L-Università ta' Malta

CPS2004 Object Oriented Programming Assignment 2023/24

Graham Pellegrini 0352804L Computer Engineering 2nd Yr

https://gitlab.com/GrahamP/oo-programming-assignment-2023_24/-/tree/master?ref_type=heads

1. A Logistics Management Application developed in C++ and Java.

(a)

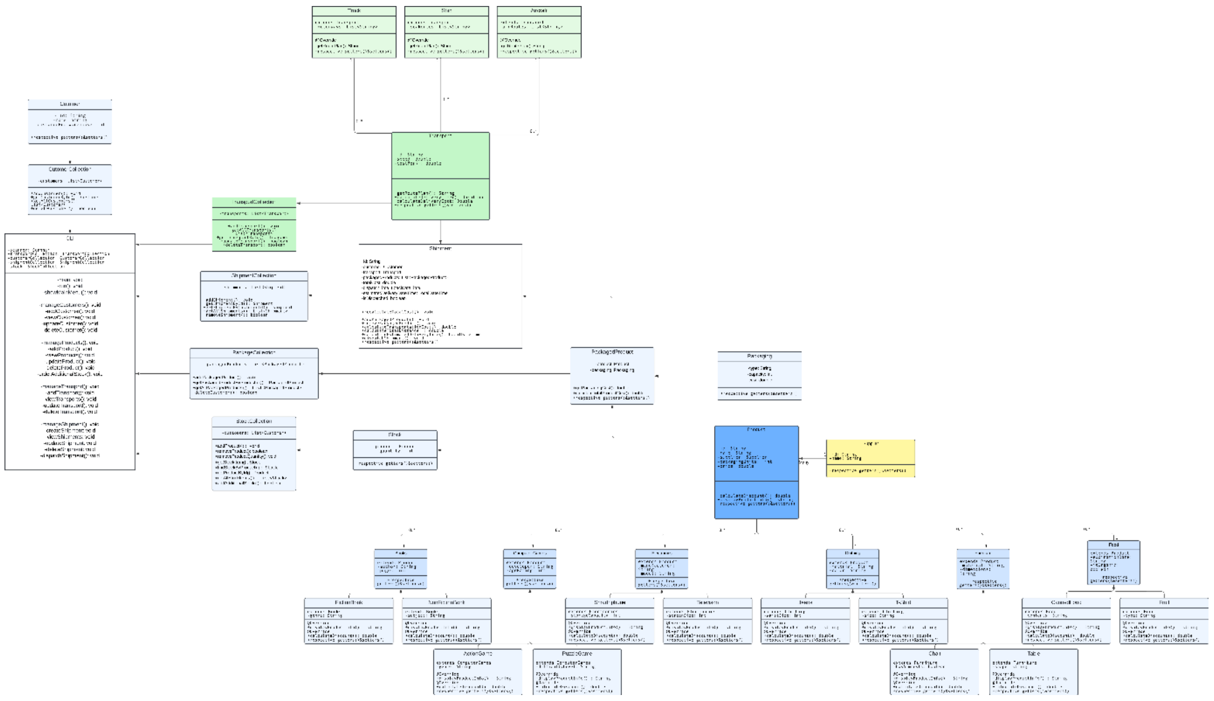


Figure 1UML Q1

https://lucid.app/lucidchart/f7e1f70c-8799-463d-944c-0ae8088cf6ab/edit?viewport_loc=-3906%2C-2751%2C4591%2C2474%2CHWEp-vi-RSFO&invitationId=inv_3e524848-fb86-48e8-b37b-35a3f9cee048

(c)

The Project is made up of the following class structures:

- main class: Q1_Java/src/main/java/CLI.java

-Product hierarchy: Q1_Java/src/main/java/Product

-Transport hierarchy: Q1_Java/src/main/java/Transport

-Shipment hierarchy: Q1_Java/src/main/java/Shipment

The Product hierarchy contains a Supplier class which is related to a Product class with a one-to-many relation respectively. From the Product class we extend 6 subclasses each with their respective folder:

Q1_Java/src/main/java/Product/Book

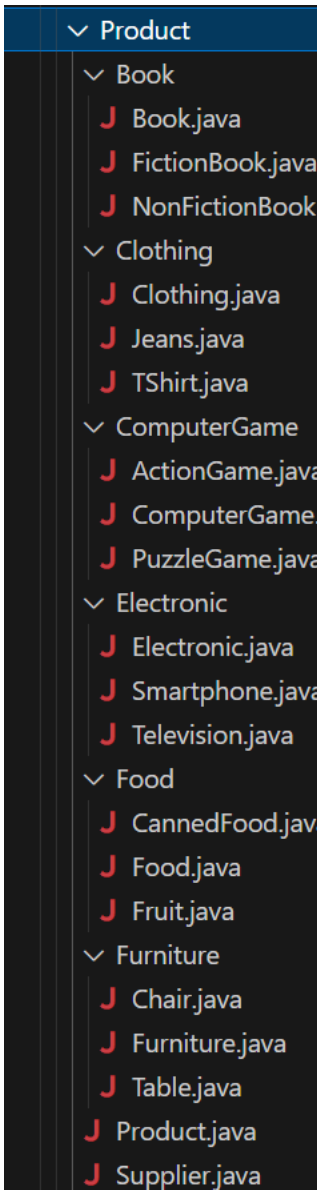
Q1_Java/src/main/java/Product/Clothing

Q1_Java/src/main/java/Product/ComputerGame

Q1_Java/src/main/java/Product/Electronic

Q1_Java/src/main/java/Product/Food

Q1_Java/src/main/java/Product/Furniture



```
Product
  Book
    Book.java
    FictionBook.java
    NonFictionBook
  Clothing
    Clothing.java
    Jeans.java
    TShirt.java
  ComputerGame
    ActionGame.java
    ComputerGame.
    PuzzleGame.java
  Electronic
    Electronic.java
    Smartphone.java
    Television.java
  Food
    CannedFood.jav
    Food.java
    Fruit.java
  Furniture
    Chair.java
    Furniture.java
    Table.java
  Product.java
  Supplier.java
```

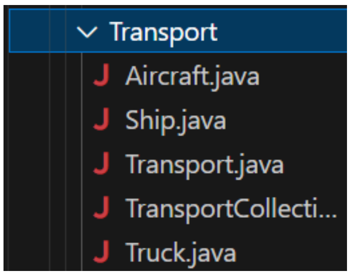
In each folder the respective Product Subclass is found and a future two child classes extend the subclasses; therefore, the Product class has 12 grandchildren classes. In these classes the respective overrides depending on the type of product are implemented as well as any additional constructer variables.

The Transport hierarchy is made up of a Parent Transport class and 3 children classes extending the parent class:

Q1_Java/src/main/java/Transport/Ship.java

Q1_Java/src/main/java/Transport/Aircraft.java

Q1_Java/src/main/java/Transport/Truck.java


Transport
Aircraft.java
Ship.java
Transport.java
TransportCollecti...
Truck.java

Here each hold respective list of routes available and the respective getters and setters for this variable.

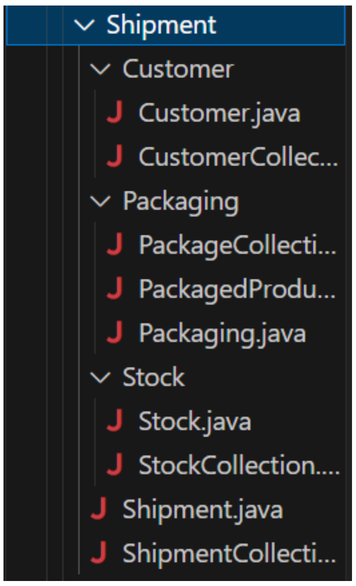
In the Shipment Folder we find:

Q1_Java/src/main/java/Shipment/Customer

Q1_Java/src/main/java/Shipment/Packaging

Q1_Java/src/main/java/Shipment/Stock

Q1_Java/src/main/java/Shipment/Shipment.java


Shipment
Customer
Customer.java
CustomerCollec...
Packaging
PackageCollecti...
PackagedProdu...
Packaging.java
Stock
Stock.java
StockCollection....
Shipment.java
ShipmentCollecti...

The customer class is a simple object to be manged and included in each shipment.

The Packaging is a class which will be used in series with a respective Product to create a PackagedProduct Object.

The Stock class will keep a track of the quantity of Product available.

The Shipment class will be in charge of all the above classes as well as related variables such as the shipment id ect... However, also has functionalities for calculating totalCost, providing timing details for estimated arrival time and dispatch information functions.

The CLI class is the main class to be run. Here a series of menus and command line interface are used to manage Customers, Products, Transport and Shipments. For the functionalities used in the CLI management respective collection classes are also found in the hierarchies. These contain list of the objects being managed as well as functions to work hand in hand with the CLI class. These being adding objects to the list, removing objects to the list, viewing items in the list, updating items in the list and other necessary specifications.

The use of collection classes help better visualize and manage functions for neatness in the CLI class.

Note declaration of objects are mostly example based. For example, creating a product has preset parameters, as well as transport and some other functions. To increase the simplicity of program.

(b)

The implementation of the project in C++ followed the same hierarchical and logical approach. However making use of different folder structures with cmake:

Q1_Cpp/.vscode

Q1_Cpp/build

Q1_Cpp/include

Q1_Cpp/lib

Q1_Cpp/src

The src was used for the .cpp files an the include folder was used for the #included .h files.

The compilation process of the C++ caused more errors this is due to the include errors that might have been present and also some translational errors that I might have made. However, all classes hold the required functionality and methods respectively. I wasn't able to finish the last debugging of the C++ due to time constraints. I am sure there is a very stupid reason for why the CLI is not running.

2. Refactoring the Logistics Management Application

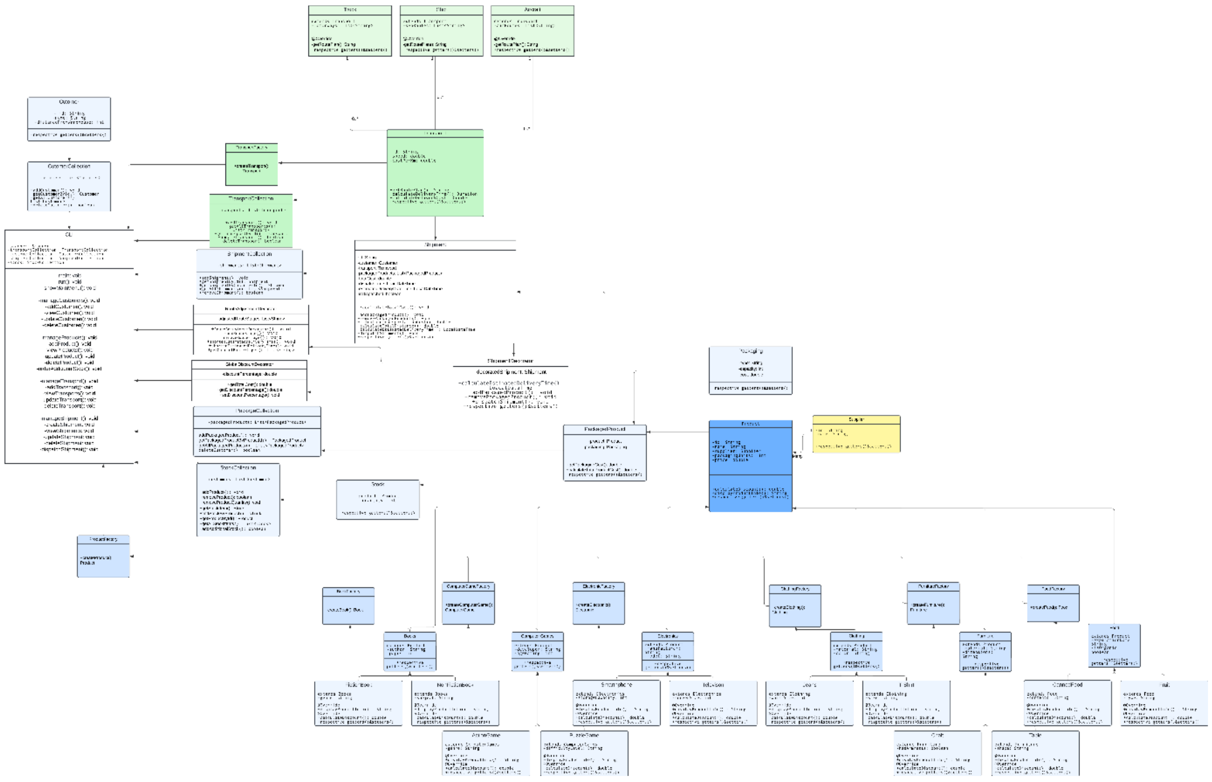


Figure 2 UML Q2

https://lucid.app/lucidchart/2cded046-8d05-4e3c-8554-f7f29a140024/edit?viewport_loc=-4325%2C-2921%2C6236%2C3361%2CHWEp-vi-RSFO&invitationId=inv_200543b3-2b68-42c8-b753-5d5547302fa3

(a)

Implementation of Factory Pattern recognition involved configuration of the Product and Transport hierarchies.

For the Product hierarchy a main Product Factory class is found which based on type parameter calls respective Subclass Factories and here a further dependency on subtype parameter calls the final class of type of product to be constructed.

Parameters are checked using a simple switch case and are passed down from the main initialization. Here the type and subtype parameters are placed in int format, and an ordered structure is kept.

The additional information found in the subclasses are passed through the use of String named additional information parameters and then are converted to the respective needed use in the lower-level factory classes.

A similar implementation for the Transport hierarchy is used. As a single Transport factory is in charge of creating transport objects, this time passing a string parameter instead for the type.

The implementation of these factory classes from a top level can be seen in the CLI class when managing products and transport as respective menu choices are needed and the type and subtype parameters are simple passed through the top most factories and the addition of the objects will be managed by the factory hierarchy constructed.

(b)

Implementation of the Decorator Pattern can be seen in the folder:

Q2/src/main/java/com/yourorganization/Shipment/Decorator

Where:

Q2/src/main/java/com/yourorganization/Shipment/Decorator/GlobalDiscountDecorator.java

Q2/src/main/java/com/yourorganization/Shipment/Decorator/RouteAdjustmentDecorator.java

Q2/src/main/java/com/yourorganization/Shipment/Decorator/ShipmentDecorator.java

Are the classes used for allowing the required functionalities to be implemented.

A main Shipment Decorator is first created. Here a basis is set up the extended shipment for the further two decorators. We override the already construed functions in the shipment to be able to decorate them respectively.

The first function needed is that of apply a Global discount. A respective Global discount decorator is created. Here a simple double percentage to be deducted from the total cost is passed through as a parameter as well as the shipment object being modified. Now we simply override the function to get the total cost and apply deduct the calculated percentage of discounted cost from the total cost.

The second function is a bit more complex. It involves adding the functionality to edit available routes even when the shipment has already been dispatched. This causing a change in the estimated delivery time. Therefore, here the respective object of shipment along with list of current routes are passed. The decorator is comprised of 4 main functions which can be split into two groups. The adding route stage function which would in turn cause a shortened estimated delivery time and the removal of route stage which would extend this time.

Respective logic and use of the Duration import functionalities are used to bring this specification to use.

These decorators can then be accessed in the CLI through an additional method created called ” applyShipmentDecorators” here a choice of which functionality would be need is presented and the respective parameters are passed. This additional method is called through the update shipment method in the managing shipment functionalities group and allows for updating features even after dispatch which in turn allows the user to edit the shipment at any stage of the process.

(c)

Factory Pattern

-Class decoupling:

The Factory Pattern decouples the creation of objects from their usage as classes don't need to know the exact types of objects they need to create. This separation leads to a more modular manner of coding, where changes in object creation logic will only alter the pattern structure rather than the classes using it. For example, if a new type of Transport where needed to be added, the major impact will be involved in the pattern installation. The classes must only be concerned on how to use the objects rather than which type is being used.

-Code modularity:

Due to the modularity created by Factory pattern, if changes are required the effect can be traced better and lessen the change of unintended consequences changes to unrelated code. It promotes consistency in object creation, which is particularly useful in large projects where different parts of the code might otherwise create objects in slightly different ways, leading to inconsistencies and potential bugs.

Decorator Pattern

-Class Decoupling:

The Decorator Pattern allows for the dynamic addition of responsibilities to objects having to change the structure extensively and over complicating the class hierarchies. This means you can extend the functionality of objects at runtime. It promotes the principle of Single Responsibility, as each decorator class is responsible for a specific feature or behaviour. This separation of concerns means that changes in one aspect of functionality shouldn’t impact others.

-Code Maintainability:

Decorators provide a flexible way for modifying exiting code following the principle of the Open/Closed by making use of subclasses for extended functionalities. This makes the codebase more maintainable, as there is less risk of introducing bugs into existing code. Each decorator focuses on a specific aspect, making the code easier to understand and maintain. This organization is desirable for readability and avoiding chaotic class handling.

In summary, both the Factory and Decorator patterns emphasize separation of concerns and encapsulation to contribute to a more robust and adaptable design. That helps in creating a system of code that is easier to manage, extend, and maintain and offers lots of modular structure benefits.

3. Portable data persistence.

(a)

The relative proto file with all the needed messages was created:

Q3/src/main/proto/Data.proto

As well as the respective configuration to the pom.xml file and the involved adding a protobuf dependency for java, a maven protobuf plugin, the os maven plugin and the respective compiler plugin.

To finally getting the respective automatically generated files on build:

Q3/target/generated-sources/protobuf/java/yourorganization/Data.java

Q3/target/generated-sources

Q3/target/generated-sources/annotations

Q3/target/generated-sources/protobuf/java/ yourorganization

Q3/target/generated-sources/protobuf/java/yourorganization/Data.java

(b)

The Facade for creating the methods to load () and save () in the CLI comprised of making save() and load() methods in the respective collection classes and calling these methods.

To save and load to a proto file we need to serialize and deserialize the information of each object. This was done through two further methods convertToData and convertFromData in the respective base classes of the objects.

For saving of objects such as Shipment a type of layering was done as the shipment needed to work down convert the shipment information which contains transport, customer and packagedProducts which all need to be converted having packagedProducts also need to convert Products and packaging.

The base layout for all classes was set-up. However due to time constraints errors are present in the classes and the methods have not been fully finished but logic should be present.

The main problem faced here was the builder class and how it worked in hand with the messages.

(C)

Here Q3_Cpp inherits the same compilation error present in Q1_Cpp however the implementation for the stock viewer application is present.

Due to the structure chosen to work with of collection classes the stock viewer choice was made very simply using the same logic as the previously constructed viewer classes found in the CLI.

As we iterate through the list of stocks in the stock collection. Checking first if empty we then proceed to reuse the product viewer in the cli manage product method group and add the additional stock variable of quantity.

4. Distributed computation.

(a)

The remote service protobuf files was defined using gRPC:

Q4/src/main/proto/RemoveService.proto

The pom.xml file was edited for the use of the gRPC. It is to be noted that the use of version 0.6.1 is used since the latest gRPC could not be found in the Maven Repository and plugins so a more outdated version has been committed. The respective gRPC and protobuf Dependencies along with a protobuf Maven plugin allowed for the required protobuff files to be generated:

Q4/target/generated-sources

Q4/target/generated-sources/protobuf

Q4/target/generated-sources/annotations

Q4/target/generated-sources/protobuf/java/shipment/RemoveService.java