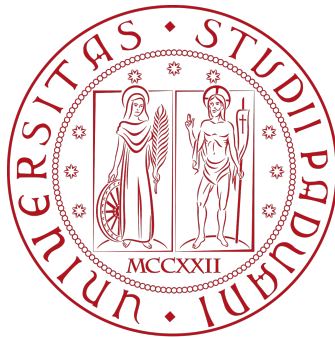


Clothing FaQTory

Object Oriented Programming Project

Elena Ferro
2042328



University of Padua
Computer Science
Academic year 2023/23

Contents

1 Introduction 2

2 Logic model 2

3 Polymorphism 5

4 Data persistance 5

 4.1 Repository pattern 5

 4.1.1 Error handling 6

5 Implemented features 6

 5.1 Functional 6

 5.2 Graphical 6

6 Hours report 8

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 aim 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 the materials is rather dynamic over time, being influenced by economy and other factors. Therefore, the software allows to easily change the cost of the prime materials 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 the calculation of the price for each concrete product is the perfect circumstance to adopt polymorphism.

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, two items 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 to compute the final price (this data can be a static constant value like the **DIAMETER** for the **Hat**, or it can be dynamic and instance-based like the **capacity** in liters of a **BackPack**). A very basic case of this calculation is the price of **Jeans**, obtained with the sum of the lateral area of two cylinders. 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 considered to be 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 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 and only holds the **id**.

Two custom containers have been created for the hierarchy entities; 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 querying 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**.

During the process of creation of new products, in order to split the logic that selects the concrete product type based on the one chosen by the user, the Factory method design pattern has been implemented (in the **ProductFactory** class).

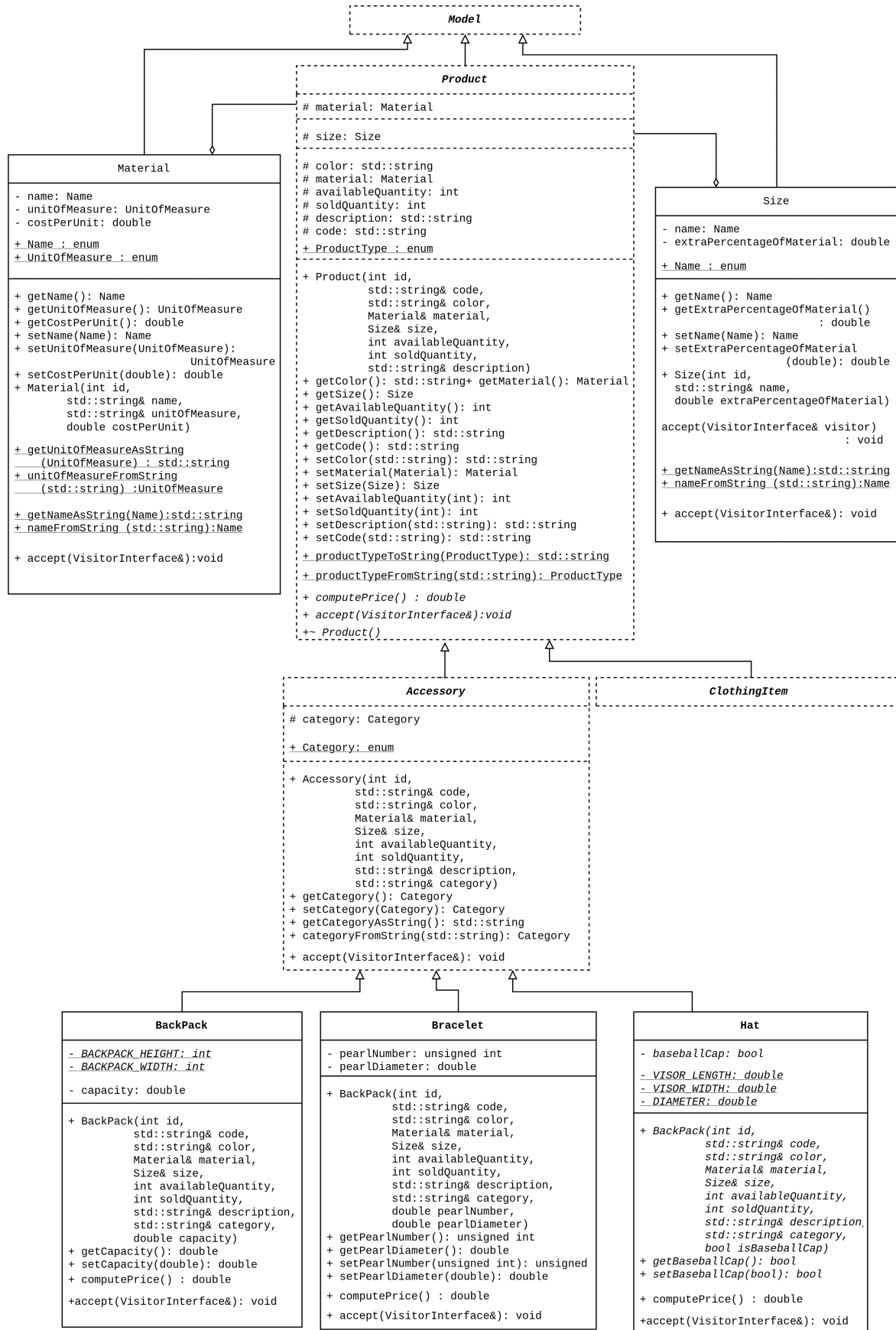


Figure 1: Logic model class UML (Accessories part)

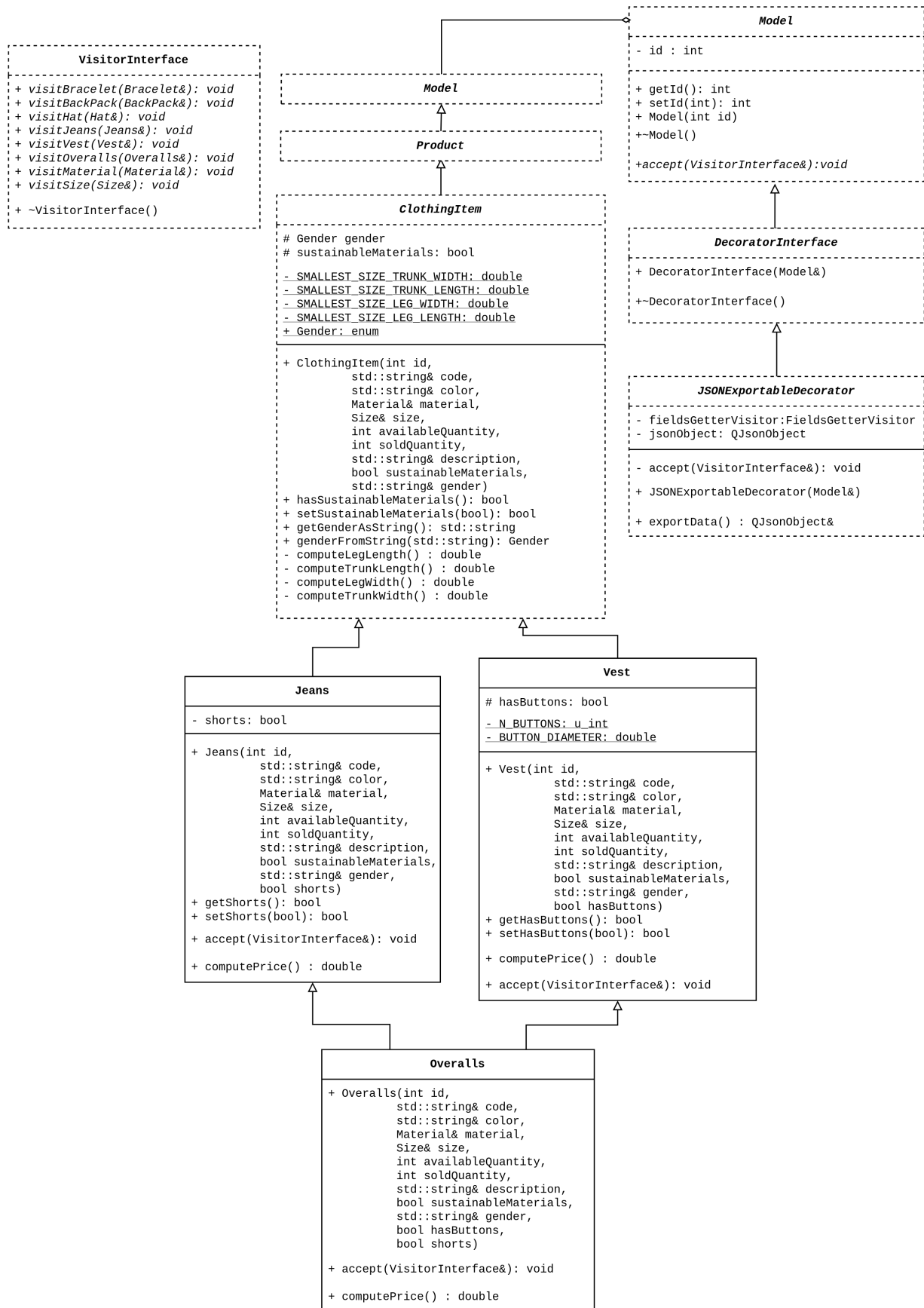


Figure 2: Logic model class UML (ClothingItems part)

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 for items creation and editing. 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 dumping 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 `JsonObject`, 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. Therefore, they all 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 persistence

The chosen method for data persistence 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 providing an abstraction layer between the database and the business logic. The project is based on a MVC architecture. Controllers are the only ones who interact with Repositories but they are totally independent from the data source (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 instance, `SizeRepository` is read-only since its records will not be modified. On the other hand, the only operations carried out directly on the `product` table are `DELETE` queries and given that the record can be eliminated equally from the `product` or its child tables, `DeleteOnlyRepository` is used for simplicity.

Repositories for each `Product` subclass must support CRUD operations. Consequently, a concrete repository extending and specializing at the same time `CRUDRepository<T>` has been implemented for each one of them.

These concrete repositories 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 this is error-prone, but rather composes them chaining calls to the **QueryBuilder** methods and executing **.build()** in the end. With an eye to possible changes concerning the database, this class would allow a non-dramatic switch from a SQL dialect to another.

4.1.1 Error handling

Repositories also check the query for errors. 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 implement **Either** class offers methods to check if the contained value, stored in a **std::variant**, is right (thus the query was successful) 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 respective 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 price
- Validation of all the form fields (in the filter panel, in the wizard to create and 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 a stylesheet to personalize 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
- 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).
 - Select the color of the product 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 improvements such as custom icons and font and optimize the mapping of entities from queries.