Implementation of UML concepts in Java

Kakarontzas George
UML – Class Diagram

Class diagram depicts the static structure of the system. This structure comprises:

- The classes
- The associations between classes
- The dependencies between classes
- System interfaces
Class

The UML symbol for a class is a box with three partitions. In the top partition we write the name of the class. In the middle partition the attributes of the class and the third partition the class methods. In the example of the figure, the name of the class is ‘Student’. The class has a ‘name’ attribute and an ‘enroll’ method.
Association

Associations are relations between instances of a class (A person works for a company. A company has a number of employees). Associations in UML are depicted as lines connecting the associated classes. In the example diagram, class Student is associated with the Course class.
Role

An association end can be named with a label. This label is the rolename and represents the role of the end’s class in the association.
Multiplicity

An association end has also a multiplicity, which is an indication of how many instances can participate in this association at any time. Multiplicity can be exactly 1, 0..1 (optional association), * (zero or more), 1..* (one or more), or any specific number (e.g. 11).
Navigability

Navigability represents the ability to traverse an association from one direction, but not the opposite direction. For example consider the UML class diagram of the figure. The arrow in the Professor-Course association means that having a professor instance we can query it for the its courses but not the opposite (course instances are unaware of the professors that teach them). Therefore the navigability is from the professor to the course.
Generalization

Generalization is a technique that allows the definition of a general class from which more specific classes are derived. These more specific classes (subclasses) specialize certain aspects of the more general class (base class or superclass). In the figure we can see a generalization example in which a Part-Time Student is defined as a subclass of the more general Student base class.
A class diagram example
In UML **sequence diagrams** objects are represented as boxes placed horizontally from left to right. Objects exchange messages that are depicted as directed arrows. The direction of arrows shows the direction of the message (which object is the source of the message and which object is the destination of the message). Time passes from top to bottom: messages higher on the sequence diagram occur before messages lower on the diagram.
Object

An object is represented by a box in which we write the name of the object and the class of the object separated by a colon. If we haven’t decided yet what is the class of an object we can omit the class name. If the name an object is not important we can omit it and simply write the class name preceded by the colon.

aCustomer : Customer
Message exchange

Message exchange is represented as a directed arrow from one object (the sender) to another object (the receiver). Message handling is provided by the receiver object implementation. The receive object will usually provide a method that will handle the message. In the sequence diagram example of the figure, class Customer will provide a public method ‘getName()’ that will be executed when the message arrives at the aCustomer object. If the message is conditional we write the condition of the message before it in brackets (e.g. [ok]getName()). If the message is repeated (as in loops) we write an asterisk before the message (as in * getName()).
Message Return

A message return is depicted as a dashed arrow from the receiver of the message back to the sender, and it means that the receiver finished processing the message. If there are any results, they can be written above the dashed line. It is not obligatory to explicitly draw returns from messages and we usually omit them.
Self calls

Self-calls are method calls in which the sender and the receiver objects are the same object.
An example sequence diagram

1: add a course
2: display
3: select course offering
4: add professor (professor id)
5: get professor (professor id)
6: add professor (professor)
Programming simple associations with navigability

```java
public class Order {
    ...
    private Customer m_Customer;
    Order() { ... }
    ...
    public Customer getCustomer() {
        ...
    }
    public void setCustomer(Customer a) { ... }
}
```
One-directional multivalued associations

public class Order {  
  ...
  //array or Vector or ...
  private Customer customers[];
  ...
  public Customer[] getCustomers() { ... }
  public void setCustomers(Customer a[]) { ... }
}

![Diagram showing one-directional multivalued association between Order and Customer classes.](image-url)
Example

As an example we will develop a small Java application comprises three files: Building.java, Owner.java and Main.java. Main is a small program that creates instances of buildings and owners and associates them by creating the appropriate links.
Class diagram

Building

owners

1..*

Owner

- name : String
- getName() : String
- setName(s : String)
A possible implementation of the Building class

```java
import java.util.*;
public class Building {
    private Vector owners;
    public Building() {owners=new Vector();}
    public Enumeration getOwners() {
        return owners.elements();
    }
    public void addOwner(Owner c) {
        owners.addElement(c);
    }
}
```
The Owner class

```java
public class Owner {
    private String name;
    Owner() {}
    public void setName(String n) {
        name = n;
    }
    public String getName() {
        return name;
    }
}
```
Usage example (1)

```java
public static void main(String[] args) {
    Owner o1 = new Owner();
    Owner o2 = new Owner();
    Owner o3 = new Owner();

    o1.setName("GIWRGOS");
    o2.setName("NIKOS");
    o3.setName("FANH");
    ...
```
Usage example (2)

// b1 is owned by o1 and o2
Building b1 = new Building();
b1.addOwner(o1);
b1.addOwner(o2);

// b2 is owned by o1 and o3
Building b2 = new Building();
b2.addOwner(o1);
b2.addOwner(o3);
...

Usage example (3)

```java
Enumeration e;
System.out.println("The owners of the first building");
e=b1.getOwners();
while (e.hasMoreElements()) {
    Owner o = (Owner) e.nextElement();
    System.out.println("\t"+o.getName());
}
System.out.println("The owners of the second building");
e=b2.getOwners();
while (e.hasMoreElements()) {
    Owner o = (Owner) e.nextElement();
    System.out.println("\t"+o.getName());
}
```
Output of the example

The owners of the first building
GIWRGOS
NIKOS
The owners of the second building
GIWRGOS
FANH
Press any key to continue...
Class diagram for the exercise
public class Student {
    private String name;

    Student() {
    }

    public String getName() {
        return name;
    }

    public void setName(String n) {
        name = n;
    }
}

public class Assignment {
    private double mark=-1;
    Assignment() {} 
    public double getMark() throws NotYetSetException {
        if (mark==-1) {
            throw new 
            NotYetSetException("Mark is not yet set");
        }
        return mark;
    }
    public void setMark( int m) {
        mark=m;
    }
}
import java.util.Vector;
import java.util.Enumeration;
public class CourseRecord {
    private Student student;
    private Vector assignments;
    CourseRecord() {assignments=new Vector();}
    public Student getStudent() {
        return student;
    }
    public void setStudent(Student s) {
        student = s;
    }
    ...
}
CourseRecord class (2)

...  

public void addAssignment(Assignment a) {  
    assignments.addElement(a);  
}

public Enumeration getAssignments() {  
    return assignments.elements();  
}

...
public class NotYetSetException
    extends Exception
{
    public NotYetSetException(String s) {
        super(s);
    }
}
Exercises for the CourseRecord class

```java
public double average() throws NotYetSetException {
    /*
     * Computes and returns the average for a student by
     * traversing the Vector of her assignments and
     * obtaining the mark for her assignment.
     * This method should handle the NotYetSetException that
     * can be thrown by getMark() by
     * ignoring the Assignment object that caused it in the
     * average computation
     */
    return 0.0;
}

public boolean canTakeFinalExam() {
    /*
     * Returns true if the student has a mark
     * for at least three assignments and false
     * otherwise.
     */
    return true;
}
```
import java.util.Vector;
import java.util.Enumeration;
public class Course {
    private String title;
    private Vector courseRecords;
    Course() {courseRecords=new Vector();}
    public String getTitle() {return title;}
    public void setTitle( String t) {title=t;}
    public void addCourseRecord(CourseRecord cr) {
        courseRecords.addElement(cr);
    }
    public Enumeration getCourseRecords() {
        return courseRecords.elements();
    }
    ...
}
Exercise for the Course class

```java
public void printCourseStudents() {
    /*
     * Prints the names of the students enrolled for this course
     */
}

public void printBestStudent() {
    /*
     * Prints the best student according to the average computed by the CourseRecord class
     * for each student
     */
}

public void printFinalExamStudents() {
    /*
     * Prints the student names that can participate in the final exam, based in the value returned
     * by the method "boolean canTakeFinalExam()" of the CourseRecord class
     */
}
```
Sequence diagram for the printBestStudent() method of the Course class

printBestStudent() → *average() → *getMark() → getName() → printBestStudent()
Composition – Inheritance – Polymorphism

Kakarontzas George
Composition

- Composition is a strong association type with additional semantics:
  - The part is considered exclusive property of one and only one whole
  - There is a life and death association between the wholes and their parts: parts are created with the whole and are deleted with it.
UML notation

- Composition is depicted with a black diamond on the side of the whole.
  - For example a Polygon owns its Points exclusively
Programming compositions

- To program compositions we use the private access indicator in order to prevent sharing of exclusive parts with other object.
- In addition we prevent the transfer of private parts by not providing direct or indirect accessor methods (e.g. get methods).
- Usually part-objects are created during construction of the whole-object inside the constructor of the whole-object.
Example

- As an example of composition we will program the Polygon-Point composition association.
The Polyon class

public class Polygon {
    private Point corners[];

    Polygon(int[][][] points)
        throws IllegalArgumentException
    {
        . . .
    }

    public void draw() { . . .}
}

Polygon’s constructor

```java
Polygon(int[][] points)
    throws IllegalArgumentException {
    if (points.length < 3 || points[0].length != 2) {
        throw new IllegalArgumentException("Not enough corners or invalid point format");
    } else {
        corners = new Point[points.length];
        for (int i = 0; i < points.length; i++) {
            corners[i] = new Point(points[i][0], points[i][1]);
        }
    }
}
```
The draw() method

The draw() method simply displays the coordinates of the points.

```java
public void draw() {
    for (int i=0; i<corners.length; i++)
        System.out.println(
            "x="+corners[i].getX()+
            ",y="+corners[i].getY());
}
```
The Point class

class Point {
    private int _x;
    private int _y;

    public Point(int x, int y) {
        _x=x; _y=y;
    }

    public int getX() { return _x; }
}

public int getY() { return _y; }
}
Notes on Polygon

- Polygon creates its own Points by the information provided in its constructor. It doesn’t accept ready-to-use Points.
- Point objects will have the same lifetime as the Polygons that contain them. This life-and-death relationship along with the exclusiveness of the parts are the main distinguishing elements between compositions and plain associations.
Exercise

In the Polygon class implement a main function that creates two polygon objects and displays the values of their points by calling the draw() method on them.
Inheritance

- The term ‘inheritance’ refers to the extension of a class by another class, in which the extending class inherits from the extended class all the methods and attributes as well as its type.

- Type inheritance is a very important concept since it allows the extending class to be used wherever objects of the extended class are expected.
  - Objects of the extending class are type-compatible to objects of the extended class.
Inheritance in Java

- A sub-class extending a super-class uses the ‘extends’ keyword in its declaration followed by the name of the extended class.

```java
class subclass_name
    extends superclass_name
{
    subclass_body
}
```
Access Modifiers

- All members (attributes and methods) of a super-class are inherited by the sub-class.
- If a member is declared as private in the super-class then although the sub-class inherits it, it cannot refer to it directly, but must do so indirectly via access (get) methods as all other clients do.
- If a member is declared as protected then the sub-class not only inherits it but can also refer directly to it by using its name. On the other hand other class are not allowed such an access.
- Members declared as public are, as usual, accessible by every other class (sub-class or other).
An inheritance example

- As a simple example we will implement a Person class and a Student sub-class that will extend the Person class.
- The Student class will provide in addition to what the Person class provides, a variable that stores the field of study for the student and methods for updating and retrieving this field.
- Student class will also override a method of the Person class, so that it performs slightly differently for students.
The Person class

```java
public class Person {
    protected String name;
    protected int age;
    public Person() {}
    public void setName(String n) {name = n;}
    public void setAge(int a) {age = a;}
    public String getName() { return name;}
    public int getAge() { return age; }
    public String who()
    {
        return 
            "I am "+name+" and I am "+age+" years old";
    }
}
```
The Student sub-class

```java
public class Student extends Person {
    private String fieldOfStudy;
    public void setField(String s) { fieldOfStudy = s; }
    public String getField() { return fieldOfStudy; }
    public String who() {
        return "I am " + name + " and I am " + age + " years old. " + "I study " + fieldOfStudy;
    }
}
```
Notes on the Student class

- The Student class has an additional variable, the fieldOfStudy variable, and get and set methods for accessing this variable.
- The Student class can refer directly to the variables name and age of the super-class (see the ‘who’ method) because these variables are declared as protected in the super-class.
- The Student class overrides the who() method by declaring it again. When the who() method is called on Student objects the more specific who() of the Student class will be executed.
Polymorphism

- Method ‘who’ is polymorphic since it has two different versions: one version for the ‘Person’ class and another for the ‘Student’ class.
- Polymorphic methods allow objects of different types to respond to the same request differently.
Exercise

- Create two Persons and a Student with the following data:
  - person1: (name, age) = ("George", 35)
  - person2: (name, age) = ("Jim", 42)
  - student: (name, age, fieldOfStudy) = ("Simon", "20", "Computing")

- Place these three objects in an array of type Person. Then traverse the array and call the polymorphic method ‘who’ on all objects of the array displaying the returned value.
Thoughts on inheritance

- Large hierarchies of class create large dependencies between the classes. Don’t forget that a class inherits from the base class all its methods and data. This creates a strong coupling between these classes, and it is therefore difficult to evolve them separately. This problem is more intense when these hierarchies are deep.

- This problem is known in the literature as ‘the fragile base class problem’. The fragile base class problem has two different versions: the semantic and the syntactic
The syntactic fragile base class problem

The syntactic fragile base class problem is about changing the syntax of the super-class (e.g. a change in the parameters of a method). Then this change is in effect also for any subclasses that this class might have. It might involve recompilation of the subclasses, if these subclasses were using the old syntax that is now different.
The semantic fragile base class problem

- The semantic fragile base class problem involves changes in the semantics of the super-class (e.g. a method is know behaving differently). This means that subclasses will also have a different behavior and this might require changes on them.
Abstract classes and abstract methods

- Sometimes we want to declare a method in the super-class but are unable to define what it does. Although we can say how this method looks like (its name, parameter list, return value and possible exceptions), its behavior will be so different in the subclasses that providing its implementation in the super-class is meaningless.

- These unimplemented methods are abstract methods and their classes are abstract classes.

- We cannot create objects of abstract classes.
When a class declares an abstract method:

- It provides only the method declaration but not its implementation. Method declaration must contain the ‘abstract’ keyword as can be seen in the following code segment:

  ```java
  public abstract void anAbstractMethod();
  ```

- We cannot create objects of this class which is also considered abstract and must be declared as such again using the abstract keyword, as in the following code segment:

  ```java
  public abstract class AClass { ... }
  ```

The subclasses of this class must define what the abstract method does (implement it), otherwise they will also be abstract.
Constraints

- An abstract class is specifically designed as the base class for inheritance. Therefore it cannot be declared as final (using the final keyword), because then we wouldn’t be able to extend it with subclasses.

- An abstract method cannot be static, private or final.
  - If the method was static that it would be possible to call it using the class name, without creating any objects and this would allow calling an unimplemented method!
  - If the method was private then it wouldn’t be possible to implement it in the subclasses because it wouldn’t be visible.
  - If the method was final then we wouldn’t be able to re-declare it in the subclasses.
Example of abstract classes and methods

- As an example we could have a class for the payment of goods called PaymentMethod that is abstract. Such a class could be used for example in an Internet store.
- The following very simple PaymentMethod class allows setting and getting of the payment amount (money), but also declares an abstract method ‘pay’ which is very different for the possible payment methods and therefore we are unable to declare it here for all of them.
The PaymentMethod class

```java
abstract class PaymentMethod
{
    protected double money;
    PaymentMethod() { money = 0; }
    public double getMoney()
    {
        return money;
    }
    public void setMoney(double d)
    {
        money = d;
    }
    public abstract void pay();
}
```
Concrete subclasses

Now we can declare concrete subclasses that will extend the abstract class PaymentMethod and provide implementations for the ‘pay’ method. We will create two such subclasses: CreditCardPayment and CheckPayment.
public class CreditCardPayment
    extends PaymentMethod
{
    public void pay()
    {
        System.out.println("Paying "+
            money+ " euros by credit card");
        // credit card payment code goes here
    }
}
public class CheckPayment extends PaymentMethod {
    public void pay() {
        System.out.println("Paying " + money + " euros by check");
        //check payment code goes here
    }
}
Testing the classes

Finally in order to test these classes we will create a class Main1 with a parametric main function.

We will pass as parameters of main (via the args[] array) the preferred payment method (1 for credit card and 2 for check payment) and the amount to be paid.
public class Main1
{
    public static void main (String[] args)
    {
        if (args.length != 2)
            stop();
        int type=0;
        double money=0.0;
        try
        {
            type = Integer.parseInt(args[0]);
            money =
                (Double.valueOf(args[1])).doubleValue();
        }
        catch (NumberFormatException e) { stop(); }
PaymentMethod p;
switch (type) {
    case 1:
        p = new CreditCardPayment();
        p.setMoney(money);
        p.pay(); break;
    case 2:
        p = new CheckPayment();
        p.setMoney(money);
        p.pay(); break;
    default: stop();
}
}
public static void stop()
{
    System.out.println(
        "Usage: java Main1 (1 ή 2) amount");
    System.exit(0);
}
} //Main1
Exercise (1)

• Create the following class hierarchy:

```java
class Employee {
    String name = "";
    String afm = "";

    void setName(String n) {
        name = n;
    }

    String getName() {
        return name;
    }

    void setAfm(String a) {
        afm = a;
    }

    String getAfm() {
        return afm;
    }

    abstract int payment();
}

class SalariedEmployee extends Employee {
    int salary;

    void setSalary(int s) {
        salary = s;
    }

    int payment() {
        return salary;
    }
}

class HourlyEmployee extends Employee {
    int hoursWorked;
    int hourlyPayment;

    void setHoursWorked(int hw) {
        hoursWorked = hw;
    }

    int getHoursWorked() {
        return hoursWorked;
    }

    void setHourlyPayment(int hp) {
        hourlyPayment = hp;
    }

    int getHourlyPayment() {
        return hourlyPayment;
    }
}
```
Exercise (2)

- Class Employee is the superclass of the classes SalariedEmployee and HourlyEmployee. The superclass has the attributes name and tax number (afm) and methods to set and get these attributes. It also has an abstract method payment() and is therefore abstract.

- The SalariedEmployee subclass has in addition to the attributes of the super-class the attribute salary and a setting method for it. This class also defines the payment method which in this case returns the salary of the employee.
Exercise (3)

The HourlyEmployee subclass has in addition to the attributes of the super-class, the hoursWorked and hourlyPayment attributes. It provides ‘get’ and ‘set’ methods for these attributes. Finally it provides an implementation of the abstract method payment, which returns the product of the hourly payment rate with the hours worked for each employee.
Exercise (4)

- After creating this class hierarchy create a Main2 class with a main function. This function will have an array of type Employee and size 2. Place in the position 0 of this array a SalariedEmployee object and in the position 1 an HourlyEmployee object. Then set some appropriate values for the two objects. Finally create a loop that calls the payment() method on all objects of the array and display the returned value.