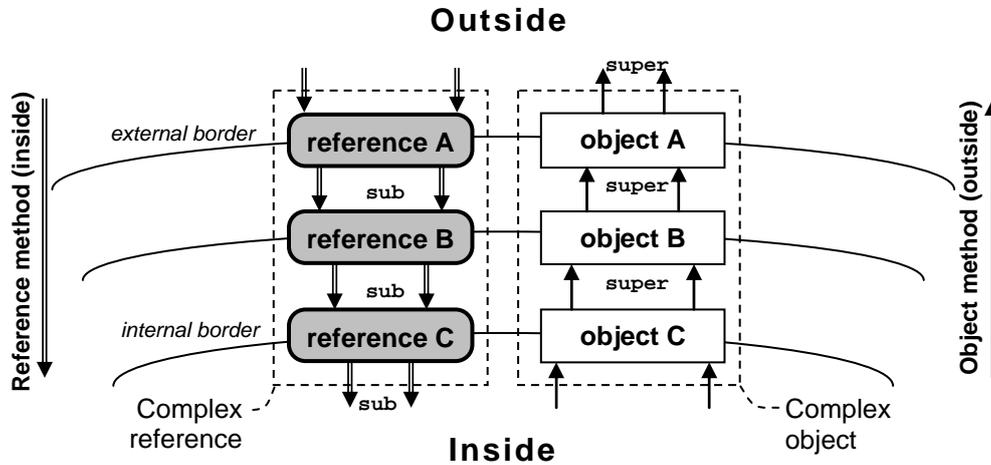


Figure 4. Dual methods.

## Inheritance and polymorphism

In CoP reference-object pairs exist in a hierarchy at run-time which is modelled by the concept inclusion relation. In the run-time hierarchy an object has one parent object (context) and many child objects (extensions). The context is shared among many extensions and is accessed via 'super' keyword. Extensions can be accessed via their local references while the current extension is accessed via keyword 'sub'. In contrast to OOP where all object segments exist together side-by-side and base object is not shared, in CoP object segments exist separately and have their own references. For example, if `Button` inherits `Panel` then in OOP each button object consists of two parts which are created and exist together having one common primitive reference. In CoP these two parts will be separate objects with their individual references. Moreover one panel segment (context) may have many button segments (so one panel includes many buttons). Thus CoP changes the semantics of inheritance. Child objects are made more specific by the fact of existing in the context of their parent (along with other child objects) rather than identifying themselves with the context. In other words, in OOP a button 'is a' panel while in CoP a button 'is in' panel. Interestingly, this approach generalizes that used in OOP because it is reduced to OOP in the case child concepts do not define their own reference classes.

In OOP polymorphism is based on overriding base methods by child class methods. Then the method executed depends on the real object class. The selection (dispatching) is performed via some special built-in mechanism like a table of function pointers (vtbl). In any case only one method will be executed. In CoP it is not so and a method invocation is actually a *sequence* of steps executed in different concepts. This sequence starts from the base reference and then proceeds to the child references. Each intermediate method makes its own contribution to the processing of the current request and then delegates it further to the child concepts. Depending on the real object type we will get different processing chains.

For example (Listing 2), let us assume that concept `SavingsAccount` is included in concept `Account` (so one account may have many savings accounts as well as other types of sub-accounts). Both concepts implement method `getBalance`. The method of `Account` checks if the child object really exists (line 5) and then either returns its own balance (line 5) or the balance of the child account (line 6). In code we can declare a variable as having base type `Account` and then the balance returned by `getBalance` method depends on the real object type. If the object is of concept `Account` (line 24) then we get one behaviour. If it is of concept `SavingsAccount` (line 26) then we get another behaviour.

Notice that `SavingsAccount` assumes that there can be also internal objects (line 15), i.e., it is implemented in the concept-oriented manner where methods are intermediate processing elements getting a request from somewhere and then dispatching them to somewhere for further processing. The polymorphic behaviour is defined by the programmer who writes intermediate methods each contributing to the overall processing. We can include a new child concept in `SavingsAccount` later for example to describe some concrete savings account type and it will be incorporated in the whole access chain by getting requests from its parent concept.

```

01 concept Account
02   reference {
03     String accNo;
04     double getBalance() {
05       if(sub == null) return balance;
06       else return sub.getBalance();
07     }
08   }
09   class { double balance = 10.0; }
10
11 concept SavingsAccount
12   reference {
13     String subAccNo;
14     double getBalance() {
15       if(sub == null) return balance;
16       else return sub.getBalance();
17     }
18   }
19   class { double balance = 20.0; }
20
21 Account account;
22 double balance;
23 account = findAccount(); // Real type is Account
24 balance = account.getBalance(); // = 10.0
25 account = findSavingsAccount(); // Type SavingsAccount
26 balance = account.getBalance(); // = 20.0

```



---

Listing 2. Dynamic polymorphism.

From this example we see that one and the same method applied to a variable of one type may produce different behaviour depending on the real type of reference stored in it. In CoP, such a method call is a sequence of actions associated with the reference segments. Each intermediate reference and object may contribute to the processing of the access request. In OOP, polymorphism is much simpler and is reduced to choosing the method defined in the real object class which completely overrides its base methods. Thus the method executed by default in OOP is only the last step in a sequence of actions executed in CoP. Interestingly, CoP does not guarantee that the last method corresponding to the real object type will be reached while in OOP it is always so. The base reference methods override child methods and may finish processing at any moment without continuation. For example, the base method may raise an exception because of security constraints or insufficient resources. Such an approach is more flexible because request processing is distributed among all constituents at different levels rather than concentrating all the functionality in one class.

### Life-cycle management

In OOP object creation involves automatic allocation of the necessary resources represented by a primitive reference followed by the new object initialization implemented in the class constructor. The former (reference creation) is a hidden procedure provided by the compiler using the available run-time environment. The latter (initialization) is controlled by the programmer via class constructor. Object deletion also involves two procedures: one for cleaning up the object in the class destructor and the other for freeing the resources (not controlled by the programmer).

In CoP *both* these tasks are implemented in concepts, i.e., concept is responsible for both the reference allocation/de-allocation and object initialization/de-initialization. Thus the programmer gets full control over these special procedures using dual methods. To create a concept instance a special *creation method* is used while deletion is implemented in a special *deletion method*. Just as other methods, creation and deletion are dual, i.e., they can be implemented in both the reference class and the object class. The object creation/deletion method is an analogue of the conventional constructor/destructor in OOP. Their role consists in initializing a new object just after its reference creation and cleaning it up just before its reference deletion. The reference creation/deletion methods are responsible for initializing/cleaning up a reference. Their role consists in allocating or freeing the associated resources such as memory.

The creation/deletion methods have the same sequence of access as other methods. For example, if we need to create a new savings account then we declare a variable and then call the creation method:

```
SavingsAccount account.create();
```

Since this object is represented by two reference segments (Account and SavingsAccount), the creation starts from the first one. An example of its implementation is shown in Listing 3. It generates a unique identifier for the new base

account (line 6) and then proceeds by allocating system resources for this object (line 7), i.e., here we really create a new object with its primitive reference. Line 7 is also the point where the object creation method (constructor) will be called (line 15). Once the account object has been created we can allow possible children to contribute to this process (line 8). In the case this object is a `SavingsAccount`, its extension will be created here. Finally, when children are created, we store the association between the new account number and the primitive reference in the map (line 9). This information is then used in the continuation method for resolving references when this account is being accessed. The creation method of `SavingsAccount` can be implemented in the same way. The only difference is that it will be executed in the context of the just created base account. In particular, the mapping from sub-account numbers to primitive references will be stored in the base object (line 14) rather than as a static variable (line 1).

```

01 static Map map = new Map();
02 concept Account
03     reference {
04         String accNo;
05         void create() {
06             this.accNo = getUniqueNo();
07             Object o.create();
08             if(sub != null) sub.create();
09             map.add(accNo, o);
10         }
11     }
12     class {
13         double balance;
14         Map map = new Map();
15         void create() { balance = 0; }
16     }

```

Listing 3. Object creation.

One important feature of this sequence of creation is that the base object needs not to be always created. Instead, we can select an already existing base object which is shared among many child objects. For example, a new savings account could be created for a person who already possesses an existing base account. Another example is where parent concept implements a container and then creating an object means selecting an existing container. A container will be created only if all the existing containers are full. Otherwise the creation procedure will try to find an existing container for a new object. Even when we have to really create a new object we do not need to call system procedures. This approach allows us to keep a pool of primitive references and other system resources in a base concept rather than allocate/free them for each object. So the programmer has full control over the creation procedure. The deletion is implemented in the same way except that the



---

operations are normally inverted (i.e., we proceed to children and then consecutively delete parents on the way back).

## 5 RELATED WORK

ORA functionality exists at all levels of system organization. For example, memory access performed by processor is not a single action but takes many internal micro-cycles. Operating system brings a new level of indirection by providing its own memory handles for identifying and accessing objects in the global or local heap. Middleware such as CORBA or RMI [Mon06] propose additional ORA mechanisms targeted at specific tasks like remote access. Language run-time environment may implement its own object container with some specific logic of indirection, e.g., Java or C#. In addition, such an environment may implement special facilities targeted at access indirection like reflection or dynamic proxies which allow for transparent interaction of method calls and dynamic implementation of arbitrary interfaces [Blo00]. A general approach that can be used to change the behaviour of language constructs, particularly, ORA functions, is metaobject protocol [Kic91, Kic93].

The described approach is aimed at providing *language* means for describing indirect ORA and hence it is important to compare it with existing language-based approaches to this problem. The simplest method of language-based indirection consists in using some discipline or pattern. One of the most wide-spread patterns is that of proxy, which is a class simulating the target class. Another pattern which can be used to indirect access is chain of responsibility. Like any pattern, these methods are rather specific and need high level of manual support because the compiler is unaware of their semantics and cannot help in their maintenance. An interesting approach to reference modelling consists in using smart pointers [Str91]. Yet it is not dedicated method but rather an adaptation of the universal mechanism of templates to the problem of reference modelling.

A much more fundamental approach is provided within aspect-oriented programming (AOP) [Kic97]. Here any object access may trigger quite complex intermediate actions, which are injected in the necessary points of code implicitly by *aspects*. However, this approach does not provide any means for modelling custom references and indirect access. In addition, the direction of module dependence is opposite to that used in CoP (see [Sav05, Sav07] for more details). Another related approach is based on using *mixins* (abstract subclasses) [Bra90, Sma98]. In particular, it is similar to CoP (and AOP) in its ability to wrap some target code into another method (using ‘around’ keyword).

The mechanism of dual methods in CoP is similar to *super/inner* methods of classes [Gol04] which is implemented in the Beta programming language. In particular, the inner methods are designed in such a way that they implement the same sequence of access as that in reference methods. However, the mechanism of *super/inner* methods is implemented as an addition to normal classes. Hence it can be viewed as an enhancement to OOP aimed at providing means for object protection from outside. In CoP, this behaviour is implemented using a completely different approach, namely, by means of concepts.

The proposed approach relates also to so called *context-oriented* methods which are aimed at bringing context dependence into programming [Con05]. These methods introduce language constructs and mechanisms which allow the programmer to put objects in a context changing their behaviour at run-time. For example, in the ContextL programming language it is done by means of the keyword 'in-layer' while in CoP we use 'in' which generalizes inheritance and 'super' to access the context. The context-orientation also relates to a technology known as dependency injection.

There exist also other mechanisms that can be used to model indirection such as annotations in attribute-oriented programming, language-oriented programming and domain languages, multi-dimensional separation of concerns, subject-oriented programming. Yet some approaches to programming having the same name are actually based on very different notions and do not relate to our work [Voi92, Mcc99].

Earlier we have already described an approach to programming based on using concepts to which we refer as CoP-I [Sav05]. The approach described in this paper (see also [Sav07]) is referred to as CoP-II. Although they use the same programming construct, the difference is significant. Namely, we changed the role of concept constituents: in CoP-II (in this paper) we assume that references represent objects of *this* concept while in CoP-I we assume that references represent objects of *child* concepts.

## 6 CONCLUSIONS

In the paper we described a new approach to programming based on using concepts instead of classes and inclusion instead of inheritance. This approach is backward compatible with OOP. One of its main achievements is that references are completely legalized and made first-class citizens of the program along with objects. The functions encapsulated in objects and references are orthogonal and this can be viewed as a continuation of a very general and deep principle of Separation of Concerns formulated by Dijkstra [Dij76]. In other words, any program consists of two types of functionality which needs to be separated in a principled manner.

By bringing in the concept of concept in programming languages, we can effectively model references and intermediate functions executed implicitly during object access. However, this also changes the way how a system is being developed. An action in CoP is not a single operation but rather a sequence of processing steps executed by intermediate objects on the way to the target. In many cases this intermediate hidden functionality accounts for most of the system complexity. This leads to a paradigm shift because we cannot follow the instantaneous action principle anymore. Instead, in CoP any action is indirect and needs some environment to propagate. Modelling such underlying environments becomes one of the main design goals (rather than modelling classes of objects in OOP). Methods in CoP are not end-points for processing but play an intermediate role: they get control from somewhere and pass it further to somewhere. They are triggered automatically whenever a request intersects the border. So the most interesting events happen under the hood without any explicit action from the programmer and CoP provides adequate means for modelling such a view of the system.



---

In this new approach objects are thought of as existing in a *virtual* space with its own structured address system. Actually this space may play more important role than the functionality of the internal objects. It is similar to a living cell border or state border which implements common functions important for any internal element. An object can be accessed only by intersecting the borders of the spaces where it exists and hence the space always intervenes in processing of any request. Designing an appropriate virtual space for a system is therefore of high importance and CoP provides effective means for doing it at the level of the programming language.

CoP is being developed together with a new approach to data modelling, called the concept-oriented model [Sav06, Sav07a]. In particular, CoP provides a mechanism for data physical representation and access. In future we are going to closer integrate these two approaches. Another goal consists in designing an experimental programming language based on the proposed concept-oriented principles.

## REFERENCES

- [Blo00] Blosser, J. Explore the Dynamic Proxy API, *Java World*, November 2000. <http://www.javaworld.com/javaworld/jw-11-2000/jw-1110-proxy.html>
- [Bra90] Bracha, G., Cook, W. Mixin-based inheritance. *Proc. OOPSLA/ECOOP'90*, ACM SIGPLAN Notices, 25(10), 303-311, 1990.
- [Con05] Constanza, P., Hirschfeld, R. Language constructs for context-oriented programming. In *Dynamic Languages Symposium, co-located with OOPSLA'05*, 2005.
- [Dij76] Dijkstra, E.W. *A Discipline of Programming*. Prentice Hall, 1976.
- [Gol04] Goldberg, D.S., Findler, R.B., Flatt, M. Super and inner: together at last! *Proc. OOPSLA'04*, 116-129, 2004.
- [Kic91] Kiczales, G., Rivieres, J., Bobrow, D.G. *The Art of the Metaobject Protocol*. MIT Press, 1991.
- [Kic93] Kiczales, G., Ashley, J.M., Rodriguez, L., Vahdat, A., Bobrow, D.G. Metaobject protocols: Why we want them and what else they can do. In: Paepcke, A. (ed.) *Object-Oriented Programming: The CLOS Perspective*, 101-118, MIT Press, 1993.
- [Kic97] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. and Irwin, J. Aspect-Oriented Programming, *Proc. ECOOP'97*, LNCS 1241, 220-242, 1997.
- [McC99] McConnell, B. Concept-oriented programming. *Dr. Dobb's Journal*, 24(6), 90-96, 1999.
- [Mon06] Monson-Haefel, R. *Enterprise JavaBeans*, O'Reilly, 2006.
- [Sav05] Savinov, A. Concept as a Generalization of Class and Principles of the Concept-Oriented Programming. *Computer Science Journal of Moldova*, 13(3), 292-335, 2005.

- [Sav06] Savinov, A. Grouping and Aggregation in the Concept-Oriented Data Model. *Proc. SAC'06*, 482-486, 2006.
- [Sav07] Savinov, A. *Concepts and their Use for Modelling Objects and References in Programming Languages*. Technical Report, Institute of Mathematics and Computer Science, Moldavian Academy of Sciences, 43pp., 2007.
- [Sav07a] Savinov, A. Concept-Oriented Model. *Encyclopedia of Database Technologies and Applications*, Editors: Rivero, L.C., Doorn, J.H., Ferraggine, V.E. (accepted).
- [Sma98] Smaragdakis, Y., Batory, D. Implementing layered designs with mixin-layers. *Proc. ECOOP'98*, 550-570, 1998.
- [Sop] Subject-Oriented Programming, <http://www.research.ibm.com/sop/>
- [Str91] Stroustrup B. *The C++ Programming Language*, 2nd Edition, Addison Wesley, 1991.
- [Voi92] Voinov A.V. Netlog - A Concept Oriented Logic Programming Language, *Proc. LPAR'92*, 357-368, 1992.

## About the author



**Alexandr Savinov** (Ph.D.) is currently a researcher at the [Department of Computer Science III](#), University of Bonn, Germany. He received his MS degree from the [Moscow Institute of Physics and Technology](#) in 1989 and his PhD from the [Technical University of Moldova](#) in 1993. His primary research interests include programming, data modelling and knowledge management methodologies with applications to distributed systems, peer-to-peer technologies, grid computing, web services, semantic web and other areas. He is an author of a new general system theory, called the concept-oriented paradigm (see <http://conceptoriented.com>). Previous research interests include expert systems, fuzzy logic and data mining. See also: <http://conceptoriented.com/savinov>