Table

Description automatically generated

Table of Contents

[OOP Design 2](#_Toc129989515)

[UML Class Diagram 4](#_Toc129989516)

[Pseudo Code 4](#_Toc129989517)

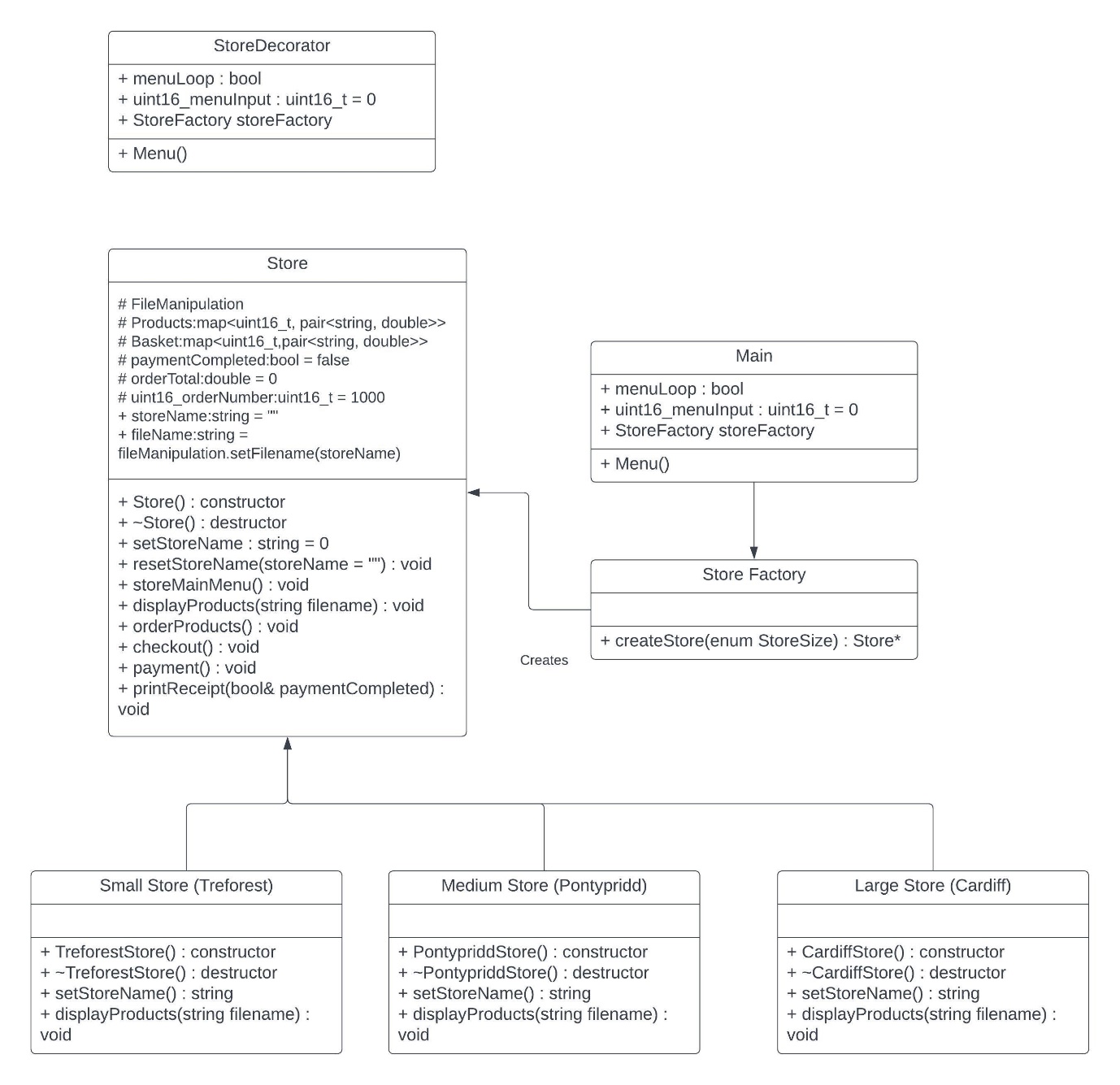
[Justification 5](#_Toc129989518)

## OOP Design

The hardware outlet application is a complex system that requires the implementation of multiple design patterns to achieve a functional, efficient, and robust Object-Oriented application. The application uses three design patterns: Factory, Observer, and Decorator. The Factory pattern creates different branches of the outlet that offer various item selections, while the Decorator pattern adds accessories to the main items for sale. The Observer pattern tracks orders in real-time. These design patterns ensure a functional, efficient, and robust Object-Oriented application.

The Factory pattern provides an interface for creating objects of a superclass or interface, while allowing subclasses to alter the type of objects that will be created. The Decorator pattern allows for the dynamic addition of new behaviour or features to an existing object. The Observer pattern defines a one-to-many dependency between objects, notifying all dependents when an object changes state. These design patterns can work together to create a scalable and robust object-oriented application for the hardware outlet. Overall, the combination of these design patterns in the application helps to create a flexible, efficient, and robust system that can be easily updated with new functionality.

### UML Class Diagram



### Pseudo Code

Ask user which store to browse.

Create object of selected store

Initialise files used by the stores for products.

Displays selected stores main menu.

Ask user what they wish to do:

* 1. View Orders
* 2. Place Order
* 3. Track Orders
* 4. Change Store
* 0. Quit

1. View Orders - Not implemented due to time constraints of 4 assessments in one month.
2. Place Order
   1. Read products into products map from stores file.
   2. Display the list to the user then call order Product function.
   3. Asks user what product they wish to add to their basket.
   4. Check if input matches a product and then either add it to the basket or display error message.
   5. Ask if user wishes to add more items to the basket.
   6. And then either continue to add items or move on to check out.
   7. Display all items currently in the basket to the user then call payment function.
   8. Confirm if user wishes to confirm or cancel the order, exit if they wish to cancel, proceed if they wish to confirm.
   9. While loop, order total greater than amount paid.
   10. Output how much is due.
   11. Accept user input.
   12. Calculate whether order is paid for or if there is more to pay, loop until complete.
   13. Upon complete payment call print receipt function
   14. Adds order to delivery tracking.
   15. Displays order number to user.
   16. Displays items to user.
   17. Displays order total to user.
   18. Sends order to orders file.
   19. Clears basket and increments order number.
   20. Returns to stores main menu.
3. Track Orders
4. Change Store – Not implemented due to time constraints of 4 assessments in one month. Requires users to quit and restart to change stores.
5. Quit
   1. Display exit message to user.
   2. Terminate program.

## Justification

The factory design pattern offers two options for the model we need, I chose the standard factory over the Abstract family due to the smaller requirments and also knowing my design meant all stores would be derived from the common store type. Thus at run time the factory can create any of the stores the user selects from the input by returning a pointer to the newly created store.

The Decorator pattern has two similar design patterns, the Adapter and Bridge patterns. Both modify an object's behaviour at runtime; however, each pattern has a unique approach as to how to achieve the modification. The Adapter pattern introduces a new interface for an existing object, allowing it to interact with other objects that it previously couldn't. The Bridge pattern separates an object's abstraction and implementation, giving both the ability to vary independently. Compared to the Decorator pattern which adds new functionality to an existing object by "decorating" it with a decorator object. Using the Decorator pattern, I can extend a stores product list without changing the original list. This makes it easier to offer specific products based on the current selection along with maintaining and updating the code in future. Additionally, since the Decorator pattern uses composition instead of inheritance, it allows for greater flexibility in adding or removing functionality from an object at runtime.