Computer Science Journal of Moldova, vol.**13**, no.3(39), 2005

Concept as a Generalization of Class and Principles of the Concept-Oriented Programming

Alexandr Savinov

Abstract

In the paper we describe a new construct which is referred to as concept and a new concept-oriented approach to programming. Concept generalizes conventional classes and consists of two parts: an objects class and a reference class. Each concept has a parent concept specified via inclusion relation. Instances of reference class are passed by value and are intended to represent instances of child object classes. The main role of concepts consists in indirecting object representation and access. In concept-oriented programming it is assumed that a system consists of (i) conventional target business methods (BMs), and (ii) hidden representation and access (RA) methods. If conventional classes are used to describe only BMs then concepts allow the programmer to describe both types of functionality including its hidden intermediate functions which are automatically executed when objects are being accessed.

1 Introduction

1.1 Object Representation and Access

Let us consider a conventional method call: `myRef.myMethod()`. In OOP it is assumed that a reference stores a target object identifier. It is allocated and managed by routines which are not directly controlled by the programmer. For example, memory handles are allocated by the operating system and Java references are provided by the runtime

environment. In such an approach the programmer is unaware of how objects are represented and what intermediate actions are performed behind the scenes after a method is called and before its first statement starts. The traditional object-oriented program functionality is concentrated in class methods. It is important that any function executed in the program is the result of some *explicit* method invocation written by the programmer somewhere in the source code. And any object that appears in the program is the result of an *explicit* instantiation. The program itself does not create any data structures and does not perform any actions for its internal use in order to maintain user defined classes.

Such an approach to programming is known to be very simple and efficient for many types of systems because the compiler or runtime takes care of all the object representation and access issues. However, this full automation has its price: the programmer is not able to influence the object representation and access mechanism and is restricted by the standard functionality. Indeed, in many cases the following question arises: What if I want to define my own format of references and access procedures which are developed specially for my system? For example, I might want to develop my own memory manager because the objects I am going to use have a very special format and properties. Or, in addition, my system might need to carry out special security checks whenever its objects are accessed. In all these and many other cases the standard mechanisms of representation and access could be too restrictive. In particular, main memory is only one possible location for objects. In general case they may well be stored in some cache, on disk or on remote computer. Even if a memory manager is very general it cannot cover all the needs of an arbitrary program.

A new approach to programming described in this paper assumes that we can define our own format of references and our own access procedures which are adapted to the purposes of each individual program. Custom references could be defined as integers, text strings or a combination of any other fields. And the corresponding access procedures may include any code because it is written by the programmer. In this case the representation and access mechanism is an integral

part of the program it is written for. However, these custom references and access procedures are not used by the programmer anywhere in the source code. It is the task of the compiler or runtime environment to activate them. Thus the method `myRef.myMethod()` does not start immediately because it is necessary to resolve the reference and to find the target object. In this case some intermediate procedures are implicitly activated and this code (which is part of the program) executes after the method call and before its first statement.

1.2 Two Types of Functionality

One of the main general assumptions of the new approach is that there exist two types of functionality:

- *business methods* (BMs) which are defined in classes and used explicitly in the program, and
- *representation and access* (RA) functionality constituting a separate cross-cutting concern and activated implicitly

Business methods or target methods are explicitly used by the programmer in order to access applied functionality of object classes in the traditional OO manner independent of how they are represented and accessed. BMs are precisely what OOP is designed for: we can easily define classes (with reuse via inheritance) and then call their object methods in the program. RA functions (or intermediate functions) introduced and studied in this paper determine how objects are represented and accessed independent of the target BM.

In the described approach we assume that a great deal of program functionality is activated and executed when objects need to be represented and accessed. It is a kind of invisible matter that cross-cuts any program because these functions are hidden, they are not called explicitly in the source code and they are executed behind the scenes. In other words, we assume that such a simple line of code as `myRef.myMethod()` may activate rather complex intermediate functions which are invisible in traditional programs. A general goal of the described approach to

programming consists in making this hidden level of functionality an integral part of the program. We need to legalize these functions because they cannot be qualified as something auxiliary while the facilities provided by the standard runtime environments are rather limited for contemporary programs. The thing is that in large program systems RA functions account for most of the program complexity. This is why the level of RA functions should be an integral part of the program that has to be dealt with and developed for this very program.

If we represent a program as consisting of internal spaces (scopes, containers, layers) where objects live then RA functions can be thought of as concentrated on this space borders and automatically executed whenever a process intersects a border on its way to the target object (Fig. 1). Target BMs are executed when the process reaches the target object. According to this analogy each method call is a sequence of steps leading to the target object. (In contrast, in the conventional programming a method call is viewed as only one step leading from the source context to the target.) The target business method specified explicitly in the program is only the last step while intermediate steps involve various RA functions which are hidden and are executed seamlessly behind the scenes. In particular, in such a program source code it would be impossible to find any explicit invocation of an intermediate method. The program might consist of a relatively small number of explicit method calls but be rather complex because of the hidden functionality.

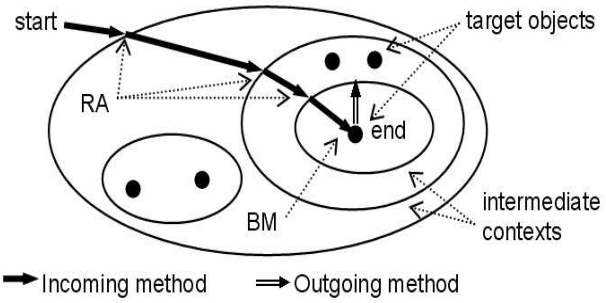

Figure 1. Intermediate borders possess important functions

One of our main assumptions is that the automatically triggered border processes account for a great deal (and even most) of the whole program complexity and reflect specific properties of this program just like its BMs. Hence the programmer needs to be able to describe these functions as an integral part of the program. In other words, a program has to consist not only of BMs (normal classes with their methods) but also include facilities for describing space borders, how objects are represented within particular subspaces and how they are accessed. In such an approach even if a program has little or no BMs at all, it may well be rather complex because of its internal space structure and RA functions associated with space borders and automatically executed when they are intersected by interacting processes.

1.3 Design Goals

It is very important that RA functionality is not associated with any target class but instead, it scatters the whole program. However, it is important that one RA mechanism be described in one place rather than distributed all over the program. This criterion is formulated as the following design goal:

DG1 [Modularization] Representation and access functions have to be described in a modular manner.

This means that if there are two custom RA methods as part of the program then they need to be described in two places. Then these RA functions can be automatically and implicitly injected in all appropriate points in the program where we find it necessary. For example, one custom reference format with its associated resolution methods should be described in one place.

Although RA functions are described in a modular manner we do not want to use them explicitly in all points in the program where they are appropriate. We want only to somehow specify those points so that the compiler could inject the necessary code automatically. It is important that all the method calls are described as usual by specifying a target object and some method. The compiler then uses declarative properties of the target class in order to choose what intermediate

actions to execute. However, in the source we do not see those intermediate actions because they are hidden. So the next design goal consists in ability to make normal method calls:

DG2 [Transparency] It is an illusion of instant access. Method calls are made as usual by specifying only a reference and a method without any explicit indication of the type of intermediate actions.

A consequence of this principle is that the use of objects and their BM does not change when we change the underlying RA mechanism. For example, we may have a huge number of BM calls like `ref1.m1()`, `ref2.m2()`, `ref3.m3()` all over the program. In OOP all these objects use one and the same default mechanism of RA. However, in our approach objects of different classes may use different and rather complex RA mechanisms assigned to them (and described in this very program in a modular way). In particular, `ref1` might be identified by a primary key and accessed via JDBC protocol, `ref2` might be represented by an absolute offset in the local heap while `ref3` might be an object on the moon represented via some Universe Unique Identifier and accessed via ISS. The transparency of access guarantees that we do not need to know how objects are represented in order to access their BM. We retain an illusion of instant access when using objects and it is the task of compiler, interpreter or an execution environment to activate all the necessary RA functionality. If we change the way how our objects are represented then the source code where they are used does not change.

It is very important that one and the same RA mechanism could be used to serve many target classes in the program. For example, if we develop a complex hierarchical persistent memory manager with special access rules then it is very natural to use it for any class of object in the program. Thus the following design goal makes sense:

DG3 [Reuse] Many target classes should be able to use one RA mechanism.

Each target class in the program can be assigned some appropriate RA mechanism. And even for individual uses of classes (instantiations) it is desirable to be able to specify how this concrete object should be represented and accessed. However, we want to do it in a declarative manner rather than to control this at run-time. The compiler then

uses the declarations in order to activate the necessary intermediate functionality:

DG4 [Declarativity] RA mechanism should be assigned to target classes in a declarative manner.

Assume that there is a module with intermediate RA functionality and a module with target BMs. We want these business methods be accessed via this intermediate functions. There is a design alternative: either (i) to indicate the target module in the context of the RA module, or vice versa (ii) to indicate the RA mechanism in the context of the target module. In other words, who knows whom: intermediate module knows target module, or vice versa? We choose the second alternative:

DG5 [Direction] RA modules do not know target classes they serve and it is each individual target class that should declare what kind of RA mechanism it needs.

The two types of functionality found in any system differ logically rather than physically. This means that they always exist together, cross-cut each other and in most cases cannot be separated. One and the same piece of code can be considered an intermediate function that is activated automatically and a normal business method that is used explicitly. In particular, we do not have one programming construct for business methods and another construct for intermediate functionality. We want to have one mechanism that is able to express both types of functionality:

DG6 [Integrity] A module should not have one dedicated purpose but rather it should be able to express both types of functionality simultaneously.

Assume that for our target classes we specified some concrete RA mechanism. The functions of this RA module are then used before the target methods will be executed. However, in many cases we want this very RA mechanism to rely on some other RA mechanism. In this case the structure of indirection will be nested.

DG7 [Hierarchy] RA functionality described in a modular manner should have a hierarchical structure where parent modules play a role of intermediate layers for their child modules which play a role of targets.

1.4 Concept

In this paper we propose a new approach, called concept-oriented programming (COP), which is based on a special construct called *concept*. Shortly, concept is a combination of one object class and one reference class. Object class is the conventional class as defined in OOP. What is new in this approach is the reference class which complements the object class just like RA functions complement BMs. By combining object class and reference class we make it possible to describe two sides of any system: explicit BMs and implicit RA functions. Object class and reference class have one name (concept name) and may define methods with the same name (any method has a definition within object class and within reference class). Instances of object class, called objects, are passed by reference. Instances of reference class, called references, are passed by value. Informally, the main idea is that references passed by value can represent objects.

Concepts are organized into a hierarchy by using inclusion relation, i.e. any concept has one parent concept. Concepts cannot exist outside an inclusion hierarchy — if a concept does not have a parent concept then it is assumed to be some default concept. Concept hierarchy plays an important role because its structure determines how objects in the program are represented and accessed. In other words, the format of reference and intermediate procedures used to access some target object depend on its position in the concept inclusion hierarchy. Parent concept always indirects representation and access to its child concepts. In order to specify what RA mechanism to use we simply need to include a concept into an appropriate parent concept.

One concept can be interpreted as a space with its own border (Fig. 1). If a concept is included into another concept then it is placed within this parent space. The external space is the root of the concept hierarchy while internal spaces represent its child concepts. Instances of the root concept are represented and accessed directly using some built-in RA mechanism (as if they were OOP objects). Instances of internal concepts are represented and accessed using their parent concepts.

Informally, the difference between classes and concepts is analogous

to that between real numbers and complex numbers. Class reflects only an explicit (real) side of software system while concept is able to describe both sides by combining in one construct one object class and one reference class. Reference class of concept describes invisible hidden functionality of a software system like imaginable part of complex numbers. In the same way as complex numbers are much more expressive and natural for mathematical tasks, concept is much more expressive and natural for computer programming.

2 Concept Inclusion Hierarchy

Concept is a generalization of conventional classes defined as a program element consisting of two parts: (i) an *object class* with instances called *objects* and passed by references, and (ii) a *reference class* with instances called *references* and passed by value. For example, Table 1 defines one concept with name `MyConcept`. Its object class has one field referencing an instance of `OtherConcept` (line 2), and reference class has one integer field (line 8) intended for identifying objects of other classes. Both object class and reference class define `myMethod` with different implementations (lines 3 and 9). Note that we do not know here what is the format of field `ref` (line 2) because it depends on how `OtherConcept` is declared. It is a general principle of the concept-oriented programming that reference format and access methods depend on the target class declaration. This reference could have any structure appropriate for our task or we might choose to use the default reference format (OOP approach). However, we can call methods of `OtherConcept` (line 4) as usual and all resolution and other intermediate functions will be executed automatically.

Table 1. Concept is a pair of object class and reference class.

```
00  concept MyConcept in ParentConcept
01    class { // Object class
02      OtherConcept ref; // Indirect reference
```

```
03      int myMethod() { // Incoming method
04        return ref.getInt();
05      }
06    }
07    reference { // Reference class
08      int id; // Identifies other objects
09      int myMethod() { // Outgoing method
10        return context.myMethod() + 5;
11      }
12    }
```

Objects are never accessible directly because they permanently exist in some kind of storage. Their position is physical in the sense that it cannot be changed. Objects are accessed via their representatives in the form of references. References on the other hand do not have a permanent position in space. They are travelling elements that move between different points by value. This property reflects the existence of two realities: (i) storage with its address system as a set of permanent addresses, and (ii) a system of information transfer which allows for interactions to be propagated all over the space of objects.

References are elements that exist and can be manipulated only by value. In other words, references do not have their own references and hence represent themselves. References are coordinates in some space or elements of an address system where the address system is a space of objects. For any space to exist two elements are needed: this space itself as an object and its addresses as references. Normally for one object there exist many references. An object can be thought of as a scope or space instance while its references are concrete addresses within this space. A concept then is aimed at describing both space structure and its address structure. In other words, object class describes how the space will look like and how it will function while reference class describes the format of addresses within this space. For example, a country could be viewed as a space where addresses are city names. Its concept then could be written as follows:

```
00  concept Country
01    class { String countryDescription; }
02    reference { String cityName; }
```

This concept means that there can be many country objects each having some description. A country then defines its own internal coordinate or address system in reference class. According to this address system any object within one country is identified via some unique city name. Note that here we do not know how countries themselves are identified because concepts define only their internal coordinate system for which they are responsible.

Another example is where we define our own memory manager where objects of any type can be stored. Concepts allow us to define such a storage at high level as an abstract space with its own name and then take responsibility for everything that happens inside this space. In particular, we can define characteristics of the space itself in object class and its address format in reference class. The current number of internal objects is kept in a field of the concept object class. Internal objects themselves are identified by unique integers:

```
00  concept MemoryManager
01    class { int objectCount; }
02    reference { int objectId; }
```

We may have many memory managers and then many objects of this concept will be created. Each such memory manager may create many references each of them representing some internal object. Note again that references are not objects because they are passed by value and hence do not have a position in space (their own reference). In contrast, objects have a position in space which is represented by some reference.

Using concepts we can define the format of objects and format of references that are intended to represent objects. Then the question is how do we determine what references represent what objects? For example, what references are used to represent country objects and what references are used to represent memory manager objects in the previous examples?

In order to solve this problem we use the mechanism of *inclusion relation* which means that each concept is included into a parent concept. Thus the whole program is not simply a number of concepts but rather a hierarchy of concepts. In this hierarchy any concept has one parent (explicitly defined) and a number of children (not directly known in its declaration). For example, in Table 1 `MyConcept` is included into `ParentConcept` using keyword '`in`'.

In order to determine what references represent what objects we use the following principle: *an object is represented by its parent concept reference*. This means that there is one-to-one correspondence between this concept objects and its parent concept references. Since references are passed by value they can be used to represent objects in other points of space. For example, in Table 1 all instances of `MyConcept` will be represented by instances of `ParentConcept` reference class. If we want countries to be represented by their country code then concept `Country` has to be included into the following parent concept:

```
00  concept CountryCode
01    class { int countryCount; }
02    reference { String countryCode; }
03  concept Country in CountryCode;
```

This means that all country objects will be represented by means of the corresponding country code. For example, variables that reference Germany will store "DE" as their values. This value will be passed to methods as parameters, returned from methods and stored in local variables and object fields. If we want our memory managers to be represented by long integers then concept `MemoryManager` has to be included into a parent concept with the long integer field in its reference class.

An advantage of such an approach is that the programmer specifies and can change the parent concept declaratively. For example, if we want to make a remote memory manager then we simply change its parent concept which supports remote references. The memory manager itself as well as all its uses in the program need not to be changed.

It is assumed that there exists one *root concept* provided by the

compiler, interpreter or an execution environment while all program concepts defined by the programmer are directly or indirectly included into the root. There may be more than one root in the case the compiler provides several standard RA mechanisms, for example, local heap, global heap, managed objects, persistent objects, remote objects etc. Classes and concepts included into the root concept are represented using the system default RA mechanism like memory handles or Java references. Conventional OO program can be viewed as consisting of classes included into the root concept. The root is normally a static concept with a single well known object instance. This is why it does not need a (dynamic) reference and needs not to be resolved. If a parent concept is not specified then by convention it is assumed to be the root concept. The compiler however needs to know what default RA mechanism to use for the root. For example, if we want all objects to be finally represented by memory handles allocated by the operating system in global heap then the root concept will look as follows:

```
00  concept Root
01    class { AllocationTable allocationTable; }
02    reference { long memoryHandle; }
```

This system level concept defines its reference class as consisting of one long field and hence all objects at this level will be represented by unique long integers. For example, assume that our custom memory manager is included into such a root concept:

```
00  concept MemoryManager in Root
01    class { Map objectIdToMemoryHandle; }
02    reference { int objectId; }
```

This concept will represent all objects of its child concepts by means of integer identifiers. However, each such custom identifier will replace some root memory handle. The mapping between integer identifiers and long memory handles is stored in the field of this concept object class. If we want our target objects to be managed by this custom memory manager then we simply include the target class or concept into it:

```
class MyTargetClass in MemoryMangaer
```

After that all instances of this child target class will be represented indirectly by means of its parent references (integer identifiers). Let us consider the following code:

```
00  void myMethod(MyTargetClass param) {
01    param.targetMethod();
02  }
```

Here the method parameter has a class that is included into the custom memory manager concept. Hence this parameter will be passed by using integer values. If we need to call some method of this object then this reference (integer value) has to be resolved into its own parent reference. In our case the integer value has to be resolved into some long integer which is a memory handle allocated by the root. The root reference is then used to make a *direct* method invocation. After that the access procedure returns.

An object (instance of this concept object class) where a reference was created is referred to as *context*. Context and its references belong to the same concept. We say that references exist in some context and one context may create many references. The current context is available in the program via keyword '`context`'. For example, line 10 in Table 1 accesses a method defined in the object class of this concept (line 3).

If an object class is static with no instances (with a single instance known at compile time) then such a concept is also said to be static. If a concept has no reference class defined then it is a normal class.

A typical object run-time structure is shown in Fig. 2. Dashed boxes represent contexts (objects of concepts) while black boxes are references within these contexts. The outer most dashed box is the root context, i.e., an instance of the root concept (such as the system default memory manager). The root context has several root references which represent internal child contexts at different depth (not necessarily direct children). In this example there are two child contexts belonging to one concept `MemoryManager`. (In general case there are

many child concepts each crating many instances.) Thus there exist two memory managers each managing its own set of objects. For example, the first memory manager allocated two integer references in order to represent two internal objects. However, these internal objects have their own internal objects and so on.

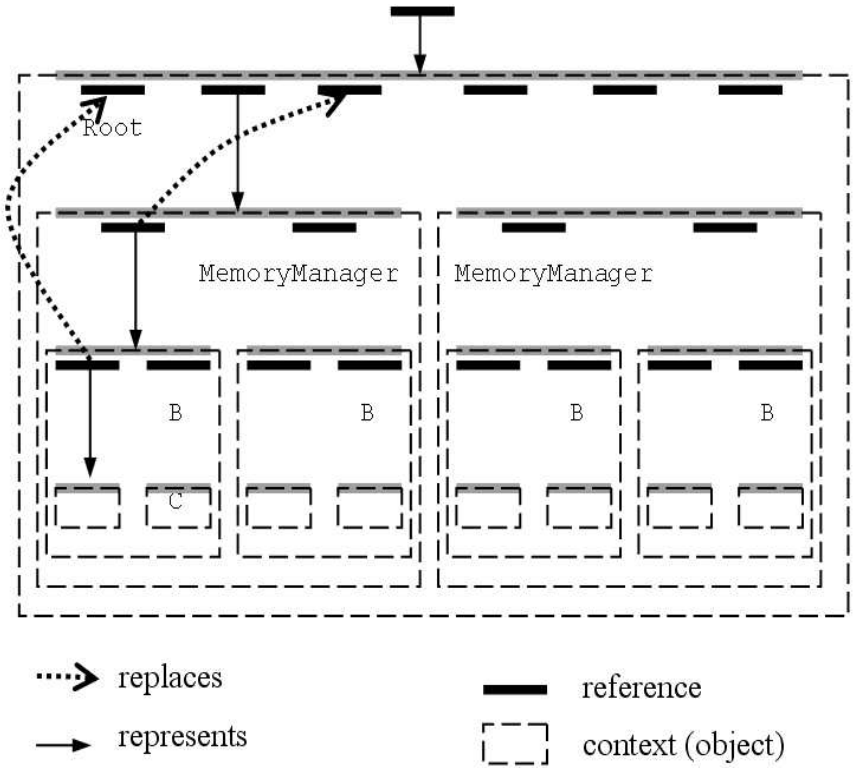

Figure 2. Context structure

It is important that any reference (black rectangle) replaces some parent reference (denoted by upward dot line pointing to a replaced parent reference). At the same time a reference represents some child object (denoted by downward line pointing to represented child object). In both cases these relationships are not necessarily direct. That is, a

reference may replace a parent reference of higher order including some root reference. And a reference may represent a child object of higher order including some target object.

3 A Sequence of Access

3.1 Reference Substitution and Resolution

The system root concept provides default format for object references which are used to *directly* represent and access all objects in the program. Here direct RA does not mean that the objects will be really accessed instantly. Rather, by direct access we mean that the programmer is not able to influence this level of RA functionality. For example, we say that Java references provide direct access because the programmer is unaware of the underlying RA mechanism which actually can be rather complex. Even physical memory addresses do not provide the ultimate direct access because they are processed at hardware level and each access requires a number of hardware clock cycles. However, in a program we can assume that such hardware addresses are used for direct access because all the program objects are resolved into them.

One important use of concepts consists in indirecting object representation and access in the program by describing custom format of references and custom access procedures. Program objects are still represented by the root references however these root references are not stored and passed anywhere in the program as representatives. Instead, objects are represented by means of custom references which replace the corresponding root references. Since concepts are organized into a hierarchy, the reference substitution has a nested nature. This means that a child reference replaces some parent reference which in turn replaces its own parent reference and so on till the root which provides direct access to the target object. For example, a street within a city might define its own local notation in order to identify houses. However, in order for the basic access mechanism to work we need to map these local identifiers into the parent city-level identification format. Thus each street-level local identifier will replace some city-level parent

identifier for a house. The same substitution mechanism can be used in custom memory managers which can introduce its own local format for object identification. However, these local identifiers replace parent system-level identifiers. Each memory manager is then responsible for storing this mapping and resolving its identifiers.

Table 2. Reference resolution.

```
00  concept A in Root
01    class { static Map map; }
02    reference {
03      int id;
04      void continue() {
05        Object o = context.map.get(id);
06        o.continue();
07      }
08    }
09  concept B in A
10    class { static Map map; }
11    reference {
12      String id;
13      void continue() {
14        A a = context.map.get(id);
15        a.continue();
16      }
17    }
```

For example, suppose that we want to develop a mechanism for representing our program objects by means of integer values (that replace the system default references). This means that all variables, method parameters, return values and object fields will store integers for those objects rather than the default references. These integer references will live in their own context which is included into the root context. In

Table 2 such a mechanism is described as concept `A` (line 0-8). Its reference class has only one integer field (line 3) that is used to identify child objects. Thus if we include any class or concept into `A` then all its instances will be represented by integers, which will be passed by value as the object representatives. It is important that these integers will replace parent references. In Table 2 integer references of concept `A` replace references provided by `Root` which have unknown format because they are provided by the compiler. For example, in Java integer reference of concept `A` would replace Java references.

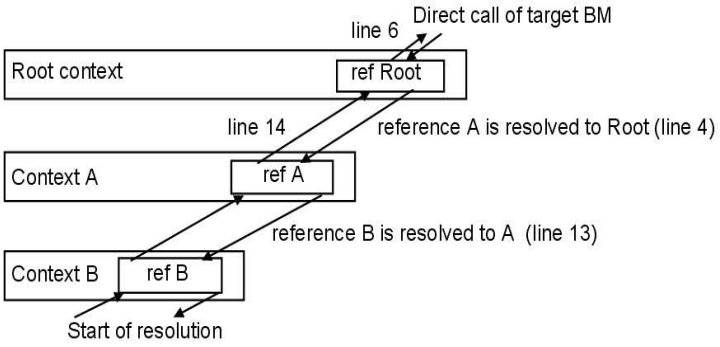

Figure 3. A sequence of reference resolution

Such a substitution is shown in Fig. 3 where the root context has one reference which directly represents some target object in a child context. However, instead of this root reference the compiler will use its substitute of integer type created in context `A`. Thus the root reference is actually not used anywhere in the program. If we declare a new child concept `B` and include it into `A` (line 9 in Table 2) then the substitution will have a nested character. Concept `B` uses text strings to identify its child objects, that is, any child object will have a text string assigned to it, which is unique within this context. This text string replaces some parent reference of integer type which in turn replaces some root reference providing a direct access to the target object. This hierarchy can be developed further by defining new concepts and including them into the existing ones. The main idea however remains the same: any

child object will be automatically represented by its parent concept reference which replaces some root reference.

It should be noted that it is not necessary that a reference replaces its direct parent reference. In particular, a reference can replace its root reference. For example, in Table 2 and Fig. 3 concept `B` might well be designed in such a way that string identifiers replace root references (rather than integers of concept `A` which then replace root references). It is important also that reference substitution is performed and makes sense within some concrete context only. In particular, the context stores all the information that is necessary to resolve indirect references into their parent counterparts. In Table 2 such information is stored in a field of the concept object class (lines 1 and 10 for concepts `A` and `B`, respectively).

Concept hierarchy is intended to describe a reference substitution order where child references replace their parent references. In this way the programmer can describe internal space with its own local coordinate system that indirects the parent coordinate system. However, one of the design goals of the concept-oriented approach is that this indirection mechanism has to be transparent when it is used (DG2). In other words, when we use our target objects we do not need to know how they are represented and what is necessary to do in order to access them. In particular, we do not need to know the format of their references and how these references are resolved. For example, if we want objects of class `C` to be represented by text strings then we include it into concept `B`. After that we use objects of class `C` as usual and it is the parent concept that is responsible for the resolution of custom references.

The mechanism of reference resolution is implemented at the level of each concept that defines its own references. Thus each concept that defines its references which replace parent references is also responsible for their resolution whenever some target object needs to be accessed. Such a resolution is implemented in the special method of reference class called `continue` (lines 4 and 13 in Table 2). The role of this method consists in providing a door or portal between level. In this sense it is not a normal method in the sense of object-oriented

programming but rather a mechanism for border intersection and context change. It is declared as a method because its implementation is provided by the programmer who decides what should happen when a process intersects this border (not necessarily only reference resolution). The continuation method takes no parameters and returns no value. It is applied to a parent reference in order to continue the current process in the parent context.

Whenever an indirectly represented object is going to be accessed, its reference is automatically and transparently resolved by the continuation method. Reference resolution (continuation) method is applied to an instance of reference class and executes in the context of its object class. For example, continuation method of reference `A` executes in the context of an object of class `A` while continuation method of reference `B` executes in the context of some object of class `B`. Let us assume that class `MyTargetClass` is included into concept `B` and then its business method `targetMethod` is called from somewhere in the program:

```
00  MyTargetClass in B {
01    void targetMethod() { ... }
02  }
03
04  void myMethod(MyTargetClass param) {
05    param.targetMethod();
06  }
```

Notice again that when we call the target method (line 5 above) we do not know how this object is represented and how it will be accessed. It is the task of compiler to find all the parent concept and use their functionality to organize the resolution procedure. In our example the target object is included into concept `B` and hence it will be represented by a text string. When a method of this object is invoked the compiler needs to resolve this text string into the corresponding root reference and then make a direct method call. Thus the compiler applies continuation method of reference `B` to the reference representing the target object (start of resolution in Fig. 3). The continuation method (lines 13-16 in Table 2) has to decide how to resolve this reference (text

string). In our example, it restores the replaced parent reference given the value of this reference using information from the context (line 14). When the parent reference is restored it simply passes the control by invoking the parent continuation method (line 15). This means that the process intersects the border and proceeds in the parent space (in concept `A`). Continuation method of reference `A` is implemented in a similar manner. Its task consists in restoring a root reference given some integer value as a key. When the root reference is found it again proceeds by invoking the parent continuation method (line 6). However, in this case it is a root reference and hence its implementation is provided by the compiler. We do not know what concretely happens in the root continuation method however its task consists in calling the target business method (direct call of target BM in Fig- 4). It is possible now to make a direct call because the object reference is completely resolved and this is precisely what happens when we call a method in OOP. When the target method finishes the whole procedure returns and the continuation method can execute some clean up procedures.

It should be noted that the continuation method is provided by the programmer and can use any resolution strategy or include any other necessary code. It is important only that the compiler will follow the concrete sequence of steps when an object is going to be accessed. In particular, the continuation method may include more complex logic then simply object resolution. Its general purpose consists in providing a mechanism for border intersection and code that will trigger automatically whenever a process wants to intersect this border. This is precisely the code that is hidden in the conventional object-oriented programming.

3.2 Context Resolution

In the previous section we described how references are resolved in continuation method using information from the current context. However, one problem is that contexts in most cases are not static. Rather, they are normal objects with their own references as shown in Fig. 2. For example, when an integer reference is being resolved we need to ac-

cess information from its memory manager which itself is represented by its parent reference. In particular, lines 5 and 14 (Table 2) cannot be directly executed because each context is represented by its parent reference (just like any other object).

An important conclusion is that it is not enough to store only an object parent reference as has been described in the previous section. For complete representation it is necessary to store also references to all the parent contexts of the target object. For example, it is not enough to store only a street name because it is specified relative to its city (context) which in turn is specified relative to its country and so on till the root context (which is static).

Such a hierarchical approach to object representation is implemented via the mechanism of *complex references*. A complex reference is a sequence of several reference *segments*. Each segment is an instance of one reference class. The very first (high) segment is of root type and represents the first context within the root where all child objects live. The next segment is of child concept type and so on till the target class. For example, in Table 2 a target object of `MyTargetClass` included in concept `B` would be represented by three segments: high root segment representing context `A`, middle segment of integer type representing child context `B`, and low string segment representing the target object. Such a reference might be equal to `<0x123, 10, "objectUniqueName">` where `0x123` is the value of the root (system default) reference, 10 is the value of reference `A` and `"objectUniqueName"` is the value of reference `B`.

What happens if we get a reference of target class `MyTargetClass` and then call some its method `c.myMethod()`? In the previous section we described this process as a resolution of the target reference into the corresponding root reference using information in the intermediate contexts. However, now our object is represented by three segments rather than only one low segment. The first two segments represent the two intermediate contexts. One approach solving this problem consists in resolving these intermediate contexts each time we need to access them from the child context. However, this technique is rather inefficient because context is supposed to be used very intensively. An

alternative approach consists in changing the sequence of access. Now high segments are resolved before low segments and the result of the resolution is accessible from all child contexts. In other words, the procedure described in the previous section is repeated for each segment of the complex reference starting from the high segment and ending with the last low segment representing the target object. The main advantage is that parent contexts are guaranteed to be resolved and directly accessible from any child context.

In our example shown in Table 2 the compiler determines that the target object of `MyTargetClass` has three parent concepts and hence is represented by three segments. Although only the last segment represents the target object, in order to resolve it, we need its two intermediate segments. So the compiler in this situation starts from resolving the very first (high) segment. It is however of the root class and hence is already in the default system format that can be used for direct access. On the second step the compiler resolves the second integer segment of concept `A`. When this segment is resolved, the corresponding root reference represents the next child context. After that the next string segment is resolved into the root reference which represents the target object.

Table 3. A sequence of access.

```
01  concept A in Root
02    class {
03      Map map;
04      void continue() { // Incoming method
05        continue(); // Outgoing (reference) method
06      }
07    }
08    reference {
09      int id;
10      void continue() {
11        Object o = context.map.get(id);
```

```
12        o.continue();
13      }
14    }
15
16  cconcept B in A
17    class {
18      Map map;
19      void continue() { // Incoming method
20        continue(); // Outgoing (reference) method
21      }
22    }
23    reference {
24      String id;
25      void continue() {
26        A a = context.map.get(id);
27        a.continue();
28      }
29    }
```

Such a sequence of access (from high to low segment) is supported by a special method of object class called `continue`. Note that this method has the same name as the method of reference class. In other words, each concept has two continuation methods: one defined in reference class used to resolve one segment (described in the previous section), and another defined in object class used to enter just resolved context. (The same is true for any other method of concept.) Object continuation method takes no parameters and returns no value (line 4 and 19 in Table 3). It is called after this object is resolved and is going to be accessed, that is, after the border intersection. Using this continuation method the process enters the scope of the object after its parent reference is resolved. In contrast, reference continuation method (line 10 and 25 in Table 3) is applied to reference and its role consists in resolving it to a system default reference that represents a child object and can be used for direct access.

A generic sequence of access using two continuation methods is

shown in Fig. 4. A target object of class `C` is represented by three segments 1, 2 and 3 created in contexts `Root`, `A` and `B` respectively. After entering the root context it is necessary to resolve segment 1 which represents the next context. However, it is already resolved because all root references (1, 4 and 6 in this example) are direct and have system default type. Therefore we follow a dash line and enter the child context `A` using its continuation method (line 4 in Table 3 and thick arrow in Fig. 4).

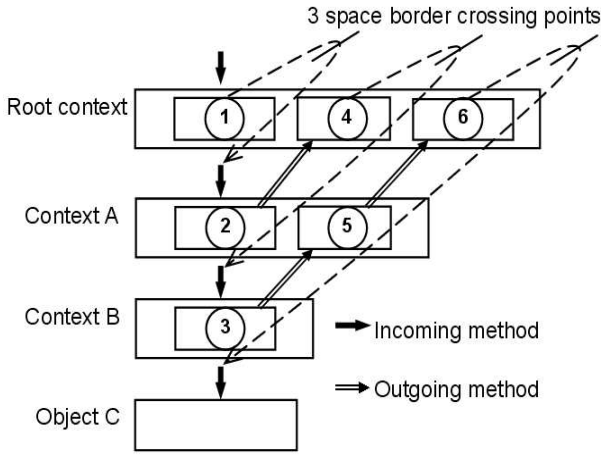

Figure 4. A sequence of context resolution

This method does nothing in our example and simply proceeds by resolving its own reference (line 5 and double arrow from 2 to 4). Root reference 4 is a direct representation of the next child context `B` so we again follow a dash line and get into the context `B` using its object continuation method (line 19 and thick line). Here we need to resolve the last segment 3 by applying continuation method of its reference (called from line 20 and defined in line 25). This method resolves reference 3 to reference 5 of its parent concept `A` using information in the context (line 26) and then calls the parent resolution method (line 27) which resolves 5 to root reference 6 in the same way. Thus reference 6 is a direct representation of the target object and we can enter its

scope and call its business method specified in the source access request. Note that the sequence ⟨reference 3, double arrow, reference 5, double arrow, reference 6⟩ is the resolution process described in the previous section, which uses reference continuation method.

3.3 Dual Methods

In the previous sections we mentioned that both reference class and object class of concept may define a method with the same name. However, in the concept-oriented program a method is invoked by using only its main name with no indication whether it belongs to a reference class or an object class. Then the following question arises: which of two versions has to be executed? One example has been considered in the previous section. The special continuation method defined by the programmer and used by the compiler to organize transparent access is defined both in reference class and object class. The reference continuation method is used to resolve the current reference while the object continuation method is executed when the process enters the current context. It is also possible to define any normal method of a concept as consisting of two parts: one within reference class and the other within object class. Methods of object class are referred to as *incoming* because they are visible from outside and are executed when any process enters this object scope (double line arrow in Fig. 5). Objects of reference class are referred to as *outgoing* because they are visible from inside and are called when this object is used by its child objects (thick arrow in Fig. 5). For example, `MyConcept` in Table 1 has incoming method (line 3) and outgoing method (line 9) which have the same name.

One principle of the concept-oriented programming is that functionality is concentrated on space borders and is automatically triggered whenever a process intersects it. The idea of dual methods is that we want to separate this functionality depending on the direction in which the process intersects the border. At the same time we want to have only one method name for both functions. This allows us to call object methods by specifying as usual only its method name while one of two

versions will be automatically chosen depending on the relative position of the source context. If we are calling the target object method from outside then one version will be executed. If the method is called from inside then the other version will be executed.

Using concept inclusion hierarchy this principle means that one of two versions will be chosen depending on whether the target object method is called from some its child object (inside) or from any other object. Thus any object defines a border or scope and if a method call comes from the side of its child object (lower part in Fig. 5) then the version defined in reference class is executed. If a method call comes from the side of its parent object (upper part in Fig. 5) then the version defined in the object class will be executed. However, in the source context the method call looks the same. We say that if an object is not yet reached and its reference is not resolved then the process is outside. As soon as the object reference is resolved the process intersects the border and gets inside this scope.

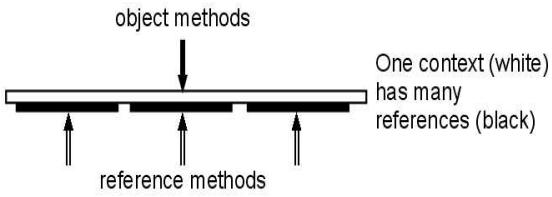

Figure 5. Object methods and reference methods of concepts

Another important idea behind dual methods is that applying a method to a reference is not considered a specification of concrete code to be executed. Methods in the concept-oriented programming are thought of as symbolic names for doors in space borders. Each intermediate border (concept) can define several doors with their symbolic names and one universal door via continuation method. Each door has two directions, name and code assigned to it. This code will be automatically executed whenever an access request with this method name intersects it. Thus method invocation in COP is a way to specify a door through which it is necessary to go rather than a concrete action

or code. Reference in this case contains a path to the target.

Below some other properties and uses of dual methods are described:

- Object methods are called from parent objects (from outside) while reference methods are called from child object (from inside).
- Object method is called after complete resolution of its parent reference, i.e., it is the first method to be executed after this object reference has been completely resolved. Entering object method means intersecting this object border.
- Object methods serve external objects while reference methods serve internal (child) objects.
- Object method can delegate or forward its task to its internal object and all requests to child objects pass through the parent objects. This is analogous to intercepting methods of extensions by base objects in OOP.
- Reference method implements functionality exposed to its child objects. This is analogous to implementing methods of extensions using base methods.
- An object accepts method calls from outside using methods of object class and then serves its child objects by providing methods of reference class.

One of the most important mechanisms that can be implemented using dual methods is life-cycle management. At least two functions are needed to implement this mechanism:

- object initialization/clean up, and
- reference initialization/clean up

Object initialization method is normally called constructor while object clean up procedure is called destructor. Reference initialization

and clean up procedures do not have their own conventional names because this mechanism is not part of OOP, which provides RA functionality as a default mechanism. In the concept-oriented programming both types of life-cycle management methods are equally important: object creation/deletion and reference creation/deletion. In this situation dual methods are precisely what we need to implement this mechanism. Namely, each object class has methods `create` and `delete` which are responsible for construction and destruction (initialization and cleaning up). On the other hand, each reference class has the same pair of methods (line 6, 22 for creation and 11, 27 for deletion in Table 4) that are responsible for reference allocation and deletion. The reference creation method is applied to a new (empty) instance of reference class. After return this reference has to contain some correct value identifying the child object for which it has been created.

Object creation and deletion follow the standard sequence of access described in the previous section (Fig. 4). An example of reference creation and deletion is shown in Table 4 (object creation is a constructor which simply initializes object fields). Just like the method of continuation, creation and deletion methods take no parameters and return no values. Assume that the `MyTargetClass` is included into concept `B` and we create its instance as follows: `MyTargetClass target.create()`. This statement is equivalent to the conventional `MyTargetClass target = new MyTargetClass()`. Variable `target` will contain a reference of parent concept `B`, i.e., a string identifier as last segment. In order to initialize this reference the compiler will apply creation method of `B` (line 22). This method will generate a unique string in order to identify new object in the current context (line 23). Then it allocates a parent reference and calls the same method of creation to initialize this new empty reference (line 24). Finally the creation method remembers an association between this id and the parent reference within the current context (line 25). This information can be then used whenever this object needs to be accessed, for example, from reference continuation method.

Line 24 is a reference creation method which is applied to empty reference of concept `A`. It is implemented in the same way. First, it

generates a unique integer (line 7), then it creates a parent reference (line 8) and finally it remembers an association between them in the current context (line 9). Line 8 is the most important because here we create a root reference which directly represents the target object being created. When line 8 is being executed the target object is really created at the system level. It is precisely the moment when the target object constructor (its object creation method) is called. Notice that if it were a normal business method call then at this moment the compiler would call the target object method. So the target object constructor plays the role of normal business method because it is the last method that is executed in the sequence of access. It is important that object constructor is executed after its root reference has been created and before the creation procedure returns. In other words, lines 9 and 25 are executed after the target object constructor.

Reference deletion is performed analogously. If we need to delete an object then deletion method is applied to its reference: `target.delete()`. If the reference consists of several segments then they need to be resolved as usual starting from high segment and ending with the last low segment which represents the target object. When the last segment (concept `B`) is reached, its reference deletion method is called (line 27 in Table 4). Here we resolve this reference and find its parent reference of concept `A` (line 28). Then this parent reference is deleted by calling the same deletion method of parent concept (line 29). And finally information about this identifier and the parent reference is deleted from the context (line 30). After that this reference is invalid and cannot be used to access the target object. Line 29 is a call of the parent deletion method (line 11) which works similarly. Line 13 is the most important here because it is where the root reference and hence the target object is really deleted (after its destructor).

Table 4. Creation and deletion.

```
01  concept A in Root
02    class { static Map map; }
```

```
03    reference {
04      int id;
05      void continue() { ... }
06      void create() {
07        id = context.getUniqueInteger();
08        Object o.create();
09        context.map.add(id, o);
10      }
11      void delete() {
12        Object o = context.map.get(id);
13        Object o.delete();
14        context.map.remove(id);
15      }
16    }
17  concept B in A
18    class { static Map map; }
19    reference {
20      String id;
21      void continue() { ... }
22      void create() {
23        id = context.getUniqueString();
24        A a.create();
25        context.map.add(id, a);
26      }
27      void delete() {
28        A a = context.map.get(id);
29        A a.delete();
30        context.map.remove(id);
31      }
32    }
```

4 Uses of Concepts

4.1 Concept as a Generalized Class

Concept as a programming construct is a pair of one object class and one reference class with special responsibilities described in the previous sections. In such a form it is a rather general instrument that has many possible uses. In other words, concept can be applied in very different forms in very different programming languages depending on the goals. One possible application of concepts is interpreting them as a generalized class. The idea is that the conventional object-oriented programming languages can be then used as usual except that concepts are used instead of classes. Concept inclusion is interpreted as a generalized inheritance. Notice that if all concepts have only an object class with empty reference class then we get the conventional object-oriented case.

One important new property of the concept-oriented approach is that each intermediate object in the hierarchy has its own reference and life-cycle. In contrast, in OOP an object has always only one reference independent of the number of its base objects. For example, if class `Circle` inherits class `Figure` then in OOP all instances of class `Circle` will have a reference which is also valid for its base object of class `Figure`. In COP in general case it is not so. For example, if concept `Figure` is included into concept `Panel` then all panels will have their own unique references which are independent of references allocated for figures. For each panel identified by some reference there can be many figures with their own references allocated by its parent panel. Within one concrete panel, figure objects are identified by their short (local) references while outside in global scope figures need to be identified by their long references consisting of two segments. In larger scope we might need even more segments. For example, if panels are included into windows then each figure reference consists of three segments starting from window reference.

Internal objects have also their own independent life-cycle. We can create and delete internal objects independent of their parent objects. In OOP it is not so, and creating/deleting an object means creat-

ing/deleting all instances starting from the root class and ending with the last extension. Independent life-cycle is maintained by means of dual creation and deletion methods. If a concept is being developed to maintain its internal coordinate system then it should define the corresponding methods for reference creation and deletion that will serve its child objects. Then its child objects will have references allocated by the parent which will be also responsible for their resolution via reference continuation method.

In the concept-oriented programming the role of concepts changes significantly with respect to the role of classes. The main role of base classes is object and functionality reuse. This means that base classes are developed as pieces of generic functionality that can be then inherited and extended from child classes. The main role of concept in COP consists in implementing a scope or space border with associated functionality. Then any object that is created within this concept inherits this behaviour and can use it at run time. Objects are created and function within a hierarchical space which determines many their properties. In OOP it is done statically at compile time while in COP this inheritance of behaviour and influence of context is performed dynamically at run time. As a scope or space border any concept has to behave like an intermediate environment rather than an end point in OOP. The main goal of such an environment consists in processing incoming and outgoing access requests (method calls). For example, a concept might accept an incoming request from outside in its object method and after some processing continue its execution in a child object (delegation):

```
00  concept Intermediate
01    class {
02      int total=0;
03      int requestCount=0;
04      void someMethod(int amount) {
05        total += amount;
06        requestCount++;
07        continue();
08        requestCount--;
```

```
09      }
10    }
11    reference { ... }
```

Here we count the number of method calls which are currently being processed in variable `requestCount` (line 3) and also sum up all the method parameters in variable `total` (line 2). Both variables are stored in the current context (object class) so we always know how many internal objects are being currently accessed via this method (door in the border). Additionally we can determine the sum of all parameters passed to this method. Line 7 is where we pass this request to the internal object for additional processing that is specific to the extension. Here we clearly see that this method implements an intermediate functionality that is however activated automatically.

4.2 Concept as an Active Namespace

Conventional namespaces are static and passive constructs that extend class naming system. One interesting use of concepts consists in interpreting concept hierarchy (without target classes) as an active and dynamic naming system. In contrast to conventional namespaces which are processed at compile time, active namespaces implemented via concepts exist at run-time. The main idea of this mechanism is that there exist two roles of concepts: concepts as target classes, and concepts as namespaces. The former role has been considered in the previous section while in this section we consider how concepts can be used as active namespaces. For simplicity we will assume that target concepts are normal classes (without reference class and other specific features of concepts). Any target class is included into some active namespace described as a concept. However, in contrast to conventional namespaces this effectively changes their behaviour at run-time because access to all class instances is intercepted and indirected by its parent namespaces. Classes included into Root namespace will be accessed directly without any intervention while each internal namespace will add its own level of indirection and its own intermediate functionality.

The easiest way to implement active namespaces consists in using

static concepts where object class has only static members and hence does not produce run-time instances. In this case each namespace can be declared as one reference class with static members belonging to object class and non-static members belonging to reference class. For example, in Table 5 we declare namespace `Persistent` (line 0). Its static fields (lines 1) belong to object class while all other (non-static) members belong to reference class. Any target object with the class included in `Persistent` namespace will be identified by a unique integer value which is supposed to correspond to this object primary key in a database. Continuation method of this namespace (line 4) is a reference method. Its task consists in transforming this primary key into the target object root reference. It has to load the target object from the database before it can be accessed (line 4) and freeing this object after the access has been finished (line 6).

Table 5. An example of active namespace.

```
00  namespace Persistent in Root {
01    static Map map; // Static member (object class)
02    int primaryKey; // Dynamic members (reference class)
03    void continue() {
04      Object o = context.restore(primaryKey);
05      o.continue();
06      context.free(primaryKey, o);
07    }
08  }
09
10  class C in Persistent {
11    void create() { ... } // Constructor
12    void delete() { ... } // Destructor
13  }
```

Using the mechanism of active namespaces the programming is reduced to designing the structure of namespace and then including target classes into them. The namespace structure accounts for a great

deal of the program functionality however its functions are used implicitly. A target class may change its behaviour depending on its parent namespace.

4.3 Applications of Concepts

Concepts are useful in applications with complex structure characterized by a great deal of intermediate functionality cross-cutting the whole system. This includes the following technologies and mechanisms:

Access interception. Frequently we need to perform some actions before the target object is reached. This can be done by defining an object class method of parent concept which then will intercept all calls. If it is necessary to intercept all calls then we define object class continuation method.

Security and object protection. Before an internal child object is reached we would like to perform some security checks. This can be done in object methods of parent concepts.

Persistence. Before a target internal object can be accessed it might need to be loaded from persistent storage or activated in some other way. This can be done in object class methods as well as in the continuation method of reference.

Debugging, tracing and logging. Incoming methods can be used for auxiliary purposes such as controlling access to objects. We can define special concepts in order to control access to internal classes by intercepting specified method calls.

Internal services. Each object may define service functions to be used exclusively from inside by internal objects by means of its outgoing methods. If classes are included into such concepts then they can use these services which are not visible from outside.

Memory and life-cycle management. We can use this mechanism to implement custom memory managers. For example, it might be necessary to create an efficient memory manager for special types of objects like a hierarchical buffer or a local heap. Persistent storage can also be viewed as a special type of memory manager.

Layered structure of containers. Concepts effectively define space borders and serve as run-time containers serving their internal child objects. Such containers are environments for their objects providing all the necessary services including life-cycle management.

Remote objects. This mechanism is very suitable for implementing network protocols and remote method invocation mechanisms. Concept can be responsible for network communication and reference resolution. Its incoming methods accept remote calls while outgoing methods provide local services for internal objects as a local context.

Protocol stacks. The hierarchical structure of concepts can be used to implement the mechanism of protocol stack which is especially useful for distributed systems. In this case a reference class describes a packet header with information about the target object position. A complex reference is viewed as a nested structure of packet headers. The first high segment of the complex reference is the first header of the external packet. The body of this packet starts from the second segment of the reference and so on. Each intermediate segment (internal packet header) represents one intermediate context. Concepts allow us to create custom protocol stack for each individual program and then use it transparently.

Lazy creation and deletion. Here object reference is initialized without the real object creation. For example, we could simply generate a unique text string as object reference and exit. And only when this object is really accessed (and we cannot resolve the string identifier) the continuation method performs the rest of the creation procedure.

Transactionality. It is convenient to develop concepts which are responsible for performing operations with internal objects as one transaction. In particular, such a concept will automatically and transparently begin a transaction for each incoming access request and end this transaction after the access is finished.

Synchronization and multi-threading. Concepts can be used to implement a complex mechanism of synchronized access to some internal resources. For example, such a concept can guarantee that only one process accesses one object. It will store a list of objects being currently accessed as well as a queue for processes waiting at the border

(in front of the door).

5 Related Work

An approach described in this paper is being developed within a new paradigm which covers several branches in computer science including programming, data modelling [1,2,3] and system design. The concept-oriented paradigm is based on several general principles that distinguish it from the currently existing theories and approaches. In the context of programming the most important concept-oriented assumption is that system functionality is concentrated on space borders. In contrast, in object-oriented paradigm it is assumed that most of functionality is concentrated in objects themselves. Concept as the main programming construct allows the programmer to describe effectively not only what happens in objects but simultaneously what happens when they are being accessed.

The concept-oriented programming can be considered a continuation of a very general and deep principle of Separation of Concerns formulated by Dijkstra [4]. The main idea of this principle is that any problem or system functionality can be viewed from different points of views or concerns. One specific feature of our approach is that we distinguish two main concerns any program consists of: BMs and RA. Currently there exist different techniques for separating business methods from representation and access functionality but they can be broken into two main categories: methods based on dedicated middleware and approaches based on programming languages.

The idea of middleware-based approaches consists in creating special software and hardware environments where a conventional program will run. Such an environment offers a number of functions that are intended to support custom RA functionality. This special environment can exist and be accessible to running programs in very different forms, for example, as part of an operating system, an object container, a service, a dynamically or statically linked library etc. However, the main property of this approach is that the programming language remains the same while the support is provided by developers of the

middleware. In particular, it is not easy to develop a new custom environment or adapt the existing environment to the purposes of each concrete program.

One wide-spread class of middleware is techniques for remote procedure calls. Examples of such middleware platforms are CORBA and RMI/EJB [5,6]. Such an environment provides facilities for creating remote references and then making transparent method calls. Although such an approach provides much more flexibility in comparison with the manual remote method invocation, they still have serious limitations. First of all we are not able to change the format of remote references and the underlying invocation protocol. Such middleware platforms may fit well to the purposes of one system but may be inappropriate for another system. Their adaptation possibilities are very limited and such an environment is separated from the rest of the program.

A more flexible approach to separating two concerns consists in using reflective environments and metaobject protocols [7,8,9]. The idea of this approach consists in providing a mechanism for changing the behavior of the language from this very language. Normally programming languages are defined in such a way that their behavior cannot be changed. In particular, we cannot change how objects are represented and accessed because it is defined at the level of the language environment. The reflective approach allows the programmer to change this environment and to influence its behavior. Such an approach can be viewed as an intermediate between middleware and programming languages because on one hand the programming language (reflective) environment is separated from the language itself like in middleware approaches. On the other hand, the programming language has special constructs for influencing and changing the environment where it will run.

In the approaches based on programming languages an environment is created within the language itself and using this very language. In other words, the program is responsible for creating and maintaining its own run-time environment. The functionality, which is normally implemented in some standard middleware, is now an integral part of the program written in the same programming language as the rest of

the system.

One wide-spread technique to automating intermediate RA functions consists in using static or dynamic proxies [10]. Proxy is a special class that emulates an interface of the corresponding target class but inserts some intermediate functionality. These intermediate functions of the proxy class are called before target methods and hence they effectively intercept all target object method invocations. The trick here consists in using proxy class instead of the target class. Thus it is not a real interception but rather a normal sequence of method calls. In other words, in the source context a reference to the proxy instance is created and hence its methods are called when it is used. Then it is the task of the proxy to decide what to do if some its method has been called. Normally, after some processing the corresponding target method is called. One disadvantage of this approach is that it requires significant manual support and is not very general. It is more a special technique or programming pattern rather than a programming paradigm. Here are other disadvantages of this approach:

- If a target class changes we need to manually change its proxy class.
- For each target class we need to develop its own proxy while in many cases proxy functions are rather general and can be used by many target classes.
- It is difficult to impose behaviour in a nested manner (creating a proxy for proxy).
- It is difficult to develop custom references which are stored by value instead of native references.

An interesting solution to the problem of developing custom references and intermediate behaviour consists in using smart pointers in C++ [11]. However, it is also a specific technique rather than a general programming approach. A more general solution consists in using the mechanism of annotations. The idea of this approach (called attribute-oriented programming) consists in marking places in code

where some intervention is needed by special tags. Other related approaches that can be used to automate intermediate RA functionality are mixins [12,13], subject oriented programming [14] and multidimensional separation of concerns [15]. All these methods allow the programmer to specify how behavioural granules (concerns) have to be distributed throughout the system.

Probably the most interesting approach to solving the problem of separation of concerns is aspect-oriented programming (AOP) [16]. Aspects describe intermediate functionality (and data) injected into the points in the program which are specified by means of regular expression. Thus aspect can be viewed as a special programming construct that modularize intermediate functionality. An important property of this approach is that aspects know explicitly the points where the intermediate functions will be injected while the target classes do not know what other code will modify their behaviour (Fig. 6). Such a structure of relationships between the module with the code to be injected and the modules where it has to be injected can be viewed as declaring in an aspect all the target classes (the target classes being unaware of this aspect). In this sense our approach is characterized by the opposite direction of this relationship (see DG5). Namely, the module with the code to be injected is unaware of the points where it will be used (the target modules). These are the target modules that declare the modifications they need.

6 Conclusions

In the paper we introduced a new programming construct called concept. Concept is defined as a pair of one object class and one reference class having their own fields and methods (possibly with the same name). Concepts are organized into a hierarchy using inclusion relation with the main purpose to specify how objects have to be represented and accessed. The main idea is that an object is represented by its parent reference which replaces a system default reference. An approach to programming based on this new construct is called concept-oriented programming. This approach assumes that a system consists of two

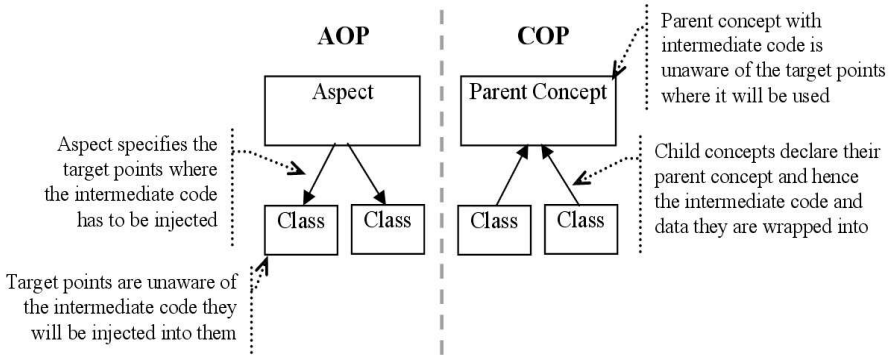

Figure 6. Aspect-oriented programming vs. concept-oriented programming

types of functionality: target BMs and intermediate RA functionality. Accordingly, it is important to be able to implement both types as an integral part of one program using one programming language. This new approach to programming can be applied to very different complex problems such as access control and interception, security and object protection, persistence, debugging, tracing and logging, memory and life-cycle management, containers, remote objects, distributed computing, protocol stacks and many others.

References

[1] Savinov, A. *Principles of the Concept-Oriented Database Model*, Institute of Mathematics and Informatics, Academy of Sciences of Moldova, Technical Report, 54pp., November 2004.

[2] Savinov, A. Hierarchical Multidimensional Modelling in the Concept-Oriented Data Model, Proc. the 3rd international conference on Concept Lattices and Their Applications (CLA'05), Olomouc, Czech Republic, September 7-9, 2005, 123–134.

[3] Savinov, A. Grouping and Aggregation in the Concept-Oriented Data Model, ACM Symposium on Applied Computing (SAC 2006), April 23-27, 2006, Dijon, France (accepted).

[4] Dijkstra, E.W. *A Discipline of Programming.* Prentice Hall, 1976.

[5] Roman, E., Sriganesh, R.P., Brose, G. Mastering Enterprise Java Beans. Wiley; 3 edition.

[6] Enterprise JavaBeans Technology, http://java.sun.com/products/ejb/

[7] Cazzola, W., Ancona, M. mChaRM: a Reflective Middleware for Communication-Based Reflection. Technical Report DISI-TR-00-09, DISI, Universita degli Studi di Genova, May 2000. 29 pages.

[8] Kiczales, G., Rivieres, J., Bobrow, D.G. The Art of the Metaobject Protocol. MIT Press, Cambridge, 1991.

[9] 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, The MIT Press, Cambridge, MA, 1993.

[10] Blosser, J. Explore the Dynamic Proxy API, *Java World*, November 2000. http://developer.java.sun.com/developer/technicalArticles/DataTypes/proxy

[11] Stroustrup B. *The C++ Programming Language*, Second Edition, Addison Wesley, 1991.

[12] Smaragdakis, Y., Batory, D. Implementing layered designs with mixin-layers. *Proc.ECOOP'98*, 550–570, 1998.

[13] Bracha, G., Cook, W. Mixin-based inheritance. *Proc. OOPSLA/ECOOP'90*, ACM SIGPLAN Notices, 25(10), 303–311, 1990.

[14] Subject-Oriented Programming, http://www.research.ibm.com/sop

[15] Multi-Dimensional Separation of Concerns, http://www.research.ibm.com/hyperspace/MDSOC.htm

[16] 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, Jyvaskyla, Finalnd, 1997.

Alexandr Savinov,
Institute of Mathematics and Informatics,
Academy of Sciences of Moldova
str. Academiei 5,
MD-2028 Chisinau, Moldova
E–mail: *savinov@conceptoriented.com*
Home page: *http://conceptoriented.com/savinov*

Received December 21, 2005