Concept-Oriented Programming
Classes and Inheritance Revisited

Alexandr Savinov
Hauptstr. 25, 01097 Dresden, Germany

Keywords: Programming Paradigms, Classes, Inheritance, Polymorphism, Cross-Cutting Concerns.

Abstract: The main goal of concept-oriented programming (COP) is describing how objects are represented and accessed. References (object locations) in COP are made first-class elements responsible for many important functions which are difficult to model via objects. COP rethinks and generalizes such primary notions of object-orientation as class and inheritance by introducing a novel construct, concept, and a new relation, inclusion. They make it possible to describe many mechanisms and patterns of thoughts currently belonging to different programming paradigms: modeling object hierarchies (prototype-based programming), precedence of parent methods over child methods (inner methods in Beta), modularizing cross-cutting concerns (aspect-oriented programming), value-orientation (functional programming).

1 INTRODUCTION

Object orientation is one of the most influential and successful paradigms in computer science. Objects have always been in the center of this methodology (hence its name) according to which it is object's functionality that accounts for most of the program complexity. Yet object-oriented programming (OOP) has one general drawback: it does not provide a means for describing how objects are *represented* and how they are *accessed*. Any object is guaranteed to get some kind of primitive reference and a built-in access procedure without a possibility to change them. Thus there is a strong asymmetry between the role of objects and references: objects are intended to implement domain-specific structure and behavior while references have a primitive form and are not modeled by the programmer. Programming means describing objects rather than references. This abstraction from reference mechanics is achieved by completely removing references and object access procedures from the scope of programming, and delegating these functions to the translator. In OOP, we are not able to model how objects exist, where they exist, and how they are accessed.

Concept-oriented programming (COP), first described in (Savinov, 2005), is a novel approach to programming the main general goal of which is to answer these questions by *legalizing references* and making them first-class elements of programming languages. In this sense, COP can be characterized as reference-oriented programming or programming focusing on what happens during access. COP assumes that references account for a great deal of the program complexity and their functions are at least as important as those of objects. To describe both references and objects, COP introduces a novel construct, called *concept* (hence the name of this approach). The main goal of concepts is to retain main functions of conventional classes by providing a possibility to model how objects are represented and accessed.

Classical inheritance cannot be easily adopted for concepts, particularly, because concept instances exist in a hierarchy (like in prototype-based programming). Therefore COP introduces a new relation, called *inclusion*. Its main purpose consists in modeling hierarchical address spaces by describing references consisting of several segments. As a result, objects in COP exist in a hierarchal space where each of them has a unique address with custom structure (like postal addresses). Defining program elements as consisting of two parts (reference and object) and existing in a hierarchical address space leads to rethinking and generalizing such fundamental notions as object identification, inheritance and polymorphism.

First version of concept-oriented programming, COP-I, is described in (Savinov, 2005). The next

Savinov A..
Concept-Oriented Programming - Classes and Inheritance Revisited.
DOI: 10.5220/0004082303810387
In *Proceedings of the 7th International Conference on Software Paradigm Trends* (ICSOFT-2012), pages 381-387
ISBN: 978-989-8565-19-8
Copyright © 2012 SCITEPRESS (Science and Technology Publications, Lda.)

version, COP-II (Savinov, 2008; Savinov, 2009), changes the interpretation of concepts and adds several new mechanisms. This paper describes a new major revision of concept-oriented programming, denoted as COP-III. Its main goal is to describe this programming model by using fewer general notions and more natural interpretations by simultaneously covering more programming patterns existing in other approaches.

The first major change in COP-III is that concepts are defined differently: instead of using two symmetric constituents – object class and reference class – we use only one component which models references. Instead of modeling objects explicitly via object classes, we propose a new general treatment of objects: *object is a function of its reference.*

Another important change is the use of two keywords for navigating through the hierarchy, *super* and *sub* (as opposed to using only super in OOP), and the existence of two opposite overriding strategies. In addition, COP-III introduces incoming and outgoing methods instead of using reference methods and object methods in previous versions. Incoming methods of concepts intercept requests from outside and outgoing methods intercept requests from inside. We also remove the continuation method and reference resolution mechanism from the programming model. Instead, access indirection relies on the ability of elements to intercept incoming and outgoing methods.

The paper has the following layout. Section 2 defines the notion of concept. Section 3 is devoted to describing inclusion relation. Section 4 describes how inheritance, polymorphism and cross-cutting concerns are implemented in COP-III using concepts and inclusion. Section 5 makes concluding remarks.

2 CONCEPTS INSTEAD OF CLASSES

Concepts and values. Concepts in COP-III describe *values*. In this sense, concepts are analogous to classes in C++ except that concepts do not have a possibility to get an address or reference for their instances. Like all values, concept instances are passed by-copy only and do not have any permanent location, address, pointer, reference or any other indirect representation. For example, the following concept describes a bank account:

```
concept Account {
  char[10] accNo;
  Person owner;
}
```

The first field will contain 10 characters while the second field will contain a value with the structure defined by the `Person` concept.

Dual methods. What makes concepts different from classes is the presence of two kinds of methods: *incoming* methods (marked by the modifier ‘in’) and *outgoing* methods (marked by the modifier ‘out’). Such a pair of incoming and outgoing methods with the same signature is referred to as *dual methods*. For example, if we would like to have a method for getting the current account balance then formally this functionality can be specified in the incoming and outgoing methods:

```
concept Account
  char[10] accNo;
  in double getBalance() {...};
  out double getBalance() {...};
}
```

It is not necessary to define both versions: if one of them is absent then it is supposed to have a default implementation. Dual methods are invoked as usual using only their name without any indication if it is an incoming or outgoing version. The main purpose of dual methods is performing different functions for different directions of access. If concepts are thought of as borders then dual methods are responsible for processing incoming and outgoing requests. In other words, a request originating from inside is processed by an outgoing method and a request originating from outside is processed by an incoming method. Scopes and directions of access are described in Section 3.

References and objects. One of the most important assumptions in COP is that references *are* values and hence modeling the structure of references is equivalent to modeling that of values. More specifically, references are values interpreted as locations or addresses of objects. References not only identify objects but also provide access to other values which are thought of as being stored in the object fields. Thus object fields can be defined as *functions* of references which return the same output value for the same input reference (but this association may change by using setters). An object is defined as a couple of two tuples: the first tuple, called reference or identity, is a number of values $s = \langle v_1, \ldots, v_n \rangle$, and the second tuple, called object or entity, is a number of values returned by functions defined on the reference (first tuple), $\langle f_1(s), \ldots, f_m(s) \rangle$. Thus an object is identified by its reference (which is some value) and has as many fields as it has functions in the entity. Importantly, only the identity part of an element is really transferred while the entity part is what the functions return. Therefore we say that values are accessed directly while objects are accessed indirectly. Note also that this definition makes references more important than objects because the identity part (reference) must always exist. (Reference can be empty if it is inherited from the parent as described later in this section.) If the entity part is empty then it is a value, that is, values are a particular case of objects without associated functions (an address or location without any other values stored at it).

Object fields as outgoing methods. Since references are values and concepts are used to model values, we can use concepts to model references. Concept fields specify the structure of references and concept methods specify functions returning other values interpreted as being stored in this object. In addition, we assume that *object fields are implemented by outgoing methods of concepts* which return the same result for the same reference (concept instance). Syntactically, we will define such methods as setters and/or getters. For example, bank accounts are uniquely identified by their numbers which is used as a reference. In addition, any bank account is supposed to have some balance which however should be stored in an object field. Such a field is defined using an outgoing method which returns balance depending on the account number.

```
concept Account {
  char[10] accNo;
  out double balance {
    get { return func(accNo); }
  }
}
```

Here we effectively defined a new object field, called `balance`, which can be used as usual:

```
Account acc = getAccount("Smith");
double currentBalance = acc.balance;
```

If there is no need in having custom references and object allocation mechanism then they can be inherited from some kind of primitive reference provided by the run-time environment as described in the next section.

3 INCLUSION INSTEAD OF INHERITANCE

Extending values and references. COP provides a possibility to extend an already existing concept by adding new fields and methods using an *inclusion relation* denoted by the keyword 'in'. Inclusion generalizes conventional inheritance and containment relations as well as has several new properties discussed in Section 4. If concept B is included in concept A then A is referred to as a *super-concept* and B is referred to as a *sub-concept*. Instances of B will extend instances of A, that is, an instance of B is a value with additional fields attached to an instance of A. What is new in inclusion is that it can be used to describe hierarchical address spaces similar to postal addresses or computer names. Here we use an important conceptual assumption: *if reference is a value then an extended value is a relative (local) reference*. Super-concepts describe spaces for their sub-concepts while concept fields define the structure of local addresses relative to the parent address space. A child instance (extension) is said to exist in the domain (also context or scope) of its parent instance. For example, bank accounts are always identified with respect to their bank. Such a hierarchical address space is described by two concepts:

```
concept Bank
  char[12] bankCode;
}
concept Account in Bank {
  char[10] accNo;
}
```

Any reference to an account will consist of two segments: a parent bank reference and a child account reference.

Primitive references and objects. If there is no need in having custom references and object allocation mechanism then they can be inherited from some kind of primitive reference provided by the run-time environment like global/local heap, remote references or persistent storage. For that purpose, the concept has to be included in the platform-specific concept. For example, if we are going to allocate our objects in memory then the standard memory manager is used as a super-concept:

```
concept Bank in MemoryHandle
  char[12] bankCode;
}
```

Now instances of the `Bank` concept (and all its sub-concepts like `Account`) will extend memory handles provided by the platform. By default (but not always), each new bank and account objects will get a separate memory handle. In particular, each variable of the `Account` concept

```
Account acc; // 3 segments
```

will consist of three segments: memory handle, bank code and account number. The compiler will automatically allocate memory handles and memory necessary to store all object fields.

Navigating inclusion hierarchy. Any concept breaks the whole space into two domains: internal and external. Internal domain consists of all its sub-concepts while external domain consists of all other concepts. If concept is thought of as a border then it can be crossed in two directions: from outside in the direction of internal domain and from inside in the direction of external domain. Each border crossing is intercepted by some concept method depending on the direction of access: *if an element is accessed from inside then its outgoing method is used, and if it is accessed from inside then its incoming method is used*. This can be viewed as a visibility rule where outgoing methods are visible from inside and incoming methods are visible from outside. It is analogous to the passport control system at airports where arriving and departing passengers pass through different gates with different procedures. Essentially, concepts provide two implementations for each method: one for external use and one for internal use. However, once two versions of a method have been defined, we can forget about their differences and use concept methods precisely as methods of conventional classes.

COP uses `super` and `sub` keywords to access super- and sub-elements, respectively. `sub` is analogous to `inner` in the Beta programming language (Goldberg et al., 2004) where clear and convincing justification for their need is also provided. Applying a method to the `sub` keyword will produce an incoming method call because we are trying to enter a domain. Applying a method to the `super` keyword will call an outgoing method of the parent concept because it is a call from inside. Thus super method calls are always outgoing methods and sub method calls are always incoming methods. For example, if a method of the `Bank` concept is called from any method of the `Account` concept then an outgoing version of this method will be executed:

```
concept Account in Bank
  out double getInterest() {
    double rate = super.getInterest();
    return rate + accRate;
  }
}
```

Here `super.getInterest()` is an outgoing method of the `Bank` concept which returns the current interest rate at this bank (the same for all accounts of this bank). An incoming version of this method might produce different interest rate for external calls (or might not be defined at all). The `getInterest` method of the `Account` concept can be called from its sub-concepts only because it is marked as an outgoing method.

Object hierarchy. One of the distinguishing features of COP is its support of object hierarchies where one object may have many child objects with different relative references. Outgoing methods produce their result depending on this instance value and the parent segment values. In the case of the same parent, outgoing methods of different children will produce different results which are interpreted as different object field values. For example, assume that one bank object has many account objects with the persistent state stored in some database. Account balance could be then defined as follows:

```
concept Account in Bank {
  char[10] accNo;
  out double balance {
    get {
      Connection db = super.getConn();
      return db.load("balance", accNo);
    }
    set {
      Connection db = super.getConn();
      db.save("balance", accNo, value);
    }
  }
}
```

Here each `Account` object is identified by its number and then its `balance` object field is defined as an outgoing method (via one setter and one getter). Account balance depends on the current bank which provides connection to the database (so different banks store their data in different databases). As an extension, it also depends on the current account number which is used as a primary key when getting values from the database. Importantly, these are only implementation details but logically all objects exist in a hierarchy where each bank has many accounts. We can read balances and update balances using account references (consisting of several segments). And these operations will be logically correct because their result depends only on references. It is analogous to object hierarchies in prototype-based programming (Borning, 1986; LaLonde et al., 1986; Lieberman, 1986) with the difference that COP is also a class-based approach where both classes and their instances exist in a hierarchy.

4 USES OF THE INCLUSION HIERARCHY

Inheritance. Inheritance is a language mechanism for defining new objects by *reusing* already existing object definitions. The most wide spread treatment of inheritance is that members of a new class are added to or extend those already defined in the base class being reused. This model of inheritance is directly supported by outgoing concept methods. More specifically, child outgoing methods are implemented using parent outgoing methods which are called via the `super` keyword. COP also supports the model of inheritance implemented in prototype-based languages where the behavior defined in a parent object (prototype) is shared among and reused by all child objects (Stein, 1987).

Inheriting concept fields also works precisely as in the classical case: child fields are simply added to the parent concept fields. In this way we can extend values by adding more fields to them. For example, if concept `Point` has two fields `x` and `y` then we can define a new concept `Point3D` which has an additional field `z`:

```
concept Point { int x; int y; }
concept Point3D in Point { int z; }
```

Extending objects is not so simple because parent objects are shared among their children and therefore child fields cannot be simply concatenated with the parent fields. The classical model for object extension can be obtained if the child concept has no fields. Since the reference is empty, only one child can exist within one parent (just because they cannot be distinguished). In this case, we can think of child object fields as simply extending the parent fields. For example, if we need to define a bank account with some additional property then it can be done as follows:

```
concept BonusAccount in Account {
  out double bonus; // Object field
}
```

It is equivalent to conventional class and class inheritance. Any instance of this class will get its own parent segment with an additional `bonus` field defined in this concept.

Polymorphism. Polymorphism allows an object of a more specific type to be manipulated generically as if it were of a base type. For example, if we declare a variable as having the type `Account` then polymorphism allows us to apply to it the method `getBalance` even though it stores a reference to a more specific type like `BonusAccount`. There exist different approaches to implementing polymorphic behavior but the currently dominating strategy consists in completely overriding parent methods by child methods. In other words, if we define a child method then it will have precedence over the parent methods. If the child still needs some parent functionality then it has to explicitly use it by means of a super call. For example, if the `Button` class has to provide a more specific implementation of the `draw` method (than its parent `Panel` class) then it is implemented as follows:

```
class Panel {
  void draw() {
    fillBackground();
  }
}
class Button extends Panel {
  void draw() {
    super.fillBackground ();
    drawButtonText("MyButton");
  }
}
```

In addition to this classical direct overriding strategy for implementing polymorphism, COP introduces a *reverse overriding strategy* by assuming that parent incoming methods have precedence over and then can call child incoming methods. Thus incoming methods of parent concepts override incoming methods of child concepts. In the above example, panel background is filled by the parent class and then the child method is called in order to add (inject) more specific behavior:

```
concept Panel {
  in void draw() {
    fillBackground();
    sub.draw();
  }
}
concept Button in Panel {
  in void draw() {
    drawButtonText("MyButton");
  }
}
```

Note that here we inject some more specific behavior from within the parent incoming methods instead of injecting more general (parent) behavior from within the child (direct overriding). It is analogous to the idea of treating sub-classes as behavioral extensions to their super-classes in the Beta programming language (Kristensen et al., 1987; Madsen & Møller-Pedersen, 1989) where super-classes provide generic behavior which extended using the keyword inner rather than overridden. Both strategies describe behavior incrementally by executing some operations and then sending a request for further processing either to the parent or child object so the difference between them is only in the direction of delegation which is also similar to the mechanism of capturing and bubbling in JavaScript. What is new in COP is that these two strategies are combined using the mechanism of dual methods which effectively isolates two directions for method call propagation.

Cross-cutting concerns. Complex programs have functions which are scattered throughout the whole source code. Such program logic that spans the whole program is referred to as a cross-cutting concern and is known to produce numerous problems in software development. Aspect-oriented programming (AOP) (Kiczales, 1997) is the most wide spread approach to modularizing cross-cutting concerns which introduces an additional programming construct, called aspect. Aspects are orthogonal to the class hierarchy so that behavior defined in aspects is injected into points defined in the class hierarchy. In this sense, aspects and classes play different roles; they are not completely unified as well as not completely independent.

COP proposes a novel solution for this problem which is based on the ability of parent methods to intercept any access to child methods. Thus cross-cutting concerns are modularized in parent incoming methods and this functionality is injected in child methods. Effectively, this mechanism allows using parent incoming methods as wrappers for child methods so that some functions are guaranteed to be executed for each access while target (child) objects are unaware of this intervention. In terms of spaces, cross-cutting concerns are thought of as functions associated with space borders and automatically triggered for each incoming request passing the border. For example, if we would like to log any access from outside to account balances then this cross-cutting concern is implemented in the `getBalance` incoming method:

```
concept Bank {
  in double getBalance() {
    logger.Debug("Balance accessed.");
    return sub.getBalance();
  }
}
```

Interestingly, the notion of cross-cutting concern can be also applied to outgoing methods which means that one and the same logic is executed for all outgoing requests. For example, if banks have some reserves and they want to log all accesses to this property from inside then it is implemented as an outgoing method:

```
concept Bank {
  protected out double reserves;
  out double getReserves() {
    logger.Debug("Reserves accessed.");
    return this.reserves;
  }
}
```

Now any access to the bank reserves from any child object (like `Account` methods) will be logged. Obviously, this pattern is easily implemented in OOP. We mention it in order to emphasize that cross-cutting behavior has dual nature which is modularized in incoming and outgoing methods.

5 CONCLUSIONS

In this paper we described a novel approach to programming, called concept-oriented programming, which revisits some classical notions like class, inheritance, referencing, polymorphism, cross-cutting concerns. COP can be viewed as a generalization and further development of OOP by retaining its main features and adding the following new mechanisms:

- Modeling values and references by concepts
- Treating objects as functions of references
- Dual methods: incoming and outgoing
- Modeling object fields by outgoing methods
- Extended reference means relative (local) address
- Modeling hierarchical address space by inclusion relation and navigating via super and sub calls
- Inclusion generalizes inheritance and containment
- Two override strategies: reverse implemented by incoming methods and direct implemented by outgoing methods
- Modularize cross-cutting concerns in incoming methods which inject behavior in all child objects
- Integration with a new unified data model (Savinov, 2011; Savinov 2012)

Taking these properties into account, COP can be used as a basis for a next generation unified programming model.

REFERENCES

Borning, A. H., 1986. Classes versus prototypes in object-oriented languages. In *Proc. ACM/IEEE Fall Joint Computer Conference*, 36–40

Goldberg, D. S., Findler, R. B., Flatt, M., 2004. Super and inner: together at last! In *Proc. OOPSLA'04*, 116–129

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J., 1997. Aspect-Oriented Programming, In *Proc. ECOOP'97*, 220–242

Kristensen, B. B., Madsen, O. L., Moller-Pedersen, B., Nygaard, K., 1987. The Beta programming language, In *Research Directions in Object-Oriented Programming*, B. Shriver, P. Wegner (Eds.), 7–48

LaLonde, W. R., Thomas, D. A., Pugh, J. R., 1986. An exemplar based Smalltalk. In *Proc. OOPSLA'86*, 322–330

Lieberman, H., 1986. Using prototypical objects to implement shared behavior in object-oriented systems. In *Proc. OOPSLA'86,* 214–223

Madsen, O. L., Møller-Pedersen, B., 1989. Virtual Classes: A Powerful Mechanism in Object-Oriented Programming. In *Proc. OOPSLA'89*, 397–406

Savinov, A., 2005. Concept as a Generalization of Class and Principles of the Concept-Oriented Programming. *Computer Science Journal of Moldova*, 13(3), 292–335

Savinov, A., 2008. Concepts and Concept-Oriented Programming. *Journal of Object Technology* 7(3), 91–106

Savinov, A., 2009. Concept-Oriented Programming. *Encyclopedia of Information Science and Technology, 2nd Edition*, Editor: Mehdi Khosrow-Pour, IGI Global, 672–680

Savinov A. (2011) Concept-Oriented Model: Extending Objects with Identity, Hierarchies and Semantics, *Computer Science Journal of Moldova*, 19(3), 254–287.

Savinov A. (2012) Concept-Oriented Model: Classes, Hierarchies and References Revisited, *Journal of Emerging Trends in Computing and Information Sciences*, 3(4), 456–470.

Stein, L. A., 1987. Delegation Is Inheritance. In *Proc. OOPSLA'87*, 138–146