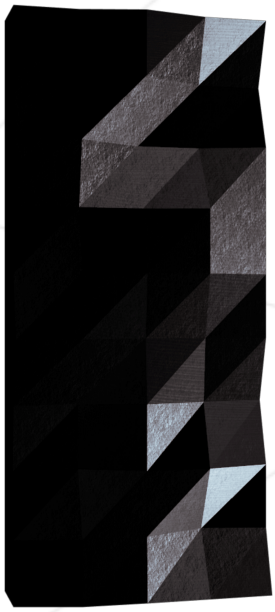

PISCINE OBJECT

Module 03 - SOLID

Summary: This document will introduce you to the five SOLID principles.

Version: 1.00

Contents

Chapter I

Preamble

The development of object-oriented programming began in the 1960s, but it wasn't until the 1990s that the concept really began to take hold in the software development industry. As the complexity of applications and systems increased, developers began to realise that it was important to create software that could evolve over time without becoming too difficult to maintain.
It was in this context that the SOLID principles were developed. They were proposed by Robert C. Martin, a software developer and computer consultant, also known as "Uncle Bob". The principles were formulated to help programmers create software that was easier to modify, maintain and reuse.

Chapter II

Introduction

We saw in the previous chapter that SOLID is a set of rules, or rather guidelines, that allow you to produce code that is easier to read, understand and maintain.

Each letter in the word SOLID refers to a specific rule, the meaning of which you will explore in this module :

- [*S*]ingle Responsibility Principle
- [*O*]pen/Closed Principle
- [*L*]iskov Substitution Principle
- [*I*]nterface Segregation Principle
- [*D*]ependency Inversion Principle

By following these principles, developers can create more flexible and modular software. It also reduces the coupling between different parts of the software, making it easier to maintain and evolve the code. The SOLID principles are therefore an essential tool for programmers who want to create high-quality, scalable and maintainable software.

To ease your time over this modules, we allow you to place methods bobies directly inside the header file, to allow you to focus mainly on what you're doing in term of architecture, and not passing too much time swapping from one file to another.

But that's only on this module ! So enjoy it while it last !

Chapter III

[S]olid - Single responsability

	Exercise 00
Exercice 00: [S]olid - Single responsability	
Turn-in directory : $ex00/$	
Files to turn in : `car.hpp, *.hpp, main.cpp`	
Allowed functions : `None`	

You must create a `Car` class that can perform the following list of actions:

- `start()`: starts the engine
- `stop()`: stops the engine and applies the brakes
- `accelerate(speed)`: increases the speed of the car by a specified value
- `shift_gears_up()`: shifts up to the next gear
- `shift_gears_down()`: shifts down to the previous gear
- `reverse()`: puts the transmission in reverse gear
- `turn_wheel(angle)`: turns the wheels by a specified angle
- `straighten_wheels()`: returns the wheels to a straight-ahead position
- `apply_force_on_brakes(force)`: applies a specified force to the brakes
- `apply_emergency_brakes()`: applies the brakes with maximum force for an emergency stop

You're required to provide a main.cpp file, showing how your code work.

You must follow the single responsability rule from SOLID to perform this exercice. During the evaluation, you will be interviewed on how and why you designed your class the way you will have compose it. You're allow to add any necessary file in this exercice repository.

Chapter IV

s[O]lid - Opened/Closed

	Exercise 01
Exercice 01: s[O]lid - Opened/Closed	
Turn-in directory : $ex01/$	
Files to turn in : `command.hpp, thuesdayDiscount.hpp, packageReductionDiscount.hpp, main.cpp`	
Allowed functions : `None`	

You must create a `Command` class to represent an order with the following attributes:

- `id`: the order ID
- `date`: the order date
- `client`: the client who placed the order
- `articles`: the articles ordered with their quantities

The class Command must contain the following methods :

- `get_total_price()` : a method to compute the price of the command

You are requested to create 2 differents discount commands :

- A class ThuesdayDiscountCommand, inheriting from Command
  This class must apply a discount of 10 percents of every command executed on a Thuesday
- A class PackageReductionDiscountCommand, inheriting from Command
  This class must apply a discount of 10 euro if the total command value exceed 150 euros

You're required to provide a main.cpp file, showing how your code work.
You must follow the Opened/Closed rule from SOLID to perform this exercice.

You're obviously not allowed to edit the Command class to perform the 10 percent or the fixed value discount

Chapter V

so[L]id - Liskov substitution

	Exercise 02
Exercice 02: so[L]id - Liskov substitution	
Turn-in directory : $ex02/$	
Files to turn in : `shape.hpp, rectangle.hpp, circle.hpp, triangle.hpp, main.cpp`	
Allowed functions : `None`	

You must create a base `Shape` class with virtual methods for calculating the area and perimeter of a shape, and you must also create three derived classes, `Rectangle`, `Triangle` and `Circle`, that inherit from `Shape` and provide their own implementations of the abstracts methods.

In order to satisfy the Liskov substitution principle, you should ensure that any code that expects an object of type `Shape` can work correctly with any object of a derived type, without knowing or caring about the specific type of the object. This means that the derived class should be able to replace the `Shape` class in any context without introducing errors or unexpected behavior.

You're required to provide a main.cpp file, showing how your code work.

Chapter VI

sol[I]d - Interface segregation

	Exercise 03
Exercice 03: sol[I]d - Interface segregation	
Turn-in directory : $ex03/$	
Files to turn in : `employeeManagement.hpp, employee.hpp, hourlyEmployee.hpp, salariedEmployee.hpp, main.cpp`	
Allowed functions : `None`	

In this exercice, you're required to create a class `EmployeeManager`, that will manage a set of `Employee`.

The `EmployeeManager` class must contain the following methods :

- void addEmployee(Employee*)
- void removeEmployee(Employee*)
- void executeWorkday()
  This method must execute a single day of work for each of the registered worker. One day equal 7 hours of work.
- void calculatePayroll()
  This method must output in the console the amount of money to give to each of the subscribed workers, over the last complete month.

The `Employee` class must contain the following attributes/methods:
Attributes:

- int hourlyValue

Methods:

- int executeWorkday()

You must now create the following 3 new Employee derived classes :

- class `TempWorker`
- class `ContractEmployee`
- class `Apprentice`

The idea is quite simple :

- The `TempWorker` is a class who must register every work hour it will do. It is also need a way to mobilise it over a certain period of time, in hour.
- The `ContractEmployee`, on the other hand, must register the hour it will NOT do (Vacantion, sickness, any reason).
- The `Apprentice`, at last, must register the hour it will NOT work, just as the ContractEmployee, but it must also log every hour where it will go to school. The Apprentice need a way to indicate how many hours of school he will do before going back to work. Those hours passed at school will be payed half the hourly value.

You're allowed to edit the base Employee class, if you may think it's necessary. You will have to explain such edition during evaluation.
Use the Liskov substitution principle previously saw to ensure that each implementation of employee management can be used interchangeably with the generic employee management interface.

Use the interface segregation principle to avoid forcing client classes of the generic interface to depend on methods that do not concern them.

You're required to provide a main.cpp file, showing how your code work.

Chapter VII

soli[D] - Dependency inversion

	Exercise 04
Exercice 04: soli[D] - Dependency inversion	
Turn-in directory : *ex*04/	
Files to turn in : `ILogger.hpp, *.hpp, main.cpp`	
Allowed functions : `None`	

In this exercice, you have to create a system of log file/output.

You must create the `ILogger` class, with at least a method `virtual void write(std::string) = 0`

You're also required to create some derived classes, following the requirements:

- a Logger that will write the string onto a file fully managed by the logger.
- a Logger that will write the string onto a std::ostream given to the logger.
- both those type of loggers `can` have a text header in front of any string. The header can be :
  - a constant string.
  - a date describing when the message is outputed.

`Loggers CAN have a header. It's not mandatory. So do not force it onto ILogger derived classes that do not need it.`

As test, you will create a std::vector<ILogger*> containing every kind of logger you may have created, in parallel of a list of string to log. Your main must iterate thought th list of string, and force every logger to output each string, to check if all your logger work correctly.

You're not authorized to add anything inside the ILogger, as it must remain a strict interface.

You must remember what you learn thought the interface segregation exercice.

```
Does the constant string header need to know how to get the date ?
Does the non-header logger need to know how to print a header ?
```

You're required to provide a main.cpp file, showing how your code work.

You must follow the dependency inversion principle to perform this exercice.

Chapter VIII

Submission and peer-evaluation

The idea is to force you to THINK of what you do, and not simply follow rules without fully understanding why there are here.

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.