



PISCINE OBJECT

Module 04 - DESIGN PATTERN

Summary: This module will introduce you to the concept of Design Pattern.

Version: 1.01

Contents

I	Preamble	2
II	Introduction	3
III	General instructions	4
IV	Exercice 00: Preparation	7
V	Exercice 01: Singleton	8
VI	Exercice 02: Factory and Command	10
VII	Exercice 03: Mediator	11
VIII	Exercice 04: Observer	15
IX	Exercice 05: Facade	17
X	Submission and peer-evaluation	19

Chapter I

Preamble

Before the Gang of Four's design patterns, programming was a bit like cooking for a broke student.

You had a small arsenal of simple recipes for making basic dishes, but if you wanted to make something more elaborate, you had to dig through cookbooks and improvise with the available ingredients.

The results were often mixed: you could get a tasty dish, but it was often difficult to reproduce or modify it without breaking everything.

But fortunately, the Gang of Four came up with their tried and tested recipes

Chapter II

Introduction

The Gang of Four (GoF) design pattern is a collection of design patterns that provide proven solutions for common problems encountered in object-oriented software design. The patterns were described in the book "Design Patterns: Elements of Reusable Object-Oriented Software" by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, also known as the Gang of Four.

The main purpose of design patterns is to provide proven solutions for common software design problems, by reusing already tested and approved design concepts.

Design patterns are not a universal solution to all software design problems, but rather a set of solutions for common problems that can be adapted to the specific needs of the project.

It is important to note that design patterns are not a recipe for software design. They do not replace the creativity, experience and common sense of the developer. Rather, design patterns are a toolkit that can help developers solve common problems in an efficient and consistent manner.

Here are some example of Gang of Four design patterns:

- Flyweight: Share objects to conserve memory.
- Template Method: Define a common method structure, but allow subclasses to override some steps.
- Adapter: Adapt an existing interface to a new expected interface.
- Visitor: Allow adding new features to an existing object without modifying its structure.
- Command: Encapsulate an action as an object, allowing the handling of this action to be parameterized.
- Builder: Separate the construction of a complex object from its representation so that the same construction process can create different types of representations.
- Bridge: Decouple an abstraction from its implementation so that they can vary independently.

By learning and using the Gang of Four design patterns, developers can improve the quality, maintainability and reusability of their code.

Practical exercises are an excellent way to consolidate your understanding of design pattern concepts and apply them in real-world situations.

Chapter III

General instructions

In this module, you will be asked to implement a group of design pattern. Some of them required you to use template, some don't. Some will make you use some inheritance and polymorphism, some don't.

So, if you're not familiar with those subjects, you should review your C++ basic skills, as you will need those here.

In this module, if your non-templated methods does not exceed 3 instructions, you're allowed to implement it directly inside the header file, such as follow :

```
class Foo
{
private:
    int _value;
    char *_string;

    void _initializeString(); //Method too long to be placed in header
public:
    Foo() : //Constructor with 0 instructions
        _value(0),
        _string(nullptr)
    {

    }
    Foo(char *p_string); //Method too long to be placed in header
    const int& value() const // Simple getter, no check, only returning the value
    {
        return (_value);
    }
    const char *string() //Getter with a call for variable initialisation method
    {
        if (_string == nullptr)
            _initializeString();
        return (_string);
    }
    const char *string() const //Getter const isn't allow to initialize variable
    {
        if (_string == nullptr)
            throw std::runtime_error("No string initialized");
        return (_string);
    }
}
```

Instructions does NOT mean lines. Instruction are unit of code :



- one conditionnal loop is one instruction
- one call for method is one instruction
- one return is one instruction.

Any exceeding instructions number method in header will result in an instant 0.



Templated methods are obviously exclude from the 3 instructions limitation, as you can't place them in .cpp files. But don't create templated methods just to espace the .cpp creation !



Don't forget to apply encapsulation in your code to ensure that your classes are properly protected and that access to variables is optimally managed. This is a crucial element for success in this exercise.



Be sure to properly define the ownership of each attribute by linking each attribute to its owner class. This will allow for efficient management of system resources, which is critical to the success of this exercise.



`/* DO NOT REPEAT YOUR CODE */`
If you end up rewriting the exact same code with only small differences, you must find a way to factor that code. Otherwise, you will get a mark of 0 on the evaluation.
`/* DO NOT REPEAT YOUR CODE */`



As this subject is focused on clean and easy to read code, you're allowed to edit anything such as type, signature, name etc, as long as it respect the required spirit of the feature. As exemple, if a method require you to return a `std::vector` of `Student*`, it mean that you're suppose to return a container, storing but not owning `Student*`. You're allowed to edit this to return a `std::list` of `std::shared_ptr` `Student`.

Chapter IV

Exercice 00: Preparation

	Exercise 00
	Exercice 00: School preparation
	Turn-in directory : <i>ex00/</i>
	Files to turn in : <i>ex00/*.cpp, ex00/*.hpp</i>
	Allowed functions : standard STD containers and structure

In this exercice, you're suppose to open the attachment folder, where you should find a datas.hpp file, containing multiples structures, classes and some enums that we nicely pre-created for you !

You must read it, and to decompose it at least into multiple .hpp.

Once you're satisfied with your enums/classes/structures repartitions, you should move to the next exercice !

Chapter V

Exercice 01: Singleton

	Exercise 01
	Exercice 01: Singletons
	Turn-in directory : <i>ex01/</i>
	Files to turn in : <i>ex01/*.cpp</i> , <i>ex01/*.hpp</i> , <i>ex01/singetons.hpp</i> , <i>ex01/main.cpp</i>
	Allowed functions : standard STD containers and structure

You must create multiples objects, for the propose of storing every student, every staff, every course and every room of your simulated school.

You should create the following objects :

- StudentList, holding every student existing in the school.
- StaffList, holding every staff existing in the school.
- CourseList, holding every course currently running in the school.
- RoomList, holding every room existing in the school.

Those object should have at least a method allowing user to access the object, a method to add things inside those objects, and one to retrieve them.

Those newly created objects need to exist only once for all possible threads of the program, and should be joinable from anywhere they have been include in.



`/* DO NOT REPEAT YOUR CODE */`
Don't forget what we said during the general instruction part :
repeating your code will result in a 0.
C++ added the template feature, it's not for nothing.
`/* DO NOT REPEAT YOUR CODE */`

You should create a main, where you need to be able to instanciate both of those object, and populate them manualy, by inserting new element inside each of the object you created, and been able to retrieve them.



the name of the exercice isn't probably here just to look cool, maybe you should look out for it, on the internet !

Chapter VI

Exercice 02: Factory and Command

	Exercise 02
Exercice 02: Factory and Command	
Turn-in directory : <i>ex02/</i>	
Files to turn in : <i>ex02/*.cpp</i> , <i>ex02/*.hpp</i> , <i>ex02/main.cpp</i>	
Allowed functions : standard STD containers and structure	

Headmasters and secretaries are workers that work hand in hand : the secretary department create forms that the headmaster receive and execute, once it have been complete with mandatory informations about what those forms need to do.

As such, you're requested to create the system that will allow Secretary to create form, depending of the enum FormType it will receive as input for the `createForm` method. Once the form is created by the secretary, you will have to fill it with relevant informations, than give it to the headmaster, to be signed and executed.

You also have to complete the 4 types of forms we have pre-created for you. For this propose, we strongly advice you to :

- Add any attributes inside those forms.
- Add any method inside both those forms to fill the new attributes with datas.
- Create the execute method for the new forms, to allow them to execute there propose once the headmaster have accepted it.



You're allow to edit the Form or any derived classes, if you may find that anything is missing inside it.

Once those forms are created, you must create a main where you instanciate some, give them to the headmaster or not, and try to execute them, to see that when they are signed, they do have an effect, and when they are not, they aren't executed properly.

Chapter VII

Exercice 03: Mediator

	Exercise 03
	Exercice 03: Mediator
	Turn-in directory : <i>ex03/</i>
	Files to turn in : <i>ex03/*.cpp</i> , <i>ex03/*.hpp</i> , <i>ex03/main.cpp</i>
	Allowed functions : standard STD containers and structure

Now that we have created those forms, it's time to animate this school!

That's the role of the headmaster: he's in charge of giving courses to professors and assigning students to courses!

The idea is quite simple : everything must pass by the headmaster hand, who will have to sign every action happening in school.



Perhaps this statement has something to do with the name of the exercise, I'm not sure...

He will, for each action, require a form from the secretaries. Once the requester has fill the form information, he will sign it and execute what's required.

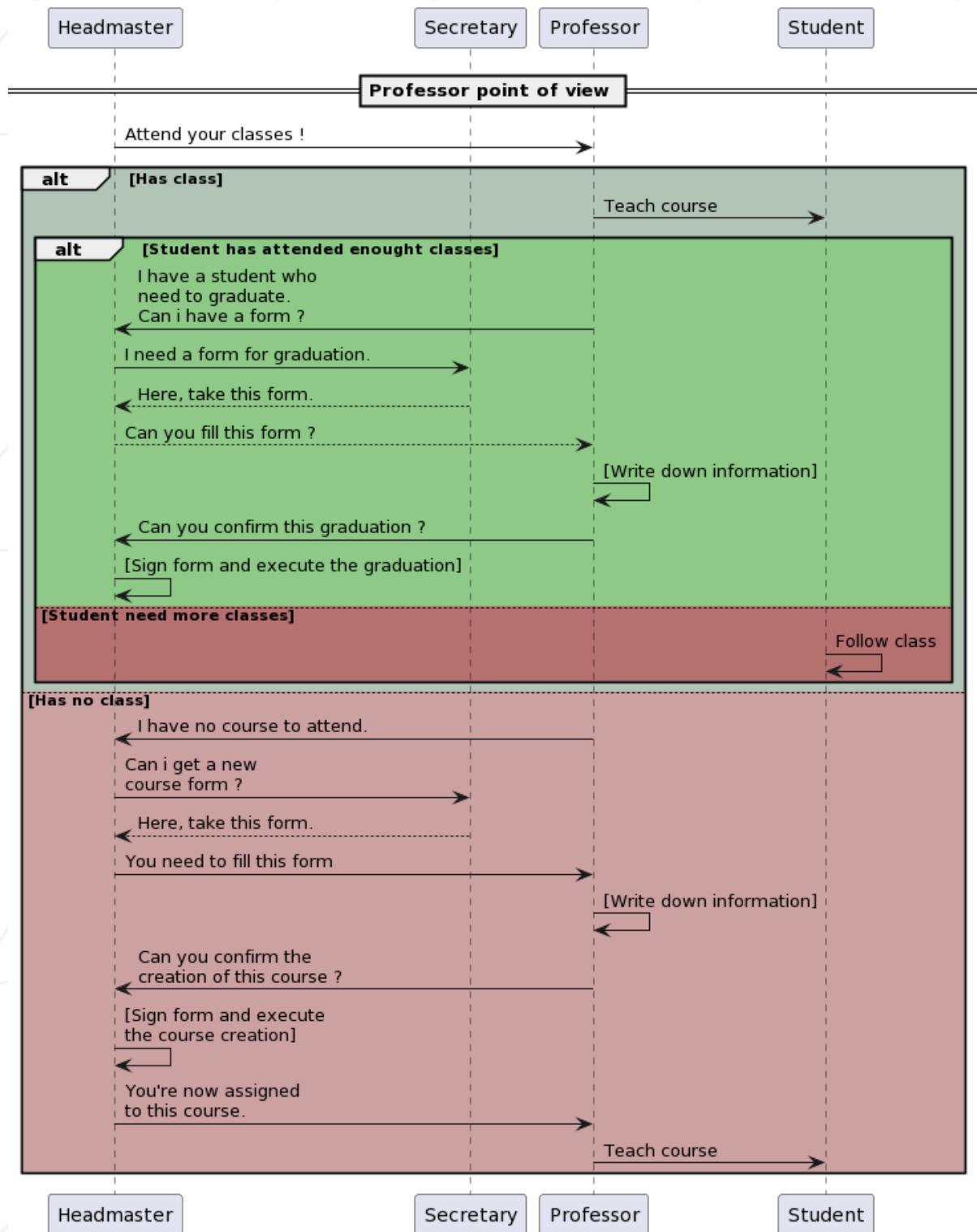
You must complete the Headmaster class to perform this behavior.

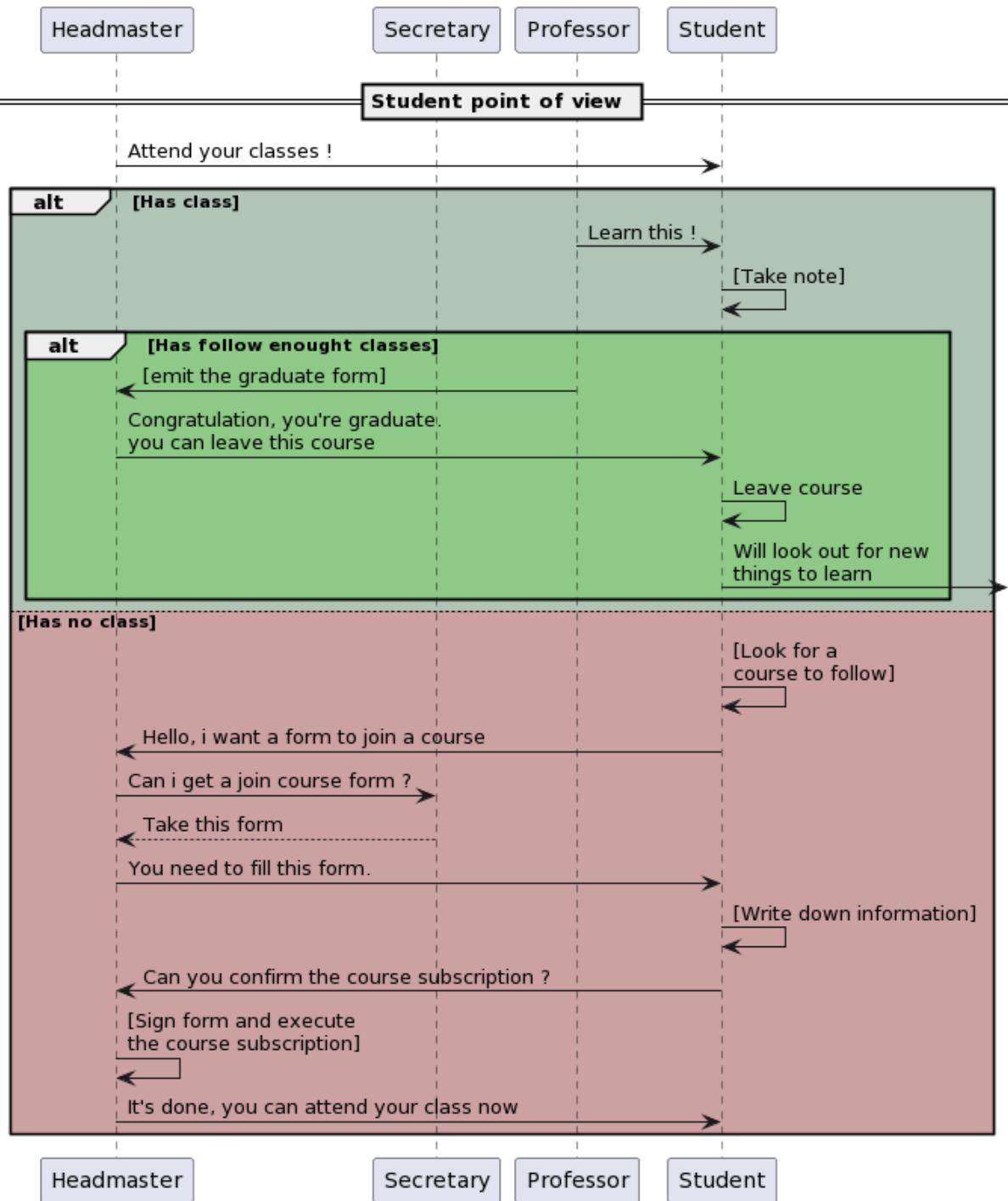
You're also required to implement a system for professors to create new Course, and for student to join them.

We will give you an example of a set of interaction between Headmaster, Secretary, professor and student, to illustrate the interaction we expect you to create.

It's express as a dialogue between them, but feel free to create a "real" sequence diagram, by interpreting this diagram !

You're required to implement the following systems :





Once you're done those 2 types of interactions, you must create the last form type : the room creation !

The idea is quite simple : if, when trying to start a class, a professor does not find an free classroom, it will be able to ask the headmaster to magically create one !

Try to guess how it will be done, following the last 2 examples.

You must provide a main that demonstrates the following features:

- The headmaster can require professors to attend their classes.
- If a professor has no course, they must be provided with a new one.
- If a professor does not find an empty class room, they must be provided with a new one.
- Students who are not enrolled in a course will try to join one.
- Once a student has completed the required number of classes, they will be graduated by the headmaster, based on the recommendation of the course professor

Chapter VIII

Exercice 04: Observer

	Exercise 04
	Exercice 04: Observer
Turn-in directory : <i>ex04/</i>	
Files to turn in : <i>ex04/*.cpp, ex04/*.hpp, ex04/observer.hpp, ex04/main.cpp</i>	
Allowed functions : standard STD containers and structure	

Now that the headmaster can give professors and students classes to attend, they will exhaust themselves at work ! We may need to give them free time, at regular interval. For this propose, you will have to simulate a bell system.

The idea is quite simple : every time students or professor hear the bell ringing, they are allowed to move out of classes.

Once there free time is passed, the bell will ring again, and they will return in class.

It will be the role of the Headmaster to ring this bell !

You have been given an enum in the entry file, the Event enum, that will represent this bell ring event.

You have to create a class where student and professor will be able to be notified when the bell is ringing.



As always, the name of the exercice isn't here for nothing, it may help you !



You MUST make the BELL indicate to the students and professors that the pause start or is over. Making the professors or students check is the pause is over will result in a 0.

You must provide a main, showing how your system work.

Chapter IX

Exercice 05: Facade

	Exercise 05
	Exercice 05: Facade
	Turn-in directory : <i>ex05/</i>
	Files to turn in : <i>ex05/*.cpp</i> , <i>ex05/*.hpp</i> , <i>ex05/main.cpp</i>
	Allowed functions : standard STD containers and structure

Now that your school seem to work correctly, with the headmaster doing the headmasterish work, the professors teaching, the secretaries doing secretary job and the students learning things, it's time to create a class to encapsulate all this complexe process !

This will do inside a new class, called School, that you must create.

This new class must contain a method to do the following actions :

- `void runDayRoutine()` : execute the school day routine, composed of :
 1. launch classes
 2. allow student and professor to go on recreation
 3. launch classes
 4. launch lunch
 5. launch classes
 6. allow student and professor to go on recreation
 7. launch classes
- `void launchClasses()` : Make the headmaster ask the students and professors to attend them classes

- `void requestRingBell()` : Make the headmaster ring the bell
- `void recruteProfessor()` : Add a new professor to the school
- `void recruteStudent()` : Add a new student to the school
- `Course* getCourse(std::string p_name)` : return a course with a given name
- `std::vector<Student*> getStudents()` : return a vector containing every student in school
- `std::vector<Professor*> getProfessors()` : return a vector containing every professor in school
- `Headmaster getHeadmaster()` : return the headmaster of the school
- `Secretary getSecretary()` : return the secretary of the school
- `void graduationCeremony()` : launch the graduation of every student that can graduate at a desired moment

As you may have guessed, maybe you will have to edit the way we've made things previously ! You may add a new Event, edit the way people graduate, i don't know !

Chapter X

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.

For this exact reason, you will be asked question about everything you've done, so think about it, and be ready to discuss choice you may have made over architecture or implementation !

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.