UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

Department of Computer Science
Faculty of Engineering, Built Environment & IT
University of Pretoria

COS214

Practical 2 Specifications

Release Date: 14-08-2025

Due Date: 02-09-2025 at 11:00

Total Marks: 120

Contents

1 General Instructions

- ***Read the entire assignment thoroughly before you start coding.***
- This assignment should be completed in pairs.
- **To prevent plagiarism, every submission will be inspected with the help of dedicated software.**
- Be ready to upload your assignment well before the deadline, as **no extension will be granted.**
- Note that plagiarism is considered a very serious offence. Plagiarism will not be tolerated, and disciplinary action will be taken against offending students. Please refer to the University of Pretoria's plagiarism page at `https://portal.cs.up.ac.za/files/departmental-guide/`.
- You can use any version of C++.
- You can import the following libraries:
  - vector
  - string
  - iostream
  - map
  - list
- You will upload your code with your main to FitchFork as proof that you have a working system.
- **If you meet the minimum FitchFork requirements (working code with at least 60% testing coverage), you will be marked by tutors during your practical session. Please book in advance (Instructions will follow on ClickUp).**
- **You will not be allowed to demo if you do not meet the minimum FitchFork requirements.**

2 Plagiarism

The Department of Computer Science considers plagiarism a serious offence. Disciplinary action will be taken against students who commit plagiarism. Plagiarism includes copying someone else's work without consent, copying a friend's work (even with permission) and copying material from the Internet. Copying will not be tolerated in this course. For a formal definition of plagiarism, the student is referred to `http://www.library.up.ac.za/plagiarism`. If you have any questions regarding this, please ask one of the lecturers to avoid misunderstanding.

3 Mark Distribution

Activity	Mark
Task 1: UML Diagrams	30
Task 2: Implementing patterns	60
Task 3: FitchFork Testing Coverage	10
Task 4: Demo preparation	20
Total	**120**

Table 1: Mark Allocation

4 Overview and Background

After visiting family abroad in Naples, Italy, Romeo came home to South Africa and dreamt of Romeo's Pizza. A family based pizza shop that puts the customer first by listening to their requests and updating the menu as needed.

Romeo uses recipes that his Nonna gave him from Naples. These wood-oven pizzas have taken over the Hatfield area with everybody wanting a taste of authentic Italian pizza.

Romeo started off with a Pepperoni pizza and a Vegetarian pizza. After listening to customer requests, Romeo introduced the Meat Lovers pizza - an extension of the Pepperoni pizza and the Vegetarian Deluxe - an extension of the normal Vegetarian pizza. Romeo wants to make sure that he can always create more pizza types or extend existing types for his customers.

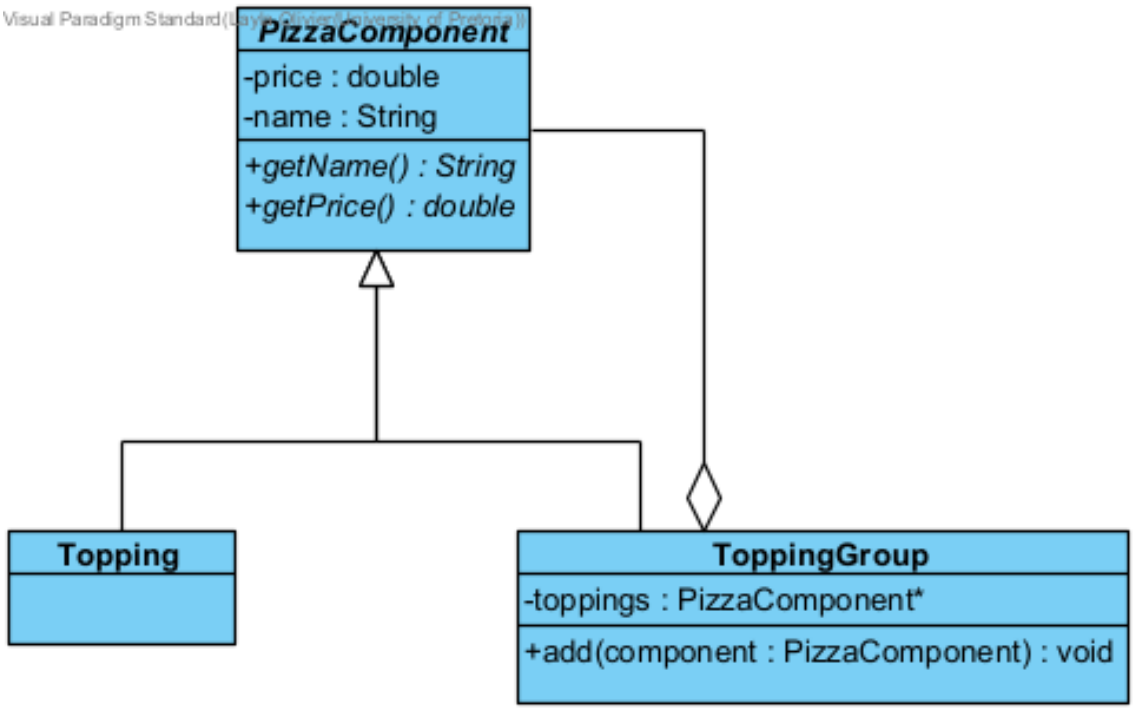
Romeo understands that customers like to customise their pizzas by either having extra cheese or a stuffed crust (for when they are really hungry). Romeo has thus incorporated this into his menu by allowing customers to add this customisation to current pizzas on the menu.

Romeo values all of his customers and wants to thank them for large purchases, so he offers a 10% discount on all orders consisting of 5 or more pizzas. He also loves his family and their support, so he gives them a 15% discount on any orders.

Read the following sections carefully and complete the tasks that follow.

4.1 Composite

Pizzas all start out the same - the dough, the sauce and the cheese. Although different pizzas consist of different toppings. When Romeo first started he only had a Pepperoni pizza and a Vegetarian pizza. The Pepperoni only has pepperoni toppings. The Vegetarian pizza has mushrooms, green peppers, and onions as toppings. The Meat Lovers is a Pepperoni pizza with beef sausage and salami added. The Vegetarian Deluxe is a Vegetarian pizza with feta cheese and olives added.



4.1.1 Classes and Attributes

1. **PizzaComponent**

   - *Attributes*
     - price: double
     - name: String

   - *Methods*
     - getName(): String
       Abstract method.
     - getPrice(): double
       Abstract method.

2. **Topping** An individual topping.

   - *Attributes*
     - price: double
       The price of the individual topping.
     - name: String
       The name of the individual topping.

   - *Methods*
     - getName(): String
       Returns the name of the topping.

- getPrice(): double
  Returns the price of the topping.

3. **ToppingGroup** A group of toppings that make up a pizza.

   - *Attributes*
     - price: double
       The price of the individual topping.
     - name: String
       The name of the individual topping.

   - *Methods*
     - getName(): String
       Returns the name of all of the toppings part of the group(s).
       E.x. `Vegetarian Deluxe (Vegetarian (Mushroom, Green Peppers, Onions), Feta, Olives)`
     - getPrice(): double
       Returns the price of all of the toppings part of the group(s).

4.1.2 Implementation Details

Creating Pizzas

The pizzas are created by grouping toppings together. The original pizzas (Pepperoni and Vegetarian) consist of less toppings than the extended pizzas. The more complex pizzas are an extension of the originals, meaning that they use the originals and then just add on additional toppings.

Toppings and Groups

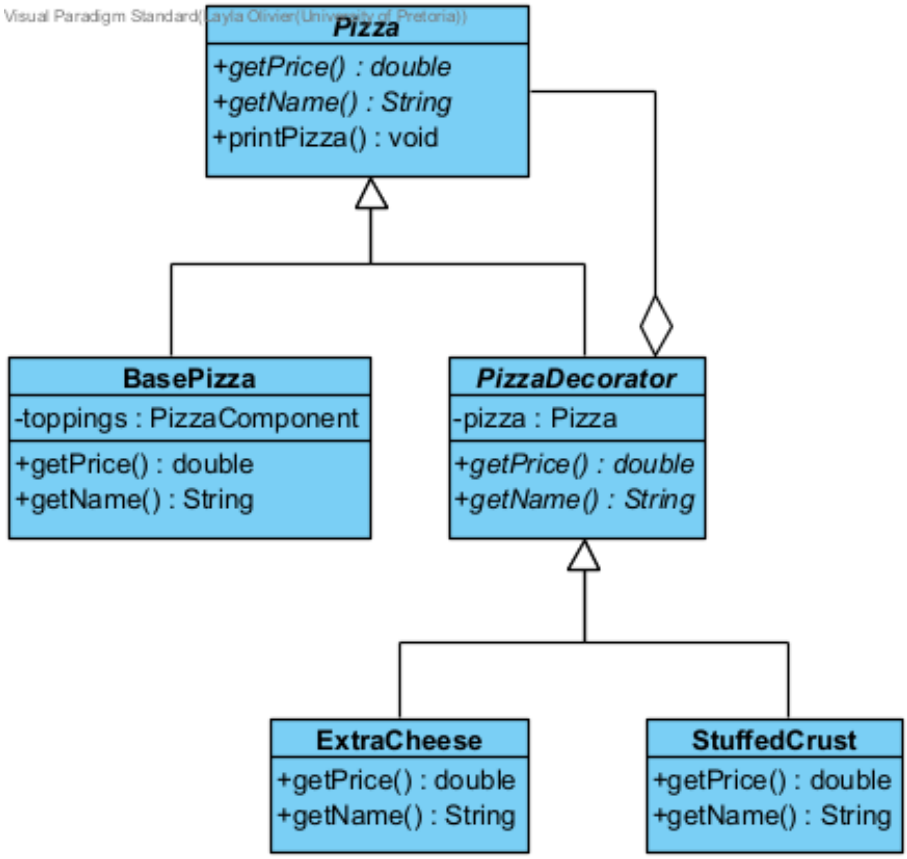
Topping represents a single ingredient (e.g., "Mushrooms", "Olives", "Pepperoni") and stores its own name and price. ToppingGroup stores a collection of PizzaComponent objects, which may be individual toppings or other groups, allowing nested composition.
The getName() method in ToppingGroup should produce a descriptive string of all components, and getPrice() should sum all prices of contained components.

Extensibility

New pizzas can be added simply by creating a new ToppingGroup or extending an existing pizza's group.
You are welcome to add more pizzas.

4.2 Decorator

Base pizzas can be modified by adding extra cheese or having a stuffed crust instead of a normal one. A pizza can also have both modifications. Each modification comes at a cost.



4.2.1 Classes and Attributes

1. **Pizza**

   - *Methods*
     - getPrice() : double
       Abstract method.
     - getName() : String
       Abstract method.
     - printPizza() : void
       Prints out the pizza. Can be also be implemented in the subclasses.

2. **BasePizza**

   - *Attributes*
     - toppings : PizzaComponent

   - *Methods*
     - getPrice() : double
       Returns the price of the pizza.
     - getName() : String
       Returns the name and the toppings on the pizza.

3. **PizzaDecorator**

- *Attributes*
  - pizza : Pizza
- *Methods*
  - getPrice() : double
  - getName() : String

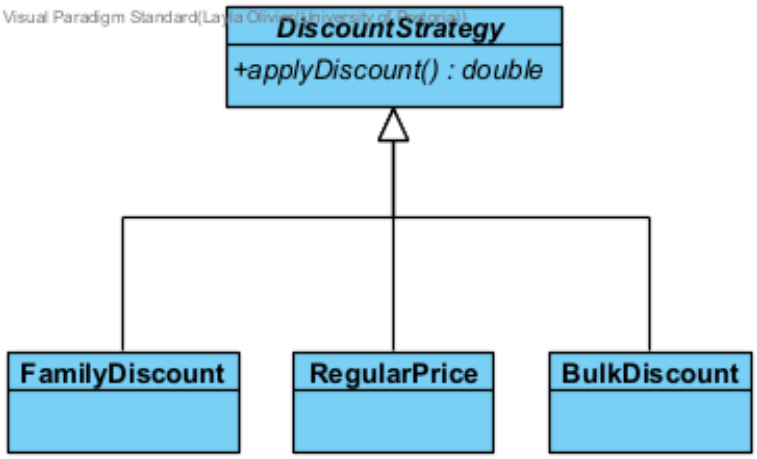
4. **ExtraCheese, StuffedCrust**
   The extra items to add to the pizza.
   - *Methods*
     - getPrice() : double
     - getName() : String

4.2.2 Implementation Details

Decorators
The value of the extra must be added on top of what the base pizza originally costs.
Chaining
The decorators can be chained or stacked. This means that a pizza can have both extra cheese and a stuffed crust.
You are welcome to add any other concrete decorators to the pizzas (e.g., extra sauce).

4.3 Strategy

To help Romeo with the discounts, the Strategy pattern is used to alter the discount logic.



4.3.1 Classes and Attributes

1. **DiscountStrategy**
   - *Methods*

- applyDiscount() : double
  Abstract method.

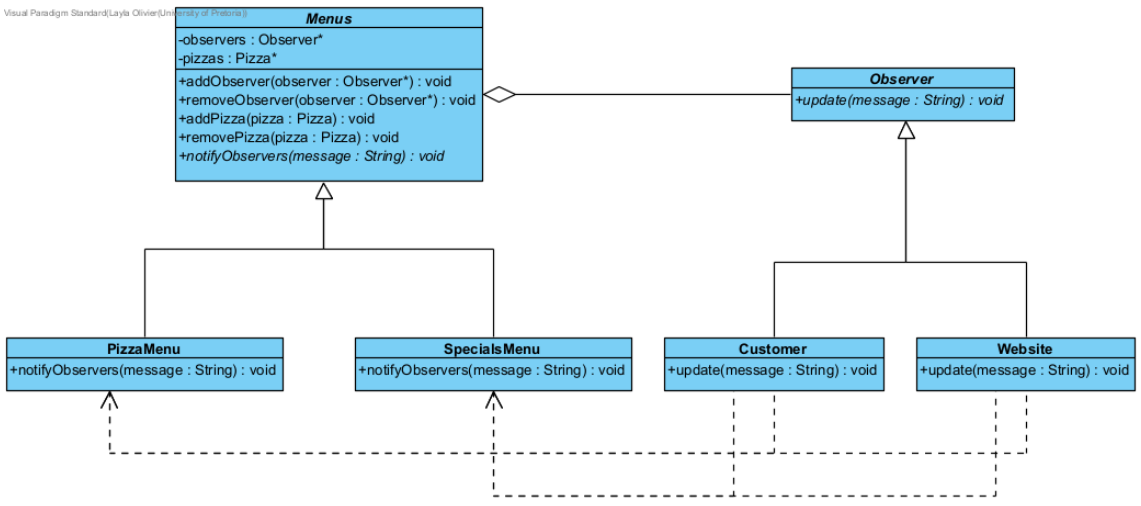
2. **FamilyDiscount, RegularPrice, BulkDiscount**
   - Each class inherits from DiscountStrategy
   - Alters the discount logic in applyDiscount().

4.3.2 Implementation Details

- Each concrete strategy class (FamilyDiscount, RegularPrice, BulkDiscount) needs to provide their own implementation for applyDiscount().
- The discounts are as follows:
  1. Regular Price: 0% discount
  2. Bulk Orders (5+ pizzas): 10% discount
  3. Family Discount: 15% discount
- It is your choice to pick the Context class. You must decide if you want to apply discounts per pizza or per the order. Be able to justify your decision.

4.4 Observer

Romeo has just introduced a Specials Menu. This menu has limited time deals (e.g., Pepperoni pizzas are 10% off on Tuesdays). Romeo has decided that he wants to notify his customers and update his website every time he adds a new persistent pizza to the normal menu or removes one from the menu. And Romeo wants to notify his customers and update the website when a new special is added to the Specials Menu. He is hoping that this will attract more customers.



4.4.1 Classes and Attributes

1. **Menus**
   Abstract Class.

   - *Attributes*
     - observers: Observer*
     - pizza: Pizza*
   - *Methods*
     - addObserver(observer: Observer*) : void
       Adds an observer to the list.
     - removeObserver(observer: Observer*) : void
       Removes an observer from the list.
     - addPizza(pizza: Pizza) : void
       Adds a pizza to the menu.
     - removePizza(pizza: Pizza) : void
       Removes a pizza from the menu.
     - notifyObservers(message: String) : void
       Abstract Method - to be implemented in concrete classes.

2. **PizzaMenu, SpecialsMenu**

   - *Methods*
     - notifyObservers(message: String) : void
       Notifies the observers of changes to the menus.

3. **Observer**
   Abstract Class.

   - *Methods*
     - update(message: String) : void
       Abstract method.

4. **Customer, Website**

   - *Methods*
     - update(message: String) : void
       Updates the customer and website to be aware of the changes.

4.4.2 Implementation Details

Menus and Observer Communication

- When a pizza is added or removed, the menu calls notifyObservers() to update all registered observers.

Notifications

- Concrete classes PizzaMenu and SpecialsMenu implement notifyObservers(message: String) to send appropriate messages about menu changes.
- Observers like Customer and Website implement the update(message: String) method to receive these notifications and react accordingly. You may change how the Observers react (e.g., change a State perhaps) if you do not want to only use print statements.

4.5 State

Currently there is no State design pattern apart of the solution. Your task is to identify a suitable place to use the State pattern and implement it.
You can add the pattern anywhere that is appropriate. Make sure that you can justify your choice to the tutors.
When deciding where to add the State pattern, consider the following:

- Ensure the intent of the pattern is clearly demonstrated in your chosen context.
- Show an object changing states and the consequences of those state changes. For example, a document in the FinalDraft state can no longer be edited.
- Include at least one state that allows transitions both forwards and backwards. For example, a contract in Editing may move to **Review**, and from there either to Accepted or back to Editing.

Ingredient Price Table

Ingredient / Topping	**Price (ZAR)**
Dough	10.00
Tomato Sauce	5.00
Cheese	15.00
Pepperoni	20.00
Mushrooms	12.00
Green Peppers	10.00
Onions	8.00
Beef Sausage	25.00
Salami	22.00
Feta Cheese	18.00
Olives	15.00
Extra Cheese (Decorator)	12.00
Stuffed Crust (Decorator)	20.00

5 Task 1: UML Diagrams

In this task, you need to construct UML Class and State diagrams for the given scenario.

5.1 Class Diagrams (20 Marks)

- Create a comprehensive UML class diagram that shows how the classes in the scenario interact and participate in various patterns.
- Ensure it includes all relevant classes, their attributes, and methods.
- Identify and indicate the design patterns each class is involved in, as well as their roles within these patterns. (Remember a class can participate in more than one pattern)
- Display all relationships between classes.
- Add any additional classes that are necessary for a complete representation.
- Do not forget to add your State design pattern UML Class Diagram.

5.2 State Diagrams (10 Marks)

- Develop a UML state diagram representing your system at a specific point in time (of your choosing) during its lifetime.
- Use the correct notation.
- There should be at least 4 nodes in your State Diagram.

6 Task 2: Implementation

In this task, you need to implement the design patterns described in the scenario.

- Use your UML class diagram as a guide to integrate the design patterns. Note that a **single class** can be part of **multiple patterns**.
- The provided scenario outlines the minimum requirements for this practical. Feel free to add any additional classes or functionalities to better demonstrate your understanding of the patterns and tasks.
- Thoroughly test your code (more details will be provided in the next task). Tutors will only mark code that runs.
- You may use arrays, vectors, or other relevant containers. Avoid using any libraries not mentioned in the general instructions, as your code must be compatible with FitchFork.
- Tutors may require you to explain specific functions to ensure you understand the pattern being implemented.

7 Task 3: FitchFork Testing Coverage

In this task, you are required to upload your completed practical to FitchFork. The FitchFork submission slot will open on 19 August 2025.

- Ensure that your code has at least 60% coverage in your testing main. This is necessary for demonstrating your work to the tutors.
- You will need to show your FitchFork submission with sufficient coverage to the tutor before being allowed to demo.
- You will be required to download your code from FitchFork for demonstration purposes.
- It might be useful to have two main files: a testing main for FitchFork and a demo main with a more user-friendly interface (e.g., a terminal menu) for demoing. This is just a suggestion, not a requirement. You are required to have at least a testing main. If you choose to have a demo main, upload it to FitchFork as well, but ensure it does not compile and run with make run.
- Name your testing main file **TestingMain.cpp**.
- If you create a demo main, name it **DemoMain.cpp** so it can be excluded from the coverage percentage. Note that you do not have to test your demo main.
- Additionally, upload a makefile that will compile and run your code with the command **make run**.

8 Task 4: Demo Preparation

You will have 10 minutes to demo your system and answer any questions from the tutors. Proper preparation is crucial.

- Ensure you have all relevant files open, such as UML diagrams and source code.
- Set up your environment so that you can easily run the code during the demo after downloading it.
- Be ready to demonstrate specific function implementations upon request.
- Practice your demo to make sure it fits within the allotted time and leaves time for questions.
- Prepare a brief overview (30 seconds) of your system to quickly explain its functionality and design.
- Make sure your testing and demo mains (if applicable) are ready to show their respective functionalities.
- Have a clear understanding of the design patterns used and be prepared to discuss how they are implemented in your code. Also, think about other potential use cases for the patterns.

- Be prepared to answer questions. If you do not know the answer, inform the marker and offer to return to the question later to avoid running out of time.

9 Submission Instructions

You will submit on FitchFork.

- FitchFork submission:
  - Zip all files.
  - Ensure you are zipping the files and not the folder containing the files.
  - Upload to the appropriate slot on FitchFork well before the deadline, as no extensions will be granted.
  - Ensure that you have at least 60% coverage.