

Documentation du modèle et de l'architecture
Système de Gestion d'Entrepôt (Warehouse Manager)

Chaymae MAKHOKH

Anas BZIOUI

6 décembre 2025

Table des matières

Introduction

Ce document décrit la structure du modèle, les principales classes métiers et les éléments de l'architecture Qt du projet *Warehouse Manager*. L'objectif est de relier clairement :

- le diagramme UML fourni,
- l'implémentation C++ dans le projet Qt (dossier `domain` et contrôleurs),
- les fonctionnalités visibles dans l'interface utilisateur (dialogs d'ajout, hiérarchie dans la `QTreeView`, etc.).

Toutes les classes du modèle sont regroupées dans le dossier `src/domain` du projet Qt, et la logique applicative dans le dossier `contrôleur`.

Les classes principales implémentées sont :

- `Product` (classe abstraite),
- `ProduitAvecCaracteristiques`,
- `ProduitAvecCycleDeVie`,
- `Conteneur`,
- `Palette`, `ElementsPalette`,
- `Entrepot`,
- `ContrainteCompatibilite`, `ReglesCompatibilite`,
- les énumérations `TypeProduit`, `TypeConteneur`, `EtatProduit`,
- les contrôleurs `ProduitControleur` et `ConteneurControleur`,
- les vues Qt : `MainWindow`, `ajouterProduit`, `ajouterconteneur`.

1 Vue d'ensemble de l'architecture

1.1 Structure métier

Le modèle représente la structure suivante :

- Un **Entrepôt** (`Entrepot`) contient plusieurs **Conteneurs** (`Conteneur`) et plusieurs **Palettes** (`Palette`).
- Chaque **Conteneur** contient une liste de **produits** (`Product*`), qui peuvent être de deux types concrets : `ProduitAvecCaracteristiques` (non périssable) ou `ProduitAvecCycleDeVie` (périssable).
- Chaque **Palette** contient un ensemble d'**éléments** (`ElementsPalette`) qui référencent des produits avec des quantités.
- Les compatibilités entre types de produits sont exprimées via des `ContrainteCompatibilite`, regroupées dans `ReglesCompatibilite`.

1.2 Structure logicielle

L'application suit une architecture inspirée MVC :

- **Modèle** : classes du dossier `domain` (`Product`, `Conteneur`, `Palette`, `Entrepot`, etc.).
- **Contrôleurs** :
 - `ProduitControleur` gère la liste des produits en mémoire et fournit des méthodes de création.
 - `ConteneurControleur` gère la liste des conteneurs et permet d'y ajouter des produits.
- **Vues Qt** :
 - `MainWindow` : fenêtre principale avec `QTreeView` pour visualiser la hiérarchie conteneurs / produits.
 - `ajouterProduit` : boîte de dialogue pour la création d'un produit.
 - `ajouterconteneur` : boîte de dialogue pour la création d'un conteneur.

Les contrôleurs servent de « pont » entre les vues Qt et le modèle métier.

2 Énumérations

Toutes les énumérations sont définies dans `enums.h` et leurs fonctions utilitaires dans `enums.cpp`. Elles sont déclarées comme métatypes Qt via `Q_DECLARE_METATYPE` afin de pouvoir être stockées dans des `QVariant` et utilisées dans les `QComboBox` de l'interface.

2.1 TypeProduit

Rôle Énumération représentant le type d'un produit.

Valeurs

- `Alimentaire`
- `Electronique`
- `Medicament`
- `Autre` (valeur par défaut)

Utilisation

- Attribut `m_type` dans `Product`.
- Attributs `m_typeA` et `m_typeB` dans `ContrainteCompatibilite`.

2.2 TypeConteneur

Rôle Énumération représentant le type d'un conteneur (normal, froid, fragile, etc.).

Valeurs

- `Normal`
- `Froid`
- `Fragile`
- `Autre`

Utilisation Attribut `m_type` dans la classe `Conteneur`, et dans l'interface pour alimenter la `QComboBox` `TypeConteneur_2`.

2.3 EtatProduit

Rôle Énumération représentant l'état d'un produit.

Valeurs

- `Stocke`
- `Expedie`

Utilisation Retour par les méthodes virtuelles `etat()` dans `Product` et stocké dans les sous-classes concrètes.

3 Modèle de produits

3.1 Product (classe abstraite)

Rôle Classe de base abstraite représentant un produit de l'entrepôt. Elle ne contient que l'identité commune, le type et la capacité maximale. Les aspects physiques et les dates sont gérés dans les sous-classes.

Attributs privés

- `QString m_idProduit`
- `QString m_nom`
- `TypeProduit m_type`
- `double m_capaciteMax`

Méthodes publiques

- Accesseurs : `idProduit()`, `setIdProduit()`, `nom()`, `setNom()`, `type()`, `setType()`, `capaciteMax()`, `setCapaciteMax()`.
- Méthodes virtuelles pures (polymorphisme) :
 - `virtual double poids() const = 0;`
 - `virtual double volume() const = 0;`
 - `virtual QDate dateEntreeStock() const = 0;`
 - `virtual QDate datePeremption() const = 0;`
 - `virtual EtatProduit etat() const = 0;`
- Méthode utilitaire : `bool estPerime(const QDate& jour) const;` qui teste la validité de `datePeremption()`.

Signal

- `void productChanged();` émis à chaque modification.

Ce design permet de manipuler tous les produits via des `Product*` (par exemple dans les conteneurs), tout en déléguant le comportement concret aux sous-classes.

3.2 ProduitAvecCaracteristiques

Rôle Produit non périssable : poids, volume, conditions de conservation et état. La date de péremption n'est pas utilisée.

Attributs spécifiques

- `double m_poids`
- `double m_volume`
- `QString m_conditions`
- `QDate m_dateEntreeStock`
- `EtatProduit m_etat`

Redéfinitions

- `double poids() const override;`
- `double volume() const override;`
- `QDate dateEntreeStock() const override;`
- `QDate datePeremption() const override;` (renvoie une date invalide pour indiquer « non périssable »).
- `EtatProduit etat() const override;`

Utilisation Ce type est choisi lorsque l'utilisateur coche « Produit avec des caractéristiques » dans la boîte de dialogue `ajouterProduit`. Les champs `Poids`, `Volume`, `ConditionsConservation`, `DateEntreeStock` sont alors utilisés, et le champ `DatePeremption` est masqué.

3.3 ProduitAvecCycleDeVie

Rôle Produit périssable : possède un cycle de vie complet avec dates d'entrée en stock et de péremption.

Attributs spécifiques

- double m_poids
- double m_volume
- QDate m_entree
- QDate m_peremption
- EtatProduit m_etat

Redéfinitions Même principe que la classe précédente, mais ici `datePeremption()` renvoie une date valide et permet d'appliquer des stratégies FEFO (First Expire First Out).

4 Conteneurs, palettes et entrepôt

4.1 Conteneur

Rôle Représente un conteneur physique dans l'entrepôt.

Attributs

- QString m_idConteneur
- TypeConteneur m_type
- double m_capaciteMax
- QList<Product*> m_produits

Méthodes principales

- Accesseurs classiques sur l'identité et le type.
- `const QList<Product*>& produits() const;` : accès en lecture.
- `double poidsTotal() const;` : somme des poids.
- `bool peutAjouter(Product *p) const;` : vérifie si le poids total reste inférieur à `m_capaciteMax`.
- `bool ajouterProduit(Product *p);` : ajoute le produit ici si `peutAjouter()` est vrai.
- `void retirerProduit(Product *p);` : retire toutes les occurrences.

Signaux

- `conteneurChanged(), produitAjoute(Product*), produitRetire(Product*)`.

4.2 ElementsPalette et Palette

Ces classes implémentent la logique de palettes d'expédition (liste d'éléments avec quantités, vérification des règles de compatibilité, etc.), en conformité avec le diagramme UML. Elles ne sont pas détaillées ici car la fonctionnalité principale développée actuellement concerne surtout la partie *conteneurs / produits* et la visualisation dans l'interface.

4.3 Entrepot

Rôle Racine logique du modèle (niveau 1 de l'arborescence). Contient les listes de conteneurs et de palettes, ainsi que l'ensemble des règles de compatibilité (`ReglesCompatibilite`).

5 Contrôleurs

5.1 ProduitControleur

Rôle Gère la collection de produits en mémoire et fournit des méthodes de création utilisées par l'interface graphique.

Attribut interne

— `QVector<std::shared_ptr<Product>> m_produits;`

Méthodes

— `std::shared_ptr<Product> ajouterProduitAvecCaracteristiques(...);`
— `std::shared_ptr<Product> ajouterProduitAvecCycleDeVie(...);`
— `void debugPrintProduits() const;` : affichage dans la sortie de débogage Qt (liste des produits avec leurs propriétés).

Les méthodes d'ajout retournent un `shared_ptr<Product>` afin que le produit puisse être immédiatement ajouté dans un conteneur via le `ConteneurControleur`.

5.2 ConteneurControleur

Rôle Gère la liste des conteneurs et permet d'y ajouter des produits.

Attribut interne

— `QVector<std::shared_ptr<Conteneur>> m_conteneurs;`

Méthodes

— `void ajouterConteneur(const QString&, TypeConteneur, double);`
— `const QVector<std::shared_ptr<Conteneur>>& conteneurs() const;`
— `bool ajouterProduitAuConteneur(const QString& idConteneur, const std::shared_ptr<Product> produit);`
— `void debugPrintConteneurs() const;`

Principe Lorsqu'un produit est créé dans la boîte de dialogue `ajouterProduit`, l'utilisateur peut choisir un conteneur dans une `QComboBox`. Le contrôleur va alors retrouver le conteneur via son identifiant et y ajouter le produit (`Conteneur::ajouterProduit`).

6 Interface Qt et intégration du modèle

6.1 ajouterconteneur

Rôle Boîte de dialogue (`QDialog`) utilisée pour créer un nouveau conteneur.

Éléments d'interface

— `IdConteneur_2` : `QLineEdit` pour l'identifiant.
— `TypeConteneur_2` : `QComboBox` alimentée par les valeurs de `TypeConteneur` via `QVariant`.
— `CapaciteMax_2` : `QDoubleSpinBox`.
— Boutons `Créer` et `Annuler`.

Fonctionnement Lors du clic sur « Créer », la méthode `on_Creer_clicked()` lit les valeurs des widgets et appelle `ConteneurControleur::ajouterConteneur`. Le dialog renvoie `QDialog::Accepted`.

6.2 ajouterProduit

Rôle Boîte de dialogue de création de produit. Elle gère deux types de produits : avec caractéristiques ou avec cycle de vie.

Éléments principaux

- Deux `QCheckBox` exclusives :
 - `checkBox` : « Produit avec des caractéristiques »
 - `checkBox_2` : « Produit avec cycle de vie »
- `TypeProduit_2` : `QComboBox` alimentée par `TypeProduit`.
- `Etat_2` : `QComboBox` alimentée par `EtatProduit`.
- Champs numériques : `CapaciteMax_2`, `Poids_2`, `Volume_2`.
- Dates : `DateEntree_2`, `DatePeremption_2`.
- `ConditionsConservation_2` : `QPlainTextEdit` pour les conditions.
- `DansQuelConteneur_2` : `QComboBox` listant tous les identifiants de conteneurs existants, obtenus auprès du `ConteneurControleur`.

Comportement dynamique

- Les deux cases à cocher sont mutuellement exclusives.
- Selon le type choisi :
 - pour **Produit avec caractéristiques** : les champs `ConditionsConservation` sont visibles, `DatePeremption` est cachée.
 - pour **Produit avec cycle de vie** : `DatePeremption` est visible, `ConditionsConservation` est cachée.
- Lors du clic sur « Créer » :
 - le dialog appelle la méthode appropriée du `ProduitControleur` pour créer l'objet métier,
 - si un conteneur est sélectionné dans `DansQuelConteneur_2`, le produit est ajouté à ce conteneur via `ConteneurControleur::ajouterProduitAuConteneur`.

6.3 MainWindow et hiérarchie dans la QTreeView

Rôle Fenêtre principale de l'application. Elle contient :

- un `QComboBox` `comboBoxTypeModel` pour basculer entre les groupes « Conteneur », « Palette » et « Produit » ;
- un `QGroupBox` avec une `QTreeView` pour afficher l'arborescence `Conteneur → Produits`.

Modèle de la TreeView La `QTreeView` est alimentée par un `QStandardItemModel` (`m_treeModel`).

La méthode `rebuildTreeView()` :

- efface le modèle,
- parcourt la liste des conteneurs fournie par `ConteneurControleur::conteneurs()`,
- pour chaque conteneur crée un item racine « Conteneur <Id> »,
- pour chaque produit du conteneur, crée un enfant « Produit <IdProduit> ».

La méthode est appelée après chaque création de conteneur ou de produit, afin de garder la vue toujours synchronisée avec le modèle.

Conclusion

Le projet *Warehouse Manager* implémente un modèle d'entrepôt structuré, basé sur des classes métiers claires (`Product`, `Conteneur`, `Palette`, etc.), enrichi par des contrôleurs qui font le lien avec l'interface Qt.

Les dialogs d'ajout permettent de créer des objets cohérents (en utilisant les énumérations du modèle) et de construire progressivement une hiérarchie conteneurs / produits visualisable dans la `QTreeView` de la fenêtre principale.

Ce document pourra être complété par la suite avec :

- la description détaillée des algorithmes FIFO/FEFO de génération de palettes,

- la gestion et l'édition des règles de compatibilité (`ReglesCompatibilite`),
- des captures d'écran de l'interface finale.