DOI: 10.5281/zenodo.4058346

MODELS OF CLASSES FOR ECONOMIC OBJECTS IN APPLICATIONS

Dănuţ-Octavian SIMION, PhD Associate Professor
Athenaeum University, Bucharest, Romania
danut_so@yahoo.com

Emilia VASILE, PhD Professor
Athenaeum University, Bucharest, Romania
rector@univath.ro

Abstract: *The paper presents the models of classes for economic objects in applications. In most cases, the economic objects have an abstract definition and so the optimal ways to describe these are interfaces used by economic applications. The information system is an information system that allows the performance of operations of collection, transmission, storage, data processing and dissemination of information thus obtained through the use of information technology and staff specialized in automatic data processing. The computer system includes all internal and external information, formal or informal used within the company as well as the data on which they were obtained, the software necessary for data processing and dissemination of information within the organization, procedures and techniques for obtaining (based on primary data) and disseminating information, the hardware platform necessary for data processing and information dissipation and staff specialized in data collection, transmission, storage and processing. The interfaces provided for classes can be extended according with the business requirements and may be changed easily for different types of activities.*

Keywords: *interfaces, implemented classes, economical modules, informational support, business flows, application modules, extended classes*
JEL classification: *C23, C26, C38, C55, C81, C87*

1. Introduction

The computer system is structured so as to meet the requirements of different groups of users such as: management factors at the level of strategic, tactical and operative management, staff involved in the process of data collection and processing and staff involved in the process of scientific research and design of new products and manufacturing technologies. Along with the definition of the business strategy, it is necessary to define the strategy of the information system and this because: the IT system supports the managers, through the information provided, in the management and control activity in order to achieve the strategic objectives of the organization, IT systems are open and flexible, constantly adapting to the imposed requirements the dynamic environment in which the company operates, promoting IT solutions supports the organization in consolidating and developing the business (eg: electronic commerce, e-banking, etc.), the computer system provides the necessary information to control the fulfillment and adaptation of the plans operational and strategic aspects of the organization, the organization must know and control the risks related to the implementation of the new ones technologies and adaptation of the computer system to the new requirements, establishing standards at the level of the information system that are meant to specify the hardware and software features and performance of the components to be purchased and what methodologies are to be used in the development of the system (Hewitt, 2019; Stover, 2019).

The level of strategic and tactical management is characterized by the request for data: adhoc, unanticipated, determined by a certain context created in which the manager is obliged to substantiate his decision, synthesized: as we climb the steps of the managerial hierarchy there is a selection and a gradual synthesis of information, predictive, allowing the anticipation of the evolution trends of the led process, external to define the economic, financial, competitive environment (Venners, 2020; Kumar, 2020).

In the case of operational management, which is characterized by structured decisions, the data provided are: default, their content covering the information need to be determined by the decisions of routine taken at this level, detailed because the manager must know in detail how to run an activity in its area of responsibility. They are obtained with a certain frequency, the moment of providing the information being predetermined.

2. The architecture of interfaces for classes

Extending classes using a legacy as a single implementation creates new types of classes. A superclass reference can indicate objects of its own type and subclasses strictly in accordance with the inheritance hierarchy. Because this relationship is linear, it excludes the multiple inheritance of the implementation, ie a subclass that inherits from several superclasses. Instead, Java provides interfaces, which not only allow the introduction of new named reference types, but also allow the inheritance of several interfaces. Defining interfaces

A top-level interface has the following general syntax:

```
<accessibility modifiers> interface <interface name>
<extends interface clause> // Interface header

{// Interface body
      <constant declarations>
      <method prototype declarations>
      <nested class declarations>
      <nested interface declarations>
    }
```

In the interface header, the interface name is preceded by the keyword interface. In addition, the interface header may specify the following information:

- purpose modifier or accessibility;
- any interface it extends.

The interface body may contain member statements that include

- constant statements;
- method prototype statements;
- nested class and interface statements.

An interface offers no implementation and is therefore abstract by definition. This means that it cannot be initiated, but classes can implement it by providing implementations for its method prototypes. The statement of an interface abstract is superfluous and is very rare.

Membership statements can appear in any order in the body of the interface. Because the interfaces are intended to be implemented by classes, the interface members have public accessibility by default and the public modifier is omitted (Hewitt, 2019; Horstmann, 2018).

Empty-body interfaces are often used as markers to label classes as having a certain property or behavior. Such interfaces are also called capacity interfaces. Java APIs provide some examples of such markup interfaces: java.lang.Cloneable, java.io.Serializable, java.util.EventListener.

Prototype Statement Method

An interface defines a contract by specifying a set of method prototypes, but no implementation. The methods in an interface are implicitly abstract and public by virtue of their definition. A prototype method has the same syntax as an abstract method. However, only abstract and public modifiers are allowed, but they are invariably omitted.

```
<return type> <method name> (<parameter list>) <throws clause>;
```

Example declaration of two interfaces: IStack and ISafeStack.

```
interface IStack {                                                    // (1)
    void   push(Object item);
    Object pop();
}

class StackImpl implements IStack {                                   // (2)
    protected Object[] stackArray;
    protected int      tos;  // top of stack

    public StackImpl(int capacity) {
        stackArray = new Object[capacity];
        tos        = -1;
    }

    public void push(Object item)                                     // (3)
        { stackArray[++tos] = item; }

    public Object pop() {                                             // (4)
        Object objRef = stackArray[tos];
        stackArray[tos] = null;
        tos--;
        return objRef;
    }

public Object peek() { return stackArray[tos]; }
}

interface ISafeStack extends IStack {                                 // (5)
    boolean isEmpty();
    boolean isFull();
}
```

```
class SafeStackImpl extends StackImpl implements ISafeStack {
// (6)

    public SafeStackImpl(int capacity) { super(capacity); }
    public boolean isEmpty() { return tos < 0; }        // (7)
    public boolean isFull() { return tos >= stackArray.
length-1; }// (8)
}

public class StackUser {

    public static void main(String[] args) {            // (9)
        SafeStackImpl safeStackRef  = new SafeStackImpl(10);
        StackImpl     stackRef      = safeStackRef;
        ISafeStack    isafeStackRef = safeStackRef;
        IStack        istackRef     = safeStackRef;
        Object        objRef        = safeStackRef;

        safeStackRef.push("Dollars");                   // (10)
        stackRef.push("SirExample");
        System.out.println(isafeStackRef.pop());
        System.out.println(istackRef.pop());
        System.out.println(objRef.getClass());
    }
}
```

Output program:

```
SirExample
Dollars
class SafeStackImpl
```

Implementing interfaces

Any class can choose to implement, in whole or in part, zero or more interfaces. A class specifies the interfaces that it implements as a comma-separated list of unique interface names in an implementation clause in the class header. Interface methods must be publicly accessible when implemented in the classroom or subclass. A class cannot restrict the accessibility of an interface method, nor can it specify new exceptions to the discard method, because attempting to do so would mean changing the interface contract, which is illegal. The criteria for overpressure methods also apply to the implementation of interface methods (Stover, 2019; Goetz, 2020).

A class can provide implementations of the methods declared in an interface, but does not take advantage of the interfaces unless the interface name is explicitly specified in its implementation clause.

In the previous example, the StackImpl class implements the IStack interface, specifying both interface names using the implement clause in its class header and providing the implementation of the interface methods. Changing the public accessibility of these methods in the classroom will lead to a compilation error, as this would reduce their accessibility.

A class can choose to implement only some of the methods of its interfaces, to provide a partial implementation of its interfaces. The class must then be declared abstract. Please note that the interface methods cannot be declared static, as they include the contract performed by the objects of the class that implements the interface. Interface methods are always implemented as instance methods.

The interfaces that a class implements and the classes that it extends directly or indirectly are called class supertypes. Instead, the class is a subtype of its supertypes. Classes that implement interfaces introduce the inheritance of multiple interfaces into their linear implementation inheritance hierarchy. However, keep in mind that no matter how many interfaces a class implements directly or indirectly, it provides only one implementation of a member that could have multiple statements in the interfaces (Venners, 2020; Horstmann, 2018).

Extension of interfaces

An interface can extend other interfaces using the extension clause. Unlike expanding classes, an interface can extend multiple interfaces. Interfaces extended by an interface, directly or indirectly, are called superinterfaces. Instead, the interface is an interface of its superinterfaces. Because interfaces define new reference types, superinterfaces and subinterfaces are also supertypes and subtypes.

A subinterface inherits all methods from its superinterfaces, because their method statements are implicitly public. A subinterface can replace the prototype statements in its superinterface method. Overworked methods are not inherited. The method prototype statements can also be overloaded, analogous to the method overload in the class.

The above example provides an example of multiple inheritance in Java. In the previous example, the ISafeStack interface extends the IStack interface. The SafeStackImpl class extends both the StackImpl class and implements the ISafeStack interface. Both the implementation and inheritance hierarchies of the interface for classes and interfaces are defined in the previous example.

In UML, an interface looks like a class. One way to differentiate between them is to use an "interface" stereotype. The interface inheritance is shown similarly to the implementation inheritance, but with a dotted inheritance arrow. Thinking in terms of types, each reference type in Java is an object type

subtype. This means that any type of interface is also an object type subtype (Hewitt, 2019; Goetz, 2020).

It is instructive to observe how the SafeStackImpl class implements the ISafeStack interface: it inherits the implementations of the push () and pop () methods from its StackImpl superclass and offers its own implementation of the isFull () and isEmpty () methods from the ISafeStack interface. The ISafeStack interface inherits two method prototypes from its IStack superinterface. All its methods are implemented by the SafeStackImpl class. The SafeStackImpl class implements the IStack interface by default: implements the ISafeStack interface that inherits from the IStack interface. This is easily evident from the diamond shape of the inheritance hierarchy. There is only one legacy of the implementation in the SafeStackImpl class.

We note that there are three different inheritance relationships in the workplace when defining inheritance between classes and interfaces:

The inheritance hierarchy of linear implementation between classes: one class extends another class subclasses - superclasses. Multiple inheritance hierarchy between interfaces: one interface extends other interfaces, subinterfaces - superinterfaces.

Multiple interface inheritance hierarchy between interfaces and classes: a class implements interfaces. Although interfaces cannot be initiated, interface type references can be declared. References to objects in a class can be assigned to references to class supertypes. In the previous example, an object of the SafeStackImpl class is created in the main () method of the StackUser class. The object reference value is assigned to the references of all object supertypes, which are used to manipulate the object.

3. The usage of interfaces in classes and objects

Constants in interfaces

An interface can also define named constants. Such constants are defined by field declarations and are considered public, static and final. These modifiers are usually omitted from the statement. Such a constant must be initiated with an initiator expression.

An interface constant can be accessed by any client, a class, or an interface using its fully qualified name, regardless of whether the client extends or implements its interface. However, if a client is a class that implements this interface or an interface that extends this interface, the client can directly access such constants without using the fully qualified name. Such a client inherits the interface constants. The typical use of constants in interfaces is illustrated in the following example, showing both direct access and the use of fully qualified names (Stover, 2019; Horstmann, 2018).

Extending an interface that has constants is analogous to extending a class with static variables. In particular, these constants can be hidden by subinterfaces. In the case of multiple inheritance of interface constants, any name conflict can be resolved using fully qualified names for the constants involved.

Example Interface variables

```
interface Constants {
    double PI_APPROXIMATION = 3.14;
    String AREA_UNITS = " sq.cm.";
    String LENGTH_UNITS = " cm.";
}

public class Client implements Constants {
    public static void main(String[] args) {
       double radius = 1.5;
       System.out.println("The area of the circle is " +
                          (PI_APPROXIMATION*radius*radius) +
                          AREA_UNITS);  // (1) Direct access.
       System.out.println("The circumference of the circle is " +
                          (2*Constants.PI_APPROXIMATION*radius) +
                           Constants.LENGTH_UNITS); // (2) Fully
qualified name.
    }
}
```

Output program:

```
The area of the circle is 7.0649999999999995 sq.cm.
The circumference of the circle is 9.42 cm.
```

Type hierarchy

Arrays are objects in Java. Array types (boolean [], Object [], StackImpl []) by default increase the inheritance hierarchy. The inheritance hierarchy can be increased by the corresponding matrix types. An array type is shown as a "class" with the note [] attached to the element type name. The SafeStackImpl class is a subclass of the StackImpl class. The corresponding array types, SafeStackImpl [] and StackImpl [], are shown as subtype and supertype, respectively, in the type hierarchy. The example also shows array types corresponding to primitive data types.

From the type hierarchy, the following can be summarized:

- All reference types are object subtypes. This applies to classes, interfaces and array types, as they include all reference types.
- All arrays of reference types are also subtypes of the Matrix type Object [], but arrays of primitive data types are not. Note that the array type Object [] is also a subtype of the object type.
- If a reference type is a subtype of another reference type, then the corresponding matrix types also have a similar subtype-supertype relationship. There is no subtype-supertype relationship between a type and its corresponding matrix type.

We can create a number of interface types, but we cannot initiate an interface (as is the case with abstract classes). In the statement below, the reference iSafeStackArray is of type ISafeStack [], (that is, a range of interfaces of type ISafeStack). The table creation expression creates an array whose element type is ISafeStack. The array object can host five ISafeStack references. The following statement does not initialize these references to indicate objects:

```
ISafeStack[] iSafeStackArray = new ISafeStack[5];
```

A matrix reference has a polymorphic behavior like any other reference, subject to its placement in the type hierarchy. However, a runtime check may be required when objects are inserted into a vector, as the following example illustrates.

The following assignment is valid because a supertype reference (StackImpl []) can denote objects of its subtype (SafeStackImpl []):

```
StackImpl[] stackImplArray = new SafeStackImpl[2];
```

Because StackImpl is a supertype of SafeStackImpl, the following allocation is also valid:

```
stackImplArray[0] = new SafeStackImpl(10);
```

The assignment at the previous example inserts a SafeStackImpl object into the SafeStackImpl [] object (that is, the SafeStackImpl array) created in (1). Because the stackImplArray [i] type, (0 i <2), is StackImpl, the following allocation should also be possible:

```
stackImplArray[1] = new StackImpl(20);              //
ArrayStoreException
```

There are no issues during compilation, as the compiler cannot deduce that the stack variable stackImplArray will actually name a SafeStackImpl [] object at runtime. However, assigning to (3) causes the ArrayStoreException to be thrown at runtime, because a SafeStackImpl [] object cannot contain StackImpl objects.

Allocation, transfer and change of reference values

Reference values, like primitive values, can be assigned, passed, and passed as arguments. For values of primitive data types and reference types, conversions may occur over time

- Mission
- passing the parameters
- explicit change

The main rule for primitive data types is that broadening conversions is allowed, but reducing conversions requires explicit distribution. The main rule for reference values is that conversions up to the type hierarchy (upcasting) are allowed, but conversions in the hierarchy require an explicit casting - downcasting. In other words, conversions that are from one subtype to its supertypes are allowed, other conversions require explicit distribution, or are illegal. There is no notion of promotion for benchmarks.

Reference value assignment conversions

In general, reference value allocations are allowed up to the type hierarchy, with the default conversion of the source reference value to that of the destination reference type.

Example for allocating and passing reference values:

```
IStack interface {/ * From the previous Example * /}
ISafeStack interface extends IStack {/ * From Previous Example * /}
class StackImpl implements IStack {/ * From Previous Example * /}
class SafeStackImpl extends StackImpl
          ISafeStack implements {/ * From the previous Example * /}

public class ReferenceConversion {

    public static void main (String [] args) {
   // Reference declarations
```

```
   Object objRef;
   StackImpl stackRef;
   SafeStackImpl safeStackRef;
   IStack iStackRef;
   ISafeStack iSafeStackRef;

   // SourceType is a class type
   safeStackRef = new SafeStackImpl;
   objRef = safeStackRef; // Always possible
   stackRef = safeStackRef; // Subclass to superclass assignment
   iStackRef = stackRef; //  StackImpl implements IStack
   iSafeStackRef = safeStackRef; // SafeStackImpl implements
ISafeStack

   // SourceType is an interface type
   objRef = iStackRef; // Always possible
   iStackRef = iSafeStackRef; // Sub- to super-interface assignment

   // SourceType is an array type.
   Object [] objArray = new Object [3];
   StackImpl [] stackArray = new StackImpl [3];
   SafeStackImpl [] safeStackArray = new SafeStackImpl [5];
   ISafeStack [] iSafeStackArray = new ISafeStack [5];
   int [] intArray = new int [10];

  // Reference value assignments
  objRef = objArray; // Always possible
  objRef = stackArray; // Always possible
  objArray = stackArray; // Always possible
  objArray = iSafeStackArray; // Possible always
  objRef = intArray; // (11) Always possible
  // objArray = intArray; // Compile-time error
  stackArray = safeStackArray; // Subclass array to superclass array
  iSafeStackArray =
               safeStackArray; // SafeStackImpl implements ISafeStack

 // Parameter Conversion
 System.out.println ("First call:");
 sendParams (stackRef, safeStackRef, iStackRef,
                   safeStackArray, iSafeStackArray); //
// Call Signature: sendParams (StackImpl, SafeStackImpl, IStack,
// SafeStackImpl [], ISafeStack []);

System.out.println ("Second call:");
sendParams (iSafeStackArray, stackRef, iSafeStackRef,
                   stackArray, safeStackArray); // (16)
// Call Signature: sendParams (ISafeStack [], StackImpl,
```

```
ISafeStack,
// StackImpl [], SafeStackImpl []);
}

public static void sendParams (Object objRefParam, StackImpl
stackRefParam,
   IStack iStackRefParam, StackImpl [] stackArrayParam,
   final IStack [] iStackArrayParam) {// (17)
// Signature: sendParams (Object, StackImpl, IStack, StackImpl [],
IStack [])
// Print class name of object denoted by the reference at runtime.
   System.out.println (objRefParam.getClass ());
   System.out.println (stackRefParam.getClass ());
   System.out.println (iStackRefParam.getClass ());
   System.out.println (stackArrayParam.getClass ());
   System.out.println (iStackArrayParam.getClass ());
    }
}
```

Output program:

```
First call:
class SafeStackImpl
class SafeStackImpl
class SafeStackImpl
class [LSafeStackImpl;
class [LSafeStackImpl;

Second call:
class [LSafeStackImpl;
class SafeStackImpl
class SafeStackImpl
class [LSafeStackImpl;
class [LSafeStackImpl;
```

The rules for assigning the reference value are declared on the basis of the following code:

```
SourceType srcRef;
// srcRef is appropriately initialized.
DestinationType destRef = srcRef;
```

If an assignment is legal, then the srcRef reference value is said to be attributable (or a compatible task) to the DestinationType reference. The rules are illustrated by concrete cases from the previous example.

If SourceType is a class type, then the reference value in srcRef can be assigned to the destRef reference, provided that DestinationType is one of the following:

- DestinationType is a superclass of the SourceType subclass.
- DestinationType is a type of interface that is implemented by the SourceType class.

```
objRef        = safeStackRef;  //  Always possible
stackRef      = safeStackRef;  //  Subclass to superclass
assignment
iStackRef     = stackRef;      //  StackImpl implements IStack
iSafeStackRef = safeStackRef;  //  SafeStackImpl implements
ISafeStack
```

If SourceType is an interface type, then the reference value in srcRef can be assigned to the destRef reference, provided that DestinationType is one of the following:

- DestinationType is the object.
- DestinationType is a superinterface of the SourceType subinterface.

```
objRef    = iStackRef;     //  Always possible
iStackRef = iSafeStackRef; //  Subinterface to superinterface
assignment
```

If SourceType is an array type, then the reference value in srcRef can be assigned to the destRef reference, provided that DestinationType is one of the following:

- DestinationType is the object.
- DestinationType is an array type, where the SourceType element type is attributable to the DestinationType element type.

```
objRef = objArray; //  Always possible
objRef = stackArray; // Always possible
objArray = stackArray; // Always possible
objArray = iSafeStackArray; // Always possible
objRef = intArray; // Always possible
// objArray = intArray; // Compile-time error
stackArray = safeStackArray; // Subclass array to superclass
array
iSafeStackArray = safeStackArray; // SafeStackImpl implements
ISafeStack
```

Allocation rules are applied at compile time, ensuring that no type conversion errors occur during runtime allocation. Such conversions are type safe. The reason the rules can be applied to compilation is that they target the reference type (which is always known at compile time) and not the actual type of the object being referenced (which is known at runtime) (Horstmann 2018; Goetz 2020).

4. Conclusions

Encapsulation of objects has important advantages in programming, because it increases the security and reliability of programs, by eliminating the possibility of accidental modification of their values due to unauthorized access from the outside. Because of this, programmers generally avoid providing public data in an object, preferring to access data only by methods. The visible or public part of the object constitutes its interface with the outer world. It is possible for two different objects to have identical interfaces, ie to present the same data and methods on the outside (Stover, 2019; Kumar, 2020). Due to the fact that the encapsulated part differs, such objects may behave differently. Aggregation is the property of objects to be able to incorporate other objects. However, the „data" contained in an object can be not only primitive data, but also objects. This allows you to create objects with increasingly complex structures (Venners, 2020; Stover, 2019). Classification is the property of objects that have the same data structure and the same behavior (the same methods) to be grouped into a class. The class is an abstraction, which contains those properties of objects that are important in one application or category of applications, and ignores the others. In order to define custom model formats, it is necessary to define multiple interfaces for later implementation of classes that maps object defined in business logic and implementation.

References

Goetz, Brian. (2020). *Java Feature Spotlight: Sealed Classes,* Available at: www.infoq.com.

Hewitt, Eben. (2019). *Java Garage*, Upper Saddle River, US: Pearson Publication.

Horstmann, Cay S. (2018). *Interfaces and Lambda Expressions in Java,* Pearson Publication,

Kumar, Mehak, Nitsdheerendra. (2020). *Interfaces in Java,* Available at: www.geeksforgeeks.org.

Stover, Ben C., Sarah Wiechers, Kai F. Muller. (2019). *JPhyloIO: a Java library for event-based reading and writing of different phylogenetic file formats through a common interface,* BMC Bioinformatics, DOI: 10.1186/s12859-019-2982-3,

Venners, Bill. (2020). *Designing with interfaces*, Available at: www.infoworld.com.