

DOI: 10.5281/zenodo.4383022

POLYMORPHISM OF CLASSES AND REFERENCE OF INSTANCES DISTRIBUTION FOR ECONOMIC OBJECTS IN APPLICATIONS

Emilia VASILE, PhD Professor

Athenaeum University, Bucharest, Romania
rector@univath.ro

Dănuț-Octavian SIMION, PhD Associate Professor

Athenaeum University, Bucharest, Romania
danut_so@yahoo.com

Abstract: *The paper presents the polymorphism of classes and reference of instances distribution for economic objects in applications. Data stored in a database is persistent data, ie data that remains stored on magnetic media, independent of the execution of application programs. Persistent database data is entered, deleted, or updated using input data (from the keyboard, from reading data files, or from receiving messages). Input data is generally non-persistent data; they are generated by users and are stored (becoming persistent data) only after they have been validated (accepted) by the DBMS. The output data of a database system is also non-persistent data; they come from database query operations and are made available to the user (in the form of impressions, printed reports, etc.). The correspondent model for database tables and entities are classes that encapsulate properties and methods. Data - acts as a bridge between machine components (hardware and software) and the human component. The database contains both operational data (the set of records being worked on) and metadata. Polymorphism can obtain new objects derived from existing classes that map custom business logic which includes specific economic operations. The data model is defined as a set of concepts used in the description of the data structure. The structure of the database means the type of data, the connection between them, the restrictions applied to the data.*

Keywords: *polymorphism, specific classes, economical data, derived classes, business flows, application modules, analysis of data*

Coduri JEL: *C23, C26, C38, C55, C81, C87*

1. Introduction

An external schema or user's view contains a conceptual sub-schema of the database, specifically the description of the data that is used by that group of users. The conceptual schema of the database (conceptual schema) corresponds to a unique (for all users) and abstract representation of the data, describing what data is stored in the database and what are the associations between them. The internal or physical schema of the database (internal schema) specifies how the data is represented on the physical medium. A database system supports an internal schema, a conceptual schema, and several external schemas; all these schemes are different descriptions of the same data collection, which exists only internally.

The external level or the visual level (user), includes a collection of external schemes, which are views of the different groups of users, there being an individual view of the data for each group;

The conceptual level - or conceptual (logical) schema of the database, describes the structure of the entire database for all users. At the conceptual level, a complete description of the database is made, hiding the details related to the physical storage and detailing the description of the entities, the types of data, the relations between them and the associated restrictions;

The internal layer contains the internal schema that describes the structure of physical data storage in the database, using a model of physical data. This level describes the full details of the storage and how to access the data (Del Nero, 2020; Campbell, 2020).

In many DBMSs a clear distinction cannot be made between the three levels, often the conceptual level is strongly developed and apparently replaces the other levels. Also, when developing applications, there is a fusion of the external level with the conceptual one.

2. The operator reference and instance distribution

The expression to throw a <reference> from <source type> to <destination type> has the following syntax:

```
(<destination type>) <reference>
```

A distributed expression checks whether the object's reference value denoted by <reference> is attributable to a reference of type <destination>, ie that <source type> is compatible with <destination type>. If not, a

ClassCastException is thrown. The null reference value can be assigned to any type of reference. The binary instance operator has the following syntax:

```
<reference> instanceof <destination type>
```

The instanceof operator returns true if the left operand (<reference>) can be thrown to the right operand (<destination type>), but always returns false if the left operand is null. If the instanceof operator returns true, then the corresponding distribution expression will always be valid. Both distribution and court operators require a compile-time check and a run-time check, as explained below.

Compile-time verification determines whether a <source> reference and a <destination> reference can indicate reference-type objects that are a common subtype of both <source> and <destination> type in the type hierarchy. If that is not the case, then obviously there is no relationship between the types and neither the distribution nor the application of the court operator would be valid. When running, it is the type of real object denoted by the <reference> that determines the result of the operation.

With <source type> and <destination type> as Product and String classes, respectively, there is no subtype-supertype relationship between <source type> and <destination type>. The compiler would reject throwing a Product reference to a String type or applying the operator instanceof, as shown in previous example. With <source type> and <destination type> as Product classes and TubeProduct, respectively, the Product and TubeProduct references can indicate objects in the TubeProduct class (or subclasses) in the inheritance hierarchy. Therefore, it makes sense to apply the operator instanceof or send a Product reference to the TubeProduct type (Hewitt, 2019; Chand, 2020).

During operation, the result of applying the instanceof operator is false, because the reference Product1 of the Product class will actually name an object of the Bulb subclass, and this object cannot be denoted by a reference of the Product1 peer class. Applying the distribution results in a ClassCastException for the same reason. This is why expressed conversions are said to be unsafe, as they could throw a ClassCastException at runtime. Note that if the result of the operator instanceof is false, the distribution involving operands will throw a ClassCastException.

In the example, the result of applying the instanceof operator is also false, because the reference Product1 will further denote an object of the Product2 class, whose objects cannot be denoted by a reference

of its subclass Product3. Therefore, applying the distribution causes a ClassCastException to be thrown at runtime.

The situation presented in the example illustrates the typical use of the court operator to determine which object denotes a reference, so that it can be performed for the purpose of carrying out special actions. The reference Product of the Product class is initiated on an object of the Product3 subclass (Ryder, 2020; Chand, 2020). The result of the operator instance is true, because the reference Product1 will denote an object of the Product 4 subclass, whose objects can also be denoted by a reference of its Product1 superclass. In the same sign, the distribution is also valid. If the result of the operator's court is true, the distribution involving the operands will always be valid.

Example instanceof and Cast Operator

```
class Products {/ * ... * /}
class Products1 extends Products {/ * ... * /}
class Products2 extends Products {/ * ... * /}
class Products3 extends Products {/ * ... * /}
class Products4 extends Products {/ * ... * /}

public class TestProduct {
    public static void main (String [] args) {
        boolean result1, result2, result3, result4, result5;
        Products1 products1 = new Products1 (); // (1)
        // String str = (String) products1; // (2) Compile-time error.
        // result1 = products1 instanceof String; // (3) Compile-time error.
        result2 = products1 instanceof TubeProduct; // (4) false. Peer class.
        // Products2 products1 = (Products) products2; // (5) ClassCastException.

        result3 = products3 instanceof Products3; // (6) false: Superclass
        // Products products3 = (products2) products1; // (7) ClassCastException

        products4 = new Products4 (); // (8)
        if (products1 instanceof Products) { // (9) true
            Products4 products4 = (products) products1; // (10) OK
            // You can use products4 to access the Products4 class.
        }
    }
}
```

As we have seen, the instance operator actually determines whether the object reference value noted by the reference on the left can be assigned to a reference of the type that is specified on the right. Note that an instance of a subtype is an instance of its supertypes. At runtime, it is the type of the actual object noted by the reference on the left, compared to the type specified on the right. In other words, what matters is the type of the actual object denoted by the reference at run time, not the type of reference (Wagner 2019; Ryder 2020).

The previous example provides several examples of a court operator. It is instructive to go through the printed statements and understand the printed results. The literal null is not a court of any kind of reference, as shown in the printing statements (1), (2) and (6). An instance of a superclass is not an instance of its subclass, as shown in the print statement (4). An instance of a class is not an instance of a totally unrelated class, as shown in the print statement (10). An instance of a class is not an instance of an interface type that the class does not implement, as shown in the print statement (6). Any non-primitive array is an Object and Object [] instance, as shown in the print statements (4) and (5), respectively.

Example - Using the operator instance

```

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

public class Identification {
    public static void main (String [] args) {
        Object obj = new Object ();
        StackImpl stack = new StackImpl (10);
        SafeStackImpl safeStack = new SafeStackImpl (5);
        IStack iStack;

        System.out.println ("(1):" +
            (null instanceof Object)); // Always false.
        System.out.println ("(2):" +
            (null instanceof IStack)); // Always false.

        System.out.println ("(3):" + // true: instance of subclass of
            (stack instanceof Object)); // Object.
        System.out.println ("(4):" +
            (obj instanceof StackImpl)); // false: Downcasting.
        System.out.println ("(5):" +
            (stack instanceof StackImpl)); // true: instance of
StackImpl.

```

```

        System.out.println("(6):" + // false: Object does not imple-
ment
            (obj instanceof IStack)); // IStack.
        System.out.println("(7):" + // true: SafeStackImpl implements
            (safeStack instanceof IStack)); // IStack.

        obj = stack; // Assigning subclass to superclass.
        System.out.println("(8):" +
            (obj instanceof StackImpl)); // true: instance of StackIm-
pl.
        System.out.println("(9):" + // true: StackImpl implements
            (obj instanceof IStack)); // IStack.
        System.out.println("(10):" +
            (obj instanceof String)); // false: No relationship.

        iStack = (IStack) obj; // Cast required: superclass assigned
subclass.
        System.out.println("(11):" + // true: instance of subclass
            (iStack instanceof Object)); // of Object.
        System.out.println("(12):" +
            (iStack instanceof StackImpl)); // true: instance of
StackImpl.

        String [] strArray = new String [10];
        // System.out.println("(13):" + // Compile-time error,
        // (strArray instanceof String); // no relationship.
        System.out.println("(14):" +
            (strArray instanceof Object)); // true: array subclass of
Object.
        System.out.println("(15):" +
            (strArray instanceof Object [])); // true: array subclass
of Object [].
        System.out.println("(16):" +
            (strArray [0] instanceof Object)); // false: strArray [0]
is null.
        strArray [0] = "Amoeba strip";
        System.out.println("(17):" +
            (strArray [0] instanceof String)); // true: instance of
String.
    }
}

```

Output program:

```

(1): false
(2): false
(3): true
(4): false
(5): true

```

```
(6): false
(7): true
(8): true
(9): true
(10): false
(11): true
(12): true
(14): true
(15): true
(16): false
(17): true
```

Convert class and interface type references

References to an interface type can be declared, and they can indicate class objects that implement that interface. This is another example of upcasting. Note that converting an interface type reference value to the class type that implements the interface requires explicit casting. This is an example of downcasting. The following code illustrates these cases:

```
IStack    istackOne = new StackImpl(5);           // Upcasting
StackImpl stackTwo = (StackImpl) istackOne;      // Downcasting
```

Using the reference `istack` An `IStack` interface type, `IStack` interface methods can be invoked on objects in the `StackImpl` class that implement this interface. However, additional members of the `StackImpl` class cannot be accessed through this reference without first sending it to the `StackImpl` class:

```
Object obj1 = istackOne.pop();           // OK. Method in IStack inter-
face.
Object obj2 = istackOne.peek();         // Nu OK. Method not in IStack
interface.
Object obj3 = ((StackImpl) istackOne).peek(); // OK. Method in Stack-
Impl class.
```

3. Polymorphism and dynamic methods

As an object, a reference will actually denote during run, it cannot always be determined at compile time. Polymorphism allows a reference to name objects of different types at different times during execution. A supertype reference has a polymorphic behavior because it can denote objects of its subtypes.

When a non-private instance method is invoked on an object, the definition of the method actually executed is determined by both the runtime object type and

the method signature. Dynamic method searching is the process of determining the method definition that a method signature makes during run, based on the object type. However, a call to a private court method is not polymorphic. Such a call can only take place within the class and is linked to the implementation of the private method at the time of compilation (Del Nero, 2020; Chand, 2020).

The inheritance hierarchy is implemented in the following example. The implementation of the draw () method is undone in all subclasses of the Shape class. The invocation of the draw () method in the two loops from (3) and (4) in the following example, is based on the polymorphic behavior of the references and the dynamic search of the method. Sheet metal shapes contain shape references indicating a circle, a rectangle and a square, as shown in (1). At runtime, the dynamic search determines the execution of the draw () to be executed, based on the type of object noted by each element in the table. This is also the case for the elements in the drawables in (2), which contain IDrawable references that can be assigned to any object of a class that implements the IDrawable interface. The first loop will still work without any changes if objects from new subclasses of the Shape class are added to the array shapes. If they did not replace the draw () method, then an inherited version of the method would be executed. This polymorphic behavior applies to whiteboard drawings, in which subtype objects are guaranteed to have implemented the IDrawable interface.

Polymorphism and dynamic method search form a powerful programming paradigm that simplifies client definitions, encourages object decoupling, and supports dynamic change of object-to-object relationships.

Example - Polymorphism and dynamic search of methods

```
IDesen interface {
    void draws ();
}

class Shape implements IDesen {
    public void draw () {System.out.println ("Draw a figure."); }
}

class Circle extends Figure {
    public void draw () {System.out.println ("Draw a Circle."); }
}

class Rectangle extends Figure {
    public void draw () {System.out.println ("Draw a Rectangle."); }
}
```

```
class Square extends Rectangle {
    public void draw () {System.out.println ("Draw a Square."); }
}

class Map implements IDesen {
    public void draw () {System.out.println ("Draw a Map."); }
}

public class PolymorphRefs {
    public static void main (String [] args) {
        Figure [] figures = {new Circle (), new Rectangle (), new
Square ()}; // (1)
        Drawing [] drawings = {new Figure (), new Rectangle (), new
Map ()}; // (2)

        System.out.println ("Draw figures:");
        for (int i = 0; i <figures.length; i ++) // (3)
            figures [i] .deseneaza ();

        System.out.println ("Draw figures:");
        for (int i = 0; i <figures.length; i ++) // (4)
            figures [i] .deseneaza ();
    }
}
```

Output program:

```
Draw figures:
Draw a Circle.
Draw a Rectangle.
Draw a Square.
Draw figures:
Draw a Figure.
Draw a Rectangle.
Draw a Map.
```

Choose between Inheritance and aggregation. Encapsulation

An object has properties and behaviors that are encapsulated inside the object. The services he offers to his clients include his contract. Only the object-defined contract is available to customers. Implementing its properties and behavior is not a customer concern. Encapsulation helps to clarify the difference between an object's contract and execution. This has major consequences for program development. The implementation of an object can be changed without implications for customers. Encapsulation also reduces complexity, because

the inside of an object is hidden by customers, who cannot influence its implementation (Hewitt, 2019; Ryder, 2020).

A UML class diagram shows several aggregation relationships and an inheritance relationship. The class diagram shows a queue defined by aggregation and a stack defined by inheritance. Both are based on linked lists. A linked list is defined by aggregation. The implementation of these data structures is presented in the following example. The example aims to illustrate inheritance and aggregation, not the implementation of industrial strength of tails and stacks. The Node to (1) class is simple, defining two fields: one indicating the data and the other indicating the next node in the list. The LinkedList class at (2) keeps track of the list by administering a head and a queue reference. Nodes can be inserted forward or backward, but deleted only from the front of the list.

Example - Implementation of data structures through inheritance and aggregation

```
class Node { // (1)
    private Object data; // Data
    private Node next; // Next node

    // Constructors for initializing the next node.
    Node (Object data, Node next) {
        this.data = data;
        this.next = next;
    }

    // Method
    public void setData (Object obj) {data = obj; }
    public Object getData () {return data; }
    public void setNext (Node node) {next = node; }
    public Node getNext () {return next; }
}

class LinkedList { // (2)
    protected Node head = null;
    protected Node tail = null;

    // Method
    public boolean isEmpty () {return head == null; }
    public void insertInFront (Object dataObj) {
        if (isEmpty ()) head = tail = new Node (dataObj, null);
        else head = new Node (dataObj, head);
    }
    public void insertAtBack (Object dataObj) {
```

```
        if (isEmpty ())
            head = tail = new Node (dataObj, null);
        else {
            tail.setNext (new Node (dataObj, null));
            tail = tail.getNext ();
        }
    }
    public Object deleteFromFront () {
        if (isEmpty ()) return null;
        Node removed = head;
        if (head == tail) head = tail = null;
        else head = head.getNext ();
        return removed.getData ();
    }
}

class QueueByAggregation { // (3)
    private LinkedList qList;

    // Builder
    QueueByAggregation () {
        qList = new LinkedList ();
    }

    // Method
    public boolean isEmpty () {return qList.isEmpty (); }
    public void enqueue (Object item) {qList.insertAtBack (item); }
    public Object dequeue () {
        if (qList.isEmpty ()) return null;
        else return qList.deleteFromFront ();
    }
    public Object peek () {
        return (qList.isEmpty ()? null: qList.head.getData ());
    }
}

class StackByInheritance extends LinkedList { // (4)
    public void push (Object item) {insertInFront (item); }
    public Object pop () {
        if (isEmpty ()) return null;
        else return deleteFromFront ();
    }
    public Object peek () {
        return (isEmpty ()? null: head.getData ());
    }
}

public class Client { // (5)
    public static void main (String [] args) {
        String string1 = "Queues!";
    }
}
```

```
int length1 = string1.length ();
QueueByAggregation queue = new QueueByAggregation ();
for (int i = 0; i <length1; i ++)
    queue.enqueue (new Character (string1.charAt (i)));
while (! queue.isEmpty ())
    System.out.print ((Character) queue.dequeue ());
System.out.println ();

String string2 = "! Reverse String";
int length2 = string2.length ();
StackByInheritance stack = new StackByInheritance ();
for (int i = 0; i <length2; i ++)
    stack.push (new Character (string2.charAt (i)));
stack.insertAtBack (new Character ('!')); // (6)
while (! stack.isEmpty ())
    System.out.print ((Character) stack.pop ());
System.out.println ();
}
}
```

Output program:

Queues!

Reverse string!

Choosing between inheritance and aggregation for model relationships can be a crucial design decision. A good design strategy argues that inheritance should only be used if the relationship is unequivocally maintained throughout the life of the objects involved; otherwise, aggregation is the best choice. A role is often confused with an is-a relationship. For example, given the class employee, it would not be a good idea to model the roles that an employee, such as a manager or cashier, can play by inheritance if these roles change intermittently. Changing roles would involve a new object representing the new role each time this happens (Wagner, 2019; Campbell, 2020).

4. Conclusions

Code reuse is best achieved by aggregation when there is no relationship. Applying an artificial is a relationship that is not naturally present, it is usually not a good idea. The class defines the operations of a queue, delegating such requests to the LinkedList base class. Customers who implement a queue in this way do not have access to the base class and therefore cannot break the abstraction (Hewitt, 2019; Campbell, 2020). Both inheritance and aggregation promote implementation encapsulation, because implementation changes are

localized in the classroom. Changing the contract of a superclass can have consequences for subclasses, called the ripple effect, and also for customers who are dependent on certain subclass behavior (Del Nero, 2020; Wagner, 2019). Polymorphism is achieved through inheritance and interface implementation. The code based on polymorphic behavior will continue to work without change if new subclasses or new classes that implement the interface are added. If there is no obvious relationship, then the polymorphism is best obtained by using aggregation with the interface implementation. In most application polymorphism offers flexibility and add value to existing models and so the users may choose the best models for their business logic and entities that are very specific.

References

- Campbell, Drew. (2020). *Everything You Need to Know About Polymorphism*, www.better-programming.org.
- Chand, Swatee. (2020). *Everything You Need to Know About Polymorphism*, www.edureka.org.
- Del Nero, Rafael. (2020). *Polymorphism and inheritance in Java*, www.infoworld.com.
- Hewitt, Eben. (2019). *Java Garage*. Upper Saddle River, US: Pearson Publication.
- Ryder, Barbara G. (2020). *Fragment Class Analysis for Testing of Polymorphism in Java Software*, www.researchgate.net.
- Wagner, Gerd. (2020). *Really Understanding Association, Aggregation, and Composition*, www.codeproject.com.