COS 214 Practical Assignment 1

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

- Date Issued: **24 August 2021**
- Date Due: **3 September 2021** at **11:00am**
- Submission Procedure: **Upload to 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 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 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 and any data files you may have created, in a single archive to ClickUP before the deadline.

4 Mark Allocation

Task	**Marks**
Defining soldiers	16
Creating soldiers	35
Clone the Zombies	5
Let the apocalypse begin....	36
TOTAL	92

5 Assignment Instructions

The zombie apocalypse has struck. You have been tasked with creating squads of survivors and pitting them against waves of zombies. Being a sensible computer scientist, you decide to model the problem using Design Patterns and UML Class diagrams and implement the solution in C++ so that you know what you're up against.

Task 1: Defining soldiers ..... (16 marks)

Soldiers can be one of four types:

- Sniper
- Berserker
- Medic
- Engineer

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

- A name.
- Health Points (HP)
- A primary weapon
- A secondary weapon

Soldiers also have an `attack` method. Although different types of soldiers have different fighting styles, all attacks follow the same basic steps. Pseudocode for this is given by:

```
function attack(Zombie)
    while still have Health Points and Zombie still alive
        if hitZombie function returns true
            Zombie died
            Soldier celebrates
        else
            if getHit function returns true
                Soldier dies.
            endif
        endif
    endwhile
endfunction
```

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

- relevant constructors and a destructor
- `attack` which take an instance of a Zombie as parameter and returns `void`
- `hitZombie` which take an instance of a Zombie as parameter and returns `bool`, defined by `bool hitZombie(Zombie* z)`
- `celebrate` which takes no parameters and returns `void`
- `gethit` which take an instance of a Zombie as parameter and returns `bool`, defined by `bool getHit(Zombie *z)`
- `die` which takes no parameters and returns `void`

1.2 Write the `Sniper`, `Berserker`, `Medic` and `Engineer` classes which inherit from the `Soldier`. These subclasses will not override Soldier's `attack` method, because their attacks always follow this same pattern. Instead, the subclasses will implement the 4 methods (primitive operations) as follows: (10)

1. `bool hitZombie(Zombie* z)` – The soldier hits the zombie by calling the Zombie's `takeDamage` method. `takeDamage` takes as a parameter the damage done by the soldier and it returns the Zombie's remaining HP. `hitZombie` should return true if the zombie is killed (i.e. when its HP is ≤ 0).
2. `void celebrate()` – If the Zombie dies, the Soldier celebrates triumphantly.

3. `bool getHit(Zombie *z)` — If the Zombie is still alive, it attacks the Soldier. The damage done by the Zombie can be obtained by calling the Zombie's getDamage method. The damage done by the zombie should be subtracted from the Soldier's HP. The getHit function should return true if the Soldier is killed.
4. `void die()` – If the zombie's hit kills the Soldier, the Soldier dies.

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

Method Name	**Sniper**	**Berserker**	**Medic**	**Engineer**
hitZombie (Zombie* z)	Sniper <Name> fires a <primaryWeapon> at the zombie.	Berserker <Name> swings a <primaryWeapon> at the zombie's head.	Medic <Name> frantically stabs at the zombie with a <primaryWeapon>	Engineer <Name> bludgeons the zombie with a <primaryWeapon>
celebrate()	<Name> exclaims "Headshot!"	<Name> slices the zombie in half!	<Name> sighs in relief.	<Name> shakes his <primaryWeapon> at the zombie's remains.
getHit (Zombie* z)	<Name> swears in 13 different languages as he takes <damage> damage.	<Name> pretends not to notice the <damage> damage he takes.	<Name> gives himself painkillers to numb the <damage> damage suffered.	<Name> hides behind the nearest rock after taking <damage> damage.
die()	<Name> lead a good life. He will be missed.	Nobody really liked <Name>'s company, anyway.	After saving so many lives, <Name> could not save himself.	<Name> was eaten by a zombie.

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

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

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

In this task, you will use a factory to create the Soldiers and initialise all their member variables. The class definitions for the concrete creator participant is given by:

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

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

};
```

2.1 Create the subclasses of the SoldierFactory. These are defined as SniperFactory, BerserkerFactory, MedicFactory and EngineerFactory (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 `createSoldier` method that requires the Soldier's name as a parameter and returns a pointer to the newly created Soldier. `createSoldier` should assign the name that has (12)

been provided as a parameter to the new Soldier.

```
Soldier* createSoldier(string name);
```

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

Attribute	**Sniper**	**Berserker**	**Medic**	**Engineer**
HP	6	10	8	7
primaryWeapon	.308 Rifle	Big Chainsaw	Syringe	Wrench
damage	5	4	2	3

Hint: In order for the Concrete Creators to be able to assign values to a Soldier's member variables, you will have to add setters to the Soldier class. For easy construction, call a parameterised constructor of the Soldier class from the derived classes.

2.3 Draw a UML Class diagram of the classes and their relationships. You will need to find a tool that enables you to draw UML Class diagrams. There are a number of tools online that work quite well. (10)

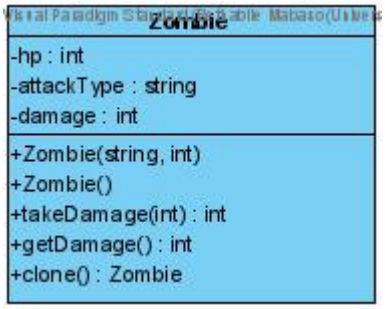
2.4 Which design pattern was implemented in this part? (1)

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

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

```
Zombie* clone();
```

The UML class diagram for this task is given below:



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

4.1 Test your Soldier and Zombie classes by running a single apocalypse simulation using the test program (*Main.cpp*) provided.

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

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

4.4 Once Soldiers and Zombies can be saved and therefore retrieved, alter the main program to run simulations for all combinations of Soldiers and Zombies in their respective arrays. That is, if you have 4 soldiers and therefore 4 zombies, you need to run 16 apocalypse simulations to cater for all the soldier and zombie combinations. Each simulation run will need to re-instate the original soldiers and zombies, assign them in the next simulation combination and run the simulation. The names of the soldiers and zombies for all simulations where the soldiers win (that is, more than double the number of soldiers are alive after the battle than zombies that are alive) must be summarised after all the simulations have been concluded. (10)

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