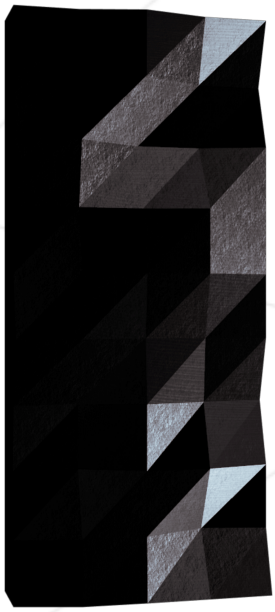

PISCINE OBJECT

Module 02 - UML

Summary: This document will introduce you to the basis of Universal Modeling Langage.

Version: 1.00

Contents

Chapter I

Preamble

In the early days of programming, there were no real "rules" about what was clean or not. It was a bit of a wild west!

Since then, engineers have started to think of a way to coordinate themselves to produce code that is easier to maintain and read.

To do this, they have created their own language, called Unified Modeling Language. This language will serve them, from this moment, to transform a software architecture into a graph, allowing to discuss, exchange and understand much more easily what has to be done!

Chapter II

Principle

UML is a graphical modeling language that allows for the visual representation of different aspects of a software system. UML diagrams are graphical representations of these aspects, and there are several types of UML diagrams that can be used depending on the modeling needs.
There are multiples types of diagram, such as :

- Class diagram
- Sequence diagram
- Activity diagram
- State transition diagram
- Use case diagram
- Deployment diagram
- Etc

Each type of UML diagram has its own characteristics and is used to represent different aspects of a system. By using a combination of these diagrams, it is possible to create a complete and detailed representation of a software system.

In this topic, you will be introduced to the class diagram, one of the most commonly used in UML. It allows the representation of classes, attributes, methods, and the relationships between these elements.

Classes are represented by rectangular boxes that contain the name of the class, its attributes, and its methods.
Attributes are represented below the name of the class and methods are represented below the attributes. Relationships between classes are represented by arrows.

There are several types of arrows that represent different relationships between classes, such as, in a non-exhaustive way :

- Inheritance arrow, represented by a solid line with an arrow pointing to the parent class.
- Implementation arrow, represented by a dotted line with an arrow pointing to the interface. This arrow indicates that the class implements the methods defined in the interface.
- Aggregation arrow, represented by an open diamond with an arrow pointing to the containing class.
- Composition arrow, represented by a solid diamond with an arrow pointing to the containing class.

Finally, it is important to note that relationships in a class diagram are generally "downward".
This means that the parent class should be on top of the diagram, and the childrens classes bellow, with arrow pointing upside.
This arrow convention is used to facilitate reading and understanding of the diagram.

Chapter III

Mandatory Part

	Exercise 00
Exercice 00: Car composition	
Turn-in directory : $ex00/$	
Files to turn in : `subject.png`	
Allowed functions : `None`	

For this exercice, you must create a diagram, as a png file, named subject.png.

This image must represent, graphicaly, the set of following objects :

- The structure `LinkablePart` is a virtual classes, requiering to implement a method `void execute(float p_pression)` to be instancied.
- The structure `Wheel` is a class containing a method `void executeRotation(float p_force);` which allows a wheel to execute a rotation with a given force.
- The structure `Gear` is a class containing an integer variable `demultiplier` which stores the demultiplier of a gear.
- The structure `GearLever` is a class that inherits from `Singleton<GearLever>` containing an array of `Gear` objects and an integer variable `level`. It has a method `void change();` which allows to change the current gear and a method `Gear* activeGear();` which returns a pointer to the current active gear.
- The structure `Transmission` is a class containing a vector of pointers to `Wheel` objects and a method `void activate(float p_force);` which activates the transmission with a given force.
- The structure `Crankshaft` is a class containing a pointer to a `Transmission` object and a method `void receiveForce(float p_volume);` which receives a force in a given volume.
- The structure `ExplosionChamber` is a class containing a pointer to a `Crankshaft` object and a method `void fill(float p_volume);` which fills the chamber with a given volume.

- The structure `Injector` is a class that inherits from `LinkablePart` containing a pointer to an `ExplosionChamber` object and a method `void execute(float p_pression);` which executes the injection with a given pressure
- The structure `Motor` is a class containing objects of type `Injector`, `ExplosionChamber` and `Crankshaft`. It has a method `void connectToTransmission(Transmission* p_transmission);` which connects the motor to a given transmission.
- The structure `Pedal` is a class containing a pointer to a `LinkablePart` object and two methods: `void setTarget(LinkablePart* p_part);` which sets the target of the pedal to a given part, and `void use(float p_pression);` which uses the pedal with a given pressure.
- The structure `Direction` is a class containing an array of `Wheel` objects and a method `void turn(float p_angle);` which turns the direction by a given angle.
- The structure `DAE` is a class containing a pointer to a `Direction` object and a float variable `force`. It has a method `void use(float p_angle);` which uses the DAE with a given angle.
- The structure `SteerWheel` is a class containing a pointer to a `DAE` object and method(s) `void turn(float p_angle)` which takes a float as input.
- The structure `Brake` is a class containing a pointer to a `Wheel` object and method(s) `void execute(float p_force)` and `void attackWheel(Wheel* p_wheel)` which take a float and a pointer to a `Wheel` object as inputs, respectively.
- The structure `BrakeController` is a class that inherits from `LinkablePart` and contains an array of `Brake` objects and method(s) `void execute(float p_pression)` which takes a float as input.
- The structure `Cockpit` is a class containing `Pedal`, `SteerWheel`, `GearLever` objects and method(s) that manage these objects.
- The structure `Electronics` is a class containing a `DAE` object.
- The structure `Car` is a class containing a `BrakeController`, `Direction`, `Transmission`, `Motor`, `Electronics`, and `Cockpit` objects.

Chapter IV

Bonuses

	Exercise 00
Bonus 00: Diagrams ameliorations	
Turn-in directory : $ex00/$	
Files to turn in : `subject.png`	
Allowed functions : `None`	

Several bonuses are proposed to go further in the modeling of the system.
First of all, it is proposed to modify the architecture of the system to correct any links that could be ascending in the example. Indeed, UML generally advises to only make descending or horizontal relationships between the different classes of a system. By modifying the architecture, it is possible to ensure that the relationships between the classes comply with UML recommendations.
Next, it is proposed to transform the system structures into classes, and to distribute their attributes and methods into public and private. This transformation can allow for better modeling of the different entities in the system, and make the code more easily understandable for developers.

It is also proposed to work on the graphical representation of the system, ensuring that the different units are distributed in distinct blocks for better readability. It is also recommended to represent association definitions (1..*, etc.) where necessary, for more precise modeling.

Finally, one or more sequence diagrams for moving the steering, braking, acceleration can be proposed. These diagrams allow for modeling the interactions between the different objects in the system in specific scenarios, and can help better understand the functioning of the system in these situations.

Once you've completed those bonuses assignament, you're free to add any bonuses you may think is needed.

The bonus part will only be assessed if the mandatory part is PERFECT. Perfect means the mandatory part has been integrally done and is validated by the evaluator. If you have not passed ALL the mandatory requirements, your bonus part will not be evaluated at all.

Chapter V

Submission and peer-evaluations

For this evaluation, you must provide any documents you may think is necessary to understand your diagram.
You won't be evaluated on anything except what's inside the repository.
The evaluation will mainly focus on explaination of what you've done in your diagram, so be prepare to discuss this subject !

Turn in your assignment in your `Git` repository as usual. Only the work inside your repository will be evaluated during the defense. Don't hesitate to double check the names of your folders and files to ensure they are correct.