Universidad Autónoma de Madrid

Escuela Politécnica Superior

Software Analysis and Design 2019-2020

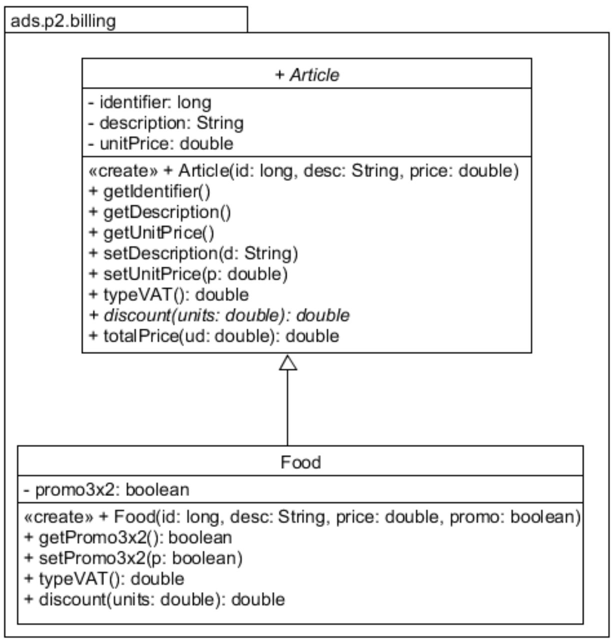
Assignment 2: Introduction to Object Oriented Design

Starting date:	The second week of February.
Duration:	2 weeks.
Submission:	**1h before the start of the next assignment, via Moodle.**
Weight in the grade:	15 %

The goal of this assignment is to learn basic concepts related object oriented design and their implementation in Java, with particular emphasis on the design of class hierarchies, associations and aggregations by means of UML class diagrams.

Part 1: (4 points)

Consider an application that stores information about items sold in stores. The class diagram to the right describes two classes: an **abstract** base class `Article` that stores the identifier, description and unit price of each item (as **private attributes**), and a subclass `Food` that models food items with an additional **private** attribute indicating if they should apply a 3x2 promotion.



ads.p2.billing
+ Article
- identifier: long
- description: String
- unitPrice: double
«create» + Article(id: long, desc: String, price: double)
+ getIdentifier()
+ getDescription()
+ getUnitPrice()
+ setDescription(d: String)
+ setUnitPrice(p: double)
+ typeVAT(): double
+ discount(units: double): double
+ totalPrice(ud: double): double
Food
- promo3x2: boolean
«create» + Food(id: long, desc: String, price: double, promo: boolean)
+ getPromo3x2(): boolean
+ setPromo3x2(p: boolean)
+ typeVAT(): double
+ discount(units: double): double

In addition to the constructor and the *getters* and *setters* methods, the `Article` class has three other **public methods**: the method *typeVAT()* returns the percentage of VAT that should be applied on its price according to the type of article; the **abstract** method *discount(units)* returns the discount applicable to the price (before taxes) of the article; and method *totalPrice(units)* calculates and returns the final price for the given number of units of the product, including the application of the discount and the corresponding VAT.

Below is the full implementation of these classes in Java. As you can see, the classes have been included in the package `ads.p2.billing`. Packages are used to organize the structure of large programs, which typically have many classes and several (nested) packages. In this way, other related classes will be declared in the same package. The structure of the packages is hierarchical and must correspond to the physical structure of directories (that is, the classes must be stored in a folder `ads/p2/billing`).

Constructors have the same name as the class, and are invoked by means of the **new** instruction, to create an object (an instance of the class) and initialize its attributes. This way, the constructor of class `Article` takes three parameters and initializes the identifier, description and unit Price attributes. Notice also how the constructor of the `Food` class, which is a subclass of `Article`, calls the superclass constructor using **super(…)** before initializing the specific `promo3x2` attribute of the `Food` subclass. Finally, notice how the `Food` class overwrites the *typeVAT()* method inherited from its superclass (also called parent class), and how it implements the abstract method *discount(units)* that had no implementation in the `Article` class.

Class Article

```
package ads.p2.billing;

public abstract class Article {
	private long identifier;
	private String description;
	private double unitPrice;

	public Article(long id, String desc, double price) {
		identifier = id;
		description = desc;
		unitPrice = price;
	}
	public long getIdentifier() { return identifier; }
	public String getDescription() { return description; }
	public double getUnitPrice() { return unitPrice; }
	public void setDescription(String desc) { description = desc; }
	public void setUnitPrice(double price) { unitPrice = price; }

	// The general VAT 21%, unless it is redefined in subclasses
	public double typeVAT() { return 0.21; }

	// Every subclass of Article will calculate the corresponding discount
	public abstract double discount(double units);

	// Total price is always calculated in the same way
	public double totalPrice(double units) {
		return ((unitPrice * units) - discount(units)) * (1.0 + typeVAT());
	}
}
```

Class Food

```
package ads.p2.billing;

public class Food extends Article {
	private boolean promo3x2;

	public Food(long id, String desc, double price, boolean promo) {
		super(id, desc, price);
		promo3x2 = promo;
	}
	public boolean getPromo3x2() { return promo3x2; }
	public void setPromo3x2(boolean promo) { promo3x2 = promo; }
	public double typeVAT() { return 0.10; }
	public double discount(double units) {
		if (promo3x2) return getUnitPrice() * (int) (units / 3);
		else return 0.0;
	}
}
```

The following example program uses the two previous classes to calculate and print the final price of 7 units of chocolate (with a unit price of 2.5€ and 3x2 promotion) and 4 units of yoghourt (1.25€ per unit, without promotion). As you can see, the objects are created by the instruction **new** followed by the name of the class and the constructor parameters (later we will see that a class can have more than one constructor). The type used in the declaration of the variable can be an abstract or a concrete class. The compiler checks that the class of the object assigned to a variable is compatible with the class used in the declaration of that variable.

Example program

```
package ads.p2;
import ads.p2.billing.*;

public class Example1 {
	public static void main(String args[]) {
		Article a1 = new Food(990034, "Chocolate", 2.5, true);
		System.out.println("Total price: "+ a1.totalPrice(7));//13.75=(7 * 2.5 - 2 * 2.5) * 1.10
		Food a2 = new Food(990268, "Yoghourt", 1.25, false);
		System.out.println("Total price: "+ a2.totalPrice(4));//5.5 = (4 * 1.25 - 0.0) * 1.10
	}
}
```

The output is the following:

```
Total price: 13.750000000000002
Total price: 5.5
```

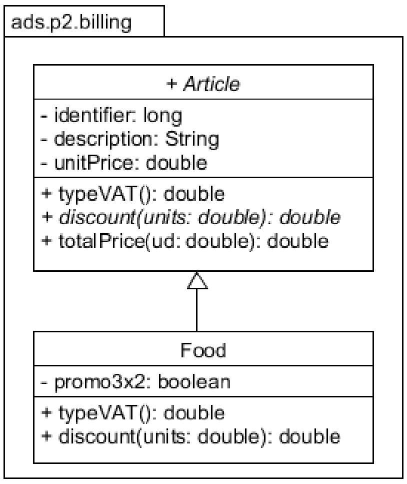
To do: Add two classes to the UML diagram and write the Java code to model books and tobacco as two other types of articles, with the following requirements. Each book has an attribute that identifies which category it belongs to: “Textbook”, “Collection”, “Best-sellers”, etc. It should be possible to change the category of a book once created. A reduced VAT rate (4%) is applied to the books and its discount is calculated according to its category: 15% for textbooks, and 50% for the third and subsequent units of collection books. Regarding tobacco, the final price is calculated without discount and applying the general VAT rate. Complete the `Article` class with a `toString()` method so that, with the code of the two new classes, the execution of this program produces the output below.

```java
package ads.p2;
import ads.p2.billing.*;

public class Example2 {
  public static void main(String args[]) {
    Article a3 = new Book(940111, "UML", 15, "Textbook");
    Article a4 = new Book(940607, "Classic readings", 10, "Collection");
    Book    a5 = new Book(940607, "Fifty fifty", 3.25, "Unclassified");
    Tobacco a6 = new Tobacco(690023, "Marlboro", 1.75);

    int cant = 2;
    System.out.println(cant + " units of " + a3 + " Final price: " + a3.totalPrice(cant));
    cant = 5;
    System.out.println(cant + " units of " + a4 + " Final price: " + a4.totalPrice(cant));
    cant = 4;
    System.out.println(cant + " units of " + a5 + " Final price: " + a5.totalPrice(cant));
    cant = 1;
    System.out.println(cant + " units of " + a6 + " Final price: " + a6.totalPrice(cant));
  }
}
```

```
2 units of Id:940111 UML Price: 15.0 Final price: 26.52
5 units of Id:940607 Classic readings Price: 10.0 Final price: 36.4
4 units of Id:940607 Fifty fifty Price: 3.25 Final price: 13.52
1 units of Id:690023 Marlboro Price: 1.75 Final price: 2.1174999999999997
```

Remark:

In practice, a class diagram is an abstraction of the final code, so normally and to facilitate its understanding, not all the details of the source code are included. For example, the setters and getters methods and the constructors are usually omitted. The following diagram shows a version of the design using this higher level of abstraction.



If you want to know more...:

Method *totalPrice()* in class `Article` is an example of the “*template method*” design pattern. This consists in writing code in a class, which invokes (possibly abstract) methods that are implemented in subclasses. You can find more information on design pattern in:

Design Patterns: Elements of Reusable Object-Oriented Software. Gamma, E., Helm, R., Johnson, R., Vlissides, J. INF/681.3.06/PAT. Addison-Wesley, 2003.

Part 2 (4 points)

You have to design part of an application for appliance sales. In particular the module you need to design is responsible for calculating the final amount that the customer must pay based on the price of the product, checking if home delivery should be made and if at the time of purchase an old appliance is delivered. The store sells three types of appliances, which need to include the following data:

- *Television*: brand, model, base price, screen size and energetic performance (a letter between A and D, see below).
- *Washing Machine*: brand, model, base price, dimensions, energetic performance and weight.
- *Refrigerator*: brand, model, base price, dimensions, energetic performance and weight.

If the customer picks up the appliance in the store, no extra amount is added to the price. However, for home delivery, some rules apply to calculate the cost of shipping, which must be added to the total price:

- *For televisions*:
  If the screen size is less than or equal to 40'', and its price is more than 500€, shipping is free.
  If the screen size is less than or equal to 40'', and the price is less than 500€ (without taking into account discounts) the shipping price is 35€.
  In any other case, the cost of shipping is 35€ plus 1€ for every inch that the screen size exceeds 40''. For example, the shipping cost of a 45'' TV is 40€.
- *For washing machines*:
  If the weight is less than or equal to 50kg, the shipping cost is 35€.
  If the weight is over 50kg, the cost is 35€ plus 0.5€ for each kg (or fraction) over 50kg. For example, the shipping cost of a 56.5kg machine is 38.5€.
- *For refrigerators*:
  The shipping cost is calculated multiplying its volume (in m^3) by 70€, and rounding it to an integer value. For example, the shipping cost of a refrigerator of 50.5cm wide, 71.9 cm depth, and 201 cm height is 60€.

All appliances in stock are catalogued with an energy performance value based on their consumption. The energy class of the appliance is represented by a letter between A and D, with Class A appliances being the ones with the lowest consumption (highest energy class) and those with class D the ones that consume the most (lowest energy class). If, at the time of purchase, the customer delivers an old appliance, a discount will be applied to the amount to be paid depending on the energy class of the purchased appliance and the one delivered (regardless of the type of both appliances). The rules for calculating this discount are:

- If the old appliance is in the same class as the new, the discount is 25€.
- If the new appliance has a lower class than the old, to the 25€ of discount we add 15€ for each energy class of difference. For example, if buying a B washing machine, and giving a D television, the discount is 55€.
- If the new appliance has higher class, to the 25€ we subtract 5€ for each energy class of difference. For example, buying a D washing machine and giving a B television, the discount is 15€.
- If the class of the old appliance is unknown, the discount is 10€.

To do: Create a UML class diagram for the application module. There is no need to include constructor, getters or setters. There is no need to include the implementation in Java.

Part 3 (2 points):

We are going to build an application for managing the staff of the public universities in Madrid. However, in this section we will only focus on the following aspects.

Some staff are *civil servants* ("*funcionarios*"), which are identified by a registry number. In addition, we should store the start date as civil servant, and the category (an integer number used to calculate the base salary). Some civil servants hold positions of administrative responsibility, such as the *centre administrator*. For each administrator, we need to know the University and the centre (department or school) she is an administrator of, and a salary complement (which is added to her normal salary).

On the other hand, we should store information about the teaching staff of the universities. Each *professor* is identified by its DNI, and belongs to some department (which are independent of the University and centre, and may belong to several Universities and centres). For each professor, we should know if she holds a PhD or not (because this is used to calculate the salary). In addition, we should maintain a registry of all her teaching activities. Hence, for each year and term, we should store the courses taught, with the type of teaching (theory, lab, project), and the number of hours per week taught for the given course in that term.

Associate professors are professors that in addition are civil servants. For these professors, we must know their 5-year teaching periods with positive evaluation, and the 6-year research periods with positive evaluation. Both types of periods provide salary complements.

Finally, *adjunct professors* are not civil servants, but work in private companies and collaborate (typically part-time) in teaching activities in the universities. For them, we should know the company, social security number and the dedication to teaching (because their University salary is calculated based on this dedication).

To do:

a) Create a UML class diagram covering the previous requirements. Don't include constructors, getters or setters. There is no need to provide the Java implementation.
b) Comment if we should do any change on the design should we implement the application in Java. If this is the case, provide the new class diagram.

Part 4 (Optional, 1 point):

Create a new UML class diagram, extending the one you built in Part 2, modelling the following extra features to offer a new support service providing phone technical assistance and a new repair service for some appliances.

The support service will only be offered for some television models (not all). These must be identified through the application. When identifying the television models for which the service is offered, the URL of the technical documentation of the TV needs to be given so that the support staff can access it and consult it if necessary. In addition to televisions, support for tablets and mobile phones will also be offered, although the possibility of doing so for desktops and laptops may be contemplated in the future. For tablets and mobile phones, the following information should be stored: brand, model and URL of the technical documentation. This service will only be provided to those customers who hire it specifically by paying an annual fee that will entitle them to make all the queries they want during the term of the contract. The collection of this rate and the management of customers who have access to the service will be carried out by the store's billing application.

The repair service will be offered for all televisions and washing machines. All catalogued television and washing machine models will have a specific price per hour of repair (that is, each one of them may have a different price). The application must calculate the repair cost applying the following rules:

- If the appliance has been repaired successfully, the cost will be calculated by multiplying the number of hours used by the cost per hour of repair of the specific television model or washing machine. Both the result of the repair (satisfactory/unsatisfactory), and the number of hours spent on it will be provided manually at the time of returning the appliance to the customer.
- If the appliance could not be repaired, nothing will be charged to the customer.

How to submit:

- You should submit parts 1, 2 y 3 (and optionally 4). Create one folder for each part in the assignment.
- The submission is to be done by ONE of the members of the team, via Moodle.
- If the exercise contains Java code, you should include the documentation generated with JavaDoc. If the exercise includes a UML design, you should provide a PDF with a brief explanation (two or three paragraphs).
- The submission should be packaged in one ZIP or RAR file, that has the following name: `GR<group_number>_<student_names>.zip`, For example, the group composed by Marisa and Pedro, of group 2261, should submit a file named: `GR2261_MarisaPedro.zip`