

Informal Introduction into the Concept-Oriented Programming

Alexandr Savinov

<http://conceptoriented.org/>

First created: 19.11.2007

Last update: 19.11.2007

Abstract. This paper describes a new approach to programming, called the concept-oriented programming (COP). It is based on using a new programming construct, called concept, which generalizes conventional classes. Concepts describe behaviour of both objects and references. Hence references are completely legalized and made first-class citizens with the same rights as objects. Using concepts the programmer can easily describe custom virtual address spaces where objects will exist. The hierarchical structure of such a space is modelled by means of concept inclusion relation which generalizes class inheritance. In COP, a great deal or even most of functions are executed implicitly during object access rather than in target objects themselves. These functions have cross-cutting nature but can be effectively separated using COP.

1 Introduction

Objects in OOP are represented by references which are provided by the compiler. Such references are referred to as *primitive* because we are almost completely unaware of their structure and behaviour. The only thing that is guaranteed is that such a primitive reference knows how to access the represented object. One property of such an approach is that all objects are identified in one and the same way using primitive references and hence they exist in one big flat space. Another property is that these references have a limited scope restricted by one program. We cannot pass or store primitive references outside this scope. Although there are two kinds of things – references and objects – OOP deals with only objects which are modelled by classes.

However, in real life both kinds of things are equally important and widely used. Indeed, we normally manipulate very different identifiers like postal addresses, account numbers, person names, car numbers, passports, computer IP addresses and so on. Thus we have a kind of paradox: in reality there is a wide variety of addresses in any problem domain while in programming there is only one built-in type of primitive references. In this sense the main goal of the new approach to programming described in this paper consists in making this picture symmetric by providing to references the same status of first class citizens as objects have. Thus the *duality* of the real world consisting of arbitrary references and arbitrary objects has to be reflected in programming languages which should support both types of things. To describe references and objects a new programming construct is used, called *concept*, and hence the whole approach is called the concept-oriented programming (COP).

In such a symmetric approach references can be modelled in approximately the same way as objects are modelled in OOP. In particular, both have structure and behaviour. The structure of references describes how objects are identified while behaviour describes functionality which is executed during object access. And here we come to an interesting conclusion: program functionality is concentrated not only in objects but also in object identifiers (references). As a result we can as usual call object methods but some intermediate functionality will be executed behind the scenes as they are being accessed. For example, if a bank account object is represented by its real account number then this data in custom format will be used to represent the object instead of a primitive reference. And when we apply a method to this reference then some intermediate procedure will be executed to find the location of the target object.

One of the most interesting general assumptions of this approach is that the hidden functionality associated with references accounts for a great deal or even most of the overall program complexity. In other words, it is frequently more important what happens behind the scenes during access than what is executed in the target object. In real life the situation is the same. For example, our goal might consist in getting a signature of some official on a document. However, the intermediate actions might involve travelling to his office using different public transports, then authenticating at the entrance, then finding his office in the building, then preparing some additional documents. The final step in

this story with signing the document takes a couple of minutes but it is precisely what we wanted to do while the intermediate actions could well be different. Say, we might send a letter with the document instead of travelling themselves or we might collect several documents and send them together using some faster transport. Here it is important that the final action has to be independent of the intermediate access procedures, i.e., these two concerns have to be separated. Accordingly, it is highly important to have support for such a design in programming languages where the programmer can create custom levels of indirections responsible for object representation and access.

Summary:

- References and objects have equal rights and both possess structure and behaviour.
- Object access is always indirect and is performed via some intermediate layers.
- Intermediate functions executed behind the scenes during access account for a great deal or even most of the program complexity.

Further reading:

- Papers:
 - [1], Section 7 (pages 35-39): Principles of the Concept-Oriented Paradigm ([PDF](#))
 - [2], Section 10 (pages 40-42): Concept-Oriented Paradigm ([PDF](#))
 - [3], Section 1.3 (pages 296-298): Design Goals ([PDF](#))
- Blog posts:
 - [Main principles of the concept-oriented programming](#)

2 Hierarchical Addresses

Let us assume that we need to send a message to someone or something using a normal postal address consisting of country name, city, street and house number. In this message we could ask for help in understanding what concept-oriented programming is. We write this message on paper, put it in envelope, write the target address and throw it in the nearest post box. It is quite natural that we do not care how this message will be processed in intermediate post offices and what transport will be chosen for delivery. For us it is only necessary to specify the target address and the message so in a programming language it could be written as follows: `target.pleaseHelp()`. The target is a variable which contains the real address for example in the following form: `target = <"Germany", "Bonn", "Concept-oriented street", 99>`. Although sending a letter looks very simple, its delivery can take quite significant time and other resources. The typical processing sequence follows the address structure (Fig. 1). First of all, the letter is sent to the target country because only within this country other parts of the address can be meaningfully interpreted. When the target country is reached, the letter is sent to the target city. From within this city the street and house number can be found. And finally the letter is handed over to the addressee.

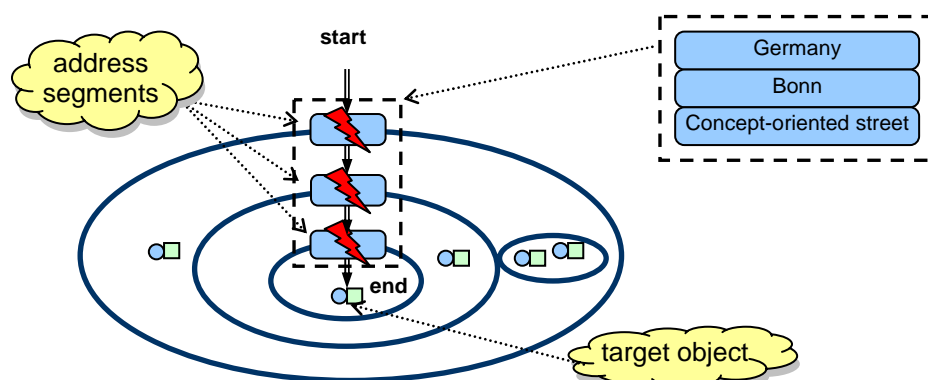


Fig. 1. Hierarchical address system and intermediate implicit processing during access.

One conclusion from analysing this procedure is that instant access is an abstraction which does not exist in nature and hence there is always some intermediate environment which is responsible for

message delivery. What is more, this environment and its functionality can account for a great deal or even most of the overall system complexity so modelling it is as important as modelling target object themselves. Thus it is important to understand that there are two types of functionality. One is associated with objects while the other is associated with addresses. For example, the procedure that interprets an address could find the location of the object represented by this address, check if we are allowed to access this object and prepare this target object for access. Another important point is that we do not want to know the peculiarities of the intermediate processing procedures when they are used to deliver messages. Instead, we want to simply specify an address and a message while all the necessary functions have to be determined automatically and performed behind the scenes. Such an approach means working in a *virtual address space* hiding the reality behind it. For example, the postal address could represent a computer in a network or any other object in a virtual environment. However, all these details are not important because the only thing we need consists in having a guarantee that our message will be somehow delivered to the target address. And one property of this procedure is that one and the same logic is used for any kind of target object. Moreover, it is defined and implemented before any target object can really exist so it cannot be associated with any object type. For example, the target address could represent a person or it could be a bank. Accordingly, we could send very different messages like invitation to dinner or money transfer order. The message delivery postal system is used for any target object and is unaware of their real abilities. So the intermediate object access functions have a cross-cutting nature and cannot be described as part of a target object type.

One of the main ideas of COP is that the mechanisms analogous to postal system are very appropriate for computer programming. Indeed, contemporary software systems manipulate very different objects with very different locations. In this case the functionality being executed during object access can account for most of the overall complexity and hence we need adequate means for its modelling. It would be very attractive to represent all objects as living in a virtual address space which is developed independent of their real functionality. After that these objects could be used as usual by specifying an address and an access request (message). For example, let us assume that the program is intended to work with bank accounts. Each account is identified by an account number in the context of some concrete bank. In order to access an account object it is necessary to resolve the bank identifier and then to resolve the account number in the context of this bank. When the account is completely resolved it is possible to access the target object. For example, if we want to get the current balance of account number "87654321" in "MyBank" then we write it as follows:

```
Account account = <"MyBank", "87654321">;  
double balance = account.getBalance();
```

Notice that here account is represented by some virtual address which has nothing to do with the real object location in memory, on disk or anywhere on computer or network. We store real bank name and real account number precisely as they are used in the problem domain and then use this object address for access. At the same time we do not care how concretely the balance will be obtained just because it is not our business: our task consists in getting balance and that is all. We do not want to deal with security issues, persistence issues, transaction issues or whatever that is different from getting account balance.

If we represent objects by their virtual addresses then they are also passed and stored in this form. For example, if want to find the owner of some bank account then we might call the procedure with this account as a parameter:

```
person = findOwner(account);
```

However, here the parameter contains bank identifier and account number rather than direct object reference. Moreover, the returned value is also not a primitive reference but rather a virtual address of the person such as person id or name and birth day combination.

Such an approach is normally used in real life because virtual addresses are much more stable and reliable than real identifiers. Indeed, we store postal addresses rather than geographical coordinates and we store bank account numbers rather than the real locations of the corresponding records in bank offices. Using virtual addresses allows us to change or update the intermediate environment at any time without changing all the rest of the system where these addresses are used. For example, post office can get new equipment or can move to some other address but we still are able to send letters. The same is very useful in programming by making our code as much independent as possible from object access procedures. For example, if our code manipulates accounts represented by their numbers then the underlying access system can be changed at any time provided that it guarantees that account

objects can be correctly resolved. An example of such a virtual address space is a well known DNS system that introduces computer names instead of their real IP addresses.

Summary:

- Objects cannot interact instantly and any access requires some intermediate environment that is responsible for object representation and access.
- Object access is a rather complex process consisting of many intermediate steps and involving quite complex functions.
- A great deal or even most of functionality has nothing to do with objects but rather is associated with references.
- Virtual address spaces with their functions are developed before and independently of the target objects.
- Addresses have a hierarchical structure.

Further reading:

- Papers:
 - [1], Section 2 (pages 3-8): Representation and Access ([PDF](#))
 - [2], Section 2 (pages 3-10): Object Representation and Access ([PDF](#))

3 References and Objects

In the previous section we demonstrated that references are highly important elements for any system because just as objects they possess structure and behaviour. However, in object-oriented programming, we model only objects while all references have one and the same primitive type provided by the compiler. So the programmer is not able to influence how objects are represented and accessed neither by changing the primitive references nor by extending them. Concept-oriented programming fills this gap and proposes to model two types of elements any program consists of: objects and references. This means that any concept-oriented program consists of and manipulates custom objects and custom references both having arbitrary structure and behaviour.

However, modelling separately objects and references is not very fruitful. In fact, there exist numerous additional mechanisms and patterns that allow us to model references using object-oriented facilities like smart pointers or proxies. (Indeed, a system cannot function without representation and access mechanism so they must be somehow modelled.) What is really new in the concept-oriented paradigm (not only programming but also in data modelling and design) is that references and objects are considered two sides of one and the same thing. In other words, there are no such things as an isolated reference and an isolated object – there exist only a pair consisting of one reference and one object. Thus any concept-oriented system consists of and manipulates object-reference pairs. The space of all elements is then broken into two parts: the entity world consisting of objects and the identity world consisting of references (Fig. 2). For example, one city has two sides or flavours: city identifiers (reference) and city itself (object). A bank account also consists of two parts: account reference is used to identify account object where both contain some fields and have some methods.

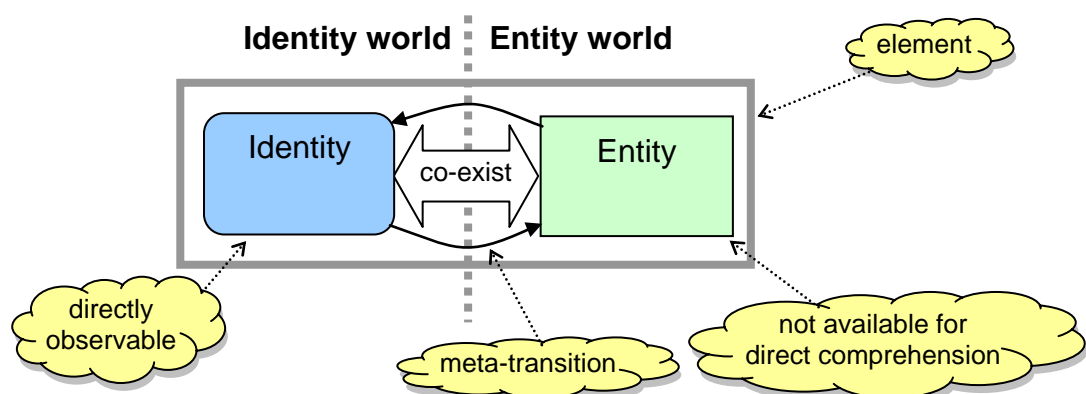


Fig. 2. Element is a pair consisting of two constituents: identity (reference) and entity (object).

This separation is of crucial importance for understanding the concept-oriented paradigm and below we provide a list of properties distinguishing references and objects.

[By-value/by-reference] References are passed and stored by-value, i.e., by copying their content, while objects are passed and stored by-reference, i.e., by using its counter-part from the identity world. This means that references do not have their own references while objects are always represented and accessed indirectly.

[Direct/Indirect] Reference represents part of reality which is directly comprehensible while object is a thing-in-itself which is not observable in its original form and hence is radically unknowable. So the only way to get information about an object or to interact with it consists in using its reference which stays between us and objective reality. In a program reference is the only element that knows how to access the object so any access passes through some reference.

[Transient/Persistent] References exist only in a transient form while objects exist persistently. This means that any reference is available only for us ourselves (in our own scope) while other subjects have their own copies of references. In particular, if we change a reference then it is not visible to anyone except for ourselves. In contrast, objects are exposed in one and the same form to all subjects. If we change an object then this change is visible for all. If we do not work with an object then it still expected to exist.

[Location] References do not have any location just because they cannot be referenced. In contrast, objects have a constant location in space which is represented by their references. For example, postal address itself does not have a location while the house object has a constant location.

[Fact of existence] Reference can be thought of as a manifestation that some object really exists. In other words, an object exists if its reference is available (even if the object is not created yet) and, vice versa, if a reference is lost then the object is considered non-existing (even there is some reality behind it). There is no other way to check if an object exists except for getting somehow its reference.

[Meta-transition] References and objects exist in different worlds and moving between them is referred to as *meta-transition*. The question then is how references perform meta-transition and provide access to objects? The general answer is that we do not know so it is a kind of magic capability of references to cross the border between two worlds. More specifically, references normally reduce this task to simpler references assuming that they can do it. However theoretically this process never ends and normally we assume that some type of reference has a built-in capability to perform meta-transition.

Above we have postulated that references and objects are two parts of one thing and hence a program is a number of reference-object pairs. However, manipulating such a big set of reference-object pairs is not very convenient because they may have very different nature. So it would be desirable to establish a structure where elements could be placed in local spaces. To establish such a structure we make the next fundamental assumption that all reference-object pairs exist within a tree-like hierarchy where any element has a parent element. This hierarchy is specified using *inclusion relation* among elements, i.e., any element is included in its parent element (Fig. 3). For example, "Bonn" is included in "Germany" and accounts "456789" and "987654" are included in element "MyBank".

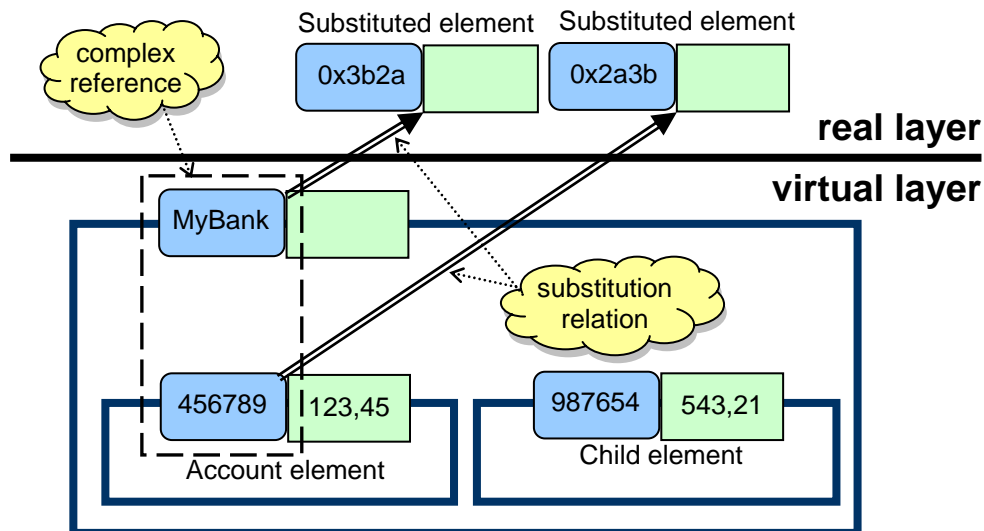


Fig. 3. Inclusion and substitution of elements.

Below we enumerate the most important properties of inclusion relation.

[Root] It is very important to understand that *any* element has a parent without exceptions however we normally stop at some level by choosing one *root* element. The root is considered the outer most scope or global space where all other elements exist. Choosing an appropriate root element depends on the problem domain and the task being solved. For example, in one case we might restrict the scope by one city while in some other application we might consider our galaxy as the root.

[Inherent] Inclusion relation is not implemented by elements themselves but rather is their inherent property. This means that elements can exist only within some other element which provides an environment for them. In particular, elements do not store a reference to their parent just as a cell in memory does not store its address – it is its inherent property to have some address.

[Permanent] Inclusion is permanent and cannot be changed during life-time of elements. Elements are created within some parent element and then exist there until they are deleted.

Any concept-oriented program is a hierarchy of reference-object pairs. Any object within this hierarchy is represented by its reference which is defined within its parent. The parent in turn is represented by its own reference and so on up to the root. Each local reference is referred to as *reference segment* while a reference consisting of several segments is referred to as *complex reference*. For example, <"MyBank" , "456789"> and <"MyBank" , "987654"> are two complex references consisting of two segments. Notice that an object then also consists of several segments and such an object is referred to as *complex object*. An important difference from OOP is that parent object segments can be shared among child segments. For example, two account segments exist within one bank (base) segment.

References may have any structure and behaviour defined by the programmer and then the question is how this data can be used to associate it with the real object. For example, if an account reference contains the real account number "123456" then it is obviously a convention used in the problem domain. Although it is very convenient to use such conventions we still need to find a way from this piece of data to the real object. As a fundamental solution we assume that any element substitutes for some other element (Fig. 3). Thus there exist two relations used to arrange elements: inclusion and *substitution relation*. Substitution is precisely what is responsible for virtualization. So we assume that elements in a virtual layer substitute element in real layer. For example, account element "456789" could substitute for some real object in memory identified by integer "0x2a3b" (say, memory handle). This means that given account number we can get account primitive reference which provides direct access to the account object. This procedure of finding the substituted reference is referred to as *reference resolution*. For example, a house number could be resolved into physical coordinates. It is important that the substituted element belongs to real world, i.e., less virtualized world with less indirection.

A concept-oriented programming is a hierarchy of elements each substituting some real element with direct access (direct meta-transition). How such a run-time structure can be described at compile-time will be discussed in the next section.

Summary:

- A concept-oriented program consists of and manipulates reference-object pairs.
- Reference-object pairs are hierarchically ordered by means of inclusion relation.
- Any reference substitutes for and resolves into a primitive references providing direct access to the represented object.

Further reading:

- Papers:
 - [1], Section 2.1 (pages 3-5): Entities and Identities ([PDF](#))
 - [2], Section 2.1 (pages 3-6): Objects and References ([PDF](#))
- Wikipedia links:
 - [Inclusion relation](#)
 - [Complex reference](#)
- Blog post:
 - [Legalizing references and their duality to objects](#)

4 From Classes to Concepts

In the previous section we have postulated that a system consists of hierarchically ordered reference-object pairs which also substitute primitive elements. Since both references and objects possess their own structure and functionality they both can be described by means of conventional classes. This means that references are described by *reference classes* and objects are described (as usual) by *object classes*. However, since references and objects are two sides of one thing these two classes are constituents of one construct, called *concept*. Thus concept is defined as a pair of two classes: one reference class and one object class. If concept has empty reference class then it is equivalent to normal classes as used in OOP. For example, a concept of bank accounts could be defined as follows:

```

01  concept Account
02      reference {
03          char[8] accountNumber; // Account identifier
04          Address someMethod(Person person) { ... } // Reference method
05          ...
06      }
07      object {
08          double balance; // Account balance
09          Person owner; // Custom reference to the owner
10          Address someMethod(Person person) { ... } // Object method
11          ...
12      }

```

Here reference class is declared using keyword ‘reference’ while object class is marked by keyword ‘object’. Both reference class and object class has their own fields, methods and other constituents of normal classes. However, they cannot be used separately. Instead, in the concept-oriented program only the whole concept can be used. Thus in the concept-oriented program concepts are used where in an OOP program classes are used, namely, for declaring types of variables, parameters, fields, return values and other elements of the program. For example, in the above code snippet, the method takes one parameter of type `Person` and returns object of type `Address`. However, since it is a concept-oriented program, these types are names of concepts. As a consequence, any element with the concept type stores a custom reference rather than a primitive reference. The format of this custom reference is defined by the reference class of the corresponding concept. For example, `person` parameter might store a person unique number or full name and birthday. The address returned from this method is also a reference with the format described in the concept `Address`. If we declare a variable or field then again concept name determines not only object structure but also reference structure. Thus instead of using primitive references, variables and fields in the concept-oriented program store data in custom format described in some reference class.

In OOP classes are rarely defined alone – normally they exist within a class hierarchy by inheriting some base classes. The same is true for the concept-oriented programming however the difference is that concepts are defined within an inclusion hierarchy which generalizes inheritance. For example, if we need to define a new concept for savings accounts than it is done by including it into the parent account concept:

```

01  concept SavingsAccount in Account
02      reference {
03          char[2] subAccountNumber; // Sub-account identifier
04          ...
05      }
06      object {
07          double balance; // Sub-account balance
08          ...
09      }

```

The parent concept is specified using keyword ‘in’ while the sub-concept is defined as any other concept using one reference class and one object class. One consequence of having the parent account is that variables, parameters, fields and other elements will contain complex references. For example, a variable of type `SavingsAccount` will contain two segments <"12345678", "01">: the main account identifier (up to 8 digits) as high segment and the sub-account number (up to 2 digits) as low segment. (In the general case it is possible to control how many segments a variable will contain using the mechanism of reference length control.) It should be noticed that although reference segments exist together side-by-side within one reference, object segments exist separately. So the main account object could be one object with its own reference while sub-account objects can have their own references and have completely different locations.

COP program uses concepts to declare types and hence objects are represented by virtual addresses in a hierarchical space. These addresses are conventions and cannot be used directly for object access. In fact, an object represented by a custom reference could reside anywhere in the world and therefore we need some mechanism to locate it given its virtual address. For example, if we need to access a bank account given its number then the account object could be loaded from a database or its state could be obtained from some remote computer. We do not want to see these details when accounts are manipulated but the access procedure must be defined somewhere in the program. More specifically, we need a procedure for resolving this reference into direct object representation by a primitive reference. In order to solve this problem each concept defines a so called *continuation method* for its reference class. This method analyses this reference, restores the primitive references and then passes control further. For example, for bank accounts it could be implemented as follows:

```

01  concept Account
02      reference {
03          char[8] accountNumber; // Account identifier
04          void continue() {
05              print(" ==> Account: Start access");
06              Object o = loadByPrimaryKey(accountNumber);
07              o.continue();
08              storeByPrimaryKey(accountNumber, o);
09              print(" <== Account: End access");
10          }
11          ...
12      }
13      object {
14          double balance; // Account balance
15          ...
16      }

```

Whenever an account object is about to be accessed the compiler will call its reference continuation method. This method loads the state of the object into main memory where it is represented by a primitive reference of type `Object` (line 6). After access is finished the continuation method stores the state of the bank account back to the database (line 8). So these two instructions wrap any access to account objects. Line 7 is where the control is passed to the method called in the source context. For example, if we set account balance given account reference containing account number:

```
Account account;
account.accountNumber = "12345678";
account.balance = 25;
```

then we will get the following output:

```
$ ==> Account: Start access
$ <== Account: End access
```

Notice that access is completely transparent so that we work with objects as if they were directly accessible like all objects in OOP. Our program is completely independent of the mechanism of representation and access defined for accounts. For example, if in future accounts will move to some other kind of persistent storage then the only place in the program that has to be changed is the concept `Account` – all the rest of the program remains unchanged. Normally however, the situation is even better because the mechanism of representation and access is developed in special concepts with only this special purpose. Concepts implementing some business logic are then included into it. For example, we might define concept `PersistentStorage` or `Remote` and then use them as parent concepts. So the mechanism of representation and access can be modularized and then easily updated or changed. For example, if security rules change then we do not need to crawl all over the program by changing their checks: we simply change one concept and then this logic is automatically used for each access.

Sub-concepts implement their own continuation method which resolves this reference segment into a primitive reference providing direct access. The only feature is that information used for translation is stored in the parent object. For example, information about sub-account locations is stored within their main account object. So the parent object plays a role of our local postal office which knows the location of the addressed street and house. For example, it is quite natural that information about accounts is stored in each concrete bank and therefore if account are included in banks the resolution procedure will rely on the parent bank object:

```
01 concept Account in Bank
02     reference {
03         char[8] accountNumber; // Account identifier
04         void continue() {
05             print("==> Account: Start access");
06             Object o = super.loadByPrimaryKey(accountNumber);
07             o.continue();
08             super.storeByPrimaryKey(accountNumber, o);
09             print("<== Account: End access");
10         }
11     }
12 }
13 object {
14     double balance; // Account balance
15     ...
16 }
```

Here we use keyword ‘super’ to access the functionality of the bank object where this account really exists. Obviously, the bank reference (high segment of the account complex reference) has to be resolved before its accounts can be accessed. For example, if complex reference involves a bank as its high segment, say, `<"MyBank" , "12345678">`, then each access will produce the following output:

```
$ ==> Bank: Start access
$ ==> Account: Start access
$ <== Account: End access
$ <== Bank: End access
```

If we access a sub-account in a bank then the resolution procedure will consist of three steps.

Since the resolution could be rather complex procedure it should not be repeated too frequently. Therefore resolved reference segments are stored in a special structure called *context stack*. The context stack grows for each next segment and then decreases on the way back when access is finished and we return from continuation methods. Primitive references from the context stack are used when an object accesses itself or parent object using keyword ‘super’. There are also other

mechanisms for optimizing access such as reference length control and specifying block of code to be executed in the context of target object.

Summary:

- Concept is a pair of one reference class and one object class.
- Concept can be included in some parent concept.
- References to objects contain data in the format defined by reference classes starting from the first parent concept and ending with the last concept specified as the variable type.
- A reference is automatically resolved into a primitive reference with direct access using the continuation method of reference.

Further reading:

- Papers:
 - [1], Section 5 (pages 25-30): Operations with References ([PDF](#))
 - [2], Section 3 (pages 11-16): Concept Definition ([PDF](#))
- Wikipeda links:
 - [Concept](#)
 - [Concept inclusion](#)
 - [Complex reference](#)
- Blog posts:
 - [Three steps to understand the main idea of the concept-oriented programming \(COP\)](#)
 - [Modelling hierarchical address space in the concept-oriented programming \(COP\)](#)
 - [Creating concept instances](#)
 - [Two versions of the concept-oriented programming: COP-I and COP-II](#)

5 Access via Dual Methods

In the previous section we described how indirect virtual addresses representing objects are resolved into primitive references providing direct access. However, access is normally performed for carrying out some useful interaction with the object. In this section we assume that the type of interaction is specified by the method applied to the object reference. For example, if we want to get a bank account balance we apply the corresponding method to the account reference:

```
Account account = findAccount("Alexandr Savinov");
double balance = account.getBalance();
```

Again, in COP we do not make any assumption about any object location so this account might reside in main memory, on disk, on tape, on remote computer, on International Space Station or in a bank on Mars. When using objects we do not care where they really are and it is the task of the representation and access layer to find the object given its reference. So in COP we have an illusion of working with objects directly at the same time retaining a possibility to control the existing and to create new levels of indirection.

Let us now consider the sequence of execution of target methods applied to an object reference. The main problem here is that each concept can provide two definitions of one and the same method signature in its reference class and object class which are called *dual methods*. The method defined in the reference class is referred to as a *reference method* while the method defined in the object class is referred to as an *object method*. Moreover, each concept in the hierarchy can also define a pair of methods having the same signature. For example, if we have concept `SavingsAccount` included in concept `Account` which in turn is included in concept `Bank` then there can be 6 definitions of each method like `getBalance`.

For a simple reference consisting of one segment, access starts from the reference method, i.e., a reference intercept any access to the object by acting as a proxy. The reference method can then decide how to proceed and what other methods to call. In particular, it can call object methods including this same method. In order to distinguish reference methods from object methods within a concept we will use special keyword 'reference' and 'object' respectively. So `reference.getBalance()` is an invocation of the reference method while

`object.getBalance()` is an invocation of the object method. (These reserved keywords belong to a so called navigation mechanism which includes such keywords as ‘super’ and ‘sub’ described later.)

An example below shows how method `getBalance` could be implemented for concept `Account`.

```

01  concept Account
02      reference {
03          char[8] accountNumber; // Account identifier
04          void continue() { ... }
05          double getBalance() { // Reference method
06              print("====> Account: Start getBalance");
07              return = object.getBalance();
08              print("<=== Account: End getBalance");
09          }
10      ...
11  }
12      object {
13          double balance; // Account balance
14          double getBalance() { // Object method
15              print("---> Account: Start getBalance");
16              return = b;
17              print("<--- Account: End getBalance");
18          }
19      ...
20  }

```

Here the reference method prints some output when it starts (line 6) and before it returns (line 8). The real work is delegated to the same method of object class (line 7) using keyword ‘object’. The object method also prints two lines when starts and before it ends (lines 15 and 17). (We use special variable named ‘return’ to assign a value that will be return when we leave the method scope.) If we apply method `getBalance` to a simple reference to an account then the following output will be produced:

```

$ ==> Account: Start getBalance
$ ---> Account: Start getBalance
$ <--- Account: End getBalance
$ <=== Account: End getBalance

```

Let us now consider what happens when a method is applied to a complex reference where each segment (each concept in the hierarchy) implements the invoked method. Here the sequence of access is different for reference methods and object methods. For reference methods we use the following rule: *parent reference methods have precedence over (override) child reference methods*. Thus parent concepts can control access to child concepts (including those added in future) by overriding some child method. Obviously, it is opposite to the rule accepted in OOP which is also valid for COP object methods: *child object methods have precedence over (override) parent object methods*.

If a method is applied to a complex reference then the first method to be executed is that defined in the very first parent reference class from which the hierarchy starts (if it implements it). For example, if concept `Account` is a parent for all sub-accounts then it can intercept in its reference class all invocations of method `getBalance`. If concept `Account` is included in concept `Bank` then the bank will be able to control when and how its accounts are accessed by overriding the corresponding methods. It is quite natural because we need some mechanism of access control when an external process enters this scope. When a parent reference method got control, it can continue either by calling its own object methods as described before or by passing request further to the child reference. In order to access child elements COP introduces a new keyword ‘sub’ which is a dual version of the standard OOP keyword for accessing parent objects (‘super’ in Java). So invocation `sub.getBalance()` means that we apply this method to the next segment of the current reference. If we are in `Account` reference then the next segment might be of `SavingsAccount` type.

At the same time COP retains the conventional direction for overriding object methods. This means that child object methods override parent object methods but can call them explicitly by means of keyword ‘super’. So invocation `super.getBalance()` means that we apply this method to the parent object. For example, if we make this call from `SavingsAccount` then this method will be applied to `Account` object which is its parent.

An example illustrating these two rules of access is shown below.

```

01 concept ParentConcept
02   reference {
03     double someMethod() { // Reference method
04       print("===> ParentConcept: Start someMethod");
05       return = sub.someMethod();
06       print("<=== ParentConcept: End someMethod");
07     }
08     ...
09   }
10   object {
11     double someMethod() { // Object method
12       print("---> ParentConcept: Start someMethod");
13       return = 999;
14       print("<--- ParentConcept: End someMethod");
15     }
16     ...
17   }
18
19 concept ChildConcept in ParentConcept
20   reference {
21     double someMethod() { // Reference method
22       print("  ===> ChildConcept: Start someMethod");
23       return = object.someMethod();
24       print("  <=== ChildConcept: End someMethod");
25     }
26     ...
27   }
28   object {
29     double someMethod() { // Object method
30       print("    ---> ChildConcept: Start someMethod");
31       return = super.someMethod();
32       print("    <--- ChildConcept: End someMethod");
33     }
34     ...
35   }

```

Here we have two concepts each implementing `someMethod` in both reference class and object class. The parent reference method (line 3) delegates the request to its child element using keyword 'sub' (line 5) being unaware what is the next concept (it may have many different child concepts). The child reference method (line 21) turns to object world by calling the same method of its object class (line 23) using keyword 'object'. Since we are now in the object world the rules of the overriding game have changed to the opposite ones. The child object method (line 29) delegates the request to its parent object using keyword 'super' (line 31). And finally the parent object method (line 11) returns some concrete value (line 13). The output produced by such a method call is shown below:

```

ChildConcept childConcept = new ChildConcept();
double value = childConcept.someMethod();

$ ===> ParentConcept: Start someMethod
$   ===> ChildConcept: Start someMethod
$   ---> ChildConcept: Start someMethod
$   ---> ParentConcept: Start someMethod
$   <--- ParentConcept: End someMethod
$     <--- ChildConcept: End someMethod
$     <=== ChildConcept: End someMethod
$   <=== ParentConcept: End someMethod

```

So the typical sequence of access starts from the base reference segment and then proceeds down to some child reference segment where the process turns to the child object. Child objects then proceed in the opposite direction by delegating the request to their parent objects (as it is normally done in OOP). Of course, we can always switch between reference and object if it is necessary using keywords 'reference' and 'object'.

Summary:

- Both reference class and object class can define one and the same method which are called dual methods of the concept.
- Reference method intercepts access to the element and then can call object methods.

- Parent reference methods have precedence over (override) child reference methods with the same signature.
- Child object methods have precedence over (override) parent object methods with the same signature.

Further reading:

- Papers:
 - [1], Section 4.2 (pages 16-20): Sequence of Access ([PDF](#))
 - [2], Section 8 (pages 34-35): Dual Methods ([PDF](#))
- Wikipedia links:
 - [Dual methods](#)
- Blog post:
 - [Modelling references](#)

6 COP vs. OOP

COP is designed to be backward compatible with OOP, i.e., it is reduced to OOP under certain simplifying assumptions. In this section we compare COP with OOP by showing where they are different and why COP can be considered a generalization of OOP.

One of the cornerstones of the object-oriented paradigm is *inheritance*. This mechanism allows us to extend existing classes by adding new properties and behaviour at the same time retaining those of the extended parent class. A program is then being developed as an inheritance hierarchy of classes where new more specific classes inherit properties of the parent class. However, at run-time all class instances exist in one space without any hierarchy, i.e., the original inheritance hierarchy effectively disappears. In particular, any object consists of segments produced by all its parent classes which cannot be shared. In other words, whenever we create a new object of some class, it gets all parent segments stored normally side-by-side in memory. So in OOP parent object segments cannot be shared among child segments and hence the original hierarchy is removed. Another feature illustrating that all objects exist in one uniform space is that all of them are represented by one and the same type of primitive references. So in OOP the situation is the same as if all houses in one country would be represented by some globally unique identifier which had to be specified for sending letters.

Such a relationship can be denote as IS-A relationship between children and parents. For example, if class `Button` inherits class `Panel` then we can say that any button object IS actually A panel (Fig. 4 left). When we create a new button we simultaneously create also a panel. Here we clearly see some asymmetry of the object-oriented approach: parent classes are shared while parent objects are not (hierarchy disappears at run-time and objects exist in flat space). In COP the design is completely symmetric because objects exist within a hierarchy which has the same structure as the hierarchy of concepts. To achieve it we use inclusion relation as a generalization of inheritance, i.e., we say that a child IS-IN its parent (or simply IN). As a result, objects can share their parents. For example, if we create a button object then it can be created IN some panel where other buttons already exist or can be created in future (Fig. 4 right). So a panel is a container for buttons as well as other child objects like check box or radio button. Any child object has its own local reference which distinguishes it from other children and also provides separate location and life-cycle. Keyword 'super' in this case provides access to a panel object segment which is shared among all its internal objects. The parent object plays a role of container or environment where child objects exist.

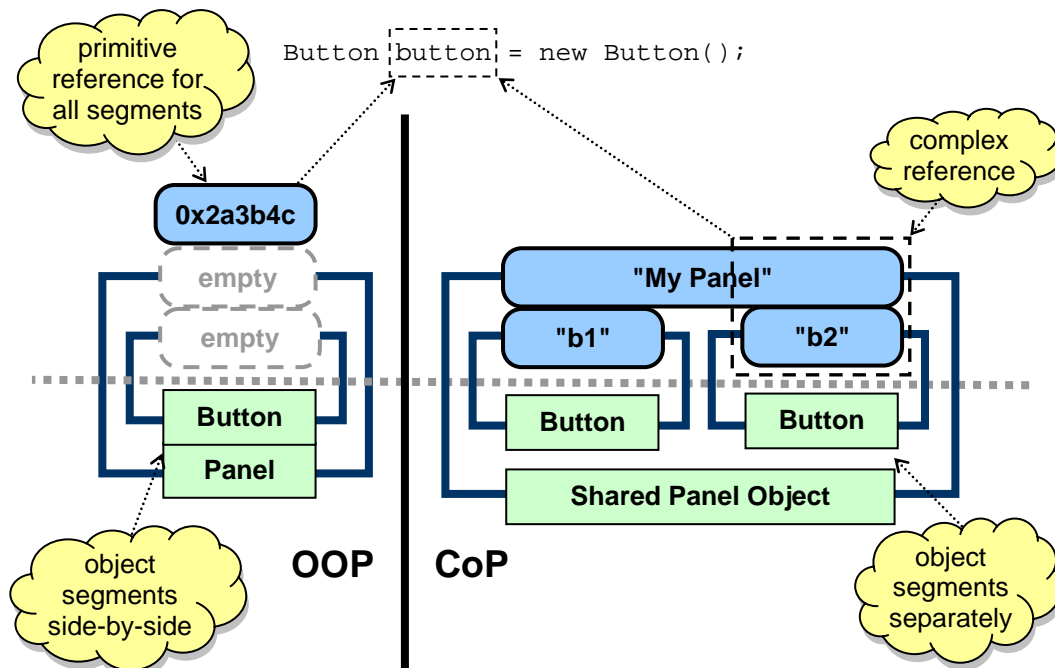


Fig. 4. Inheritance in OOP and inclusion in COP.

One interpretation of the fact that in COP objects exist within a hierarchical structure is that it makes the whole approach completely different from OOP. However, an amazing property of inclusion relation is that it actually generalizes inheritance and is backward compatible with the conventional object-oriented approach. The thing is that under certain simplifying conditions the hierarchy of objects at run-time (COP) is reduced to a flat space of objects (OOP). This happens when a child concept does not define fields in its reference class (in Fig. 4 it is shown as empty reference classes). This effectively means that objects of this concept cannot be identified and distinguished in the context of the parent object just because they do not have their own references. Hence child objects have to reuse the parent object reference, i.e., the parent reference is used to identify also its child and hence it can be considered one object precisely as it is done in OOP. For example, if concept `Button` does not have its reference class then it reuses references provided by concept `Panel` for identifying both segments. If concept `Panel` also does not define its reference class then it inherits primitive references which are used for identifying objects within this hierarchy (buttons or panels). It is precisely what we have in OOP.

Polymorphism is a mechanism which allows the programmer to manipulate objects of more specific types as if they were of the base type. For example, if we declare a variable having type `Account` then we still can apply method `getBalance` to it while the real behaviour depends on the type of reference stored currently in this variable. In particular, if it stores a reference to a savings account then this method returns the balance on this sub-account rather than on the main account. In OOP this mechanism is based on overriding parent methods so that it is guaranteed that the executed method corresponds to the real object type. In this case base objects can contribute to the processing only if their functions are used from the executed child method.

In COP polymorphism is based on two mechanisms: overriding reference methods and overriding object methods. In particular, we postulated that parent references override child references and hence the base reference method will always be executed first independent of the real type of the object. For example, if concepts `SavingsAccount` and `CheckingAccount` are included in `Account` then independent of the real reference type one and the same method of concept `Account` will be executed. Thus here we get one and the same behaviour independent of the real object type. However, normally parent objects do their own work and then make it possible for child objects to contribute to the overall processing. And it is precisely the moment where the final behaviour depends on the real object type. The parent reference method delegates the request further to the child using 'sub' to point to the next reference segment. If the next segment is of concept `SavingsAccount` then one definition of the method will be executed while if it is of concept `CheckingAccount` then some

other version will be executed. It is also possible that there is no child segment at all. In this case the parent method can call its own object method to get the balance of the main account. So the main idea of this approach is that depending on the real composition of the reference we get different results by applying one and the same method.

Let us consider an example where each intermediate element helps in implementing a method. In the following listing we define concept `Button` which is included in concept `Label` which in turn is included in concept `Panel`. Reference classes and object classes of these concepts define method `draw`.

```

01  concept Panel
02      reference {
03          void draw() {
04              object.draw();
05              sub.draw();
06          }
07      }
08  }
09  object {
10      Color color;
11      void draw() { fillBackground(color); }
12  }
13
14  concept Label in Panel
15      reference {
16          void draw() {
17              object.draw();
18              sub.draw();
19          }
20      }
21  object {
22      String text;
23      void draw() { drawText(text); }
24  }
25
26  concept Button in Label
27      reference {
28          void draw() {
29              object.draw();
30              sub.draw();
31          }
32      }
33  object {
34      Border border;
35      void draw() { drawBorder(border); }
36  }

```

Any invocation of this method is intercepted by the panel reference which implements its own logic in its object method. In particular, it can fill the panel background using the panel colour. After that the panel makes it possible for the child object to contribute to processing this request (line 5). If the next segment is a label then it draws a text and then again sends this request further to the next possible child (line 18). If the third segment is a button then it contributes by drawing a border using (like the previous methods) its own object method (line 29). Since the button expects that it can be only a parent of some more specific object, it passes this call further to the child (line 30). (We assume that non-existence of an element is represented by null and results in no-operation when used.)

In this example we used reference methods to dispatch the request while object methods implement the real logic specific to this object. Of course, we might also call super-methods from objects by requesting support from the parent. For illustrating polymorphic behaviour it is only important that the final behaviour depends on the real type of references. A distinguishing feature of COP is that different segments can make their own contribution as the request is being processed.

Summary:

- In COP, objects exist within a hierarchy having the same form as the hierarchy of concepts.
- Parent objects can be shared among many child objects.
- The behaviour of a method depends on the real reference composition, i.e., the type and number of its segments.

- Each reference segment and object segment can contribute to the overall processing.

Further reading:

- Papers:
 - [1], Section 6 (pages 30-35): COP as a Generalization of OOP ([PDF](#))
 - [2], Section 7 (pages 31-34): Inheritance and Polymorphism ([PDF](#))
- Blog posts:
 - [Concept-oriented programming \(COP\) and smart pointers in C++](#)

7 COP vs. AOP

One property of complex programs is that one and the same functionality is used in numerous places. For example, whenever we want to use a memory block represented by a handle we have to lock it and when the access is finished the memory has to be unlocked. So one and the same set of instructions is repeated for each access. If we want to read data from a file then it has to be opened and then closed. Any access to a database is performed within a transaction which has to be opened and closed. There are actually two problems connected with this phenomenon. The first is that it is difficult to maintain such a code which involves repeating fragments. For example, if the logic of transaction management has changed then we have to change many points in the program. The second problem is more general. The thing is that these examples of repeating code represent one type of functionality while the places where this code is used deals with a different type of issues. So the problem is that different concerns are mixed. For example, changing account balance belongs to business logic while transaction management and security belong to completely different types of functionality. Strictly speaking we should not mix these concerns in one module. Ideally, we want to change an account balance in one method without dealing with transactionality, security and other non-relevant issues.

Aspect-oriented programming provides a mechanism for solving the problem of the separation of concerns on the basis of a new programming construct called *aspect*. Shortly, an aspect specifies code that has to be automatically injected in a set of points in this program which are also specified in this aspect. For example, an aspect might say that each account method has to be executed within a transaction. After that we can use accounts as usual but some intermediate code will be automatically triggered.

In COP the same effect is reached by means of parent concepts which are able to intervene into the process of object access. The intervention can be performed either when a reference is being resolved for any type of access or when a parent reference method intercepts the child method. For example, we might open transaction before account number is resolved and close transaction when access is finished.

If we need some common functionality to be injected into many different concepts then it is possible to define it as a base concept into which child concepts are included. For example, we might develop parent concept `Logging` which intercepts method calls or other types of access to child objects. If some concept needs to be logged we simply include it into this concept. Or we might develop concept `Persistent` which implements the logic of access to persistent objects. If some child concept is included into it then its objects are automatically made persistent. This means that before an object is accessed its state is loaded into memory, transaction is opened and access rights are checked. Notice that child concepts deal with only their own business logic while all the auxiliary routines are modularized in parent concepts. Thus cross-cutting concern from parent concepts is automatically injected in many different child concepts.

The main difference of AOP from COP is that in AOP aspects store a set of target points into which the code has to be injected while these target points (classes, methods etc.) are completely unaware that their behaviour will be changed. In contrast, in COP, target points themselves have to specify what kind of intervention and active support they need by pointing to a parent concept. So parent concept in this sense is analogous to aspect. But in contrast to AOP, parent concepts do not store a list of child concepts but rather each child concept specifies its parent. If COP followed the style of AOP then each parent would declare a set of all its child concepts. Such a style is a kind of forward links to what may not yet exist because parent concepts cannot know what child concepts will be added in future. This problem actually exists in AOP where aspects make forward pointers to points which can be added in future. One consequence is that the code is error-prone because we cannot predict how new code will work.

Another difference of COP from AOP is that in AOP we assume one concrete type of functionality that has a cross-cutting nature, namely, object representation and access. In other words, we say that direct instantaneous access does not exist in nature and there are always some operations executed during access. And it is precisely what cross-cuts the whole system because these functions are the same for many different types of objects. In AOP there are no any assumptions about the nature of cross-cutting concerns, i.e., we define any code and then inject it in any set of points during program execution.

The next difference is that aspects are known to be orthogonal to classes, i.e., it is an additional mechanism defined independently of the object-oriented principles. For example, in Fig. 5 (left) functionality of an aspect propagates horizontally and is orthogonal to the class hierarchy. Thus aspects are able to inject their functions in arbitrary points in the program. In contrast, COP smoothly generalizes object-oriented constructs and mechanisms rather than adds new ones. In particular, concept can behave like a class under certain conditions while inheritance and polymorphism take a new more general form in COP. In addition cross-cutting behaviour propagates along the inclusion hierarchy while in AOP the direction of propagation can be arbitrary. For example, in Fig. 5 (right) the functionality of a parent concept propagates down to its child concepts in parallel with the concept hierarchy.

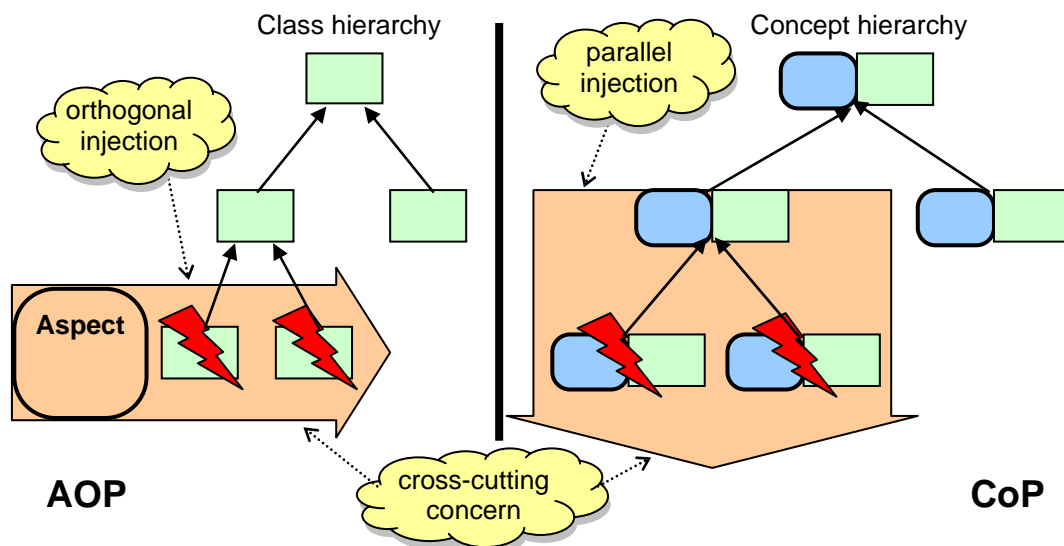


Fig. 5. Propagation of cross-cutting concerns in AOP and COP.

Summary:

- Aspect stores a set of target points into which some code has to be automatically injected.
- It is the responsibility of the target (child) concept to point to the code that has to be injected into it (parent concept).
- Object representation and access (ORA) functionality is the cross-cutting concern that is modularized in COP.

Further reading:

- Papers:
 - [1], Section 8.4 (pages 42-45): Language Level ([PDF](#))
 - [2], Section 9 (pages 36-40): Related Work ([PDF](#))
- Blog posts:
 - [Concept-oriented programming vs. aspect-oriented programming](#)

8 Concept-Oriented Model

The amazing thing about COP is that it establishes a basis for a new approach to data modelling called the concept-oriented model (COM). The most interesting general property of COM is that it splits data modelling into two orthogonal directions which correspond to the division of all things into identity and entity parts (described in Section 3). More specifically, data in COM is described by two structures:

[Identity modelling] Physical or hard structure describes how data is represented and accessed. In particular, physical structure of data has a hierarchical form where each element has one parent and may have many children. The elements are described by concepts and *inclusion relation*. In this sense it is precisely what COP deals with.

[Entity modelling] Logical or soft structure describes data semantics. Formally, it is based on *ordering relation* between data items, i.e., for two data elements we can say that one is greater than the second. The ordering relation is then used to derive various mechanisms that are normally used in data modelling. In particular, we can interpret it in terms of the object-attribute-value setting or as a multi-dimensional model.

Thus in COM the data modeller can design any inclusion hierarchy where elements have arbitrary identifiers. The whole approach to physical structure design is analogous to COP and can differ only in language means and implementation forms. For example, we could define a table with banks which consists of accounts which in turn consist of sub-accounts. Or we could define a table with countries where each country consists of cities where each city consists of streets and a street consists of houses. Actually we get a structure similar to one of the oldest hierarchical data model. Also this hierarchical structure can be used to model inheritance and class hierarchy as it is done in OOP and the object-oriented data model. Notice however, that in OO data models it is not possible to model references – precisely what concept-oriented approach is intended for.

The main role of physical structure consists in describing how data elements are *located* in a hierarchical virtual address space. Once a data item is created it gets some permanent address which is then used to represent it in other contexts (in other entities). However, this structure cannot represent data semantics (what data means) and for that purpose the concept-oriented model introduces logical structure which is dual to physical structure. Logical structure is not used in COP because in programming we normally manage data semantics manually in the program and do not need any support. However, when data needs to be managed automatically, the database has to possess proper knowledge of data semantics which is necessary for correct data manipulations. The main distinguishing feature of COM data semantics is that it is based on ordering relation and the theory of ordered sets. This relation makes this model somewhat similar to the network model and ontologies.

Fig. 6 provides an example of a simple concept-oriented data model which is intended to demonstrate the division between physical and logical structures. In contrast to the rest of the paper, here physical structure spreads horizontally so that sub-concepts are positioned to the right of their parent concept. (Another difference from the rest of the paper is that identities are shown *under* entities.) For example, concept Bank is used to create a table with bank records and concept Account is used to create a table with account records. Creation of tables could be written as follows:

```
CREATE TABLE Banks CONCEPT Bank
CREATE TABLE Accounts CONCEPT Account
```

It is important however that each account record will be associated with one bank record (one bank has many accounts). As a consequence, one record of any concept has a local identifier described in its reference class which is valid only in the context of its parent record. In order to store a record identifier in other data items it is necessary to use a complex reference consisting of several segments. In the same way we can model other identities such as persons (concept Person), addresses (concept Address) and account owners (concept AccountOwner). All the concepts have one root which is not shown but has to have the left most position. Notice that all arrows representing inclusion relation have leftward direction. It is important that at this stage we deal with *identity modelling*, i.e., we define how our data items will be represented and accessed. Here we do not care what is their meaning which is expressed in entities (object fields described via object classes).

Logical structure is defined by object classes of concepts and it spreads vertically in the diagram. More specifically, it is assumed that if object class *B* references concept *A* via some its field then *B* is less than *A*: $B < A$. In diagrams *B* is positioned below *A* and references are shown as dotted arrows having always upward direction. Concept *B* is also called sub-concept while concept *A* is called super-

concept (do not confuse them with their physical counter-parts). Since one concept can reference many other concepts and one concept can be referenced by many other concepts this structure has a multi-dimensional hierarchical form (there can be many super-concepts). It is also assumed that there is one common super-concept called *top concept* and one common sub-concept call *bottom concept* (not shown in Fig. 6). For example, a relationship between accounts and persons is established via concept `AccountOwner` which references one account and one person via its fields. Thus `AccountOwner` is a logical sub-concept for both concepts `Account` and `Person`. On the diagram we see this relationship because concept `AccountOwner` is positioned below both concepts `Account` and `Person`, and two arrows lead from the sub-concept upwards to their super-concepts.

Once this vertical logical structure is established we can manipulate data using such operations as *projection* and *de-projection*. For example, in order to find a set of accounts related to some address we should de-project this address (concept `Address`) downwards to persons (concept `Person`), then again de-project the result downwards to account owners (concept `AccountOwners`) and finally project this result upwards to accounts (concept `Account`):

```
accounts = someAddressItem
  <- address <- Person // De-project
  <- owner <- AccountOwner // De-project
  -> account -> Account // Project
```

In this query left arrow denotes de-projection and right arrow means projection. We start from a data item stored in `someAddressItem` (it could be a collection of addresses returned from some other query) and then de-project it two times along dimensions `address` and `person`. Then the result is projected along dimension `account`. The formal semantics of the concept-oriented model can be used in very different mechanisms such as constraint propagation, grouping, aggregation, inference, multi-dimensional modelling, OLAP and many others.

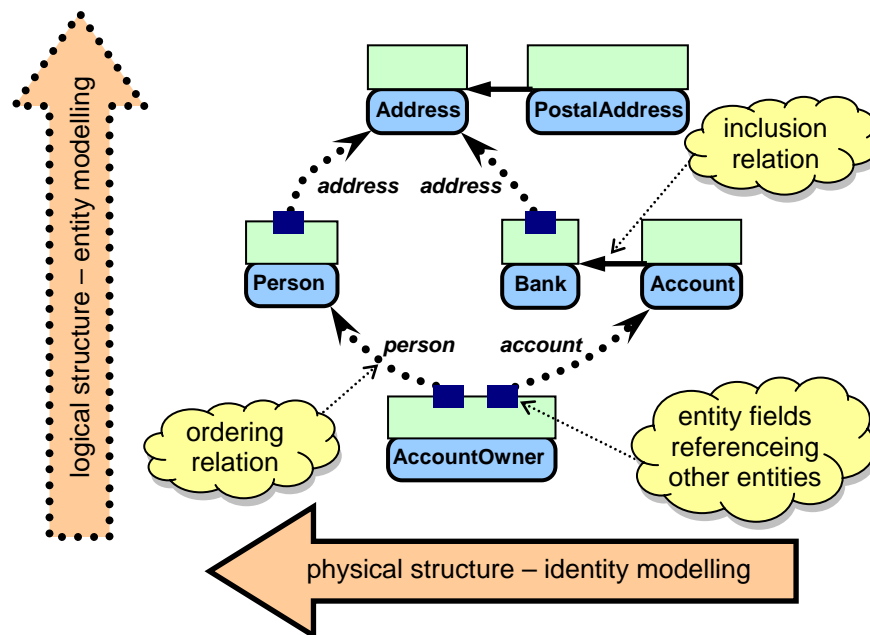


Fig. 6. Identity and entity modelling in the concept-oriented model.

Summary:

- COP is analogous to physical structure of COM which are both based on inclusion relation, i.e., identity modelling in COM is performed using principles of COP.
- Logical structure of COM uses ordering relation for describing data semantics and other data modelling mechanisms.

Further reading:

- Papers:
 - [Informal Introduction into the Concept-Oriented Data Model \(PDF\)](#)
- Wikipedia links:
 - [Concept-Oriented Model](#)
- Blog posts:
 - [Main principles of the concept-oriented data model](#)

9 Conclusions

Concept-oriented programming is a new programming paradigm which generalizes OOP. Its main programming construct, called concept, has two constituents: one reference class and one object class. Such a composition allows us to reflect the duality of the real systems consisting of reference-object pairs. In other words, in COP it is possible to describe structure and behaviour of both reference and objects by modelling not only entities but also identities. References in COP are completely legalized and made first class citizens. As a result the programmer can define custom virtual address space for the objects and then use these objects as if they were directly accessible.

The solution based on concepts could be informally compared with the introduction of complex numbers in mathematics which also have two constituents: real part and imaginary part. As a result of such mathematical generalization formulas and manipulations get much simpler and more natural in comparison with the conventional real numbers. The same effect is achieved in programming by introducing concepts instead of classes: programs get much simpler and their logic is expressed more naturally and elegantly when we manipulate pairs of references (imaginary part) and objects (real part).

References

- [1] A. Savinov, An Approach to Programming Based on Concepts, Institute of Mathematics and Computer Science, Moldavian Academy of Sciences, Technical Report RT0005, 49pp., 2007 ([PDF](#)).
- [2] A. Savinov, Concepts and their Use for Modelling Objects and References in Programming Languages, Institute of Mathematics and Computer Science, Moldavian Academy of Sciences, Technical Report RT0004, 43pp., 2007 ([PDF](#)).
- [3] A. Savinov, Concept as a Generalization of Class and Principles of the Concept-Oriented Programming, Computer Science Journal of Moldova, Vol. 13, No. 3, 292-335, 2005 ([PDF](#)).
- [4] A. Savinov, Two-Level Concept-Oriented Data Model, Institute of Mathematics and Computer Science, Moldavian Academy of Sciences, Technical Report RT0006, 40pp., 2007 ([PDF](#)).
- [5] A. Savinov, Concepts and Concept-Oriented Programming, Journal of Object Technology, vol.7, no.3, March-April 2008, pp. 91-106 http://www.jot.fm/issues/issue_2008_03/article2/

Forthcoming publications

- [6] A. Savinov, Concept-Oriented Programming, Encyclopedia of Information Science and Technology, 2nd Edition, Editor: Mehdi Khosrow-Pour, IGI Global, 2009.
- [7] A. Savinov, Concept-Oriented Model, Handbook of Research on Innovations in Database Technologies and Applications: Current and Future Trends, Editors: Viviana E. Ferragine, Jorge H. Doorn, Laura C. Rivero, IGI Global, 2009.