Projet MPQL - Gestion d'un Distributeur Automatique de Boissons

1. Diagramme de Classes UML

Description du Système

Le système de distributeur automatique est conçu autour d'une architecture modulaire où chaque composant a une responsabilité bien définie. Au cœur du système se trouve la classe **DistributeurAutomatique** qui agit comme le chef d'orchestre de toutes les opérations.

Rôle de Chaque Classe et Relations Principales

DistributeurAutomatique - Classe Contrôleur Principal

Rôle : Point d'entrée et orchestrateur de tout le système. Cette classe centralise la logique métier et coordonne toutes les opérations du distributeur.

Responsabilités principales :

- Gestion du catalogue des boissons disponibles
- Interface avec les utilisateurs pour les opérations d'achat
- Coordination entre le stock, le portefeuille et les transactions
- Mise en service et arrêt du distributeur

Attributs clés : `id`, `nom`, `enService`, `catalogueBoissons`, `gestionnaireStock`, `portefeuille`, `journalVentes`

Méthodes essentielles :

- `afficherBoissonsDisponibles()` : Liste les boissons disponibles
- `acheterBoisson(int, int, double)` : Gère l'achat d'une boisson
- `ajouterNouvelleBoissonboisson(Boisson)` : Ajoute une boisson au catalogue
- `supprimerBoisson(idBoisson, int)` : Retire une boisson du système
- `mettreEnService()` et `mettreHorsService()` : Contrôle l'état du distributeur

Boisson - Entité Métier Fondamentale

Rôle : Représente les produits vendus par le distributeur avec toutes leurs caractéristiques commerciales.

Responsabilités :

- Stockage des informations produit (nom, prix, description)
- Catégorisation des boissons (sodas, jus, eau, etc.)
- Gestion des informations de marque
- Calcul des prix selon différents critères

Attributs : `id`, `nom`, `prix`, `description`, `categorie`, `marque`

Méthodes : `getPrix()`, `getNom()`, `getId()`, `getDescription()`, `mettreAJourPrix(nouveauPrix)`, `toString()`

GestionnaireStock - Gestionnaire d'Inventaire

Rôle : Contrôle exclusif de l'inventaire des boissons, garantissant la cohérence des stocks.

Responsabilités :

- Suivi des quantités disponibles pour chaque boisson
- Validation de la disponibilité avant vente
- Gestion des réapprovisionnements
- Alertes de rupture de stock

Attributs : `stockBoissons` (Map<Integer, Integer> - idBoisson -> quantité)

Méthodes principales :

- `obtenirQuantite(idBoisson, int)` : Consulte la quantité disponible
- `ajouterStock(idBoisson, int, quantite, int)` : Ajoute du stock
- `retirerStock(idBoisson, int, quantite, int)` : Retire du stock lors d'une vente

Portefeuille - Gestionnaire Monétaire

Rôle : Simule le système de paiement du distributeur, gérant les espèces et la monnaie.

Responsabilités :

- Gestion des pièces et billets acceptés
- Calcul et rendu de monnaie
- Validation des montants insérés
- Maintien du fonds de caisse pour les rendus

Attributs : `montantInsere`, `montantTotal`, `piecesAcceptees`, `billetsAcceptes`

Méthodes :

- `insererMontant(montant, double)` : Enregistre l'argent inséré
- `calculerMonnaie(prixBoisson, double)` : Calcule la monnaie à rendre
- `rendreMonnaie(montant, double)` : Effectue le rendu de monnaie

JournalVentes - Système de Traçabilité

Rôle : Assure la traçabilité complète de toutes les opérations commerciales du distributeur.

Responsabilités :

- Enregistrement de chaque transaction
- Calcul des statistiques de vente
- Génération de rapports pour la gestion
- Historique pour audit et analyse

Attributs : `transactions`, `chiffreAffaireJournalier`, `nombreVentesJournalieres`, `dateCreation`

Méthodes :

- `ajouterTransaction(transaction, TransactionAchat)` : Enregistre une nouvelle transaction
- `obtenirTransactionsParDate(date, LocalDate)` : Filtre les transactions par date
- `calculerChiffreAffaire(dateFin, LocalDate)` : Calcule le CA sur une période
- `obtenirBoissonsLesPlusVendues()` : Statistiques de vente

TransactionAchat - Enregistrement Transactionnel

Rôle : Représente une opération d'achat individuelle avec tous ses détails.

Responsabilités :

- Conservation des détails de chaque vente
- Traçabilité client-produit-paiement
- Validation et confirmation des transactions
- Calcul des montants et rendus

Attributs : `idTransaction`, `boisson`, `montantInsere`, `prixBoisson`, `monnaieRendue`, `dateTransaction`, `statut`

Méthodes :

- `validerTransaction()` : Confirme la transaction
- `annulerTransaction()` : Annule la transaction
- `calculerMonnaieRendue()` : Calcule le rendu
- `estReussie()` : Vérifie le statut de réussite

Utilisateur - Gestion Clientèle

Rôle : Modélise les clients du distributeur pour un service personnalisé et la fidélisation.

Responsabilités :

- Identification des utilisateurs
- Historique personnel des achats
- Gestion des droits d'accès (client/personnel)
- Authentification pour les opérations sensibles

Attributs : `id`, `nom`, `typeUtilisateur`, `historiqueAchats`

Méthodes :

- `authentifier(motDePasse, String)` : Authentification utilisateur
- `ajouterAchat(transaction, TransactionAchat)` : Ajoute un achat à l'historique
- `obtenirHistoriqueAchats()` : Récupère l'historique
- `estAdministrateur()` et `estClient()` : Vérifie le type d'utilisateur

Relations Principales et Architecture

Relations de Composition (1:1)

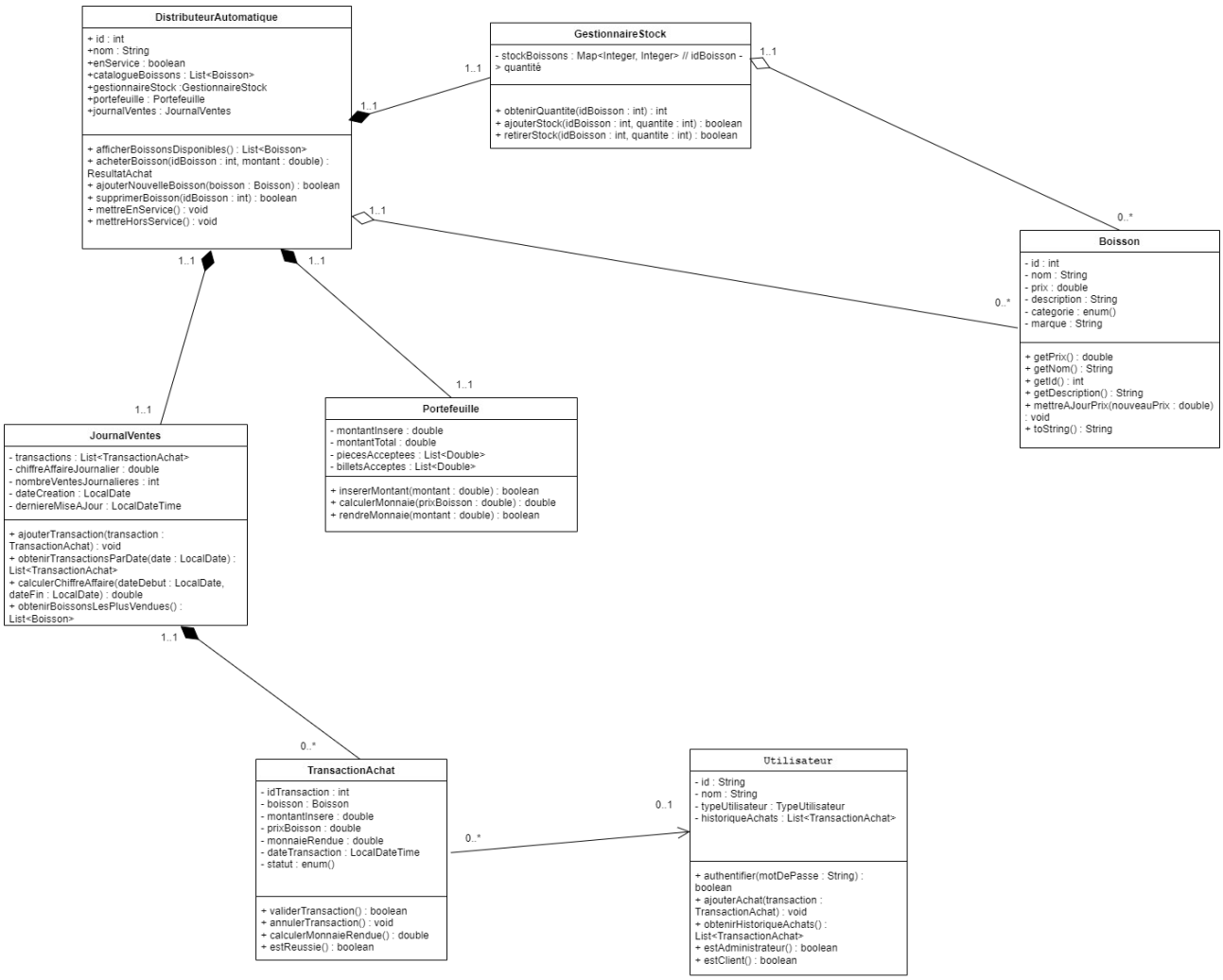
- **DistributeurAutomatique ↔ GestionnaireStock** : Le distributeur possède son propre gestionnaire de stock
- **DistributeurAutomatique ↔ Portefeuille** : Chaque distributeur a son système monétaire intégré
- **DistributeurAutomatique ↔ JournalVentes** : Un journal unique par distributeur

Relations d'Agrégation (1:*)

- **DistributeurAutomatique → Boisson** : Le distributeur référence plusieurs boissons dans son catalogue
- **GestionnaireStock → Boisson** : Le stock associe des quantités aux boissons
- **JournalVentes → TransactionAchat** : Le journal archive toutes les transactions

Relations d'Association

- **TransactionAchat → Boisson** : Chaque transaction référence la boisson achetée
- **Utilisateur → TransactionAchat** : Un utilisateur peut effectuer plusieurs achats
- **TransactionAchat → Utilisateur** : Traçabilité client par transaction



Flux Opérationnel Type

1. **L'Utilisateur** interagit avec le **DistributeurAutomatique**
2. Le **DistributeurAutomatique** consulte le **GestionnaireStock** pour vérifier la disponibilité
3. Si disponible, le **Portefeuille** gère le paiement et le rendu de monnaie
4. Une **TransactionAchat** est créée et enregistrée dans le **JournalVentes**
5. Le **GestionnaireStock** met à jour les quantités disponibles

Cette architecture modulaire respecte les principes SOLID, avec une séparation claire des responsabilités, facilitant la maintenance, les tests et l'évolution du système.

---

2. Liste des Tests Unitaires Proposés

BoissonTest :

testCreationBoisson
Ce test valide que le constructeur de la classe Boisson initialise correctement tous les attributs d'une boisson.

```
@Test
void testCreationBoisson() {
   Boisson boisson = new Boisson(1, "Café Noir", 2.50,
         "Café noir intense", Boisson.Categorie.CAFE, "Nespresso");

   assertEquals(1, boisson.getId());
   assertEquals("Café Noir", boisson.getNom());
   assertEquals(2.50, boisson.getPrix(), 0.001);
   assertEquals("Café noir intense", boisson.getDescription());
   assertEquals(Boisson.Categorie.CAFE, boisson.getCategorie());
}
```

testMettreAJourPrix
Ce test vérifie que la méthode mettreAJourPrix(double nouveauPrix) met bien à jour le prix d'une boisson lorsque le nouveau prix est valide (positif).

```
@Test
void testMettreAJourPrix() {
   Boisson boisson = new Boisson(2, "Thé Vert", 1.80,
         "Thé vert bio", Boisson.Categorie.THE, "Lipton");
   boisson.mettreAJourPrix(2.00);
   assertEquals(2.00, boisson.getPrix(), 0.001);
}
```

testMettreAJourPrixNegatif
Ce test vérifie que la méthode mettreAJourPrix déclenche une exception (IllegalArgumentException) lorsqu'on essaie de définir un prix négatif.

```
@Test
void testMettreAJourPrixNegatif() {
   Boisson boisson = new Boisson(3, "Eau Minérale", 1.00,
         "Eau minérale naturelle", Boisson.Categorie.EAU, "Evian");
   assertThrows(IllegalArgumentException.class,
```

```
                () -> boisson.mettreAJourPrix(-1.00));
        }
```

DistributeurAutomatiqueTest :

testAffichageBoissons
Ce test vérifie que si la méthode afficherBoissonsDisponibles() retourne la liste des boissons ajoutées, ici une seule .

```
@Test
void testAffichageBoissons() {
   assertEquals(1, distributeur.afficherBoissonsDisponibles().size());
}
```

testAchatAvecSucces
Ce test vérifie :

- L'utilisateur paie 2.00 pour une boisson à 1.50
- La transaction est acceptée (statut = "REUSSIE")
- La monnaie rendue est correcte (2.0 - 1.5 = 0.5).

testAchatMontantInsuffisant
Ce test vérifie :

- Si un utilisateur paie moins que le prix de la boisson (ici 1.0 < 1.5)
- alors l'achat est annulé
- le montant payé est remboursé intégralement.

```
@Test
void testAchatAvecSucces() {
   TransactionAchat transaction = distributeur.effectuerAchat(1, 2.0, "U123");
   assertEquals("REUSSIE", transaction.getStatut());
   assertEquals(0.5, transaction.getMonnaieRendue(), 0.01);
}
```

testBoissonNonTrouvee
Ce test vérifie :

- Quand l'utilisateur tente d'acheter une boisson avec un ID inexistant (999)
- alors une exception IllegalStateException est levée
- et le message d'erreur doit contenir "Boisson non trouvée".

```
@Test
   void testAchatMontantInsuffisant() {
      TransactionAchat transaction = distributeur.effectuerAchat(1, 1.0, "U123");
      assertEquals("ANNULEE", transaction.getStatut());
      assertEquals(1.0, transaction.getMonnaieRendue(), 0.01);
   }
```

testAjoutBoissonHorsService
Ce test vérifie :

- Si le distributeur est mis hors service (mettreHorsService())

- alors il est interdit d’ajouter une nouvelle boisson
- et cela déclenche une IllegalStateException.

```
@Test
   void testAjoutBoissonHorsService() {
      distributeur.mettreHorsService();
      assertThrows(IllegalStateException.class, () -> {
         distributeur.ajouterNouvelleBoisson(new Boisson(2, "Thé", 1.2, "chaud",
Boisson.Categorie.THE, "img.png"));
      });
   }
```

testMontantNegatif

Ce test vérifie :

- Si un utilisateur essaie d’acheter une boisson en passant un montant négatif
- alors le système doit rejeter cette valeur invalide
- en lançant une IllegalArgumentException.

```
@Test
   void testMontantNegatif() {
      assertThrows(IllegalArgumentException.class, () -> {
         distributeur.effectuerAchat(1, -1.0, "U123");
      });
   }
```

GestionnaireStockTest :

testAjouterStock

Ce test vérifie, lorsqu’on ajoute 5 unités de la boisson avec l’ID 1, la quantité enregistrée devient bien 5.

```
@Test
void testAjouterStock() {
   stock.ajouterStock(1, 5);
   assertEquals(5, stock.obtenirQuantite(1));
}
```

testRetirerStock

Ce test vérifie :

- On ajoute 10 unités à une boisson (ID 2), puis on en retire 4.
- Le système doit ensuite indiquer qu’il reste 6 unités.

```
@Test
   void testRetirerStock() {
      stock.ajouterStock(2, 10);
      stock.retirerStock(2, 4);
      assertEquals(6, stock.obtenirQuantite(2));
   }
```

testStockEpuise

Ce test vérifie, lorsqu’une boisson (ID 3) n’a jamais été ajoutée, elle est considérée comme en rupture de stock.

```
@Test
void testStockEpuise() {
```

```
    assertTrue(stock.stockEpuise(3));
}
```

testObtenirQuantiteBoissonInexistante

Ce test vérifie que pour un ID de boisson non présent (ici 99), obtenirQuantite(...) retourne bien 0 et pas une exception ou une valeur aléatoire.

```
@Test
void testObtenirQuantiteBoissonInexistante() {
    assertEquals(0, stock.obtenirQuantite(99));
}
```

testAjouterQuantiteInvalide

Ce test vérifie :

- Ajouter 0 (ou potentiellement une quantité négative) déclenche une IllegalArgumentException.
- On essaye de protéger la méthode contre des appels invalides.

```
@Test
    void testAjouterQuantiteInvalide() {
        assertThrows(IllegalArgumentException.class, () -> stock.ajouterStock(1, 0));
    }
```

testRetirerTropDeStock

Ce que ce test vérifie :

- On tente de retirer plus d'unités qu'il n'y en a en stock (5 alors qu'il n'y en a que 2).
- Une exception IllegalStateException doit être levée.

```
@Test
void testRetirerTropDeStock() {
    stock.ajouterStock(4, 2);
    assertThrows(IllegalStateException.class, () -> stock.retirerStock(4, 5));
}
```

JournalVentesTest :

testObtenirTransactionsParDate

Ce test vérifie que toutes les transactions ajoutées aujourd'hui sont bien retrouvées via la fonction obtenirTransactionsParDate().

```
@Test
void testObtenirTransactionsParDate() {
    LocalDate date = LocalDate.now();

    journal.ajouterTransaction(achat1);
    journal.ajouterTransaction(achat2);
    journal.ajouterTransaction(achat3);

    List<TransactionAchat> transactions = journal.obtenirTransactionsParDate(date);
    assertEquals(3, transactions.size());
}
```

testGetTransactionsRetourneCopie

Ce test vérifie que getTransactions() retourne une copie de la liste à chaque appel, et pas une référence directe.

```
@Test
void testGetTransactionsRetourneCopie() {
   journal.ajouterTransaction(achat1);
   List<TransactionAchat> liste1 = journal.getTransactions();
   List<TransactionAchat> liste2 = journal.getTransactions();
   assertNotSame(liste1, liste2);
}
```

testToStringContientInfo

Ce test vérifie :

Que la méthode toString() renvoie une représentation textuelle utile du journal, contenant :

- La liste des transactions
- Le chiffre d'affaires (CA)
- Le nombre de ventes.

```
@Test
   void testToStringContientInfo() {
      journal.ajouterTransaction(achat1);
      String s = journal.toString();
      assertTrue(s.contains("transactions="));
      assertTrue(s.contains("CA="));
      assertTrue(s.contains("ventes="));
   }
```

testAjouterTransactionNullLanceException

Ce test vérifie que si on essaie d'ajouter null comme transaction, le système doit rejeter l'opération avec une exception (IllegalArgumentException).

```
@Test
void testAjouterTransactionNullLanceException() {
   assertThrows(IllegalArgumentException.class, () -> journal.ajouterTransaction(null));
}
```

PortefeuilleTest :

testInsererMontantAugmenteMontantInsere

Ce test vérifie qu'après l'appel à insererMontant(5.0), la méthode getMontantInsere() retourne bien 5.0.

```
@Test
void testInsererMontantAugmenteMontantInsere() {
   portefeuille.insererMontant(5.0);
   assertEquals(5.0, portefeuille.getMontantInsere(), 0.0001);
}
```

testCalculerMonnaieRetourneDifference

Ce test vérifie que si on insère 10.0 et qu'un achat coûte 7.5, calculerMonnaie(7.5) doit retourner 2.5.

```
@Test
void testCalculerMonnaieRetourneDifference() {
   portefeuille.insererMontant(10.0);
   double monnaie = portefeuille.calculerMonnaie(7.5);
   assertEquals(2.5, monnaie, 0.0001);
}
```

testMontantSuffisantRetourneVraiSiAssez (Vérifier la validation logique d'un montant suffisant,et s'assurer que montantSuffisant(prix) retourne correctement true ou false selon la comparaison entre le montant inséré et le prix).

Ce test vérifie que :

- Si on insère 2.0 :
  - Un achat à 1.5 est possible → true.
  - Un achat à 2.5 est impossible → false.

```
@Test
void testMontantSuffisantRetourneVraiSiAssez() {
   portefeuille.insererMontant(2.0);
   assertTrue(portefeuille.montantSuffisant(1.5));
   assertFalse(portefeuille.montantSuffisant(2.5));
}
```

TransactionAchatTest :

testTransactionReussie

Ce test vérifie que:

- La méthode validerTransaction() retourne true, indiquant que la transaction a été acceptée.
- Le statut est bien changé à "REUSSIE".
- La monnaie rendue est bien 3.0 - 2.0 = 1.0.

```
@Test
   void testTransactionReussie() {
      Boisson b = new Boisson(1, "Chocolat", 2.0, "chaud",
Boisson.Categorie.CHOCOLAT, "img.png");
      TransactionAchat t = new TransactionAchat(1, b, 3.0, "U1");
      assertTrue(t.validerTransaction());
      assertEquals("REUSSIE", t.getStatut());
      assertEquals(1.0, t.getMonnaieRendue(), 0.01);
   }
```

testTransactionAnnulee

Ce test vérifie que:

- Lorsqu'on appelle annulerTransaction(), le statut devient "ANNULEE".
- Le montant inséré (0.5) est entièrement remboursé.

```
@Test
   void testTransactionAnnulee() {
      Boisson b = new Boisson(2, "Eau", 1.0, "froide", Boisson.Categorie.EAU,
"img.png");
      TransactionAchat t = new TransactionAchat(2, b, 0.5, "U2");
      t.annulerTransaction();
      assertEquals("ANNULEE", t.getStatut());
```

```
        assertEquals(0.5, t.getMonnaieRendue(), 0.01);
    }
```

testToStringEtDetails

Ce que ce test vérifie :

- La méthode obtenirDetailsTransaction() retourne une chaîne contenant des informations claires sur la transaction (ici on vérifie la présence de "Transaction 3").
- La méthode toString() renvoie une représentation textuelle de l'objet contenant l'id.

```
@Test
    void testToStringEtDetails() {
        Boisson b = new Boisson(3, "Jus", 2.5, "froid", Boisson.Categorie.JUS,
"img.jpg");
        TransactionAchat t = new TransactionAchat(3, b, 3.0, "U3");
        t.validerTransaction();
        assertTrue(t.obtenirDetailsTransaction().contains("Transaction #3"));
        assertTrue(t.toString().contains("TransactionAchat{id=3"));
    }
```

___

UtilisateurTest :

testAuthentificationAdminCorrecte

Ce test vérifie que:

- L'administrateur s'authentifie avec le mot de passe correct ("admin123").
- La méthode authentifier () renvoie true.

```
@Test
    void testAuthentificationAdminCorrecte() {
        assertTrue(admin.authentifier("admin123"));
    }
```

testAuthentificationAdminIncorrecte

Ce test vérifie que :

- Un administrateur ne peut pas s'authentifier avec un mot de passe incorrect.
- La méthode renvoie false.

```
@Test
    void testAuthentificationAdminIncorrecte() {
        assertFalse(admin.authentifier("wrongpass"));
    }
```

testAuthentificationClientToujoursValide

Ce test vérifie que pour un client, l'authentification est toujours acceptée (quel que soit le mot de passe).

```
@Test
void testAuthentificationClientToujoursValide() {
   assertTrue(client.authentifier("nimportequoi"));
}
```

testAjouterAchatEchoue (Vérifier que seuls les achats réussis sont enregistrés dans l'historique)

Ce test vérifie que :

- Un achat échoué ("ECHOUÉE") ne doit pas être ajouté à l'historique du client.

- Et que liste des achats reste vide après ajout.

```
@Test
void testAjouterAchatEchoue() {
    client.ajouterAchat(achatRate);
    List<TransactionAchat> historique = client.obtenirHistoriqueAchats();
    assertTrue(historique.isEmpty());
}
```

testEstAdministrateurEtClient (S'assurer que le rôle de chaque utilisateur est correctement identifié par les méthodes estAdministrateur() et estClient())

Ce test vérifie que :

- L'utilisateur avec rôle "ADMINISTRATEUR" est bien reconnu comme administrateur.
- Un client (par défaut) est reconnu comme client.
- Aucun des deux ne doit être détecté comme l'autre type.

```
@Test
void testEstAdministrateurEtClient() {
    assertTrue(admin.estAdministrateur());
    assertFalse(admin.estClient());
    assertTrue(client.estClient());
    assertFalse(client.estAdministrateur());
}
```

#. Liste des Tests D'acceptation Proposés

```
Fonctionnalité: Gestion du catalogue de boissons
  En tant que gestionnaire de distributeur
  Je veux gérer les différentes boissons disponibles
  Afin de maintenir une offre actualisée pour les clients

  Règle:
* Une boisson doit avoir un prix positif
* L'ID d'une boisson doit être unique
* Les champs obligatoires sont : nom, prix, catégorie, marque

  Scénario: Création d'une nouvelle boisson valide
    Étant donné les détails suivants pour une nouvelle boisson:
      | id | nom    | prix | description            | categorie | marque     |
      | 4  | Fanta | 800  | Boisson gazeuse orange | SODA      | Coca-Cola  |
    Quand je crée cette boisson
    Alors elle doit être disponible dans le catalogue avec:
      | id | nom    | prix | description            | categorie | marque     |
      | 4  | Fanta | 800  | Boisson gazeuse orange | SODA      | Coca-Cola  |

  Scénario: Mise à jour valide du prix
    Étant donné la boisson existante:
      | id | nom    | prix |
      | 1  | Sprite | 1000 |
    Quand je mets à jour son prix à 1200
    Alors son prix doit être 1200
    Et ses autres propriétés doivent rester inchangées

    Scénario: Tentative de mise à jour avec prix invalide
      Étant donné la boisson existante "Jus d'orange" avec un prix de 500
      Quand j'essaie de mettre à jour son prix à -100
      Alors une erreur "Le prix doit être positif" doit être levée
      Et le prix doit rester à 500

  Scénario: Tentative de mise à jour avec prix invalide
```

```
    Étant donné la boisson existante:
      | id | nom          | prix |
      | 3  | Jus d'orange | 500  |
    Quand j'essaie de mettre à jour son prix à -100
    Alors une erreur "Le prix doit être positif" doit être affichée
    Et le prix doit rester à 500

  Scénario: Consultation des détails d'une boisson
    Étant donné la boisson existante:
      | id | nom          | prix | description           | categorie | marque
|
      | 2  | Cappuccino | 500  | Café avec mousse de lait | CAFE      |
Starbucks  |
    Quand je consulte ses détails
    Alors je dois voir:
      """
      Boisson [id=2, nom=Cappuccino, prix=5.00, description=Café avec mousse de
lait, catégorie=CAFE, marque=Starbucks]
      """
```

```
import static org.junit.Assert.*;
import org.junit.Test;
import org.junit.Before;

public class BoissonTestAcceptance {
    private Boisson sprite;
    private Boisson cappuccino;
    private Boisson jusOrange;

    @Before
    public void setUp() {
        // Préparation commune pour les tests avec tous les paramètres requis
        sprite = new Boisson(1, "Sprite", 1000, "Boisson gazeuse citronnée",
                Boisson.Categorie.SODA, "Coca-Cola");
        cappuccino = new Boisson(2, "Cappuccino", 500, "Café avec mousse de
lait",
                Boisson.Categorie.CAFE, "Starbucks");
        jusOrange = new Boisson(3, "Jus d'orange", 500, "Jus 100% naturel",
                Boisson.Categorie.JUS, "Tropicana");
    }

    @Test
    public void testCreationBoissonValide() {
        // Vérification des attributs de Sprite
        assertEquals(1, sprite.getId());
        assertEquals("Sprite", sprite.getNom());
        assertEquals(1000, sprite.getPrix(), 0.001);
        assertEquals("Boisson gazeuse citronnée", sprite.getDescription());
        assertEquals(Boisson.Categorie.SODA, sprite.getCategorie());
        assertEquals("Coca-Cola", sprite.getMarque());
    }

    @Test
    public void testMiseAJourPrixValide() {
        // Sauvegarde du prix initial pour vérification
        double prixInitial = cappuccino.getPrix();

        // Mise à jour du prix
        cappuccino.mettreAJourPrix(600);

        // Vérification du nouveau prix
        assertNotEquals(prixInitial, cappuccino.getPrix(), 0.001);
        assertEquals(600, cappuccino.getPrix(), 0.001);
```

```
    }

    @Test(expected = IllegalArgumentException.class)
    public void testMiseAJourPrixInvalide() {
        // Tentative de mise à jour avec prix invalide
        jusOrange.mettreAJourPrix(-1.0);
    }

    @Test
    public void testPrixInchangeApresEchecMiseAJour() {
        // Prix avant tentative de modification
        double prixInitial = jusOrange.getPrix();

        try {
            jusOrange.mettreAJourPrix(-1.0);
            fail("Devrait lancer une exception");
        } catch (IllegalArgumentException e) {
            // Vérification que le prix n'a pas changé
            assertEquals(prixInitial, jusOrange.getPrix(), 0.001);
        }
    }

    @Test
    public void testToStringComplet() {
        String resultat = sprite.toString();
        assertTrue(resultat.contains("Sprite"));
        assertTrue(resultat.contains("1000"));
        assertTrue(resultat.contains("SODA"));
        assertTrue(resultat.contains("Coca-Cola"));
    }
}
```

Classe Porte-feuille : ***Tests d'acceptance***

```
Fonctionnalité: Gestion du portefeuille
  En tant qu'utilisateur du distributeur
  Je veux gérer les transactions monétaires
  Afin de payer mes boissons et recevoir la monnaie

  Scénario: Insertion d'un montant valide
    Étant donné un portefeuille initialisé
    Quand j'insère un montant de 2000
    Alors le montant inséré doit être 2000


  Scénario: Insertion d'un montant négatif
    Étant donné un portefeuille initialisé
    Quand j'insère un montant de -5.0
    Alors une erreur "Le montant doit être positif" doit être levée

  Scénario: Vérification de montant suffisant
    Étant donné un portefeuille avec 500 inséré
    Quand je vérifie si le montant est suffisant pour un prix de 200
    Alors le résultat doit être vrai

  Scénario: Vérification de montant insuffisant
    Étant donné un portefeuille avec 50 inséré
    Quand je vérifie si le montant est suffisant pour un prix de 500
    Alors le résultat doit être faux
```

```java
import static org.junit.Assert.*;
import org.junit.Test;
import org.junit.Before;

public class PortefeuilleTestacceptance {

    private Portefeuille portefeuille;

    @Before
    public void setUp() {
        portefeuille = new Portefeuille();
    }

    @Test
    public void testInsertionMontantValide() {
        // Étant donné un portefeuille initialisé (fait dans setUp)

        // Quand j'insère un montant de 2000
        portefeuille.insererMontant(2000);

        // Alors le montant inséré doit être 2000.
        assertEquals(2000, portefeuille.getMontantInsere(), 0.001);
    }

    @Test(expected = IllegalArgumentException.class)
    public void testInsertionMontantNegatif() {
        // Étant donné un portefeuille initialisé (fait dans setUp)

        // Quand j'insère un montant de -5.0
        portefeuille.insererMontant(-5.0);

        // Alors une erreur "Le montant doit être positif" doit être levée
        // (gérée par l'annotation expected.)
    }

    @Test
    public void testVerificationMontantSuffisant() {
        // Étant donné un portefeuille avec 500 insérés.
        portefeuille.insererMontant(500);

        // Quand je vérifie si le montant est suffisant pour un prix de 200
        boolean resultat = portefeuille.montantSuffisant(200);

        // Alors le résultat doit être vrai
        assertTrue(resultat);
    }

    @Test
    public void testVerificationMontantInsuffisant() {
        // Étant donné un portefeuille avec 50 insérés
        portefeuille.insererMontant(50);

        // Quand je vérifie si le montant est suffisant pour un prix de 500
        boolean resultat = portefeuille.montantSuffisant(500);

        // Alors le résultat doit être faux
        assertFalse(resultat);
    }
}
```

Classe GestionnaireStock : Tests d’acceptance

```
Fonctionnalité: Gestion du stock des boissons
En tant que gestionnaire de stock
Je veux gérer les quantités de boissons disponibles
Afin de maintenir un stock optimal dans le distributeur

    Scénario: Ajout de stock avec quantité positive
    Étant donné un stock initial vide pour la boisson #101
    Quand j'ajoute 5 unités de la boisson #101
    Alors la quantité disponible pour #101 doit être 5

    Scénario: Tentative d'ajout avec quantité négative
    Étant donné un stock initial vide pour la boisson #102
    Quand j'essaie d'ajouter -3 unités de la boisson #102
    Alors une exception doit être levée
    Et le stock pour #102 doit rester à 0

    Scénario: Retrait de stock avec quantité suffisante
    Étant donné un stock initial de 10 unités pour la boisson #103
    Quand je retire 4 unités de la boisson #103
    Alors la quantité disponible pour #103 doit être 6
```

```java
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;

public class GestionnaireStockTestAcceptance {
    private GestionnaireStock gestionnaireStock;

    @Before
    public void setUp() {
        gestionnaireStock = new GestionnaireStock();
        // Initialisation spécifique pour le scénario de retrait
        gestionnaireStock.ajouterStock(103, 10);
    }

    //  Ajout de stock avec quantité positive
    @Test
    public void testAjoutStockQuantitePositive() {
        // Étant donné un stock initial vide pour la boisson #101
        assertEquals(0, gestionnaireStock.obtenirQuantite(101));

        // Quand j'ajoute 5 unités de la boisson #101
        gestionnaireStock.ajouterStock(101, 5);

        // Alors la quantité disponible pour #101 doit être 5.
        assertEquals(5, gestionnaireStock.obtenirQuantite(101));
    }

    //  Tentative d'ajout avec quantité négative
    @Test(expected = IllegalArgumentException.class)
    public void testAjoutStockQuantiteNegative() {
        // Étant donné un stock initial vide pour la boisson #102
        assertEquals(0, gestionnaireStock.obtenirQuantite(102));

        // Quand j'essaie d'ajouter -3 unités de la boisson #102
        gestionnaireStock.ajouterStock(102, -3);

        // Alors une exception est levée (vérifiée par l'annotation)
        // Et le stock pour #102 doit rester à 0.
        assertEquals(0, gestionnaireStock.obtenirQuantite(102));
    }

    //  Retrait de stock avec quantité suffisante
```

```
    @Test
    public void testRetraitStockQuantiteSuffisante() {
        // Étant donné un stock initial de 10 unités pour la boisson #103
        assertEquals(10, gestionnaireStock.obtenirQuantite(103));

        // Quand je retire 4 unités de la boisson #103
        gestionnaireStock.retirerStock(103, 4);

        // Alors la quantité disponible pour #103 doit être 6.
        assertEquals(6, gestionnaireStock.obtenirQuantite(103));
    }
}
```

classe TransactionAchat : Tests d'acceptance

```
Fonctionnalité: Gestion des transactions d'achat
En tant que distributeur automatique
Je veux gérer les transactions d'achat
Afin de suivre les ventes et gérer les paiements

    Scénario: Transaction réussie avec monnaie rendue
    Étant donné une boisson "Coca" au prix de 300fr
    Et une transaction #1001 avec 500fr insérés
    Quand je valide la transaction
    Alors le statut doit être "REUSSIE"
    Et la monnaie rendue doit être 200fr
    Et la durée de transaction doit être enregistrée

    Scénario: Transaction échouée par fonds insuffisants
    Étant donné une boisson "Eau" au prix de 400
    Et une transaction #1002 avec 250 insérés
    Quand je valide la transaction
    Alors le statut doit rester "EN_COURS"
    Et la monnaie rendue doit être 0.00
    Et la transaction ne doit pas être marquée comme réussie

    Scénario: Annulation de transaction
    Étant donné une boisson "Café" au prix de 150fr
    Et une transaction #1003 avec 200fr insérés
    Quand j'annule la transaction
    Alors le statut doit être "ANNULEE"
    Et la monnaie rendue doit être 200fr
    Et la date de transaction doit être enregistrée
```

```
import static org.junit.Assert.*;
import org.junit.Test;
import java.time.LocalDateTime;

public class TransactionAchatTestAcceptance {

    @Test
    public void testTransactionReussieAvecMonnaieRendue() {
        // Étant donné une boisson "Coca" au prix de 300
        Boisson coca = new Boisson(1, "Coca", 300, "Boisson gazeuse",
                Boisson.Categorie.SODA, "Coca-Cola");

        // Et une transaction #1001 avec 500 insérés
        TransactionAchat transaction = new TransactionAchat(1001, coca, 500,
"user123");

        // Quand je valide la transaction.
        boolean resultat = transaction.validerTransaction();
        LocalDateTime dateFin = transaction.getDateTransaction();
```

```
        // Alors le statut doit être "REUSSIE"
        assertEquals("REUSSIE", transaction.getStatut());

        // Et la validation doit réussir.
        assertTrue(resultat);

        // Et la monnaie rendue doit être 200
        assertEquals(200, transaction.getMonnaieRendue(), 0.001);

        // Et la durée de transaction doit être enregistrée (positif)
        assertTrue(transaction.getTempsTransaction() > 0);

        // Et la date de transaction doit être récente
        assertTrue(dateFin.isAfter(LocalDateTime.now().minusMinutes(1)));
    }

    @Test
    public void testTransactionEchoueeFondsInsuffisants() {
        // Étant donné une boisson "Eau" au prix de 400
        Boisson eau = new Boisson(2, "Eau", 400, "Eau minérale",
                Boisson.Categorie.EAU, "Evian");

        // Et une transaction #1002 avec 250 insérés
        TransactionAchat transaction = new TransactionAchat(1002, eau, 250,
"user123");
        LocalDateTime dateInitiale = transaction.getDateTransaction();

        // Quand je valide la transaction.
        boolean resultat = transaction.validerTransaction();

        // Alors le statut doit rester "EN_COURS".
        assertEquals("EN_COURS", transaction.getStatut());

        // Et la validation doit échouer
        assertFalse(resultat);

        // Et la monnaie rendue doit être 0.00
        assertEquals(0, transaction.getMonnaieRendue(), 0.001);

        // Et la transaction ne doit pas être marquée comme réussie
        assertFalse(transaction.estReussie());

        // Et la date de transaction ne doit pas changer
        assertEquals(dateInitiale, transaction.getDateTransaction());
    }

    @Test
    public void testAnnulationTransaction() {
        // Étant donné une boisson "Café" au prix de 150
        Boisson cafe = new Boisson(3, "Café", 150, "Café noir",
                Boisson.Categorie.CAFE, "Nespresso");

        // Et une transaction #1003 avec 200 insérés
        TransactionAchat transaction = new TransactionAchat(1003, cafe, 200,
"user123");
        LocalDateTime dateInitiale = transaction.getDateTransaction();

        // Quand j'annule la transaction.
        transaction.annulerTransaction();
        LocalDateTime dateApresAnnulation = transaction.getDateTransaction();

        // Alors le statut doit être "ANNULEE"
        assertEquals("ANNULEE", transaction.getStatut());
```

```
        // Et la monnaie rendue doit être 200
        assertEquals(200, transaction.getMonnaieRendue(), 0.001);

        // Et la date de transaction doit être enregistrée et inchangée
        assertEquals(dateInitiale, transaction.getDateTransaction());
        assertNotNull(dateApresAnnulation);
    }
}
```

Classe Utilisateur : Tests d'acceptance

```
Fonctionnalité: Gestion des utilisateurs du distributeur
  En tant que système de gestion des utilisateurs
  Je veux gérer différents types d'utilisateurs
  Afin de contrôler l'accès et suivre les activités

  Scénario: Création simplifiée d'un client
    Étant donné que je crée un utilisateur avec ID "C123" et nom "Atta Fall"
    Quand j'utilise le constructeur simplifié
    Alors le type d'utilisateur doit être "CLIENT"
    Et l'ID doit être "C123"
    Et le nom doit être "Atta Fall"
    Et la date de création doit être aujourd'hui
    Et la date du dernier accès ne doit pas être nulle

  Scénario: Création d'un administrateur
    Étant donné que je crée un utilisateur avec ID "A176", nom "Yacine Seck" et
type "ADMINISTRATEUR"
    Quand l'utilisateur est créé
    Alors le type d'utilisateur doit être "ADMINISTRATEUR"
    Et l'ID doit être "A176"
    Et le nom doit être "Yacine Seck"
    Et il doit être identifié comme administrateur
    Et il ne doit pas être identifié comme client

  Scénario: Authentification des utilisateurs
    Étant donné un administrateur avec ID "A176"
    Et un client avec ID "C123"
    Quand l'administrateur s'authentifie avec le mot de passe "admin123"
    Alors l'authentification doit réussir
    Quand l'administrateur s'authentifie avec le mot de passe "perdu"
    Alors l'authentification doit échouer
    Quand le client s'authentifie avec n'importe quel mot de passe
    Alors l'authentification doit toujours réussir

  Scénario: Vérification des permissions
    Étant donné un client de type "CLIENT"
    Et un administrateur de type "ADMINISTRATEUR"
    Et un technicien de type "TECHNICIEN"
    Quand je vérifie les permissions
    Alors le client doit être identifié comme client mais pas comme
administrateur
    Et l'administrateur doit être identifié comme administrateur mais pas comme
client
    Et le technicien ne doit être identifié ni comme client ni comme
administrateur
```

```
import org.junit.Test;
import static org.junit.Assert.*;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.util.List;
```

```java
public class UtilisateurTestAcceptance
{

    // Création simplifiée d'un client
    @Test
    public void testCreationSimplifieeClient() {
        // Étant donné que je crée un utilisateur avec ID "C123" et nom "Atta
Fall"
        Utilisateur client = new Utilisateur("C123", "Atta Fall");

        // Alors le type d'utilisateur doit être "CLIENT".
        assertEquals("CLIENT", client.getTypeUtilisateur());

        // Et l'ID doit être "C123"
        assertEquals("C123", client.getId());

        // Et le nom doit être "Atta Fall"
        assertEquals("Atta Fall", client.getNom());

        // Et la date de création doit être aujourd'hui
        assertEquals(LocalDate.now(), client.getDateCreation());

        // Et la date du dernier accès ne doit pas être nulle
        assertNotNull(client.getDernierAcces());
    }

    //  Création d'un administrateur
    @Test
    public void testCreationAdministrateur() {
        // Étant donné que je crée un utilisateur avec ID "A176", nom "Yacine
Seck" et type "ADMINISTRATEUR"
        Utilisateur admin = new Utilisateur("A176", "Yacine Seck",
"ADMINISTRATEUR");

        // Alors le type d'utilisateur doit être "ADMINISTRATEUR".
        assertEquals("ADMINISTRATEUR", admin.getTypeUtilisateur());

        // Et l'ID doit être "A176"
        assertEquals("A176", admin.getId());

        // Et le nom doit être "Yacine Seck"
        assertEquals("Yacine Seck", admin.getNom());

        // Et il doit être identifié comme administrateur
        assertTrue(admin.estAdministrateur());

        // Et il ne doit pas être identifié comme client
        assertFalse(admin.estClient());
    }

    //  Authentification des utilisateurs
    @Test
    public void testAuthentificationUtilisateurs() {
        // Étant donné un administrateur avec ID "A176"
        Utilisateur admin = new Utilisateur("A176", "Yacine Seck",
"ADMINISTRATEUR");

        // Et un client avec ID "C123"
        Utilisateur client = new Utilisateur("C123", "Atta Fall");

        // Quand l'administrateur s'authentifie avec le mot de passe "admin123"
        // Alors l'authentification doit réussir.
        assertTrue(admin.authentifier("admin123"));
```

```
        // Quand l'administrateur s'authentifie avec le mot de passe "perdu"
        // Alors l'authentification doit échouer
        assertFalse(admin.authentifier("perdu"));

        // Quand le client s'authentifie avec n'importe quel mot de passe
        // Alors l'authentification doit toujours réussir.
        assertTrue(client.authentifier("anypassword"));
        assertTrue(client.authentifier(""));
        assertTrue(client.authentifier(null));
    }


    // Vérification des permissions
    @Test
    public void testVerificationPermissions() {
        // Étant donné un client de type "CLIENT"
        Utilisateur client = new Utilisateur("C123", "Client", "CLIENT");

        // Et un administrateur de type "ADMINISTRATEUR"
        Utilisateur admin = new Utilisateur("A176", "Admin", "ADMINISTRATEUR");

        // Et un technicien de type "TECHNICIEN"
        Utilisateur technicien = new Utilisateur("T001", "Tech", "TECHNICIEN");

        // Alors le client doit être identifié comme client, mais pas comme
administrateur
        assertTrue(client.estClient());
        assertFalse(client.estAdministrateur());

        // Et l'administrateur doit être identifié comme administrateur, mais
pas comme client
        assertTrue(admin.estAdministrateur());
        assertFalse(admin.estClient());

        // Et le technicien ne doit être identifié ni comme client ni comme
administrateur
        assertFalse(technicien.estClient());
        assertFalse(technicien.estAdministrateur());
    }
}
```