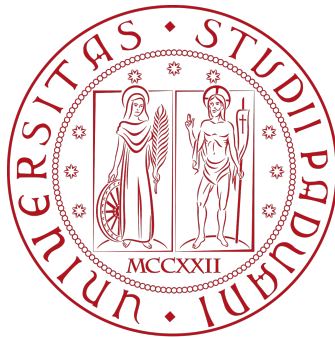# Clothing FaQTory
Object Oriented Programming Project

Elena Ferro

2042328

University of Padua

Computer Science

Academic year 2023/23

1. Introduction

ClothingFaQTory is a management software designed for a company that produces clothing items and accessories. The core functionalities it provides are managing an inventory of the different types of products and calculating their production cost. Its goal is to help managers and business analysts to establish the final selling price of the aforementioned items.

The calculation is based on the cost of the prime materials per unit (such as the price of denim per meter) which is then multiplied by the approximate surface of the final product, in order to estimate the quantity of material needed.

The price is not stored with all the other information: the cost of materials is rather dynamic over time, being influenced by economy and other factors. Therefore, the software allows to easily change these costs and obtain an immediate recalculation of the product prices, without having to update all the records in the database. In a real world scenario, with possibly thousands of products, this operation would be really onerous.

I chose this topic because as a job I develop software for a company that produces accessories, therefore I have a little bit of experience in this field.

2. Logic model

The logic model is subdivided in two parts: the products hierarchy itself, whose diagram is shown in Figure 1 and Figure 2 (splitted for reasons of space), and all the classes to interact with the database and handle errors, described in the Data Persistence section.

The products hierarchy models all the items that the company sells. Each of them has generic information, such as `color`, `description`, `size`, `code` etc. The `code` is not unique, since the software offers the possibility to store different variations of the same product (for instance, an item could be manufactured in different colors or sizes).

A product can be either an `Accessory` or a `ClothingItem`. Both of them are implemented by three concrete classes, each of which has additional data used to compute the final price. A very basic case of this calculation is the price of `Jeans`, obtained with the sum of the lateral area of two cilinders. On the other hand, a more elaborated example would be the `Bracelet`: its cost is computed through the volume of each pearl (approximately a sphere) multiplied by the specific weight of the pearl's material.

A diamond inheritance case has been inserted in the hierarchy: `Overalls` are a composition of the top part (a `Vest`) and the bottom part (a pair of `Jeans`). Hence, the final price is the sum of both its parents cost.

The `Size` of the product affects the final price too: a clothing item of larger size obviously requires more material than a smaller one, therefore the manufacturing cost is higher. A percentage, named `extraPercentageOfMaterial`, is assigned to each size. This represents how much extra material is needed to fabricate an element of that size compared to one of the smallest size. For instance, if the field `extraPercentageOfMaterial` for the size `M` is set to 20, it means that a clothing item of size `M` requires 20% more material to be produced than one of size `XS` (which is the smallest). This percentage is supposed to be constant over time.

`Product`, `Size` and `Material` all extend the `Model` class, which represents a generic database entity.

Two custom containers have been developed; the first one is `Map<K, V>`, which stores ordered key-value pairs. Likewise `std::map`, it is based on a self-balancing red-black tree and thus provides efficient search, insert and delete operations in $O(\log n)$ time. Its main usages are holding the products subdivided by their concrete type and supporting a rudimentary caching system in `SizeRepository` and `MaterialRepository`, which simply stores the entities that have already been retrieved by their id, to avoid repeatedly quering the database for the same data.

Since some functions of `Map`, such as `keys()` and `values()`, rely on a list, I also implemented a plain `LinkedList<T>` container, to make `Map` independent from `std::list`.

In order to isolate the logic which chooses the concrete product class that will be created, the Factory method design pattern has been implemeted (in the `ProductFactory` class).
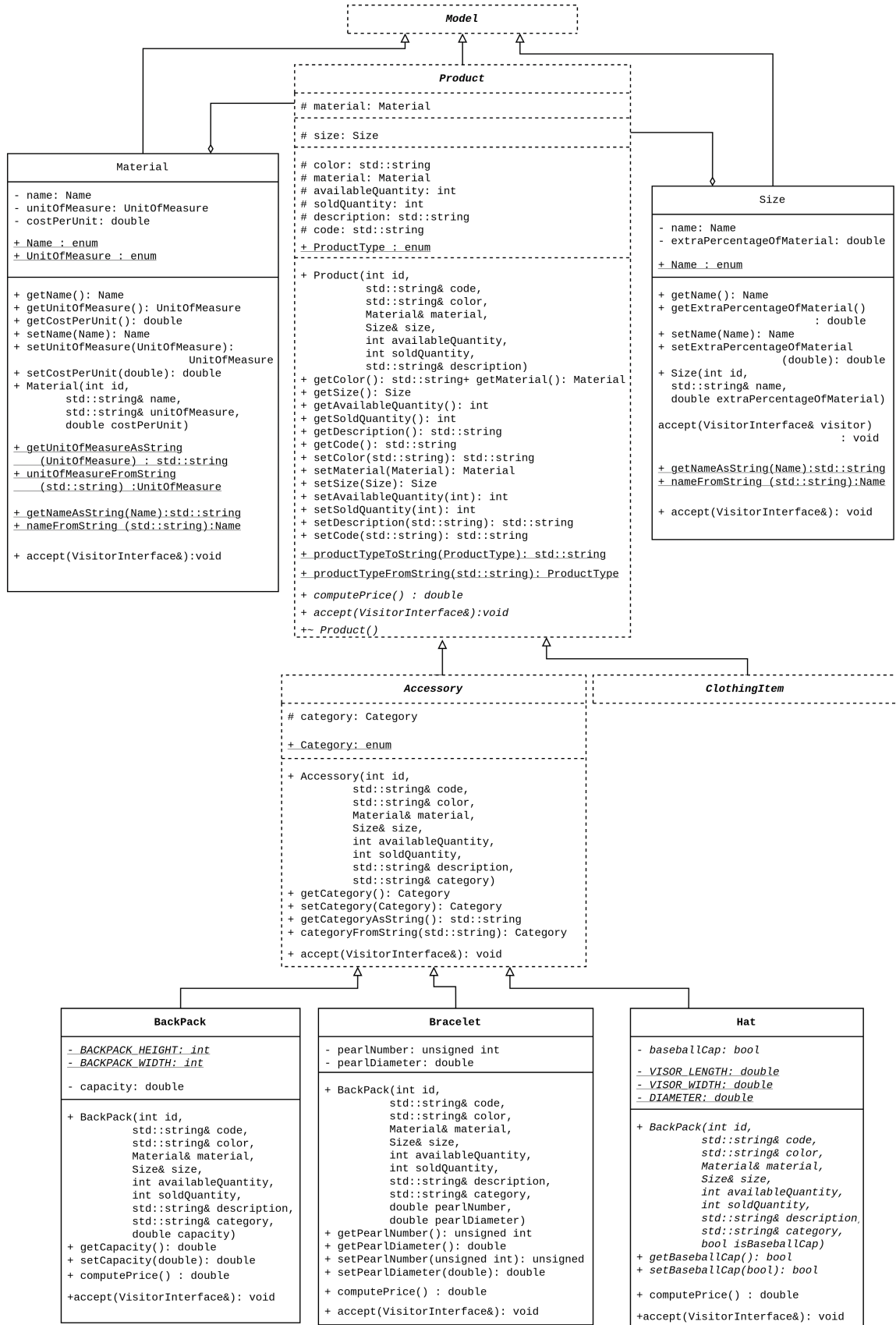
Model

Product

material: Material

size: Size

color: std::string
material: Material
availableQuantity: int
soldQuantity: int
description: std::string
code: std::string
+ ProductType : enum

+ Product(int id,
          std::string& code,
          std::string& color,
          Material& material,
          Size& size,
          int availableQuantity,
          int soldQuantity,
          std::string& description)
+ getColor(): std::string+ getMaterial(): Material
+ getSize(): Size
+ getAvailableQuantity(): int
+ getSoldQuantity(): int
+ getDescription(): std::string
+ getCode(): std::string
+ setColor(std::string): std::string
+ setMaterial(Material): Material
+ setSize(Size): Size
+ setAvailableQuantity(int): int
+ setSoldQuantity(int): int
+ setDescription(std::string): std::string
+ setCode(std::string): std::string
+ productTypeToString(ProductType): std::string
+ productTypeFromString(std::string): ProductType

+ computePrice() : double
+ accept(VisitorInterface&):void
+~ Product()

Material

- name: Name
- unitOfMeasure: UnitOfMeasure
- costPerUnit: double
+ Name : enum
+ UnitOfMeasure : enum

+ getName(): Name
+ getUnitOfMeasure(): UnitOfMeasure
+ getCostPerUnit(): double
+ setName(Name): Name
+ setUnitOfMeasure(UnitOfMeasure):
                        UnitOfMeasure
+ setCostPerUnit(double): double
+ Material(int id,
         std::string& name,
         std::string& unitOfMeasure,
         double costPerUnit)

+ getUnitOfMeasureAsString
      (UnitOfMeasure) : std::string
+ unitOfMeasureFromString
      (std::string) :UnitOfMeasure

+ getNameAsString(Name):std::string
+ nameFromString (std::string):Name

+ accept(VisitorInterface&):void

Size

- name: Name
- extraPercentageOfMaterial: double
+ Name : enum

+ getName(): Name
+ getExtraPercentageOfMaterial()
                           : double
+ setName(Name): Name
+ setExtraPercentageOfMaterial
                (double): double
+ Size(int id,
    std::string& name,
    double extraPercentageOfMaterial)

accept(VisitorInterface& visitor)
                            : void

+ getNameAsString(Name):std::string
+ nameFromString (std::string):Name

+ accept(VisitorInterface&): void

Accessory

category: Category

+ Category: enum

+ Accessory(int id,
          std::string& code,
          std::string& color,
          Material& material,
          Size& size,
          int availableQuantity,
          int soldQuantity,
          std::string& description,
          std::string& category)
+ getCategory(): Category
+ setCategory(Category): Category
+ getCategoryAsString(): std::string
+ categoryFromString(std::string): Category

+ accept(VisitorInterface&): void

ClothingItem

BackPack

- BACKPACK_HEIGHT: int
- BACKPACK_WIDTH: int

- capacity: double

+ BackPack(int id,
          std::string& code,
          std::string& color,
          Material& material,
          Size& size,
          int availableQuantity,
          int soldQuantity,
          std::string& description,
          std::string& category,
          double capacity)
+ getCapacity(): double
+ setCapacity(double): double
+ computePrice() : double

+accept(VisitorInterface&): void

Bracelet

- pearlNumber: unsigned int
- pearlDiameter: double

+ BackPack(int id,
          std::string& code,
          std::string& color,
          Material& material,
          Size& size,
          int availableQuantity,
          int soldQuantity,
          std::string& description,
          std::string& category,
          double pearlNumber,
          double pearlDiameter)
+ getPearlNumber(): unsigned int
+ getPearlDiameter(): double
+ setPearlNumber(unsigned int): unsigned
+ setPearlDiameter(double): double

+ computePrice() : double

+ accept(VisitorInterface&): void

Hat

- baseballCap: bool

- VISOR_LENGTH: double
- VISOR_WIDTH: double
- DIAMETER: double

+ BackPack(int id,
          std::string& code,
          std::string& color,
          Material& material,
          Size& size,
          int availableQuantity,
          int soldQuantity,
          std::string& description,
          std::string& category,
          bool isBaseballCap)
+ getBaseballCap(): bool
+ setBaseballCap(bool): bool

+ computePrice() : double

+accept(VisitorInterface&): void

Figure 1: Logic model class UML (Accessories part)

2

Model

- id : int

+ getId(): int
+ setId(int): int
+ Model(int id)
+~Model()

+accept(VisitorInterface&):void

VisitorInterface

+ visitBracelet(Bracelet&): void
+ visitBackPack(BackPack&): void
+ visitHat(Hat&): void
+ visitJeans(Jeans&): void
+ visitVest(Vest&): void
+ visitOveralls(Overalls&): void
+ visitMaterial(Material&): void
+ visitSize(Size&): void

+ ~VisitorInterface()

Product

ClothingItem

Gender gender
sustainableMaterials: bool

- SMALLEST_SIZE_TRUNK_WIDTH: double
- SMALLEST_SIZE_TRUNK_LENGTH: double
- SMALLEST_SIZE_LEG_WIDTH: double
- SMALLEST_SIZE_LEG_LENGTH: double
+ Gender: enum

+ ClothingItem(int id,
            std::string& code,
            std::string& color,
            Material& material,
            Size& size,
            int availableQuantity,
            int soldQuantity,
            std::string& description,
            bool sustainableMaterials,
            std::string& gender)
+ hasSustainableMaterials(): bool
+ setSustainableMaterials(bool): bool
+ getGenderAsString(): std::string
+ genderFromString(std::string): Gender
- computeLegLength() : double
- computeTrunkLength() : double
- computeLegWidth() : double
- computeTrunkWidth() : double

DecoratorInterface

+ DecoratorInterface(Model&)

+~DecoratorInterface()

JSONExportableDecorator

- fieldsGetterVisitor:FieldsGetterVisitor
- jsonObject: QJsonObject

- accept(VisitorInterface&): void

+ JSONExportableDecorator(Model&)

+ exportData() : QJsonObject&

Jeans

- shorts: bool

+ Jeans(int id,
            std::string& code,
            std::string& color,
            Material& material,
            Size& size,
            int availableQuantity,
            int soldQuantity,
            std::string& description,
            bool sustainableMaterials,
            std::string& gender,
            bool shorts)
+ getShorts(): bool
+ setShorts(bool): bool

+ accept(VisitorInterface&): void

+ computePrice() : double

Vest

hasButtons: bool

- N_BUTTONS: u_int
- BUTTON_DIAMETER: double

+ Vest(int id,
            std::string& code,
            std::string& color,
            Material& material,
            Size& size,
            int availableQuantity,
            int soldQuantity,
            std::string& description,
            bool sustainableMaterials,
            std::string& gender,
            bool hasButtons)
+ getHasButtons(): bool
+ setHasButtons(bool): bool

+ computePrice() : double

+ accept(VisitorInterface&): void

Overalls

+ Overalls(int id,
            std::string& code,
            std::string& color,
            Material& material,
            Size& size,
            int availableQuantity,
            int soldQuantity,
            std::string& description,
            bool sustainableMaterials,
            std::string& gender,
            bool hasButtons,
            bool shorts)

+ accept(VisitorInterface&): void

+ computePrice() : double

Figure 2: Logic model class UML (ClothingItems part)

3

3. Polymorphism

One of the uses of polymorphism is the implementation of the Visitor design pattern, through the `VisitorInterface` class. It allows performing different operations on the hierarchy, depending on their dynamic type. This interface is extended by three different concrete classes. Two of them are used to build different UI layouts: `InfoDialogVisitor`, to display a dialog with the complete product information, and `SpecificProductInfoVisitor` to populate the form to create and edit items. The third usage is the `FieldGetterVisitor`, which takes a `Product` subclass as a parameter and creates a key-value map between its field names and their current value. This comes in handy in different circumstances: in the aforementioned dialog to show the product information, but also to export the entities as JSON and to dynamically build the `INSERT` queries.

With the purpose of saving data to files, the Decorator design pattern has been utilized, coupled with the aforementioned `FieldGetterVisitor`. The `JSONExportableDecorator` class (extending `DecoratorInterface`) enriches the `Model` class adding the functionality to export an entity as a `QJsonObject`, while keeping the hierarchy itself totally independent from any QT class (and thus potentially reusable with another framework).

Another example is the computation of the product price, which is different for each concrete class. For this reason, all the concrete product classes redefine the virtual abstract `computePrice` method.

The Strategy pattern is another significant use of polymorphism. An abstract `Mapper` defines the blueprint for an algorithm used to map query results, from `QSqlQuery` to a generic `Model`. For each entity, a concrete class extending `Mapper` and overriding `operator()` has been defined.

4. Data persistance

The chosen method for data persistance is a PostgreSQL database. The primary motivation for this choice is that Postgres natively supports both single and multiple hierarchy between tables and is generally more efficient than other DBMS, such as MySQL.

The database structure is therefore symmetrical to the C++ hierarchy, having the table `product` as a parent to all the others, with foreign keys referencing the `size` and `material` tables.The Entity-Relationship diagram and the schema of the database can both be found in the project files, inside the `docs` directory.

Postgres connections become idle after a certain timeout of inactivity, which obviously results in the inability to use the software. To overcome this problem without having to set an unnecessarily large timeout, the `ConnectivityManager` class regularly pings the database every minute. To avoid slowing down the other operations, this is done in a separate thread.

4.1 Repository pattern

The purpose of the repository pattern is to provide an abstraction layer between the database and the business logic. The project is based on a MVC architecture. Controllers are the only ones that interact with repositories and they are totally independent from the data source (precisely, thank to the repository pattern), hence the possibility to effortlessly substitute the source of the data set (for instance, with an API integration) leaving the logic and frontend code entirely untouched. A UML class diagram for this pattern can be found inside the `docs` directory.

The `Repository` class is abstract, and only exposes static utility functions to execute queries and check for errors. It is then extended by `ReadOnlyRepository<T>` and `DeleteOnlyRepository`, whose common child is `CRUDRepository<T>`.

Each concrete repository implements the Singleton design pattern, which means that just a single instance will be created, and then referenced everywhere.

This structure provides different types of concrete repositories. For example, `SizeRepository` is read-only since its records will not be modified. On the other hand, given that any row of an entity of the product hierarchy can be eliminated equally from the parent `product` or the child table, `DeleteOnlyRepository` is used for simplicity (without having to check which concrete Repository should be used to delete that entity).

Repositories for each `Product` subclass must support CRUD operations. Consequently, a concrete class extending and specializing at the same time `CRUDRepository<T>` has been implemented for each one of them. Those take care of passing the correct table name and `Mapper` to its parents. Furthermore, in an hypothetical extension of this project, it might become necessary to have a specific query for a given entity. With separated concrete repositories, this method can conveniently be implemented in the corresponding concrete class, preserving the separation of concerns.

Another layer of abstract is added through the `QueryBuilder` utility. No repository class uses directly SQL statements as strings since, with an eye to the project maintainment, this is quite error-prone. They are instead composed chaining calls to the `QueryBuilder` methods and executing `.build()` in the end. Forecasting possible changes concering the database, this class would allow a non-dramatical switch to another SQL dialect.

4.1.1   Error handling

Repositories take care of checking queries for errors too. Being the use of exceptions highly debated in C++, I preferred to implement a functional-languages-inspired error handling strategy: each function returns an object of type `Either<Error, Entity>`, which contains an `Error` if it has occurred, otherwise the requested entity. The implemented `Either` class offers methods to check if the contained value is right (thus the query was successfull) or left (the query resulted in an error) and the method `fold`, which accepts two lambda functions, `ifLeft` and `ifRight`.

If an error was detected, the controller which called the repository method will take care of handling it, emitting a `databaseError` signal, received by the view.

5.   Implemented features

The following list includes additional features, compared to those strictly required.

5.1   Functional

- Export entities as JSON, divided by type

- Filter the products by type, code or price range (and eventually combine these filters)

- Sort the products by available quantity, sold quantity or code. This feature is available in the filter panel.

- Obtain an esteem of the products prices.

- Validation of all the form fields (in the filter panel, in the wizard to create/edit the products and in the dialog to set the material price)

5.2   Graphical

- Usage of a tab bar to distinguish between operations on the `product` hierarchy and on the `material` table.

- Usage of a toolbar in the Products tab to provide `Create new`, `Filter` and `Export to JSON` functionalities, with the respective keyboard shortcuts.

- Usage of a custom font.

- Usage of stylesheets to customize the tab bar and the buttons in the materials cost tab.

- Usage of icons in the tab bar, toolbar, products tree view and materials cost tab.

- Usage of a tree view widget to display the products divided by product type. When expanded, the general information for the items is shown, together with icon buttons to edit, delete and view also the specific product information (not shown in the main view). A square icon with the preview of the product color is also created for each row.

- Asking confirmation through a dialog before deleting a product

- Presence of a button to clear all the filters (appears only when filters are applied).

- Wizard to create and edit the products:

   - Possibility to generate a random code during the creation process (clicking ⌗ beside the code field).
   - Selection of the product color with the color picker widget. Once the color is picked, a preview of the color and its hex code will be available, with the button text adapting to the background, in order to be readable with both dark and light colors.

– Keyboard shortcuts to navigate the wizard pages.

– Validation of the mandatory form fields, which turn red if the user tries to click next but they are not filled.

– Based on the chosen product type, the user will be prompted to insert the specific fields for that product (in addition to the generic information common to all the items).

– Asking confirmation through a dialog when *Cancel* is pressed.

6.   Hours report

Activity	Planned hours	Actual Hours
Planning project structure	5	5
Develop code for models and DB interaction	12	14
Study QT framework	8	11
UI code development	15	16
Testing and debugging	6	8
Report	4	4
Total	50	**58**

As shown in the table, the project required 8 extra hours to be completed as compared to the initial evaluation. This is attributable to the necessity to deepen my knowledge of QT regarding both classes to interact with the database and UI widgets, but also to introduce graphical enhancements (such as custom icons/font) and to improve the efficiency of the queries and mapping methods.