COS 214 Practical Assignment 1

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

- Date Issued: **11 August 2020**
- Date Due: **25 August 2020** at **8:00am**
- Submission Procedure: **Upload via Clickup**
- Submission Format: **archive (zip or tar.gz)**

1 Introduction

1.1 Objectives

In this practical you will:

- implement the Template Method design pattern;
- implement the Factory Method design pattern
- implement the Abstract Factory design pattern
- implement the Prototype design pattern;
- implement the Memento pattern; and
- integrate the patterns.

1.2 Outcomes

When you have completed this practical you should:

- understand the Template Method and be able to use C++ concepts like virtual functions and inheritance to implement it;
- understand how the Abstract Factory pattern provides an interface for creating families of related objects without specifying concrete classes;
- understand how the Factory Method delegates object creation to its subclasses;
- notice the difference between the Prototype and the Factory Method and understand which to use where; and
- apply the Memento to store the state of objects and re-instate the state at a later stage.

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 (that is `.h` and `.cpp`), your Makefile, UML diagrams as individual or a single PDF document, a text file labelled ”readme” explaining how to run the program and any data files you may have created, in a single archive to Clickup before the deadline.

4 Mark Allocation

Task	**Marks**
Defining predators	24
Creating predators	35
Clone the Prey	5
Let the hunting begin...	36
TOTAL	100

5 Assignment Instructions

Predator vs Prey. You have been tasked with creating hunting packs, putting different predators to the test. Being a sensible computer scientist, you decide to model the problem using Design Patterns and UML Class diagrams and implement the solution in C++ to determine the best predator around.

Task 1: Defining predators .... (24 marks)

Predators can be one of four types:

- Lion
- Cheetah
- Wolf
- Wild Dog

1.1 Create an abstract class `Predator`. Each Predator has: (5)

- Health points (HP)
- A primary hunting method
- The damage inflicted
- A speciality

Predators also have a `hunt` method. Although different types of predators have different hunting styles, all hunts follow the same basic steps, with the prey and predator both standing a chance to inflict damage. Pseudocode for this is given by:

```
Function hunt(Prey)
    while Predator and Prey still alive
        if Predator health points less than 5
            Predator uses speciality
        endif
        if catchPrey function returns true
            Prey caught
            if getAttacked function returns true
                Predator dies.
            else
                Predator attacks
            endif
        else
            Predator loses a health point
        endif
    endwhile
endfunction
```

Each Predator therefore has the following operations (functions or methods):

- relevant constructors and a destructor
- relevant getters and setters
- `hunt` which takes an instance of a Prey as parameter and returns `void`
- `catchPrey` which takes an instance of a Prey as parameter and returns `bool`, defined as `bool catchPrey(Prey* p)`
- `getAttacked` which takes an instance of Prey as parameter and returns `bool`, defined by `bool getAttacked(Prey *p)`
- `attack` which takes an instance of Prey as parameter and returns `bool`, defined by `bool attack(Prey *p)`
- `die` which takes no parameters and returns `void`

- `speciality` which takes no parameters and returns `void`

1.2 Write the `Lion`, `Cheetah`, `Wolf` and `Wild Dog` classes which inherit from the `Predator` class. These subclasses will not override Predator's `hunt` method, because their hunts always follow this same pattern. Instead, the subclasses will implement the 5 methods (primitive operations) as follows: (10)

1. `bool catchPrey(Prey *p)` – The predator attempts to catch the prey by calling the Prey's `run` method.
2. `bool getAttacked(Prey *p)` – If the Prey is caught, it's `fight` method is called. It returns an int =< 0. If the returned value equals 0, the prey doesn't fight, if the returned value is greater than 0, the prey inflicts damage on predator. The damage done by the Prey can be obtained by calling the Prey's `getDamage` method. The damage done by the Prey should be subtracted from the Predators's HP. The `getAttacked` function should return true if the Predator is killed.
3. `bool attack(Prey *p)` – The Predator attacks the Prey by calling the Prey's `takeDamage` method which takes the Predator's damage inflicted value as a parameter and returns the Prey's health points. If the Prey is killed (ie. when it's health points ≤ 0), the attack method should return true.
4. `void die()` – If the Prey's hit kills the Predator, the Predator dies.
5. `void speciality()` – The Predator's special ability may be used when health points are less than 5. This function increases the predators attack value by 10%. The increase remains in effect until either the Prey or Predator is killed.

The following table shows what each of the operations should output for each of the Predators:

Method Name	**Lion**	**Cheetah**	**Wolf**	**Wild Dog**
catchPrey	The lion pounces into action to catch the <preyType>.	The cheetah sprints forward with its eye on the <preyType>	The wolf sneaks up to the <preyType>	The wild dog howls as it measures up the <preyType>
getAttacked	The <preyType> stands on the lions tail inflicting <damage> damage!	The <preyType> side steps the cheetah, kicks back and causes <damage> damage in the process.	The <preyType> spots the wolf, jumps onto it's back imposing <damage> damage.	The <preyType> rams into the wild dog removing <damage> health points.
attack	The lion uses <huntingMethod> to inflict <damage> damage on the <preyType>.	The cheetah causes <damage> damage to the <preyType> by using <huntingMethod>.	The wolf's <huntingMethod> caused <damage> damage to the <preyType>.	The wild dog's <huntingMethod> pays off leaving it's <preyType> with <damage> health points less.
die	Long lived the King.	The hunter becomes the hunted.	Why so afraid of the big bad wolf?	No more hunting for this old dog.
speciality	The injured lion uses <speciality>.	The tired cheetah uses <speciality>.	The wolf cunningly uses <speciality>.	The wild dog plays dead before using <speciality>.

1.3 Write your own main program to test your code. You can make use of the provided `Prey` class given in *Prey.h* and *Prey.cpp* to test your code.

1.4 Which design pattern have you just implemented? (1)

1.5 Draw the class diagram of the Predator hierarchy. (8)

Task 2: Creating predators .... (35 marks)

In this task, you will implement an abstract factory class – PredatorFactory. You will also create the concrete creator participants and initialise all their member variables. The class definitions for the abstract creator participant is given by:

```
class PredatorFactory
{
public:
    PredatorFactory() {}
    virtual ~PredatorFactory() {}

    // This pure virtual function should be overridden by subclasses.
    // Notice that it returns a pointer to a Predator.
    // Remember this in your implementation.
    virtual Predator* createPredator(string, string) = 0;

};
```

2.1 Create the class PredatorFactory and the subclasses of the PredatoryFactory. These are defined as LionFactory, CheetahFactory, WolfFactory and WildDogFactory (the ConcreteCreator participants). These subclasses need to be separated into different `.h` and `.cpp` files named according to their classnames to be able to work with the final given Main.cpp file. (12)

2.2 The subclasses must implement the `createPredator` method that requires the Predator's hunting type, and speciality as parameters and returns a pointer to the newly created Predator. `createPredator` should assign the given variables to the new Predator. (12)

```
Predator* createPredator(string huntingMethod, string speciality);
```

Below is a table containing the values to which the member variables of the different Predators should be initialised (by their respective `createPredator` functions).

Attribute	**Lion**	**Cheetah**	**Wolf**	**Wild Dog**
HP	13	11	8	6
damage	5	4	2	3

Hint: For easy construction, call a parameterised constructor of the Predator class from the derived classes. All other member variables should should be initialised as you see fit.

2.3 Draw a UML Class diagram of the classes and their relationships. You will need to use Visual Paradigm for this. (10)

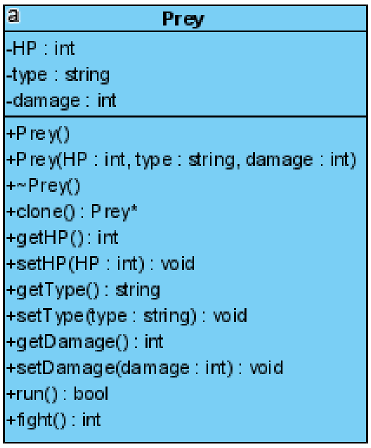
2.4 Which design pattern was implemented in this part? (1)

Task 3: Clone the Prey .... (5 marks)

Add a `clone` function to the Prey class. The clone function should return a pointer to a new Prey. The member variables of the new Prey should be initialised to the same values as those of the Prey it was cloned from.

```
Prey* clone();
```

The UML class diagram for this task is given below:



a Prey

-HP : int
-type : string
-damage : int

+Prey()
+Prey(HP : int, type : string, damage : int)
+~Prey()
+clone() : Prey*
+getHP() : int
+setHP(HP : int) : void
+getType() : string
+setType(type : string) : void
+getDamage() : int
+setDamage(damage : int) : void
+run() : bool
+fight() : int

Task 4: Let the hunting begin... .... (36 marks)

4.1 Test your Predator and Prey classes by running a single hunt simulation using the test program (*Main.cpp*) provided.

4.2 Design and implement two stores, one for Predators and one for Prey. Make use of the Memento pattern. (10)

4.3 Alter the given test program to "save" the Predators and Prey created, using the Memento pattern before any simulation of the hunt has been run. (6)

4.4 Once Predators and Prey can be saved and therefore retrieved, alter the main program to run simulations for all combinations of Predators and Prey in their respective arrays. That is, you have 4 predators and 2 prey, you need to run 8 hunting simulations to cater for all the predator and prey combinations. Each simulation run will need to re-instate the original predator and prey, assign them in the next simulation combination and run the simulation. The outcome from each simulation must be summarised after all simulations have been run. (10)

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