COS 214 Practical Assignment 3

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

- Date Issued: **16 August 2022**
- Date Due: **30 August 2022** at **11:00**
- Submission Procedure: **Upload via ClickUP**
- Submission Format: **zip or tar + gzip archive (tar.gz)**

1 Introduction

1.1 Objectives

During this practical assignment you will be required to implement the *Abstract Factory*, *Strategy*, *State*, *Composite* and *Decorator* design patterns.

1.2 Outcomes

After successful completion of this assignment you should be comfortable with the following:

- Creating UML class, object and state diagrams from your classes
- Understanding the Abstract Factory Design Pattern
- Understanding the Strategy Design Pattern
- Understanding the State Design Pattern
- Understanding the Composite Design Pattern
- Understanding the Decorator Design Pattern
- Understand how polymorphism works with virtual functions in C++

2 Constraints

1. You must complete this assignment individually.
2. You may ask the Teaching Assistants for help but they will not be allowed to give you the solutions.

3 Submission Instructions

You are required to upload all your source files and your exported class and state diagrams as a single tar.gz archive to ClickUP before the deadline. You are required to implement all makefiles, headers and source files yourself. You should create a Main.cpp to test and demonstrate the functionality of your code.

4 Mark Allocation

Task	**Marks**
The Confectionery Hierarchy	20
Managing Confectionary Production	30
Confectionery Hampers	25
The Pokemon Class	6
The State Pattern	12
The Strategy Pattern	12
Pokemon Battle	20
TOTAL	125

5 Assignment Instructions

5.1 SECTION A

Your are requested to assist an industrial engineer in modelling the manufacturing of various confectioneries by three of the biggest organisations worldwide, namely Cadbury, Nestle and Lindt. You are will be modelling two products, namely chocolate and aerated chocolate.

This section consists of four tasks that build on each other:

1. The confectionery hierarchy
2. The abstract factory pattern
3. A main function to illustrate the use of the design pattern
4. UML class diagrams

Task 1: The Confectionery Hierarchy ..... (20 marks)

In this task you will create the classes for the different types of confectioneries.

- An abstract `Confectionery` Class.
- `Chocolate` and `AeratedChocolate` classes that inherit from `Confectionery`.
- Classes for the concrete units: `DiaryMilk` (Cadbury chocolate), `DiaryMilkBubbly` (Cadbury aerated chocolate), `MilkyBar` (Nestle chocolate), `Areo` (Nestle aerated chocolate) and `Lindor` (Lindt chocolate).

1.1 **The Confectionery Class**

The Confectionery class should have at least the following members:

- **Private:**
  - manufacturer - a string eg. Cadbury
  - price - a double
  - type - a string eg. Bar, Aerated
  - id - initialise this based on a static counter in the Confectionery class that is incremented with each call to the constructor.
- **Public:**
  - A constructor that takes 3 arguments to initialise the manufacturer, price and type.
  - A getDescription function that returns a string describing the confectionery. This string should contain all the member variables.

1.2 **Chocolate**

The Chocolate class inherits from Confectionery.

- It should have a private member variable 'slab' (a boolean)
- The constructor should take 3 arguments: manufacturer, price and slab
- You should override the description function to include the slab variable (whether the chocolate is a slab or mini bar of chocolate). Your overridden function should include a call to the parent class description function.

Hint: Make sure the Chocolate's description function is called even when you store one of your chocolates in a `Confectionery*`

1.3 **AeratedChocolate**

Similarly to the Chocolate class, the AeratedChocolate class should inherit from Confectionery.

- It should have a private member variable 'bubblespccm' (bubbles per cubic centimeter) (an integer)
- The constructor should take 3 arguments: manufacturer, price and bubblespccm.

- As in the Chocolate class, you should override the description function to include the bubbles per cubic centimeter.

1.4 **The Concrete Items**
The DiaryMilk, MilkyBar and Lindor classes inherit from the Chocolate class, while the DiaryMilkBubbly and Areo classes inherit from the AeratedChocolate class. These classes will fulfil the role of the concrete products in the pattern, and should each have only a constructor. This constructor calls the constructor of the base class with the correct values. You can choose appropriate values for the price, while the slab and bubblespccm members of the different confectionery items should be chosen by the user in your main program and passed in the constructor.

Task 2: Managing Confectionary Production ..... (30 marks)

In this task you will need to use the abstract factory design pattern to create your different confectioneries items. You will need the following classes:

- An abstract ConfectioneryFactory class with functions to create a Chocolate and AeratedChocolate item.
- Concrete factories for Cadbury, Nestle and Lindt.

2.1 Notice that Lindt does not produce an AeratedChocolate product. The GoF description of the pattern assumes that there is always full parity for all the AbstractProducts, but this is often not the case in industry. Explain in your submitted PDF one way to handle this scenario using fundamental programming concepts as introduced in COS 132 up to and excluding the introduction of classes, i.e. using procedural concepts. (3)

2.2 In the previous question you were required to provide an imperative programming solution for product classes not having full parity. Explain in your submitted PDF one way to handle this scenario using an object-oriented approach, i.e. using classes. Implement this solution in your classes. (3)

2.3 Create a main function that can create different confectionery items based on user input for manufacturer, type, another variable specific to the type of confectionery and amount. It should illustrate the usage of the abstract factory design pattern. Ensure to accommodate for the scenario where a manufacturer doesn't produce a specific abstract product. (14)

In your main you should store your different items in a `Confectionery**`. All dynamic memory should be deallocated so that your program does not cause memory leaks.

Hint: Place an output statement in your factory constructor to clearly show when confectionery items are created and output a Confectionery item's details when you create it. Similarly include an output statement in your destructors to illustrate which items get destroyed when the associated memory is freed.

2.4 Finalise a complete class diagram using Visual Paradigm from all the classes of this section. Annotate the diagram with the roles the different classes have in the Abstract Factory design pattern. Export the class diagram include it in your PDF submission. (10)

Note: You should be designing the classes in UML before implementing. This is just the finalisation of the class diagram. You should not be drawing it for the first time for this question.

Task 3: Confectionery Hampers ..... (25 marks)

3.1 Lynnwood Florest offers a service where they allow clients to request the creation of confectionery hampers for gifts. Amend your code using the Composite pattern to allow a user to create a confectionery gift basket that consists entirely out of `Confectionery` objects. Amend your main.cpp from Question 2.3 to request the user which items to add to the confectionery basket. As items are added to the hamper, utilise your factories to produce the confectionery items. (6)

3.2 Add a recursive function to your main to calculate the total price of the confectionery gift basket and the list of confectionery items in the hamper. (4)

3.3 If the gift hamper is requested on a recognised public holiday, certain discounts can be applied to the final basket price. Use the Decorator pattern to implement the discount mechanism. (5)

- Valentine's Day (14 February) – 5% discount

- Mother's Day (14 May) – 3.5% discount
- Spring Day (1 September) – 7% discount

Important: Only one discount decorator class may be applied to the gift hamper.

3.4 The florest allows clients to add some extra add-ons to the gift hamper. Some add-ons may have a cost associated while others do not. For some add-ons, discounts are allowed, while for others, discounts should not apply. For more information refer to the table below. Amend your main.cpp file to print out the add-on applied to the hamper, as well as any additional information about the add-on, such as color, message, etc. (10)

Add-on	Price	Discount Allowed	Multiple Allowed	Member Variable	Variable Description
Ribbon	7.75	No	Yes	Color	Color of the ribbon
Note	Free	Not Applicable	No	Message	Message on the note
Card	25.90	Yes	No	Message	Message on the card
Flower	3.99	Yes	Yes	Variety	Variety/type of the flower

Hint: The order in which the decorators are applied will be critical in ensuring that the above business rules are obeyed. Recursion could be of crucial value here to determine whether a certain decorator can be applied or not.

5.2 SECTION B

In this section, you must make use of the Strategy and State design patterns to illustrate the different play-styles that occur depending on the state of a Pokemon during a turn based console battle game.

This section consists of five tasks that build on each other:

3. The Pokemon class
4. The State pattern
5. The Strategy pattern
6. A main function to illustrate the use of the design patterns used
7. UML object and state diagrams

Task 4: The Pokemon Class ..... (6 marks)

In this task you will create a very basic Pokemon class to demonstrate the use of the Strategy and State design patterns. You will expand this class in the following tasks.

The Pokemon class should have at least the following members:

- **Private:**
  - name - a string e.g. "Pikacu"
  - HP (Health Points) - an integer
  - damage - an integer

Task 5: The State Pattern ..... (12 marks)

In this task you will implement the state pattern to illustrate the attack behaviour that can occur during battles.

5.1 **The BattleState Class**

The BattleState class will fulfil the role of the State participant in the state design pattern, and should have the following members:

- **Protected:**
  - battleStyle - a string. This will always be either "normal", "agile" or "strong".
- **Public:**
  - handle - this is an abstract method that takes two parameters. The first is the name of the Pokemon and the second is an integer parameter `damage`. This abstract function returns the amended damage based on the battle style.
  - getBattleStyle - this function returns the value stored in the `battleStyle` member variable for state identification during a run.

5.2 **The Concrete BattleState Classes**
You will implement the NormalBattleState, AgileBattleState and StrongBattleState concrete state classes, which will inherit from the BattleState class.

Each of these classes will have at least the following members:

- **Public:**
  - a constructor which take no parameters and initialises the `battleStyle` member variable to the corresponding value stated in the previous section ("normal", "agile" or "strong")
  - handle - This method must be implemented for all three concrete battle state classes and will print out the current battle state and damage that will be dealt. The amended damage value is also returned, as follows for each different class:
    - NormalBattleState class - "<name> has no special battle state, normal battle attack will deal <damage> points." The damage dealt is the same as the parameter value.
    - AgileBattleState class - "<name> has selected an agile battle state, and is allowed two battle attacks in one turn and will deal <damage> points". The damage dealt is $\lfloor \frac{3}{4} \rfloor$ of the parameter value.
    - StrongBattleState class - "<name> has selected a strong battle state, and will inflict <damage> points on the next battle attack but misses the following attack turn." The damage dealt is double the parameter value.

5.3 **Updating the Pokemon Class**
The Pokemon class should be updated with the following:

- **Private:**
  - add the appropriate private member variable to the Pokemon class so it can fulfil the role of the Context participant in the State design pattern.
- **Public:**
  - selectBattleState - this method will select a battle state for the chosen Pokemon and call the concrete BattleState's handle method to print out the battle state and damage that will be dealt.

Task 6: The Strategy Pattern ..... (12 marks)

In this task you will implement the strategy pattern to illustrate a Pokemon's different play-styles.

6.1 **The PlayStyle Hierarchy**
The PlayStyle hierarchy will consist of four classes:

- The PlayStyle class - This class represents the Strategy participant in the pattern, and will have an abstract `attack()` method, which takes no parameters and has a String return type.
- The AttackPlayStyle, PhysicalAttackPlayStyle and RunPlayStyle classes. These classes will represent the ConcreteStrategy participants in the pattern, and inherit from the PlayStyle class.
- The AttackPlayStyle, PhysicalAttackPlayStyle and RunPlayStyle classes will implement the PlayStyle class's `attack()` method to return a string unique to each class. Keep in mind that these strings will be combined with the Pokemon's name, so only partial sentences will be returned here:

- AttackPlayStyle class will return the string: "is attacking the opposing Pokemon."
- PhysicalAttackPlayStyle class will return the string: "is using physical ability to attack."
- RunPlayStyle class will return the string: "decides life is better than death and leaves the battle."

6.2 **Updating the Pokemon Class**
The Pokemon class should be updated with the following:

- **Private:**
  - add the appropriate private member variable to the Pokemon class so it can fulfil the role of the Context participant in the Strategy design pattern.
- **Public:**
  - add a constructor to the Pokemon class, that takes a name, HP, damage and a concrete PlayStyle, and initialises the appropriate member variables. Also remember that a Pokemon will always begin with a normal battle state.
  - attack() - a method that takes no parameters and returns the damage dealt to the opposing Pokemon depending on the current BattleState. This method utilises the Pokemon's specific PlayStyle to output a string consisting of the Pokemon's name, along with the PlayStyle's unique sentence, as described in the previous part, e.g. "Pikacu is attacking the opposing Pokemon." The play method should, however, only output this string if the Pokemon hasn't fainted, that is HP is greater than zero. In the case that a Pokemon has fainted, output "<name> has fainted and can not attack."
  - Add a function to allow the setting of a Pokemon's PlayStyle.
  - Add a function to allow a Pokemon to takeDamage.

Task 7: Pokemon Battle ..... (20 marks)

7.1 Create a simple turn based Pokemon console battle game, between yourself and the NPC Pokemon's. Ensure to allow a user to create a single or group of Pokemon in your game, depending on your implementation. Your game should clearly demonstrate your implementation of the State and Strategy design patterns by using the Pokemon's `select()` and `attack()` methods to show how changes in battle states changes the damage dealt and how changes in play style causes changes in the battle. Bonus marks will be awarded for effort and creativity. (10)

7.2 Draw the final UML class diagram showing all the classes and relationships between the classes for your game. Save the diagram as *SystemUMLClassDiagram.pdf* and make sure you upload it along with all your source code and other files. (10)