COS 214 Practical Assignment 1

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

- Date Issued: **1 August 2023**
- Date Due: **15 August 2023** 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 Memento pattern.
- implement the Template Method pattern.
- implement the Factory Method pattern.
- implement the Abstract Factory Method pattern.
- implement the Prototype pattern.

1.2 Outcomes

When you have completed this practical you should:

- apply the Memento to store the state of objects and re-instate the state at a later stage.
- create objects using Factories and the Prototype to store.
- draw class diagrams in Visual Paradigm 17.1.

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 Makefiles, and 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**
Memento and prototype	25
Template Method	20
Abstract Factory	25
TOTAL	70

5 Assignment Instructions

Task 1: Memento and prototype ..... (25 marks)

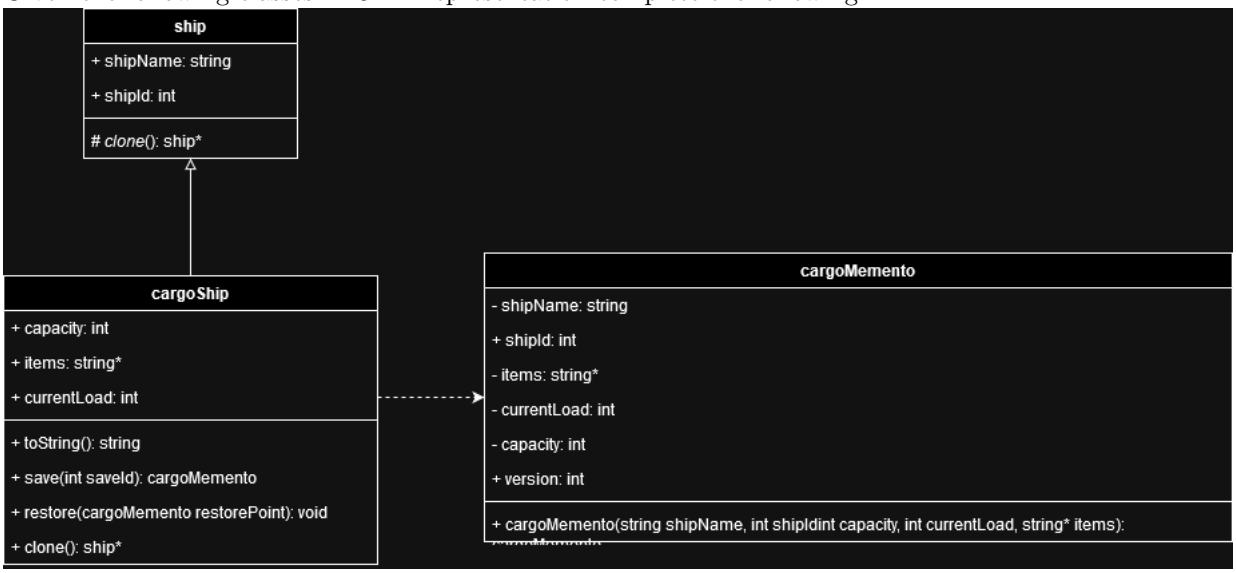
Memento pattern allows us to save and restore the state of objects, which is very useful when designing systems or applications where a user may want to restore an earlier version of an object. We can think of this in any space where an object represents a set of user customisable data i.e. documents(docx, xlsx, tex etc), the state of a saved game or any other persistent data.

We can consider also allowing a memento object to have a method that allows it to stream its internal state to export or import data or allow for data persistence to a hard drive so that it can be restored if the application gets terminated and restarted.

For this task we will consider saving the state of a game object for the purpose of restoring past saved games. Consider a naval game with ships that can equip items to use in the game, we will model these items merely as strings. Each ship can have a variable amount of items i.e. one ship will have capacity for 6 items of which only 3 items may be on board where another ship may have a different capacity.

We will also be using the prototype pattern to allow us to clone the ships used during runtime

Given the following classes in UML representation complete the following:



1.1 Implement the UML in code: (15)

- along side the defined methods, add getters and setters for all attributes of the ship and cargoShip classes.
- add getters for all attributes of the memento class
- the toString method should output similar to this (everything in braces {} is replaced with the value of the attribute *i.e.* "`ID: {shipId}`" will become "`ID: 1`"):

```
Ship {shipId}:  {shipName}
load:  {currentLoad}/{capacity}
Items:
item 1:  {items[0]}
item 2:  {items[1]}
...
```

- include an insert and delete member for the ship items, when deleting bubble up the items so that no gaps remain.

1.2 extend the UML diagram to include a new ship type "militaryShip", extract a common memento interface and create new concrete Mementos. (10)

Task 2: Template Method ..... (20 marks)

Using Template Method we can extract common code that only varies for a select few cases, this is useful when we want a common flow of business logic but need to consider special cases in a manner that we need not use conditional logic that may be evaluated often enough that it impacts performance but allows for code with little repetition. This will also allow for easy extension of existing code whenever a new specialised case may arise; this often happens when a business expands their scope or new technologies get implemented that we may need to cater for in our business logic; this also allows us to create frameworks that can be extended by 3rd-party developers if we were to publish the framework so that they can alter the behaviour to their needs without rewriting the framework ground-up.

In this case we may evaluate frameworks such as JPA that allows us to interact with a variety of databases, requiring special logic, but yields a standardised representation of the data retrieved from those databases. Here it would make sense to use template methods so that should a new database be developed we can easily extend the framework to also support the new database by only implementing the code in which the new database differs from other databases.

In the following tasks we will consider a framework that yields a representation of data we queried from the internet as an object for use in our application. When retrieving information in this manner we want to ensure that we can cater for the various standards implemented online such as returning data as a JSON object or as an OData standard XML file.

Factory method can allow us to correct for some challenges faced when implementing the template method, namely our core application still needs logic to determine which subclass to instantiate and use; This may arise for example when we are implementing a framework such as JPA. If we do not use any design pattern to remedy the issue we will have a block of code in our business logic to read our configuration to determine which database we are connecting to and create the correct subclass that can then fetch and operate on our data.

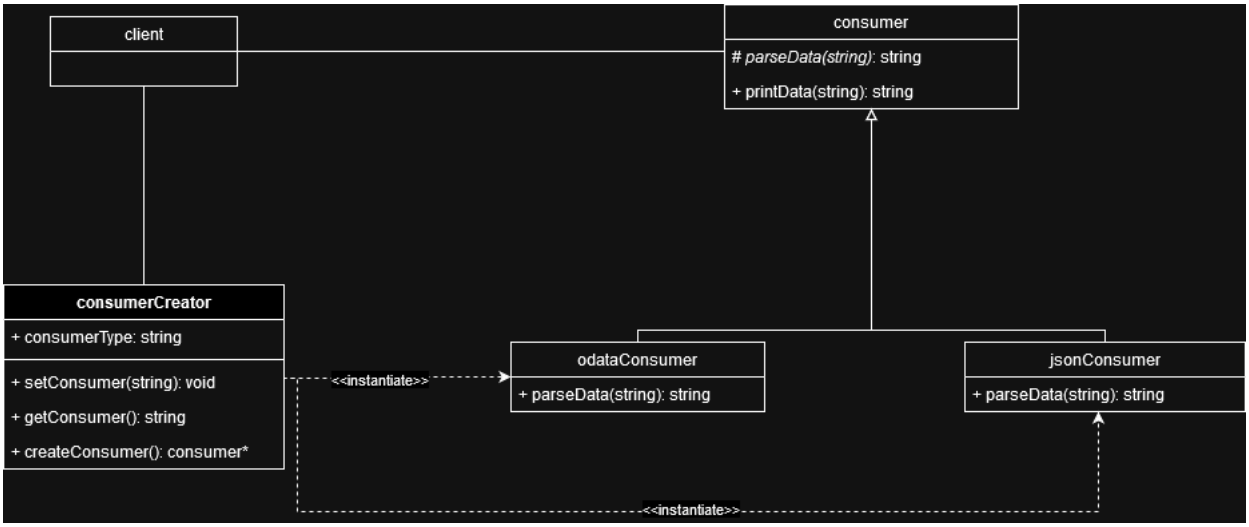
It may look something like this:

```
int main()
{
    db* Database;
    if(config.database == "Postgress"){
        Database = new postgresJPA();
    }else if(Config.database == "MySQL"){
        Database = new mysqlJPA();
    }
    ...
    business logic
    ...
}
```

This example may be further complicated if any other special processing needs to be done such as creating a port to connect to the database and passed to the constructor or other parameters need to be fetched from the environment and used.

We can implement this all in our client/engine class but then we start mixing business logic and configuration/implementation logic that may result in messy code that is difficult to extend at a later time and is difficult to read/understand which is of vital importance when working in teams or onboarding new team members.

Given the following UML:



2.1 implement the UML in code: (20)

- the consumerCreator is our factory method, we have the setConsumer member function that we use to pass configuration from out main.cpp to the factory method. if consumerType is "OData" the odataConsumer will be created and returned. if consumerType is "JSON" the jsonConsumer is returned. if another value is passed do not set the consumerType value.
- the printData function is out template method, it will simply receive a string that will be formatted in a specific manner. it will call the parseData method which will be overriden by the subclasses. and then print the resulting data to the console.
- the odata consumer will parse an XML like structure with only only tag <section> and its closing tag </section>. below is an example:

```
<section>
    section 1
    <section>
        subsection 2
        subitem 3
    </section>
    subitem 4
</section>

will output:

section 1
    subsection 2
    subitem 3
subitem 4
```

- the json consumer will parse a JSON like structure where we ignore the

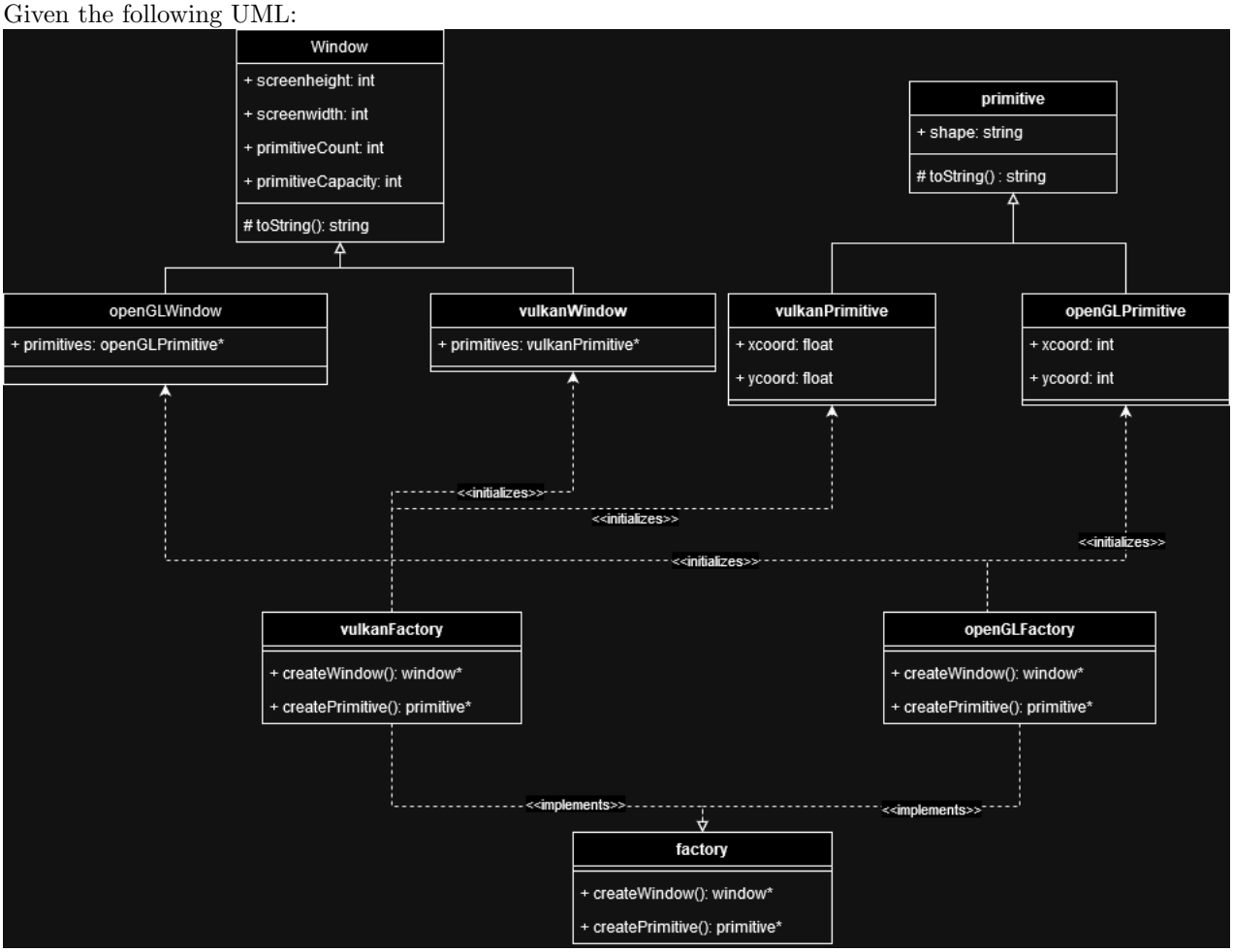
  operator and only use braces{} to configure the structure. the string must start and end with braces and have matching sets of braces to be valid.

```
{
    section 1
    {
        subsection 2
        subitem 3
    }
    subitem 4
}

will output:

section 1
    subsection 2
    subitem 3
subitem 4
```

Task 3: Abstract Factory ..... (25 marks)

Abstract Factory allows us too easily manage the creation of a family of objects with different variants. This is especially useful when considering a cross platform game engine as an example: we may want to be able to support different graphics frameworks such as openGL or vulkan. Both frameworks require unique logic to create a window for example, but will both yield a similar window that can be wrapped by a common interface, decoupling our business logic and our creation logic and allowing a concreteFactory to create only the appropriate objects for our context *i.e.* a vulkan based primitive *vs.* an openGL based primitive.

Given the following UML:



3.1 Label the given UML with the appropriate names for the abstract factory pattern *i.e.* concrete product *etc.* (5)

3.2 implement the given UML in code: (20)

- include getters and setters for all member attributes.
- the toString method for the primitives will include the following, replace "`{identifier}`" with the appropriate value:

```
an openGL primitive:
OpenGL {shape}:
    X coordinate: {xcoord}
    Y coordinate: {ycoord}

and a vulkan primitive:

Vulkan {shape}
    X coordinate: {xcoord}
    Y coordinate: {ycoord}
```

- the window toString method will output the following:

```
an openGL window:
OpenGL window {width}x{height}:
    {print all primitive toStrings}

and a vulkan window:

Vulkan window {width}x{height}
    {print all primitive toStrings}
```

- The openGL window will only allow up to 3 primitives to be specified
- the vulkan window will allow up to 9 primitives.