COS 214 Practical Assignment 2

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

- Date Issued: **31 August 2021**
- Date Due: **14 September 2021** at **08:00**
- 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* and *State* 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
- 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 Merchandise Hierarchy	18
Abstract Factory Pattern	8
Main	10
UML Class Diagram	14
The SoccerPlayer Class	2
The State Pattern	13
The Strategy Pattern	13
Main	10
UML Object and State Diagram	12
TOTAL	100

5 Assignment Instructions

5.1 SECTION A

Three different soccer clubs (Chelsea, Arsenal and Liverpool) have approached you, and tasked you with helping them produce their merchandise.

Each club wants you to help them produce soccer-balls and shirts that are specific to the club.

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

1. The merchandise 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 Merchandise Hierarchy ..... (18 marks)

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

- An abstract Merchandise Class.
- SoccerBall and Shirt classes that inherit from Merchandise.
- Classes for the concrete units: ChelseaSoccerBall, ArsenalSoccerBall, LiverpoolSoccerBall, ChelseaShirt, ArsenalShirt and LiverpoolShirt.

1.1 **The Merchandise Class**

The Merchandise class should have at least the following members:

- **Private:**
  - club - a string eg. Chelsea
  - price - a double
  - type - a string eg. Ball
  - id - initialise this based on a static counter in the Merchandise class that is incremented with each call to the constructor.
- **Public:**
  - A constructor that takes 3 arguments to initialise the club, price and type.
  - A getDescription function that returns a string describing the piece of merchandise. This string should contain all the member variables.

1.2 **SoccerBall**

The SoccerBall class inherits from Merchandise.

- It should have a private member variable 'inflated' (a boolean)
- The constructor should take 3 arguments: club, price and inflated
- You should override the description function to include the inflated variable (whether the ball is inflated or not). Your overridden function should include a call to the parent class description function.

Hint: Make sure the SoccerBall's description function is called even when you store one of your soccer-balls in a `Merchandise*`

1.3 **Shirt**

Similarly to the SoccerBall class, the Shirt class should inherit from Merchandise.

- It should have a private member variable 'size' (a string)

- The constructor should take 3 arguments: club, price and size.
- As in the SoccerBall class, you should override the description function to include the size.

1.4 **The Concrete Items**
The ChelseaSoccerBall, ArsenalSoccerBall, LiverpoolSoccerBall classes inherit from the SoccerBall class, while the ChelseaShirt, ArsenalShirt and LiverpoolShirt classes inherit from the Shirt 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 inflated and size members of the different merchandise items should be chosen by the user in your main program and passed in the constructor.

Task 2: Abstract Factory Pattern ..... (8 marks)

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

- An abstract MerchandiseFactory class with functions to create a SoccerBall and Shirt item.
- Concrete factories for Chelsea, Arsenal and Liverpool Merchandise.

Task 3: Main ..... (10 marks)

Create a main function that can create different merchandise items based on user input for type, club, another variable specific to the type of merchandise and amount. It should illustrate the usage of the abstract factory design pattern.

In your main you should store your different items in a `Merchandise**`. 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 merchandise items are created and output a Merchandise item's details when you create it.

An output example is included at the end of the document. You do not need to follow it exactly.

Task 4: UML Class Diagram ..... (14 marks)

Create a complete class diagram 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 to a png image and upload it with your code.

Software: You can use Visual Paradigm to create your class diagrams. The license details have been made available on the COS214 page on the CS website.

5.2 SECTION B

In this section, you must make use of the Strategy and State design patterns to illustrate the different play-styles and different states a soccer player can be in during a soccer match.

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

5. The SoccerPlayer class
6. The state pattern
7. The strategy pattern
8. A main function to illustrate the use of the design patterns used
9. UML object and state diagrams

Task 5: The SoccerPlayer Class ..... (2 marks)

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

The SoccerPlayer class should have at least the following members:

- **Private:**
  - name - a string eg. "John"

Task 6: The State Pattern ..... (13 marks)

In this task you will implement the state pattern to illustrate the different punishments that can be dealt to soccer players when they commit fouls.

6.1 The CardState Class

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

- **Protected:**
  - cardColour - a string. This will always be either "red", "yellow" or "none".
- **Public:**
  - handle - this is an abstract method that does not have any parameters or a return value.
  - changeCardState - this is an abstract method that does not have any parameters and returns an instance of the CardState class.
  - getCardColour - this function returns the value stored in the cardColour member variable for state identification during a run.

6.2 The Concrete CardState Classes

You will implement the NoCardState, YellowCardState and RedCardState concrete state classes, which will inherit from the CardState class.

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

- **Public:**
  - a constructor which takes no parameters and initialises the cardColour member variable to the corresponding value stated in the previous section ("none", "yellow" or "red")
  - handle - this method must be implemented for all three concrete card classes and will print out the current card colour and the next card state, as follows for each different class:
    - NoCardState class - "The player hasn't committed any previous fouls, and will now be given a yellow card."
    - YellowCardState class - "The player has already received a yellow card, and will now be given a red card."
    - RedCardState class - "The player has already been sent off with a red card."
  - changeCardState - this method must be implemented for all three concrete card classes and will return a new instance of the concrete CardState which follows the current CardState:
    - NoCardState class - return a new instance of the YellowCardState class.
    - YellowCardState class - return a new instance of the RedCardState class.
    - RedCardState class - return null.

6.3 Updating the SoccerPlayer Class

The SoccerPlayer class should be updated with the following:

- **Private:**
  - add the appropriate private member variable to the SoccerPlayer class so it can fulfil the role of the Context participant in the State design pattern.
- **Public:**

- commitFoul - this method will call the SoccerPlayer's concrete CardState's handle method to print out how the foul affects the player's card status, and then change the SoccerPlayer's CardState to the next appropriate card.

Task 7: The Strategy Pattern ..... (13 marks)

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

7.1 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 play() method, which takes no parameters and has a String return type.
- The AttackPlayStyle, DefendPlayStyle and PossessionPlayStyle classes. These classes will represent the ConcreteStrategy participants in the pattern, and inherit from the PlayStyle class.
- The AttackPlayStyle, DefendPlayStyle and PossessionPlayStyle classes will implement the PlayStyle class's play() method to return a string unique to each class. Keep in mind that these strings will be combined the SoccerPlayer's name, so only partial sentences will be returned here:
  - AttackPlayStyle class will return the string: "is attacking the opposition's goal with the ball."
  - DefensivePlayStyle class will return the string: "is defending their team's half of the field."
  - PossesionPlayStyle class will return the string: "is helping their team keep possession of the ball by passing it to their teammates."

7.2 Updating the SoccerPlayer Class

The SoccerPlayer class should be updated with the following:

- **Private:**
  - add the appropriate private member variable to the SoccerPlayer class so it can fulfil the role of the Context participant in the Strategy design pattern.
- **Public:**
  - add a constructor to the SoccerPlayer class, that takes a name and a concrete PlayStyle, and initialises the appropriate member variables. Also remember that a soccer player will always begin in a state where they haven't received any cards yet.
  - play - a method that takes no parameters and does not have a return value. This method utilises the SoccerPlayer's specific PlayStyle to output a string consisting of the player's name, along with the PlayStyle's unique sentence, as described in the previous part, e.g. "John is attacking the opposition's goal with the ball.". The play method should, however, only output this string if the player hasn't been sent off with a red card. In the case that a player has been sent off, output "The player cannot demonstrate their play-style, as they have been sent off."
  - Add a function to allow the setting of a SoccerPlayer's PlayStyle.

Task 8: Main ..... (10 marks)

Create a main function that can create a soccer player with a specific name and play-style, and then demonstrate your implementation of the state and strategy design patterns by using the soccer player's functions to show how a player's play-style can be changed, and what happens when a player continues making fouls.

Task 9: UML Object and State Diagram ..... (12 marks)

9.1 Object Diagram

Create an object diagram for a complete instance of the SoccerPlayer class that you have created in your main.

9.2 UML State Diagram

Take a scenario where a soccer player can be in different states based on the cards that they receive as described in this practical. However, when a player commits a foul, it is given a foulScore between 1 and 10, based on how severe the foul was. Players begin a game without having received cards previously, and their games end as soon as they receive a red card, which means they are sent off. Additionally to the

practical, players can also immediately receive a red card when a foul receives a foulScore of 7 or higher. Create a UML state diagram depicting this scenario.

Export the object and state diagrams to PNG images and upload them with your code.

```
Choose the type of merchandise you want to create: Ball=1, Shirt=2 > 2
Choose the club which merchandise should be created: Chelsea=1, Arsenal=2, Liverpool=3 > 1
Choose the size of the shirt(s) to be created > Medium
How many merchandise items should be created? > 3
Creating - Item #1: Chelsea Shirt, Price: 50, Size: Medium
Creating - Item #2: Chelsea Shirt, Price: 50, Size: Medium
Creating - Item #3: Chelsea Shirt, Price: 50, Size: Medium

Enter 1 to create more items or 0 to stop > 1

Choose the type of merchandise you want to create: Ball=1, Shirt=2 > 1
Choose the club which merchandise should be created: Chelsea=1, Arsenal=2, Liverpool=3 > 2
Should the ball(s) be inflated? (y/n) > n
How many merchandise items should be created? > 2
Creating - Item #4: Arsenal Ball, Price: 150, Inflated: False
Creating - Item #5: Arsenal Ball, Price: 150, Inflated: False

Enter 1 to create more items or 0 to stop >
```

An output example for Section A.