COS 214 Practical Assignment 2

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

- Date Issued: **15 August 2023**
- Date Due: **29 August 2023** at **9:30am**
- Submission Procedure: **Upload to ClickUP**
- Submission Format: **archive (zip or tar.gz)**

1 Introduction

1.1 Objectives

In this practical you will:

- implement the State, Strategy, Composite and Decorator patterns.
- draft a UML State diagram.
- draft a UML Sequence diagram.

1.2 Outcomes

When you have completed this practical you should:

- Be able to model and implement processes using the State design pattern
- Be able to use the Composite pattern to model and perform operations on composite objects
- be able to use the Strategy and Decorator patterns to modify the behaviour or specific implementation of logic used at runtime.

2 Constraints

1. You must complete this assignment individually or in groups of two.
2. You may ask the Teaching Assistants for help but they will not be allowed to give you the solutions.
3. You must demo the system in the Labs at the conclusion of the practical assignment during your booked demo slots. Both partners need to be present if working in pairs

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. A demonstration of the implementation will be conducted in the labs. It is recommended to create a main that allows for the proper demonstration of the patterns.

4 Mark Allocation

Task	**Marks**
Smart Contract	20
Organizational management	15
Testing Framework: Decorator & Strategy	20
TOTAL	55

5 Assignment Instructions

Task 1: Smart Contract ..... (20 marks)

The State design pattern models systems where a system may determine their behaviour based on some internal state. This can be seen easily with the idea of a finite state automata where we can model a system as a set of states and a transition function that determines how we change between states based on an input or action.

These systems are important as many business processes can be modelled in this manner. Any system where, based on the current progress of the process, different actions need to be taken which may or may not require various parties or resources, can be broken down into a set of states to track what the next actions should be.

To this end we are going to be implementing a "Smart contract" (Typically deployed and tracked on a blockchain such as Etherium).

Our contract will have the following states and actions available to them:

- `Negotiation`
  In this state we 'draft' a contract and negotiate on conditions until we are satisfied by the conditions set out.
  - Add condition
  - Remove condition
  - Accept contract (transition to `Tentatively accepted`)
  - Reject contract (transition to `Rejected`)
  - Complete contract (raise an exception as only accepted contracts can complete)
- `Tentatively accepted`
  In this state at least one party involved in the contract has demonstrated that they are satisfied with the conditions as they have been set out.
  - Add condition (transition back to `Negotiation` state, clear the 'Agreeing parties')
  - Remove condition (transition back to `Negotiation` state, clear the 'Agreeing parties')
  - Accept contract (self transition if not all parties have accepted, otherwise transition to `Accepted` state)
  - Reject contract (transition to `Rejected` state)
  - Complete contract (raise an exception as only accepted contracts can complete)
- `Accepted`
  - Add condition (raise exception, Contract is already Accepted)
  - Remove condition (raise exception, Contract is already Accepted)
  - Accept contract (raise exception, Contract is already Accepted)
  - Reject contract (raise exception, Contract is already Accepted)
  - Complete contract (self transition if not all parties have voted to complete, else transition to `Completed` state)
- `Rejected`
  This is a terminal state in which no alterations can be made to the contract but we keep track of it for audit purposes
- `Completed`
  This is a terminal state in which we consider the contract complete, also kept for audit purposes

We will also implement a `toString` on the contract to output in the format (everything in braces { } is replaced with the value of the attribute i.e. "`State: {state}`" will become "`State: ACCEPTED`"):

```
Contract {contract name}:
    State: {state}
    {if state=="Tentatively Accepted" or state=="Accepted" ->
    Votes:
        {vote[0]}
        {vote[1]}
        ...
    }
    Conditions:
        {condition[0]}
        {condition[1]}
        ...
```

1.1 Draft a UML state diagram for the described system. (10)

1.2 Implement the described system. (10)

Task 2: Organizational management ..... (15 marks)

The Composite design pattern allows us to bundle objects together in a tree like structure to allow for the exploitation of recursion to perform certain tasks. We may model quotes for customers based on the material costs of a construction

For example, a car is composed of an engine, driveshaft, gear train etc. an engine is composed of fuel injectors or carburettors, pistons, spark plugs etc.

If we write a point of sales system for a manufacturer we can compose quotes using a tree like model where we specify specifics such as a model number for each part or where it gets produced. We can then ask the model to generate a quote at which point each component of the tree will calculate its price based on the components it is composed of. That is, an engine would cost as much as the sum of the sparkplugs, pistons etc. The leaf nodes would either know their price before hand or handle the querying of the data from APIs or databases depending on the business logic.

For this exercise we will consider a simple example seen in enterprise software: Human Capital Management (HCM). An integral component of many HCM systems is the concept of an organisational unit (org unit). An org unit can represent a team of individuals, or a group of teams. We can group teams together to represent an office, then another grouping above that may be a region such as South Africa and can keep abstracting as far as needed to describe the hierarchy in a business.

2.1 Draft a UML Class diagram for the system. (5)

2.2 Implement an org structure with `orgStructure` as the composite and `individual` as the leaf. An `individual` will have a *uniqueId*, a *cost center* and a *cost to company*. To demonstrate the use of the Composite pattern we will have two methods that delegate work to child components. These methods are: (10)

- `getTotalExpenditure`: calculates the total expenditure of all child components combined
- `getExpenditureByCostCenter`: calculates only the expenses related to a cost center.

Task 3: Testing Framework: Decorator & Strategy ..... (20 marks)

The Decorator and Strategy patterns allow us to do some really nice abstractions that can clean up the logic in our functions. For example, if we need to create an order for a shipping company we would: generate and send an invoice then process the sale. We don't really care *HOW* the invoice gets sent, just that it does get sent. So, we would extract that functionality to a different function... But now that function would need to determine the manner in which we send the invoice: email or SMS? and still have the implementations of both of those cases... We can extract it again to two different methods but we end up with fragmented code that is hard to read and extend. So we can use the Strategy pattern to extract the implementation of logic that may vary at runtime.

We can use it, for example, in code that encrypts a file. If we have a function: `writeData(data, algorithm)` using the Strategy pattern, doesn't need to care what encryption we are using but we can implement any number of different algorithms and easily extend our code to support new algorithms using a common interface.

Where Strategy pattern allows us to control the inner algorithm that is used at runtime, Decorator pattern allows us to extend an object at runtime with new behaviours without modifying the object.

An example of this could be a system that needs to dynamically create queries at runtime and execute them, our *concreteComponent* might be responsible for querying the relevant data but then we can use the Decorator pattern to add functionality to check if a user is supposed to have access to the data and only output the portion of the data they have access to. Maybe after this data is queried we need to perform some aggregations. All of this can be done using the Decorator pattern.

In this exercise we will utilise both patterns, Strategy and Decorator, to create a rudimentary testing framework.

We will implement two classes that we want to test, one a simple calculator taking a string input of numbers and the operations + - * /. The other a boolean calculator taking in an input of True or False and the operations AND, OR, NOT and grouping symbols (,).

Our *test* interface will define the method `executeTest()` which will be implemented by the concrete Strategies. The concrete Strategies will essentially be unit tests where we input data to the given calculator class and evaluating and confirming its response.

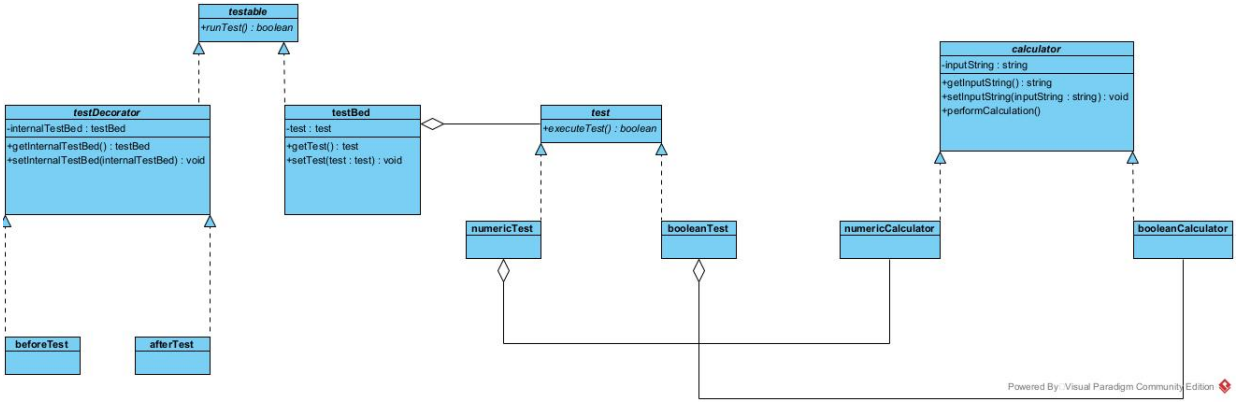
Our *context* class is the testbed class. In this example a testbed will only execute one test but by using composition we can extend a testbed to have a number of tests that it is responsible for.

We will use the Decorator pattern to implement before and after decorators, allowing us to extend tests with behaviour that should occur before or after a test. This is typically used in testing frameworks to allow a user to mock data or perform some setup and housekeeping after a test.

For example, if we want to test a method in our API that allows a user to buy a product, the before decorator might be used to create a user and a product. The after decorator might remove the user and product to ensure future tests are unaffected by the setup we needed to perform.

For this exercise we will only be printing out a message before and after a test has been run. Extend the given UML Class diagram so that you can set the string to be displayed and then output that to screen before the test is run and its output is shown on screen.

When a test is run print out the input string, the expected result, the actual result, and the status of the test i.e. pass or fail.



3.1 Draft a UML Sequence diagram of a test being performed. (5)

3.2 Discuss how best the actions of the before and after decorators can be altered (5)

3.3 Implement the given UML to build the test framework. Make note of any assumptions made or extensions to the given UML to achieve the desired functionality. (10)