Object-oriented aspects of information systems: Ada'95 and Java as advanced tools for implementation

Dr. habil. sc. ing. Janis Osis
Bikernieku iela 77 - 34, Riga, LV-1039, Latvia,osis@egle.cs.rtu.lv
Mag. sc. ing. Pavel Rusakov
Jurmalas gatve 93/2 - 54, Riga, LV-1029, Latvia, rusakovs@egle.cs.rtu.lv

Abstract: this article is fully devoted to the describing of some important aspects in the implementation of complex informational systems. Two modern advanced object-oriented programming languages – Ada'95 and Java were chosen as possible tools for this goal. Basic mechanisms for applied programming: abstraction, encapsulation and hierarchy are analysed in this article. Programs code examples were actively used to demonstrate these opportunities. The resulting table of languages object-oriented mechanisms is given at the end of the work.

Keywords: object-oriented, Ada'95, Java, abstraction, encapsulation, hierarchy, inheritance.

Introduction: Modern informational systems are mainly very complex systems. Object-oriented model is the most effective model to implement such systems. Object-oriented programming languages are excellent tools for programming using object-oriented approach to problem. Object-oriented opportunity first appeared in the programming language Simula-67 [1]. Objects were used to simulate some aspects of reality. The main goal of the system designer in this case – to decompose the problem into the collection of collaborating objects using the hierarchy principles. So, in program objects *will be simulating* real parts of systems, messages, and a lot of other aspects of work.

There are four major aspects of the object-oriented model: abstraction, encapsulation, hierarchy and polymorphism [2]. We also have some facultative elements

of this model: modularity, typing, concurrency and persistency.

Why were Ada'95 and Java chosen for the analysis? Ada'95 is the first object-oriented programming language with the officially accepted international standard [3]. The first standard, Ada'83, was only object-based (without inheritance) and was developed for safety-critical military systems and real-time embedded systems. The object-oriented language Java was developed as the safe platform independent programming language with included network mechanisms, on the base of C++, SmallTalk and Objective C, with some good ideas from other programming languages.

1. Abstraction and encapsulation

Abstraction focuses upon the outside view of an object, but *encapsulation* (information hiding) prevents clients from seeing its inside view. A *class* is a set of *objects* that share a common structure and a common behaviour [4]. An object is the instance of the class.

1.1. Ada'95.

The main mechanism for abstraction and encapsulation in Ada'95 is the *package*. The package allows to group information about the logically related data and methods for working with these data. Theoretically, we can speak about the package as the class. Practically, we have no opportunity to implement an instance of this class (or create an element, static or dynamic). The Ada'95 package is *only named space*. This problem is mainly related to the main purpose of the Ada - creating the safety-critical systems. Pointers to the packages may cause inconsistency in the language standard. This mechanism is very expensive, too.

The package has two parts: an interface and an implementation (**package** and **package body**). The part **package** defines opportunities of the package, but the **package body** implements these opportunities. There are two reserved words for the package interface: **private** and **limited private**. These words restrict the visibility of some elements of the package. The mechanism **private** allows creating an element of the given type, passing it as a parameter, checking on equality (not equality) and assigning element value of such type. The mechanism **limited private** allows only to

create an element and to pass it as a parameter. The package specification has two parts: visible without this package and not visible without this package (**private**). Both **private** and **limited private** elements must be defined in the second part.

So, to make the hide of the information more effective, each class must have two parts: an *interface* and an *implementation*. The interface of a class captures only its outside view, but the implementation of a class comprises the representation of the abstraction as well as the mechanisms that achieve the desired behaviour. Separate using of package specification (file with extension ADS) and package body (file with extension ADB) is the big advantage of Ada both standards in comparing to Java. At first, we have the largest flexibility in the encapsulation. For instance, a package specification and a package body can use different packages. Second, the interface – the most important information for user – placed very compactly and it is not necessary to see both names of methods and implementation in this case.

We want to demonstrate this logic on a simple example. For instance, we have on a discrete flat, a point with integer coordinates X and Y. We have some defined operations with this point: *get* X and Y coordinates, *set* X and Y coordinates, and *print information* about this point. Let us to demonstrate the *interface* part of this example on Ada'95.

```
package CP_Pack is
   -- this type will be described in the part private
   type CoordPoint is tagged private;
   -- The method InitPoint is a constructor equivalent
   procedure InitPoint (ParmCoordPoint : in out CoordPoint;
                        ParmX : in Integer; ParmY : in Integer);
   -- The method GetX is used to get X coordinate
   function  GetX (ParmCoordPoint : in CoordPoint) return Integer;
   -- The method GetY is used to get Y coordinate
   function  GetY (ParmCoordPoint : in CoordPoint) return Integer;
   -- The method SetX is used to change X coordinate
   procedure SetX (ParmCoordPoint : in out CoordPoint; ParmX : in Integer);
   -- The method SetY is used to change Y coordinate
   procedure SetY (ParmCoordPoint : in out CoordPoint; ParmY : in Integer);
```

```
    -- The method PrintPoint is used to print coordinate point
    procedure PrintPoint(ParmCoordPoint : in CoordPoint);
  private
    -- coordinate point with coordinates X and Y (**)
    type CoordPoint is tagged record
      X : Integer;                    -- X coordinate
      Y : Integer;                    -- Y coordinate
    end record;
  end CP_Pack;
```

As it is seen, all information is organised very logically and compactly. The *public* methods are defined in the beginning of package. The private tagged type CoordPoint declared only in the beginning of package, but defined only in the special part **private** (the reserved word **tagged** means that we can have an opportunity to inherit this type for more complex constructions in future).

We also want to demonstrate a fragment of the implementation part – **package body.** The implementation of the initialisation method will demonstrated here.

```
  with Text_IO;
  use  Text_IO;
  package body CP_Pack is
    -- the new package IO_Integer will be organised to output integer numeric
    -- generic package Integer_IO is the base of new package
    package IO_Integer is new Integer_IO(Integer);
    use IO_Integer;
    procedure InitPoint (ParmCoordPoint : in out CoordPoint;
                         ParmX : in Integer; ParmY : in Integer) is
    begin
      ParmCoordPoint.X := ParmX;      -- X coordinate initialisation
      ParmCoordPoint.Y := ParmY;      -- Y coordinate initialisation
    end InitPoint;
    ...                               -- other methods
  end CP_Pack;
```

1.2. Java

Java uses the special reserved word **class** for the class organisation (as in the C++). We can freely create dynamical instances of this class and to work with them in the form *<object>.<attribute>* (or *<object>.<method>)*. Our example with the coordinate point may be implemented on Java as follows:

```
class CoordPoint {
  /* these attributes will be inherited, but its implementation
  is hidden from the user                                    */
  private protected int X;                    // X coordinate
  private protected int Y;                    // Y coordinate
  /* these methods are public and visible from all parts of the program */
  CoordPoint (int ParmX, int ParmY) {         // constructor
    X = ParmX;
    Y = ParmY;
  };
  public void finalize() {                    // destructor
  }
  public int  getX() {                        // X getting
    return X;
  }
  public int  getY() {                        // Y getting
    return Y;
  }
  public void setX (int ParmX) {              // X changing
    X = ParmX;
  }
  public void setY (int ParmY) {              // Y changing
    Y = ParmY;
  }
  public void printPoint() {// The printing of coordinate point
    System.out.println("X is : " + X + "   Y is : " + Y);
```

```
    }
};
```

As it is seen, the interface and the implementation are not logically separated. From the authors point of view, it may cause problems with clearity and visibility if methods are not so short as in this example and if we have not very much comments in the program. May be that old construction in C++ was more reliable. We had the opportunity to define methods in class as below:

```
class CoordPoint {
  protected:
    int X;      // X coordinate
    int Y;      // Y coordinate
  public:
    CoordPoint (int ParmX, int ParmY);  // constructor
    ~ CoordPoint ();                    // destructor
    int  getX();                        // X getting
    int  getY();                        // Y getting;
    void setX (int ParmX);              // X changing
    void setY (int ParmY);              // Y changing
    void printPoint();                  // The printing of coordinate point
};
```

That part was the interface part. But now comes the fragment of the implementation part:

```
CoordPoint :: getX()
{
  return (X);
}
```

Authors think, that this concept was more effective. Of course, one of the reasons to unite interface and implementation was the problem with the so called *header-files*. These files were containers for classes and it was possible easily to change a protection

of attribute or method (for instance, to replace **private** on **public**). But we can *unit* both interface and implementation in one file and *not to lose* the common conception of this principal separating of functionality. From the authors point of view, the next construction may be more rationally for Java standard:

1. To avoid interface and implementation in the different files (as in the C++), this operation were be successfully implemented.
2. To organise interface and implementation in one file, as in C++.

Telling about our personal experience, we can mention modern object-oriented programming languages mainly use interface and implementation separately. For instance, this concept takes place in the Object Pascal, C++, Borland Delphi, Borland C++ Builder, Ada'95. The Java concept is presented in DBMS Visual FoxPro 5.0 (not "free" programming language).

But we must declarate that Java has more flexible opportunities to restrict information's visibility. Ada'95 has only *three* opportunities for this goal. Two of them were mentioned above, but the third one – by default – is **public** equivalent in C++ and Java. Java has *five* opportunities for visibility restriction..

2. Inheritance

Inheritance defines a relationship among the classes, wherein one class shares the structure or behavior defined in one or more classes (*single inheritance* and *multiple inheritance*, respectively) [4]. Only a single inheritance will be compared in this article, because both Ada'95 and Java languages don't include standard opportunities to organize multiple inheritance. Multiple inheritance may be achieved in both cases using the special receptions. For instance, *interfaces* mechanism exists in Java for this goal (the *protocol* concept was inherited from Objective-C). Interfaces have its own hierarchy, and not intersect the class hierarchy of inheritance. This feature gives an opportunity to use one interface in different classes, not related with class inheritance. Interfaces related with class methods not with class attributes. Interfaces replace C++ multiple inheritance in the large degree. Ada'95 has some informal opportunities for multiple inheritance but all these opportunities related only with applied programming, not with language standard.

2.1. Ada'95

There are three main mechanisms of inheritance in Ada'95:

1. Tagged types.

2. Child packages.

3. Generic packages.

The mechanisms (1) and (2) are new for Ada'95 standard. The mechanism (3) is inherited from Ada'83 standard.

2.1.1. Inheritance between types

Now we want to describe the mechanism (1). For instance, we want to define a new type *DisplayPoint* based on type *CoordPoint* (with the new field *Colour*). Type *CoordPoint* was defined in the package *CP_Pack* (1.1). This type was defined as tagged using the reserved word **tagged.** For instance, the new type may be implemented as follows:

```
type DisplayPoint is new CoordPoint with record
   Colour : Integer;
end record;
```

But in our case we have no opportunity to organise such type in the main program using our package *CP_Pack*. That's because the supertype *CoordPoint* was defined in the **private** part of this package.

It is very important to mention that we have the inheritance between the *types* not between the *classes* in this case. This specific feature is a very strong part of the Ada'95 standard. So, in old versions of Pascal and in Ada'83 we had an opportunity to implement "simulation of inheritance" using the *case records* and *discriminant mechanisms*. The serious problems take place in this case: we may have *only one case part* in this case and we have this fictive inheritance *only on one level* into deep. Using **tagged** mechanism we can organise complex types with not-restricted into deep inheritance. That is real not simulated inheritance. It is not necessary to pass discriminant value as the parameter for the case part. But feature "**tagged**" may be used more effectively with the child packages. It may be a very powerfull *complex of*

inheritance. Now we will discuss this approach.

2.1.2. Inheritance between classes

Ada'95 standard has a new inheritance mechanism in packages - *child packages* [5]. We have an opportunity to build a complex system of the smallest subsystems; each of them is a package (and not compulsory a child package). In that case, changes in a small subsystem will influence other subsystems very loose. There is a very big information hiding in Ada'95 packages. C++ has a special keyword **friend**, which allows to indicate friendly classes, methods of those have access to the data of the given class. The analogue of this keyword is missing in both Ada standards. Visibility exists only in the child packages parts: parts **private** of the child packages can see the corresponding parts of its parents.

Let us demonstrate this opportunity with the example. The package *CP_Pack* was defined in the part 1.1. We want to organise the child package *CP_Pack.DP_Pack* to encapsulate our new type: DisplayPoint. We want also to inherit the base methods from the package *CP_Pack*, such as *GetX*, *GetY*, *SetX*, *SetY*. Two methods: *InitPoint* and *PrintPoint* will be overwritten in this case. We will also have two new methods: *GetColour* and *SetColour*. The interface part of this package will be organised as below:

```
package CP_Pack.DP_Pack is
   -- the new type for display point interface
  type DisplayPoint is new CP_Pack.CoordPoint with private;
   -- the method to initialise the display point
  procedure InitPoint (ParmDisplayPoint : in out DisplayPoint;
   ParmX : in Integer; ParmY : in Integer; ParmColour : in Integer);
   -- the method to get Colour of the display point
  function  GetColour (ParmDisplayPoint : in DisplayPoint) return Integer;
   -- the method to set Colour of the display point
  procedure SetColour (ParmDisplayPoint : in out DisplayPoint;
                       ParmColour : in Integer);
   -- the method to print display point
  procedure PrintPoint(ParmDisplayPoint : in DisplayPoint);
private
```

```
   -- the description of the type of display point
   type DisplayPoint is new CP_Pack.CoordPoint with record
      Colour : Integer;
   end record;
end CP_Pack.DP_Pack;
```

As it is seen, the prefix of the package – superclass (*CP_Pack*) is used to access the supertype *CoordPoint*. This inheritance between the child packages also have no restrictions into deep. The more complex example with the child packages is described in [6].

The *generic packages* mechanism will not be discussed in this article. The main goal of this mechanism is to make types by transferability, perform the same logical function on more than one type of data [7]. It is only the base for working packages. We can not use these packages immediately (see example with *CP_Pack* package body in 1.1).

2.2. Java

Java has more powerful opportunities in the single inheritance in the comparing with C++. Our example with the display point (2.1.1.) may be implemented on the base of coordinate point (1.2.) as follows:

```
final class DisplayPoint extends CoordPoint {  // inheritance from the base class
   private int Colour;   // colour of point
   DisplayPoint (int ParmX, int ParmY, int ParmColour) {// constructor
      super(ParmX, ParmY); // to call the constructor of the superclass
      Colour = ParmColour;                    // the initialisation of the Colours
   }
   public void setColour (int ParmColour) {    // method to set colour
      Colour = ParmColour;
   }
   public int  getColour() {                   // method to get colour
      return Colour;
   };
```

```
    public void printPoint() {                    // The printing of DisplayPoint
      System.out.println("X is : " + X + "   Y is: " + Y + "   Colour is : " + Colour);
    }
}
```

We can *forbid changing* of the methods and attributes in future in Java. Modifier **final** is included in the language for this goal. Modifier **final** before method or attribute defines that all the next inherited subclasses will use only this definition of method or attribute. Modifier **final** before the class means that this class will not have subclasses [8]. This mechanism is very actual to the commercial classes to avoid its extension by users. We have no such a mechanism in C++.

It is necessary to mark one important feature in the Java's inheritance: exceptions are organised as classes and may be effectively inherited. Java is significantly better than Ada'95 in this relation. We have only reserved word **exception** in Ada'95 to describe our own exception. Exceptions in Ada'95 are only parts of packages (classes). In Java our exceptions will be inherited from the base class *Exception*. For instance:

class LowValueException **extends** Exception {...}

3. Resulting table

Aspect	Ada'95	Java
Abstraction and encapsulation	**9**	**10**
Dynamical objects creation	-	+
Inheritance		
Single	**10**	**8**
Multiple	**6**	**8**
Common Result:	**25**	**26+**

All described object-oriented mechanisms will be compared now. The maximal quantitative mark of each aspect is 10, minimal – 0. We have "+" in the table cell if the

quantitative aspect is represented, and '-' in the other case

Conclusion: some common aspects of two advanced object-oriented languages were analysed in this article. All aspects are described and demonstrated with examples of the program code. Logical comments were made in this context. The main result of the work is the table with estimates of each object-oriented aspect.

This work includes only common features of Ada'95 and Java (or features which we can compare correctly). For instance, Java standard includes platform-independent users interface (*java.awt*), internet tools (*java.applet*), and some other important mechanisms. The common estimate of Java will be significantly greater if to compare these features with missing of it in Ada'95. Ada'95 standard must be extended in these directions for more effective concurrency in commercial programming from the authors point of view.

References

1. K. Nygaard, O.-J. Dahl. The development of the Simula language (in History of Programming Languages). R Wexelblat, ed. New York, Academic Press, 1981, pp. 439-493.
2. Grady Booch. Coming of Age in an Object-Oriented World. *IEEE. Software*, November, 1994, page 33 – 41.
3. David A. Wheeler. Ada lessons: What is Ada, 1995. Internet address: http://cortex.dote.hu/Ada_lessons/s1-1.html
4. Grady Booch. Object Oriented Design With Applications. The Benjamin/Cummings Publishing Company, Inc. 1991.
5. David L.Moore. Object-Oriented Facilities in Ada 95. *Dr. Dobb's journal*. October, 1995, num. 235, p. 28-35.
6. J. Osis, P. Rusakovs. Comparing of Some Object-Oriented Programming Languages. MOSIS'97 Proceedings (Modelling and Simulation of Systems, April 28 - 30, 1997, Hradec nad Moravici, Czech Republic, pp.197 - 202).
7. Dirk Craeynest. Some information on the Ada programming language (Reusability), 1997. Internet address: http://www.cs.kuleuven.ac.be/~dirk/ada-belgium/ada.html.

8. Patrick Naughton. The JAVA Handbook. Osborne McGraw-Hill, 1996.

All codes examples were compiled and executed using next compilers:

1. Ada'95 (GNAT 3.05) (1996) .
2. Java 1.0.2 (1996) and 1.1.3 (1997).