Rapport du Projet Génie Logiciel

Thème :

Réaliser une application Java native autour du panier de fruits commencé en TD/TP

Réaliser par : Groupe 11

IDIR Amzal

EL MAGHOUM Fayçal

TAHIR Ilyas

RIAK Ilyas

AZDAD Mohamed Oualid

KHOMSI EL HASSOUNI Abdelmajid

Encadré par :

Mme Roudet Celine

Mr El Vaigh Cheikh Brahim

Remerciement

Nous tenons à remercier chaleureusement nos deux professeurs, **Mme Céline Roudet** et **M. Cheikh Brahim El Vaigh,** pour leur excellente pédagogie et leur engagement à expliquer clairement les concepts du cours, des travaux dirigés et des travaux pratiques. Leur capacité à rendre les sujets complexes accessibles a grandement facilité notre compréhension et notre progression dans ce projet.

Résumé

Ce rapport présente un projet de développement d'une application de gestion de paniers de fruits en Java, réalisé par des étudiants en Génie Logiciel. L'analyse des besoins, la planification, et l'utilisation du modèle MVC sont mis en avant. Des extensions, des tests, et des Design Patterns sont envisagés pour enrichir l'application. L'accent est mis sur la qualité, la performance et la sécurité. L'objectif est de créer une IHM conviviale et évolutive.

TABLE DES MATIERES

I. Introduction Générale

La conception et le développement d'Interfaces Homme-Machine (IHM) jouent un rôle central dans le domaine du génie logiciel, offrant des solutions pour simplifier les interactions entre les utilisateurs et les systèmes informatiques. Notre projet a pour ambition de créer des IHM dédiées à la gestion de fruits et de paniers, visant à répondre de manière exhaustive aux besoins des utilisateurs. Ces interfaces ont pour objectif de simplifier la gestion des produits frais, d'offrir des fonctionnalités avancées telles que la création de paniers personnalisés et la production de jus, tout en garantissant une expérience utilisateur conviviale et performante.

Ce rapport se penche sur les extensions que nous avons élaborées et proposées après la formation des groupes lors des science de travaux dirigés. Ces extensions visent à améliorer la qualité et la pertinence de nos IHM en tenant compte de divers aspects essentiels. Nous explorerons en détail ces extensions dans les sections suivantes, en mettant en lumière leurs avantages potentiels pour les utilisateurs de nos IHM de gestion de fruits et de paniers. Ce projet représente une opportunité majeure pour renforcer notre compréhension du développement logiciel, appliquer des concepts clés, et proposer des solutions novatrices pour répondre aux besoins réels des utilisateurs.

II. Cahier de charge

Dans ce deuxième chapitre nous passerons a une études et spécifications de besoins.

A. Spécification et analyse des besoins

1. Besoins fonctionnels

Après avoir définir le cadre général de notre projet nous passerons à une étude d'analyse et spécification de tous ce qui est demandé.

La spécification et l'analyse des besoins sont les étapes initiales cruciales dans le processus de développement de logiciels. Elles visent à définir précisément ce que l'application doit accomplir, en identifiant les parties prenantes et les sources de données. Cette phase est essentielle pour garantir que le résultat final répondra de manière optimale et satisfaisante aux attentes.

Dans cette section, nous débuterons par une spécification détaillée des besoins de l'application. Cela signifie que nous clarifierons les fonctionnalités requises, les comportements attendus.

Notre application propose à l'utilisateur de :

- Visualiser tous les éléments de son panier
- Ajouter un fruit depuis un catalogue dans son panier
- Supprimer le dernier fruit de son panier
- Retirer un fruit spécifique du panier
- Voir prix total de son panier ainsi le prix de chaque fruit
- Ajouter un Jus/Macédoine depuis un catalogue dans son panier
- Augmenter la capacité du panier

- Ajouter un fruit dans le catalogue
- Modifier un fruit du catalogue
- Supprimer un fruit du catalogue
- Ajouter un élément dans le catalogue
- Supprimer un élément du catalogue
- Choisir un pays à boycotter dans le catalogue

- Réinitialiser le panier pour qu'il soit vide
- Réinitialiser le catalogue avec une liste d'éléments par default
- Fait des modifications sur le catalogue afin qu'il présente :
    - Que des fruits à moins de 2 euros
    - Que des fruits locaux (de France)
    - Que des fruits (pas de Jus/Macédoine)

2. Besoins non fonctionnels

Les besoins non fonctionnels sont des exigences qui ne concerne pas spécifiquement le comportement du système mais plutôt identifiant des contraintes internes et externes du système

Les principaux besoins non fonctionnels de notre application se résume dans les points suivants :

- **Maintenance :** Le code source doit être bien documenté et structuré de manière à faciliter la maintenance continue de l'application par l'équipe de développement.
- **Tests et Qualité du Code :** Des tests unitaires, d'intégration et d'acceptation doivent être mis en place pour garantir la qualité du code et détecter les erreurs potentielles.
- **Évolutivité :** L'application doit être capable de s'adapter à une augmentation de la charge et de l'utilisation sans compromettre les performances.
- **Performance :** L'application doit répondre de manière rapide et efficace, en minimisant les temps de chargement et de réponse, même lorsqu'elle traite de grandes quantités de données.

III. Etude technique

Dans ce troisième chapitre nous présentons, en premier lieu, l'environnement de travail et nous justifions les choix du langage de programmation et la technologie de développement appliqué.

A. Outils et technologie de développement

1. LANGAGE JAVA

Java est un langage de programmation orienté objet créé par James Gosling et Patrick Naughton, employés de Sun Microsystems, avec le soutien de Bill Joy (cofondateur de Sun Microsystems en 1982), présenté officiellement le 23 mai 1995

Java est un langage de programmation orienté objet utilisable sur divers systèmes d'exploitation. Il est assez robuste, rapide, et fiable. Une particularité de Java est que les logiciels écrits dans ce langage sont compilés vers une représentation binaire intermédiaire qui peut être exécutée dans une machine virtuelle Java (JVM) en faisant abstraction du système d'exploitation.

2. Git

Git est un système de gestion de versions distribué largement utiliser dans le développement logiciel. Il permet de suivre les modifications apportées au code source d'un projet, de collaborer efficacement avec d'autres développeurs, et de gérer les différentes versions du code. Git offre un historique complet des changements, la possibilité de travailler sur des branches pour isoler des fonctionnalités ou des correctifs, et la capacité de fusionner ces branches en toute sécurité.

3. MVC (Modèle-Vue-Contrôleur)

Le modèle **MVC**, abréviation de Modèle-Vue-Contrôleur, est un modèle architectural de conception de logiciels qui vise à organiser de manière structurée le code des applications. Il offre une méthode claire et modulaire pour séparer les préoccupations, améliorer la maintenabilité, et faciliter l'évolution de l'application.

- **Modèle (Model) :** Le modèle représente la couche de données de l'application. Il gère la logique métier, les données et l'état interne. Cette composante est responsable de la manipulation des données et offre des méthodes pour accéder, mettre à jour et traiter ces données. Elle encapsule la structure sous-jacente des données.
- **Vue (View) :** La vue est la couche d'interface utilisateur de l'application. Elle est responsable de la présentation visuelle des données au(x) utilisateur(s). La vue affiche les informations du modèle de manière à ce qu'elles soient claires et compréhensibles. Elle peut également recueillir les entrées de l'utilisateur et les transmettre au contrôleur.
- **Contrôleur (Controller) :** Le contrôleur agit comme un intermédiaire entre le modèle et la vue. Il reçoit les entrées de l'utilisateur, traite ces entrées en fonction de la logique métier définie, met à jour le modèle en conséquence, et rafraîchit la vue pour refléter ces changements. Il gère également la navigation et la coordination des actions de l'application.

Design pattern utilisé

(1) Factory/Fabrique

Le patron de conception Factory, ou Fabrique, est un modèle de conception qui appartient à la catégorie des modèles de création. Il offre un moyen de créer des objets de manière centralisée, cachant ainsi la complexité de leur création aux clients. Le rôle principal de ce modèle est de séparer la logique de création d'objets du reste du code en fournissant une interface commune pour la création d'objets. Cette interface peut être mise en œuvre par des sous-classes ou des méthodes spécifiques. En résumé, le modèle Factory contribue à simplifier le code client en évitant aux clients d'avoir à connaître les détails de la création d'objets, tout en offrant une approche centralisée et cohérente pour instancier des objets.

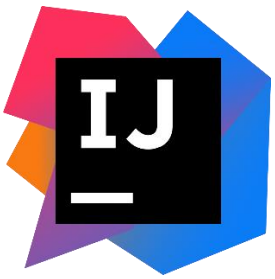
(2) Observable/Observer

Le modèle de conception Observable/Observer, également connu sous le nom de modèle Observateur, est un patron de conception comportemental. Il permet d'établir une relation de dépendance un-à-plusieurs entre objets, où un objet (l'observable) informe ses observateurs

des changements de son état, de manière automatique et décentralisée. Les observateurs, qui s'enregistrent pour être notifiés, réagissent aux changements de l'observable en effectuant des actions spécifiques. Ce modèle est couramment utilisé pour implémenter des mécanismes de notification, d'abonnement et de diffusion d'informations, améliorant ainsi la modularité et la flexibilité du code en permettant aux observateurs de réagir aux changements de l'observable sans dépendre étroitement de lui. En résumé, Observable/Observer facilite la mise en œuvre de systèmes de notification événementielle et de réaction aux changements d'état.

B. Environements de Développement

1. IntelliJ IDEA



IntelliJ IDEA est un environnement de développement intégré (IDE) très performant conçu par JetBrains. Il offre un large éventail d'outils pour la programmation, le débogage, la gestion de versions, et l'assistance intelligente à la rédaction de code. Grâce à ses fonctionnalités avancées, cet IDE est populaire auprès des développeurs Java et d'autres langages, facilitant le processus de développement logiciel en automatisant de nombreuses tâches fastidieuses

2. Neatbeans

NetBeans est un environnement de développement intégré (IDE) open source, polyvalent et extensible, largement utilisé pour la programmation en Java, C/C++, et d'autres langages. Il offre des fonctionnalités de développement, de débogage, et de gestion de projets, avec un écosystème de modules qui le rendent adaptable à divers besoins. NetBeans est apprécié pour sa facilité d'utilisation, sa compatibilité multiplateforme, et sa capacité à accélérer le processus de développement en fournissant des outils et des fonctionnalités pratiques.

C. Framework

1. SPRING

Spring est un Framework open source pour construire et définir l'infrastructure d'une application Java [11], dont il facilite le développement et les tests. En 2004, Rod Johnson a écrit le livre Expert One-on-One J2EE Design and Développent qui explique les raisons de la création de Spring

Spring IoC :

Spring Ioc étant un Framework très complet et complexe, il ne sera décrit en détail qu'une des multiples manières de faire de l'injection de dépendance avec Spring.

Le fonctionnement de Spring Ioc repose sur 3 éléments clés :

- La configuration XML pour lier les implémentations à leurs interfaces
- Les classes sont des *beans* et possèdent donc des getters et setters pour les champs à injecter
- L'injection en même est effectuée par un conteneur de Spring

2. Junit

JUnit est un framework de test unitaire pour Java (et d'autres langages) qui permet aux développeurs de créer des tests unitaires efficaces pour leurs applications. Il fournit un cadre structuré pour l'écriture, l'exécution et la vérification de tests, ce qui aide à garantir la qualité du code en identifiant rapidement les erreurs et les comportements indésirables. JUnit est largement utilisé dans le développement logiciel pour la pratique du test-driven développement (TDD) et l'intégration continue.

3. Mokito

Mockito est une bibliothèque Java open source permettant de créer des objets fictifs (mocks) pour faciliter les tests unitaires et l'isolation des dépendances. Elle offre des moyens simples et puissants pour simuler le comportement des composants externes d'une classe pendant les tests unitaires. Mockito est apprécié pour sa facilité d'utilisation et son intégration avec des frameworks de test comme JUnit. Il permet aux développeurs de s'assurer que chaque composant de leur code fonctionne correctement de manière indépendante, ce qui favorise la qualité et la robustesse des applications.

D. Plugins

1. Maven

Apache Maven est un outil de gestion de projets et de construction logicielle largement utilisé pour simplifier le cycle de vie du développement logiciel. Il automatise la compilation, le packaging, le déploiement, et la gestion des dépendances d'un projet, en se basant sur des fichiers de configuration XML appelés POM (Project Object Model). Maven facilite le processus de construction et de gestion de projets en standardisant les pratiques de développement, en rendant la gestion des dépendances plus transparente, et en offrant une structure modulaire et cohérente pour les projets. Les plugins Maven sont des extensions qui permettent d'ajouter des fonctionnalités spécifiques, étendant ainsi la capacité de Maven à gérer divers aspects du développement logiciel, comme la génération de rapports, le déploiement, ou l'intégration continue.

2. Jacoco

Le plugin JaCoCo (Java Code Coverage) est un outil open source conçu pour évaluer la couverture de code dans les applications Java. Intégré à des environnements de développement tels qu'IntelliJ IDEA, Eclipse et d'autres IDE populaires, JaCoCo offre des fonctionnalités de suivi de la couverture de code, permettant aux développeurs de déterminer quels éléments de leur code source ont été exécutés pendant les tests. En fournissant des rapports détaillés sur la couverture de code, JaCoCo aide les développeurs à identifier les zones non testées, à améliorer la qualité du code, et à garantir une meilleure robustesse des applications Java.

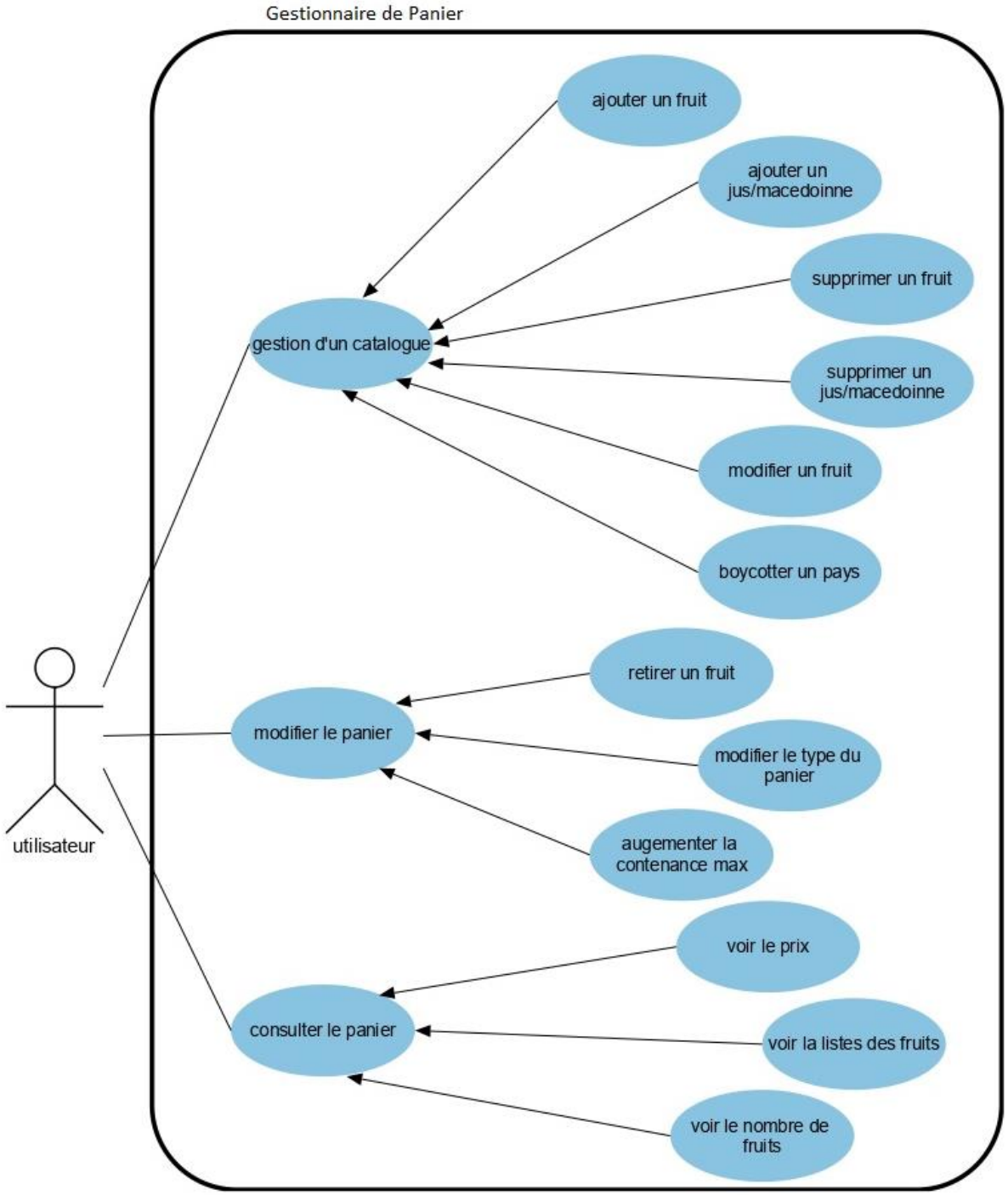
IV. Analyse et conception

Dans ce quatrième chapitre nous allons analyser les besoins que nous avons cités dans la partie précédente, et les modéliser par la suite.

A. Diagramme des cas d'utilisation

Le diagramme de cas d'utilisation est une représentation graphique qui permet de modéliser les interactions entre les acteurs (utilisateurs externes) et un système logiciel. Il sert à décrire les différents scénarios ou cas d'utilisation auxquels le système doit répondre. Chaque cas d'utilisation représente une fonctionnalité ou une action spécifique du système, généralement déclenchée par un acteur.

Donc pour cette partie nous avons utilisé le langage de modélisation UML [10] pour analyser les besoins que nous avons cité dans la partie précédente, et les modéliser par la suite. La modélisation d'un système nous permet de mieux le comprendre, du coup maitriser sa complexité pour assurer sa cohérence.


Gestionnaire de Panier
ajouter un fruit
ajouter un jus/macedoinne
supprimer un fruit
gestion d'un catalogue
supprimer un jus/macedoinne
modifier un fruit
boycotter un pays
retirer un fruit
modifier le panier
modifier le type du panier
augementer la contenance max
voir le prix
consulter le panier
voir la listes des fruits
voir le nombre de fruits
utilisateur

Le diagramme des cas d'utilisation revêt une importance cruciale dans notre projet, car il permet d'offrir une représentation claire et systématique des interactions prévues entre les utilisateurs et l'interface homme-machine (IHM) du système. Grâce à ce diagramme, nous pouvons capturer de manière exhaustive l'ensemble des fonctionnalités que nous proposons aux utilisateurs. Cela comprend des actions telles que l'ajout, la suppression, et la gestion des fruits, la consultation du contenu du panier, la gestion du catalogue de fruits, le boycotter des fruits en provenance de certains pays, l'ajout de jus et de mélanges de fruits, la suppression de fruits, de jus, et de mélanges de fruits, le changement du type de panier, ainsi que l'augmentation de la capacité maximale du panier.

Chaque cas d'utilisation présenté dans ce diagramme détaille des scénarios d'utilisation spécifiques, décrivant en profondeur comment les utilisateurs interagiront avec le système. Cette identification minutieuse des besoins spécifiques, à travers le diagramme des cas d'utilisation, nous a conduit à envisager des extensions pour répondre de manière encore plus complète aux exigences fonctionnelles. Ces extensions visent à enrichir l'expérience utilisateur en offrant davantage de fonctionnalités et de personnalisation, tout en garantissant que le système réponde pleinement aux attentes des utilisateurs.

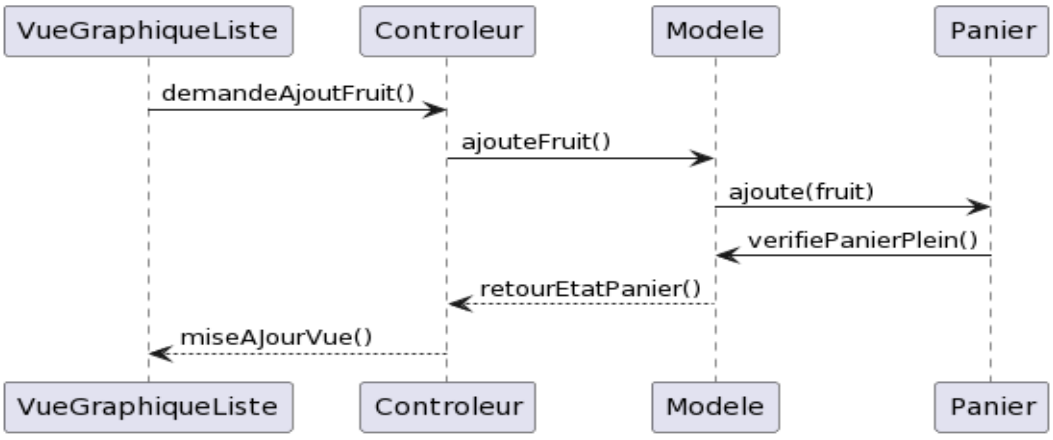
B. Diagrammes de séquence

Au sein de notre projet, l'utilisation de multiples diagrammes de séquence s'avère essentielle pour explorer minutieusement les interactions complexes entre les différents composants du système. Ces diagrammes fournissent une représentation chronologique des opérations et des échanges de messages entre les objets, se concentrant sur des aspects spécifiques de l'application. Cette approche détaillée nous permet d'analyser en profondeur les dépendances fonctionnelles, les synchronisations entre les composants et le cheminement des données, renforçant notre compréhension du comportement du système.

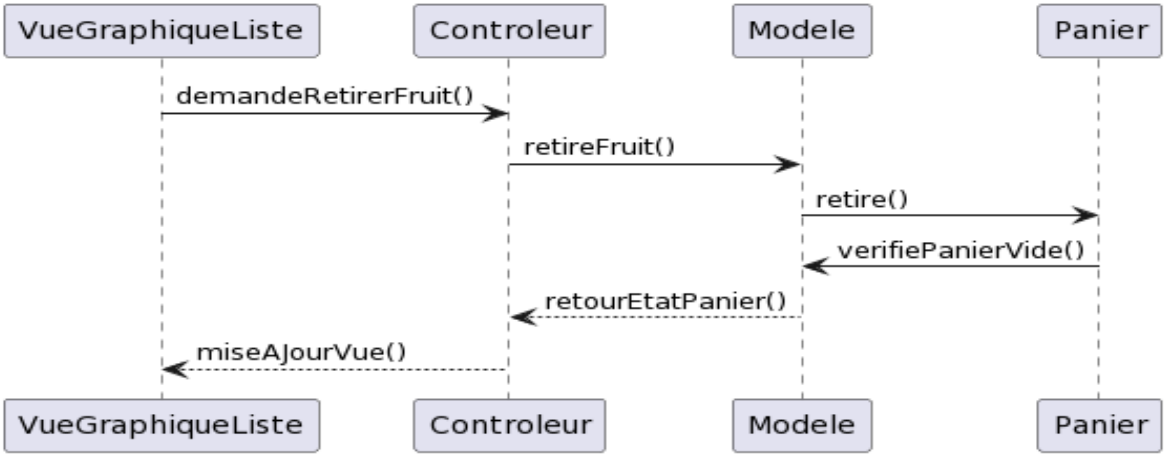
En parallèle, nous prévoyons de réaliser une variété de tests, notamment des tests unitaires, des tests de mock, des tests d'intégration et des tests d'acceptation. Ces tests seront adaptés en fonction des interactions et des dépendances mises en évidence dans nos diagrammes de séquence. Ils joueront un rôle fondamental dans la garantie de la qualité et de la fiabilité de notre solution, en identifiant et en résolvant les éventuels problèmes à chaque étape du développement.

Pour illustrer de manière approfondie la structure de la solution que nous proposons pour nos extensions, nous avons élaboré un diagramme de classes complet. Ce diagramme offre une vue détaillée de la hiérarchie des objets, des relations entre les différentes classes, ainsi que des dépendances au sein de notre système. Chaque classe est représentée avec ses attributs, ses méthodes et ses associations, fournissant une image complète de l'architecture logicielle. L'utilisation de ce diagramme de classes nous aide à mieux comprendre comment les composants de notre système interagissent et coopèrent pour mettre en œuvre les fonctionnalités étendues, tout en identifiant clairement les classes responsables des nouvelles fonctionnalités ou des modifications. Cette approche facilite grandement le processus de développement et renforce la cohérence de notre solution.

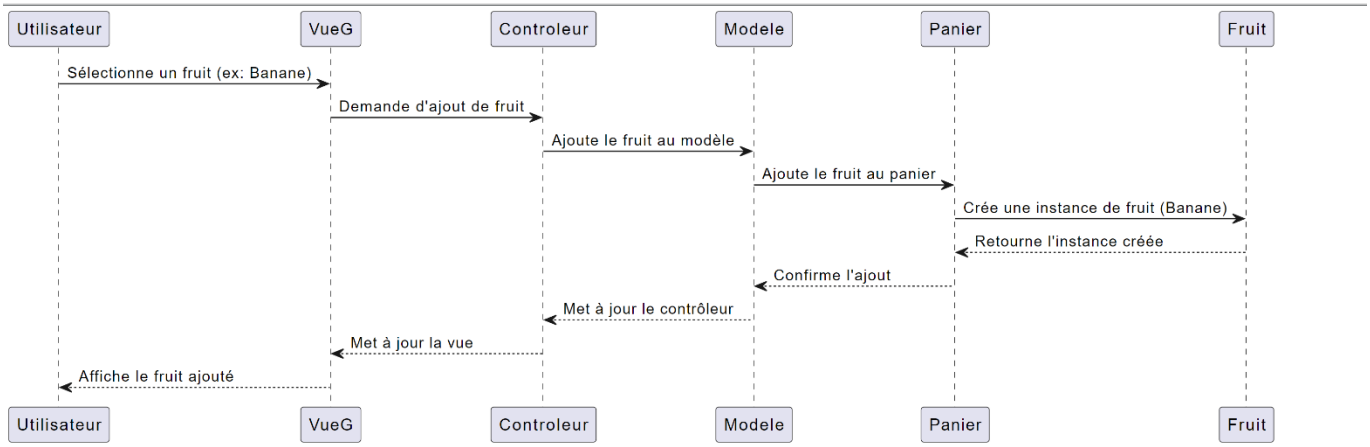
1. Diagramme de conception : ajout d'un fruit au panier



2. Diagramme de conception : retrait d'un fruit du panier

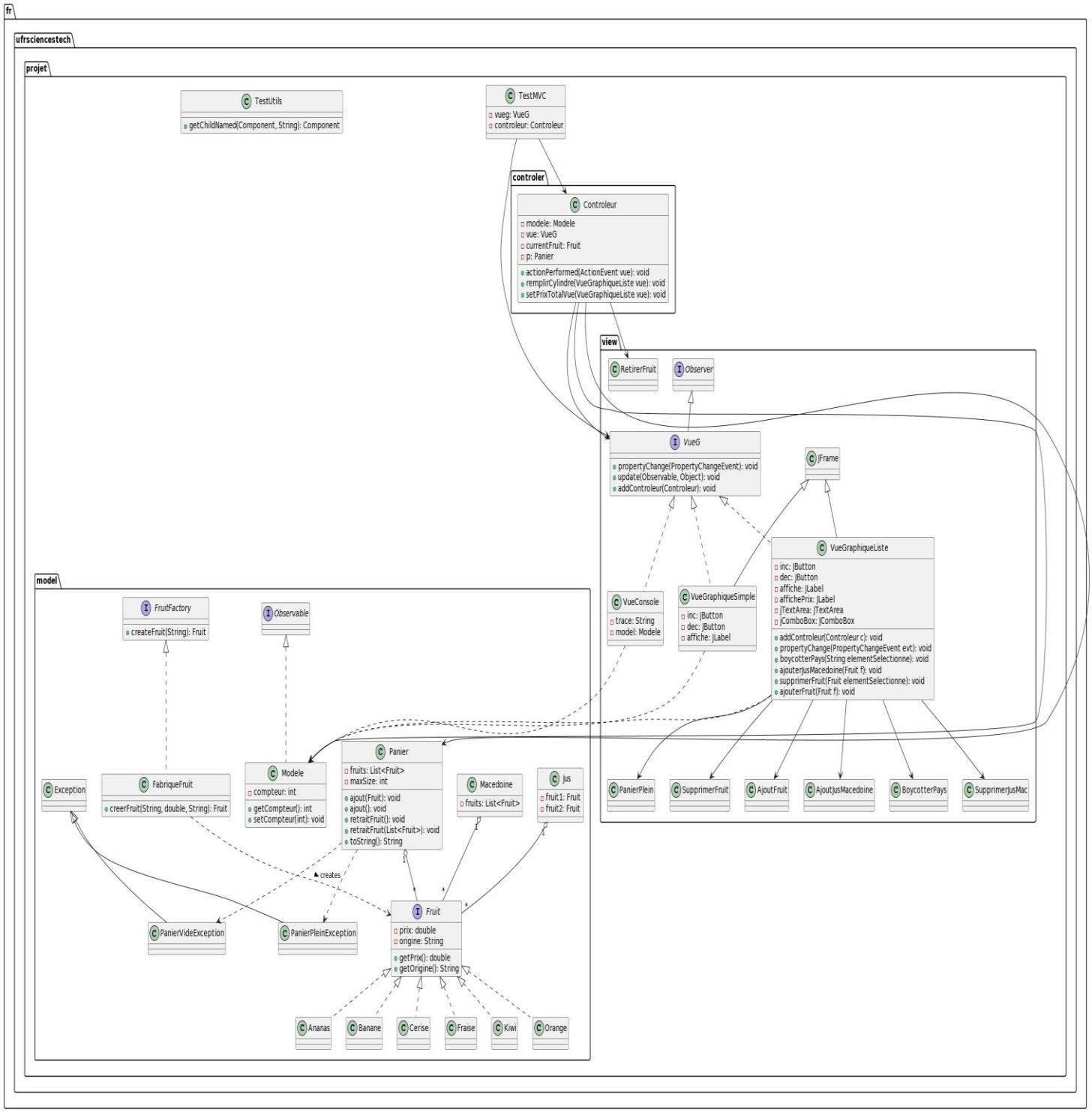


3. Diagramme de conception : ajout d'un nouveau fruit



C. Diagramme des classes

Le diagramme de classes permet de visualiser clairement la structure de la solution proposée pour les extensions. En incorporant des design patterns tels que le modèle de conception de la fabrique, notre conception devient plus modulaire et évolutive. Les fabriques permettent de créer des objets de manière dynamique en fonction des besoins, favorisant ainsi la flexibilité du système. De plus, nous avons appliqué des règles de conception rigoureuses pour assurer la qualité et la maintenabilité des extensions. Cela inclut la cohérence dans l'organisation des classes, la minimisation des dépendances, l'encapsulation appropriée des fonctionnalités et le respect des principes de conception orientée objet tels que le principe de responsabilité unique. Ces éléments sont essentiels pour garantir que les extensions s'intègrent harmonieusement dans l'IHM existante et sont prêtes à évoluer en réponse aux besoins changeants de l'utilisateur.


fr
ufrsciencestech
projet
TestUtils
getChildNamed(Component, String): Component
TestMVC
vueg: VueG
controleur: Controleur
controler
Controleur
modele: Modele
vue: VueG
currentFruit: Fruit
p: Panier
actionPerformed(ActionEvent vue): void
remplirCylindre(VueGraphiqueListe vue): void
setPrixTotalVue(VueGraphiqueListe vue): void
view
RetirerFruit
Observer
VueG
propertyChange(PropertyChangeEvent): void
update(Observable, Object): void
addControleur(Controleur): void
JFrame
VueGraphiqueListe
inc: JButton
dec: JButton
affiche: JLabel
affichePrix: JLabel
jTextArea: JTextArea
jComboBox: JComboBox
addControleur(Controleur c): void
propertyChange(PropertyChangeEvent evt): void
boycotterPays(String elementSelectionne): void
ajouterJusMacedoine(Fruit f): void
supprimerFruit(Fruit elementSelectionne): void
ajouterFruit(Fruit f): void
VueConsole
trace: String
model: Modele
VueGraphiqueSimple
inc: JButton
dec: JButton
affiche: JLabel
PanierPlein
SupprimerFruit
AjoutFruit
AjoutJusMacedoine
BoycotterPays
SupprimerJusMac
model
FruitFactory
createFruit(String): Fruit
Observable
Exception
FabriqueFruit
creerFruit(String, double, String): Fruit
Modele
compteur: int
getCompteur(): int
setCompteur(int): void
Panier
fruits: List<Fruit>
maxSize: int
ajout(Fruit): void
ajout(): void
retraitFruit(): void
retraitFruit(List<Fruit>): void
toString(): String
Macedoine
fruits: List<Fruit>
Jus
fruit1: Fruit
fruit2: Fruit
creates
Fruit
prix: double
origine: String
getPrix(): double
getOrigine(): String
PanierVideException
PanierPleinException
Ananas
Banane
Cerise
Fraise
Kiwi
Orange

V. Réalisation

Ce chapitre est consacré à la description de la phase de mise en œuvre de l'application. Nous y décrivons l'architecture de notre application et nous illustrons certaines fonctionnalités assurées à travers les interfaces.

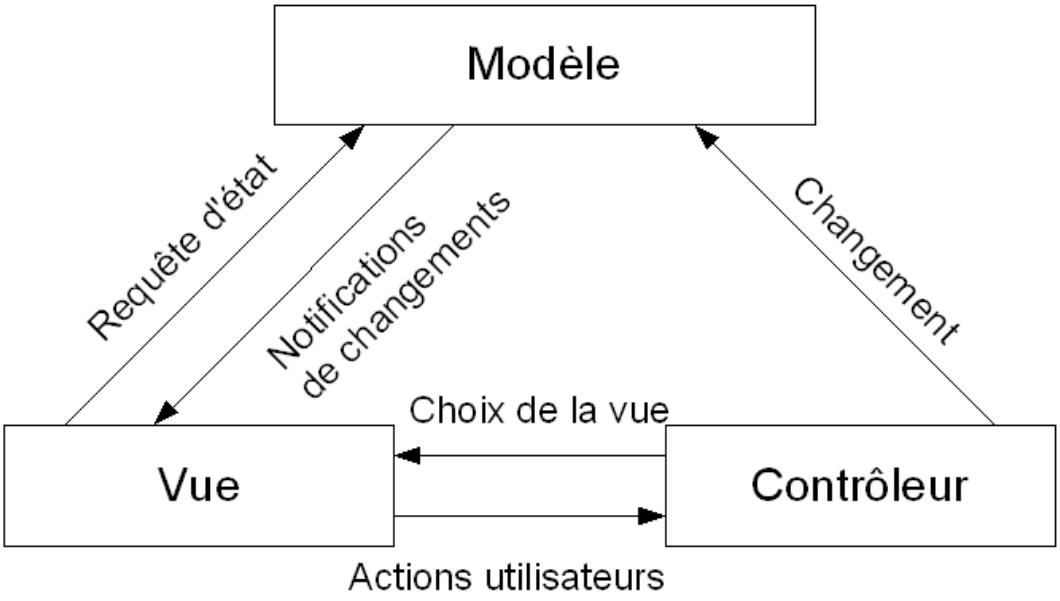
A. Architecture générale de l 'application

1. Couche Controler

La classe contrôleur permet de gérer les interactions de l'utilisateur avec l'interface graphique, telles que l'ajout et la suppression de fruits du panier, la réinitialisation du panier, la mise à jour de la vue, et la gestion du modèle.

On vous présente une description des principaux éléments et méthodes du code :

La classe "Controleur" agit en tant que contrôleur dans le modèle MVC. Elle comprend des variables d'instance pour le modèle, la vue, le fruit actuellement sélectionné, et le panier de fruits. Le constructeur initialise le contrôleur avec le modèle et la vue, et configure la vue en tant qu'auditeur des changements de propriété du modèle. La méthode "actionPerformed" gère les actions de l'utilisateur, telles que la réinitialisation du panier, l'ajout ou la suppression de fruits, et la mise à jour de la vue. Des méthodes utilitaires sont fournies pour définir et obtenir le modèle, la vue, le panier, le fruit actuel, et pour rechercher des composants spécifiques dans la vue. Les méthodes "remplirCylindre" et "setPrixTotal" sont utilisées pour mettre à jour la barre de progression du panier et afficher le prix total dans l'interface graphique. Enfin, la méthode "afficherErreurPanierPlein" affiche un message d'erreur si le panier est plein. Dans l'ensemble, ce code illustre la séparation des préoccupations et la gestion des interactions utilisateur dans une application de panier de fruits en suivant le modèle MVC.


Modèle
Requête d'état
Notifications de changements
Changement
Choix de la vue
Vue
Contrôleur
Actions utilisateurs

2. Couche model

Notre projet est structuré autour de plusieurs classes bien organisées. Tout d'abord, la classe abstraite ‘Fruit’ définit la structure de base pour tous les types de fruits, offrant des méthodes communes pour obtenir le prix, l'origine et vérifier la présence de pépins. Cela simplifie la gestion des différents types de fruits en utilisant l'héritage.

Pour créer des instances de fruits, on a la classe ‘FabriqueFruit’, qui implémente le modèle de conception Factory. Cette classe crée des objets fruitiers à partir d'un nom de fruit, d'un prix et d'une origine, garantissant une création cohérente et simplifiée des fruits.

L'interface 'FruitFactory' déclare la méthode 'creerFruit', offrant une structure commune pour la création de fruits. Cela facilite l'ajout de nouveaux types de fruits en conservant une méthode de création uniforme.

Les classes 'Jus' et 'Macedoine' représentent respectivement un jus de fruits et une macédoine. Ils peuvent contenir un mélange de fruits différents et calculent leur prix total en fonction des fruits qu'ils contiennent. Ces classes offrent également des méthodes pour ajouter, retirer et obtenir la liste des fruits qu'elles contiennent.

Le modèle de conception MVC est mis en œuvre dans la classe ‘Modele’, qui maintient un compteur utilisé pour suivre diverses opérations. Elle est observée par les vues pour maintenir l'état global de l'application.

La classe centrale du projet est le ‘Panier’ , qui représente un panier de fruits. Il peut être rempli, vidé et son prix total peut être calculé. La classe utilise un mécanisme de notification pour informer les vues de tout changement d'état. De plus, des exceptions personnalisées, ‘PanierPleinException’ et ‘PanierVideException’ , sont utilisées pour gérer les erreurs lorsque le panier est plein ou vide.

En résumé, ces classes forment une architecture solide pour notre projet. Elles simplifient la création, la manipulation et le suivi des fruits dans un panier, tout en gérant les exceptions qui peuvent survenir lors de ces opérations. Le modèle de conception Factory facilite l'ajout de nouveaux types de fruits, garantissant la cohérence et la maintenabilité de votre application.

```java
public interface Fruit {
    /**
     * Predicat indiquant si le fruit a ou non des pepins
     * @return True si le fruits n'a pas de pepin sinon False
     */
    public boolean isSeedless();

    /**
     * Methode pour definir le prix unitaire du fruit (en euros)
     * @param prix Le prix du fruit
     */
    public void setPrix(double prix);

    /**
     * Methode pour recuperer le prix unitaire du fruit (en euros)
     * @return Le prix du fruit
     */
    public double getPrix();

    /**
     * Methode pour definir l'origine d'un fruit
     * @param origine L'origine du fruit
     */
    public void setOrigine(String origine);

    /**
     * Predicat pour recuperer le pays d'origine du fruit
     * @return L'origine du fruit
     */
    public String getOrigine();

    /**
     * Predicat pour tester si 2 fruits sont equivalents
     * @return true s'il sont equivalent et false sinon
     */
    @Override
    public boolean equals(Object o);

    /**
     * Methode d'affichage d'un fruit
     * @return Le fruit et ses attributs origine et prix
     */
    @Override
    public String toString();
}
```

a) La classe 'Fruit' est l'interface commune implémentée par tous les types de fruits

```java
public class Modele extends Observable{
    private int compteur;   //compteur toujours positif
    PropertyChangeSupport support;

    /**
     * Constructeur Modele
     * Initialisation des attributs de Modele : le compteur de fruit et le PropertyChangeSupport
     */
    public Modele(){
        support = new  PropertyChangeSupport(this);
        compteur = 0;
    }

    /**
     * Methode update
     * Elle permet la communication entre les objets observés et les observateurs
     * Met a jour le compteur et reveille les observateurs avec la valeur du compteur
     */
    public void update(int incr) {
        if(incr==2){
            this.compteur=0;
        }else{
            int old = this.compteur;
            compteur += incr;
            if(compteur<0){
                compteur=0;
            }
            System.out.println("compteur = "+compteur);
            support.firePropertyChange("value",old,this.compteur);
            setChanged();
            notifyObservers(getCompteur());
        }
    }
    /**
     * Methode
     * @return Retourne le compteur
     */
    public int getCompteur() {
        return compteur;
    }

    /**
     * Methode addPropertyChangeListener(PropertyChangeListener listener)
     * Methode servant à ajouter un listener
     */
    public void addPropertyChangeListener(PropertyChangeListener listener) {
        support.addPropertyChangeListener(listener);
    }
```

b) Le modèle 'Model' est utilisé pour la gestion de la comptabilisation des actions effectuées sur le panier. Il étend Observable pour permettre aux observateurs de suivre les changements dans le modèle.

```java
    /**
     * Methode removePropertyChangeListener(PropertyChangeListener listener)
     * Methode servant à retirer un listener
     */
    public void removePropertyChangeListener(PropertyChangeListener listener) {
        support.removePropertyChangeListener(listener);
    }

    /**
     * Methode setCompteur(int newCounter)
     * Methode servant à remplacer la valeur de l'attribut compteur par une autre en parametre
     * Elle met a jour la nouvelle valeur du compteur et reveille les observateurs avec cette valeur
     */
    public void setCompteur(int newCounter) {
        int old = this.compteur;
        this.compteur = newCounter;
        if(this.compteur<0){
            this.compteur=0;
        }
        support.firePropertyChange("value", old, newCounter);
        setChanged();
        notifyObservers(getCompteur());
    }
}
```

3. Couche view

Les classes qu'on va présenter illustrent la mise en œuvre du modèle de conception MVC (Modèle-Vue-Contrôleur) dans notre projet. Elles démontrent comment organiser et séparer la logique de l'application en trois composants distincts : le modèle, la vue et le contrôleur.

Tout d'abord, la classe 'VueG' joue un rôle clé en tant qu'interface définissant les méthodes requises pour gérer les vues graphiques. Elle hérite des interfaces 'PropertyChangeListener' et 'Observer', permettant ainsi à d'autres classes d'implémenter ces fonctionnalités pour être informées des changements dans le modèle ou le contrôleur. De plus, elle comprend des méthodes pour réagir aux modifications d'attributs du modèle et ajouter un contrôleur pour gérer les interactions utilisateur.

La classe 'VueGraphiqueListe' représente l'interface graphique d'une application de gestion de panier. Elle hérite de 'JFrame' pour créer une fenêtre graphique avec divers composants tels que des boutons, des menus, des barres de progression et une zone de texte. L'objectif principal est de permettre à l'utilisateur d'ajouter, retirer et interagir avec des éléments dans un panier. Elle utilise des menus pour gérer des actions telles que l'ajout de fruits, le retrait de fruits, la réinitialisation du panier et le boycott de pays. Cette classe implémente l'interface 'VueG' et inclut des méthodes pour mettre à jour l'interface en fonction des modifications du modèle, ajouter un contrôleur pour gérer les interactions utilisateur, et appliquer des options spécifiques pour filtrer les fruits dans le catalogue.

La classe 'VueGraphiqueSimple' représente une interface graphique plus élémentaire qui affiche un compteur. Elle propose deux boutons pour incrémenter et décrémenter le compteur, ainsi qu'un label pour afficher sa valeur. Cette classe suit également le modèle de conception MVC, avec un rôle de vue. Elle implémente l'interface 'VueG' pour permettre l'association d'un contrôleur, mettre à jour l'affichage en fonction des modifications du modèle et réagir aux événements de l'utilisateur, tels que les clics sur les boutons.

Les classes suivantes sont des composants essentiels dans notre projet. Elles offrent des fonctionnalités diverses pour interagir avec le catalogue de fruits, les paniers et les actions de l'utilisateur.

La classe ‘AjoutJusMacedoine’ est une boîte de dialogue permettant à l'utilisateur de créer et d'ajouter des macédoines ou des jus dans le catalogue principal de l'IHM. L'utilisateur peut choisir entre les deux options et sélectionner les fruits à inclure dans sa préparation. Cette classe simplifie la création de ces éléments pour l'utilisateur et les ajoute ensuite au catalogue.

La classe ‘BoycotterPays’, permet à l'utilisateur de boycotter un pays d'origine spécifique en fonction de la sélection dans une liste déroulante. Les origines des fruits sont obtenues depuis l'IHM principale, et lorsqu'un pays est choisi, la liste des fruits d'origine correspondante est mise à jour. 'L'utilisateur peut ainsi gérer son boycott de pays.

La classe ‘RetirerFruit’ est une boîte de dialogue qui permet de retirer des fruits d'un panier. Les fruits à retirer sont sélectionnés dans une liste, et l'opération peut être annulée. Le modèle est mis à jour en conséquence, et une exception est gérée si le panier est plein après le retrait.

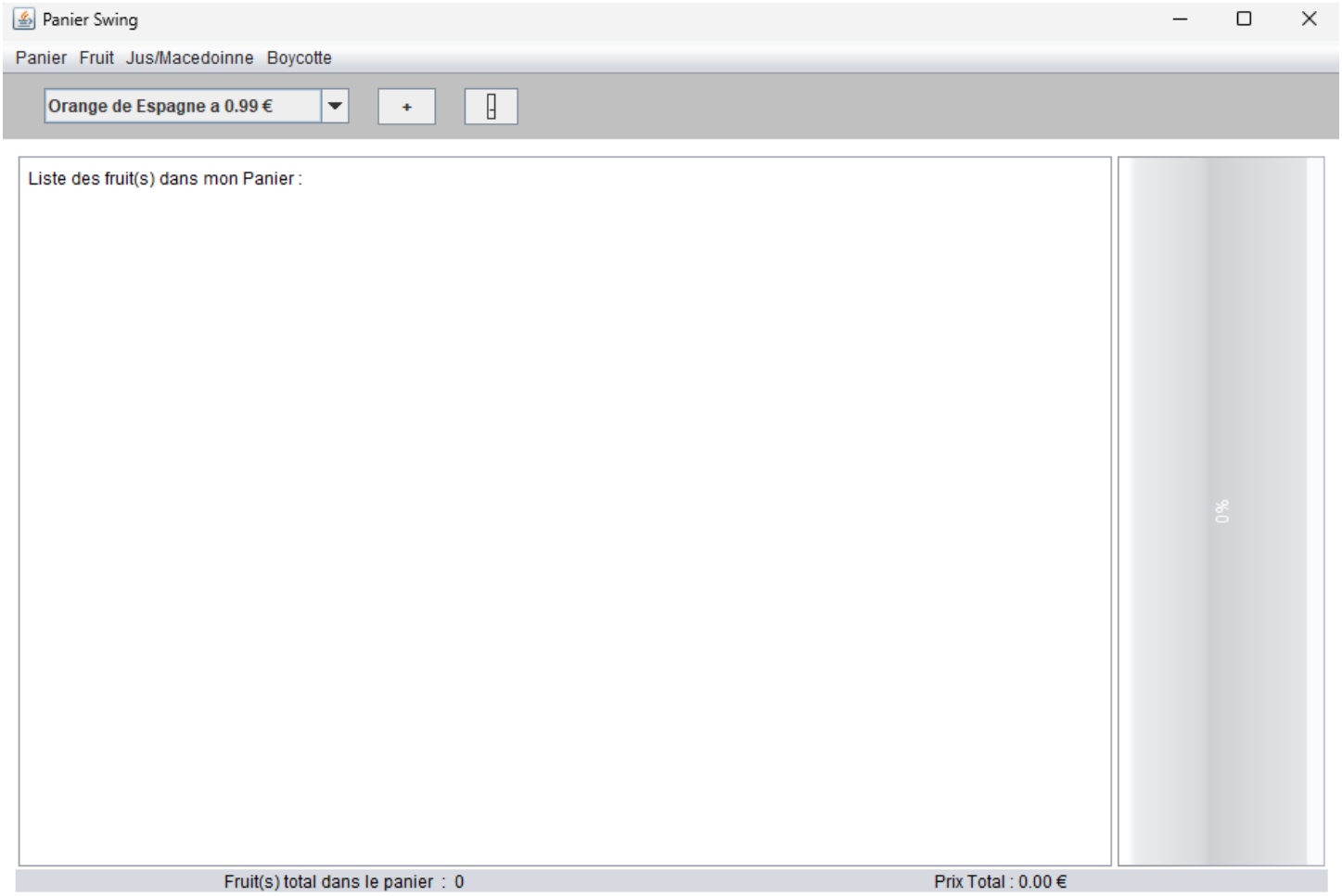
La classe ‘SupprimerFruit’ est une autre boîte de dialogue qui offre la possibilité de supprimer un fruit spécifique de la liste des fruits disponibles. Cette classe simplifie la gestion du catalogue en permettant à l'utilisateur de sélectionner le fruit à supprimer à partir d'une liste déroulante.

La classe ‘SupprimerJusMac’ permet de supprimer des éléments de type "Jus" ou "Macedoine" du catalogue. Les éléments correspondants sont obtenus depuis l'IHM principale, et l'utilisateur peut les retirer de la liste.

La classe ‘AjoutFruit’ est probablement une boîte de dialogue qui permet à l'utilisateur d'ajouter de nouveaux fruits au catalogue. Elle peut contenir des champs où l'utilisateur saisit le nom du fruit, son origine et d'autres informations pertinentes. Une fois que l'utilisateur a rempli ces informations, il peut appuyer sur le bouton "Ajouter" pour inclure ce nouveau fruit dans la liste des fruits disponibles. Cette classe simplifie le processus d'ajout de fruits au catalogue de l'application.

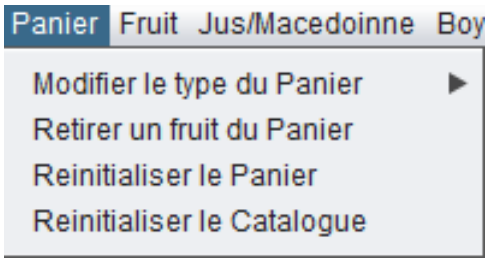
La classe ‘PanierPlein’ est une fenêtre d'erreur qui s'affiche lorsque le panier est plein, c'est-à-dire que le nombre maximum d'articles autorisés dans le panier a été atteint. Lorsque cet événement se produit, l'application peut afficher cette fenêtre d'erreur pour informer l'utilisateur que le panier ne peut pas contenir plus de fruits. Un bouton "Fermer" est généralement inclus pour que l'utilisateur puisse fermer cette fenêtre d'erreur et revenir à l'application principale.

(i) vue générale



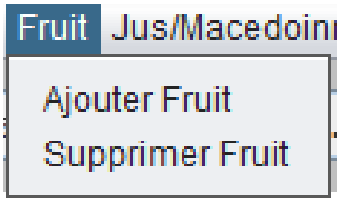
a) Boite Panier

Avec les options : Modifier/Retirer un fruit/ Réinitialiser



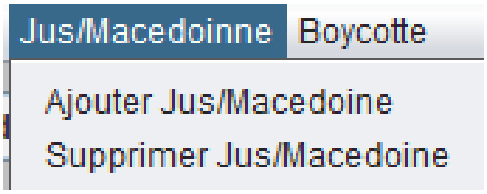
b) Boite Fruit

Avec les options Ajout/Supprimer Fruit



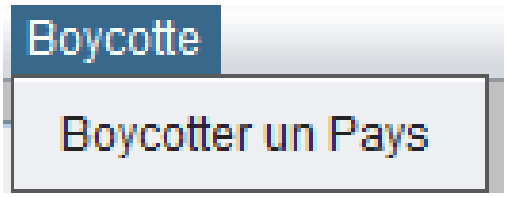
b) Boite Jus/Macédoine

Contient les options Ajout /Supprimer Jus/Macédoine



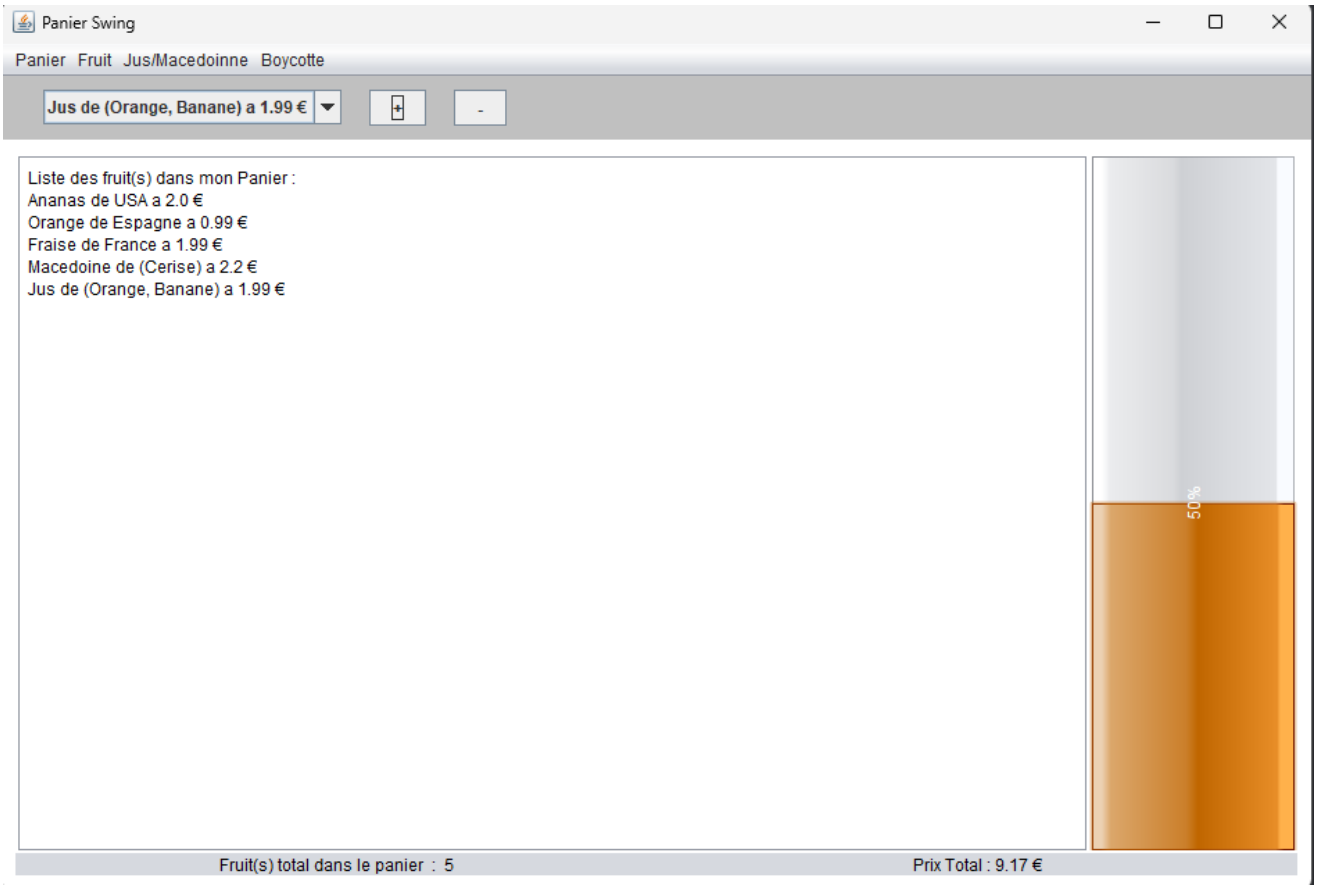
c) Boîte Boycotte

Pour choisir un pays à boycotter



d) Exemple de remplissage du panier

En remplissant progressivement le panier, un cylindre dynamique se remplit.



4. Test Unitaire

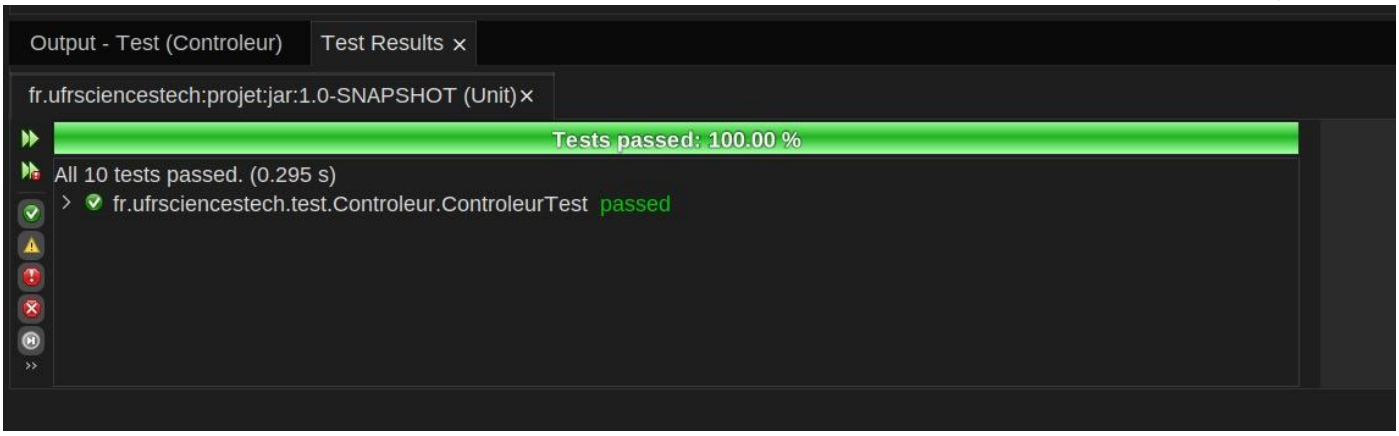
a) Outils et Méthodologies

Pour réaliser ces tests, nous avons utilisé le framework de tests unitaires JUnit et la bibliothèque de simulation Mockito. JUnit nous a permis de structurer nos cas de test et d'effectuer des assertions sur le comportement des méthodes. Mockito a été utilisé pour simuler les dépendances, permettant ainsi de tester les classes de manière isolée.

b) Résumé des Tests Effectués

- Résultats des Tests du Contrôleur

- **testAjoutFruitAuPanier :** Vérifie que les fruits sont ajoutés au panier lorsquel'utilisateur clique sur le bouton d'ajout.
- **testRetraitFruitDuPanier :** Confirme que les fruits peuvent être retirés du panier.
- **testChangementFruitCourant :** Assure que le fruit sélectionné est correctement misà jour dans le contrôleur.
- **testSetPanier :** Vérifie que le panier peut être mis à jour dans le contrôleur.
- **testSetCurrentFruit :** Teste la mise à jour du fruit courant dans le contrôleur.
- **testSetModele :** Confirme que le modèle peut être mis à jour dans le contrôleur.
- **testSetVue :** Vérifie que la vue peut être mise à jour dans le contrôleur.
- **testRemplirCylindre :** Assure que la barre de progression est correctement mise àjour pour refléter l'état du panier.
- **testSetPrixTotal :** Vérifie que le prix total affiché est correct après l'ajout d'un fruit.
- **testAfficherErreurPanierPlein** : Confirme que l'erreur est affichée lorsque le panierest plein.



- ## Tests des Classes de Fruits (Modèle)

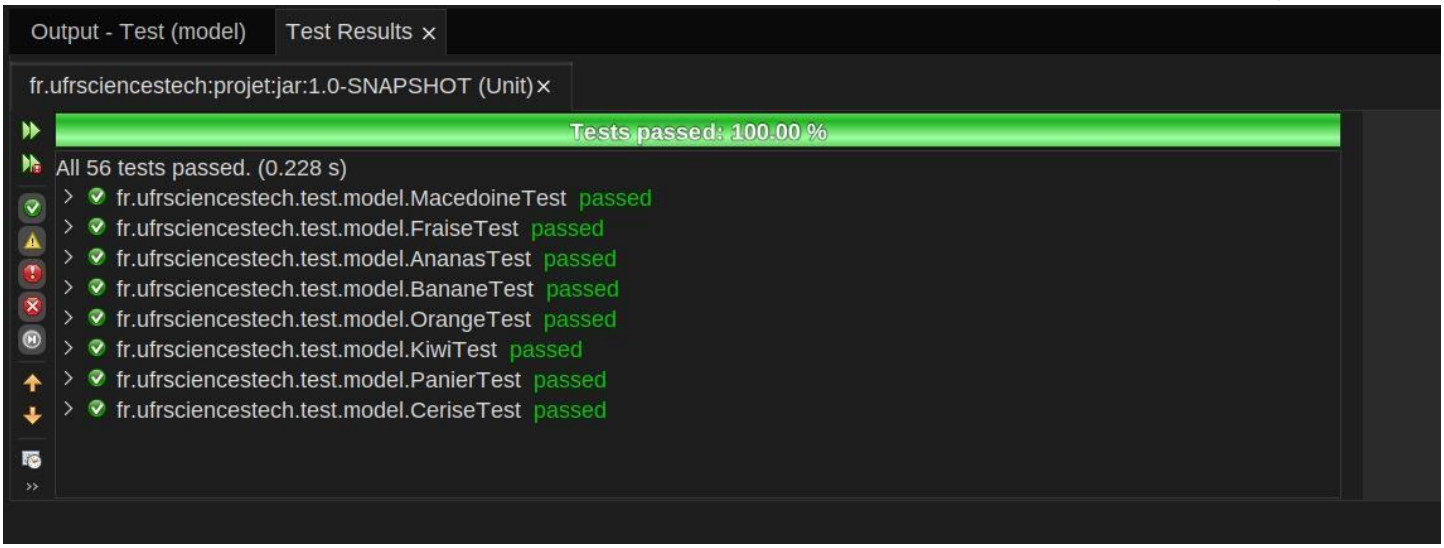
- ## Introduction aux Tests des Classes de Fruits

Les classes de fruits représentent la partie "Modèle" de notre architecture MVC et sont cruciales pour la gestion des données de l'application. Chaque classe de fruit aété testée pour s'assurer de la cohérence des données et de la logique métier.

- ## Résultats des Tests des Classes de Fruits

Les tests ont été réalisés sur les classes Ananas, Banane, Fraise, Cerise, Kiwi, et Orange. Chaque classe a été soumise à des tests similaires pour vérifier :

- La création correcte d'instances avec et sans paramètres.
- La gestion des valeurs négatives pour les prix.
- La correction des origines vides ou incorrectes.
- La représentation textuelle correcte de l'objet fruit (toString).
- L'égalité entre les instances de fruits basée sur leurs attributs
- Les caractéristiques spécifiques à chaque fruit, telles que l'absence de pépins.



- ## Tests de la Classe Macedoine

La classe Macedoine représente une salade de fruits dans notre application. Les tests suivants ont été conçus pour s'assurer que la création et la manipulation des objets Macedoine se comportent comme attendu.

- ## Résultats des Tests de la Classe Macedoine

- **testCreationMacedoineVide :** Confirme qu'une nouvelle Macédoine sans fruits à un prix de 0.0 et ne contient aucun fruit.
- **testCreationMacedoineAvecUnFruit :** Vérifie qu'une Macédoine créée avec un seulfruit contient bien ce fruit et reflète son prix.
- **testCreationMacedoineAvecDeuxFruits :** Assure que l'ajout de deux fruits à une Macedoine donne un prix total correct et que les deux fruits sont présents.
- **testAjoutFruitAMacedoine :** Teste l'ajout d'un fruit à une Macédoine existante,vérifiant le prix et la présence du fruit.
- **testMacedoineToString :** Vérifie que la méthode toString retourne la représentationtextuelle attendue d'une Macédoine.
- **testMacedoineEquals :** Confirme que deux Macédoine sont considérées égales si ellescontiennent les mêmes fruits.
- **testMacedoineIsSeedless :** Teste que la Macédoine est considérée sans pépins,conformément à la spécification.

- ## Test de la Classe Panier

La classe Panier est essentiel pour gérer la collection de fruits que l'utilisateurpeut assembler. Les tests suivants garantissent que le panier fonctionne commeprévu.

- ## Résultats des Tests de la Classe Panier

- **testCreationPanierVide :** Vérifie qu'un nouveau panier est vide et n'est ni plein nivide selon les indicateurs d'état.
- **testPrixTotalDuPanier :** Vérifie que le prix total du panier est la somme des prix desfruits contenus.
- **testBoycottOrigine :** Teste la capacité du panier à retirer tous les fruits d'une originespécifique.
- **testEqualsPanier** : Confirme que deux paniers sont égaux s'ils contiennent les mêmesfruits.
- **testPanierPlein :** Vérifie que le panier est considéré comme plein lorsqu'il atteint sacontenance maximale.

- ## Classe AjoutJusMacedoine

Cette classe, responsable de la création de jus et de macédoines de fruits, a été testée pour s'assurer que les sélections multiples étaient gérées correctement et que les objets créés étaient ajoutés à la liste parente.

- Cas de Test :

Sélection et ajout d'un mélange de fruits.
Validation de la fermeture de la fenêtre après l'ajout.

- **Classe BoycotterPays**

Les tests sur cette classe visaient à confirmer que le boycotte d'un pays entraînait la suppression des fruits correspondants de la liste.

- Cas de Test :

Sélection d'un pays et application du boycotte. Vérification de la mise à jour de la liste des fruits.

- **Classe SupprimerFruit**

Cette classe permet de supprimer des fruits, des jus ou des macédoines. Les tests sont concentrés sur la suppression correcte des éléments sélectionnés.

- Cas de Test :

Suppression d'un élément sélectionné. Annulation de la suppression.

- Classe PanierPlein

Les tests ont validé que la fenêtre d'erreur s'affichait correctement avec le message approprié lorsque le panier était plein.

- Cas de Test :

Affichage et fermeture de la fenêtre d'erreur.

- Classe RetirerFruit

Cette classe a été testée pour s'assurer que les fruits pouvaient être retirés du panier et que la liste était mise à jour en conséquence.

- Cas de Test :

Retrait de fruits sélectionnés. Annulation du retrait.

- Classe SupprimerJusMac

Les tests ont vérifié que seuls les jus et les macédoines pouvaient être supprimés et que les éléments non valides étaient gérés correctement.

- Cas de Test :

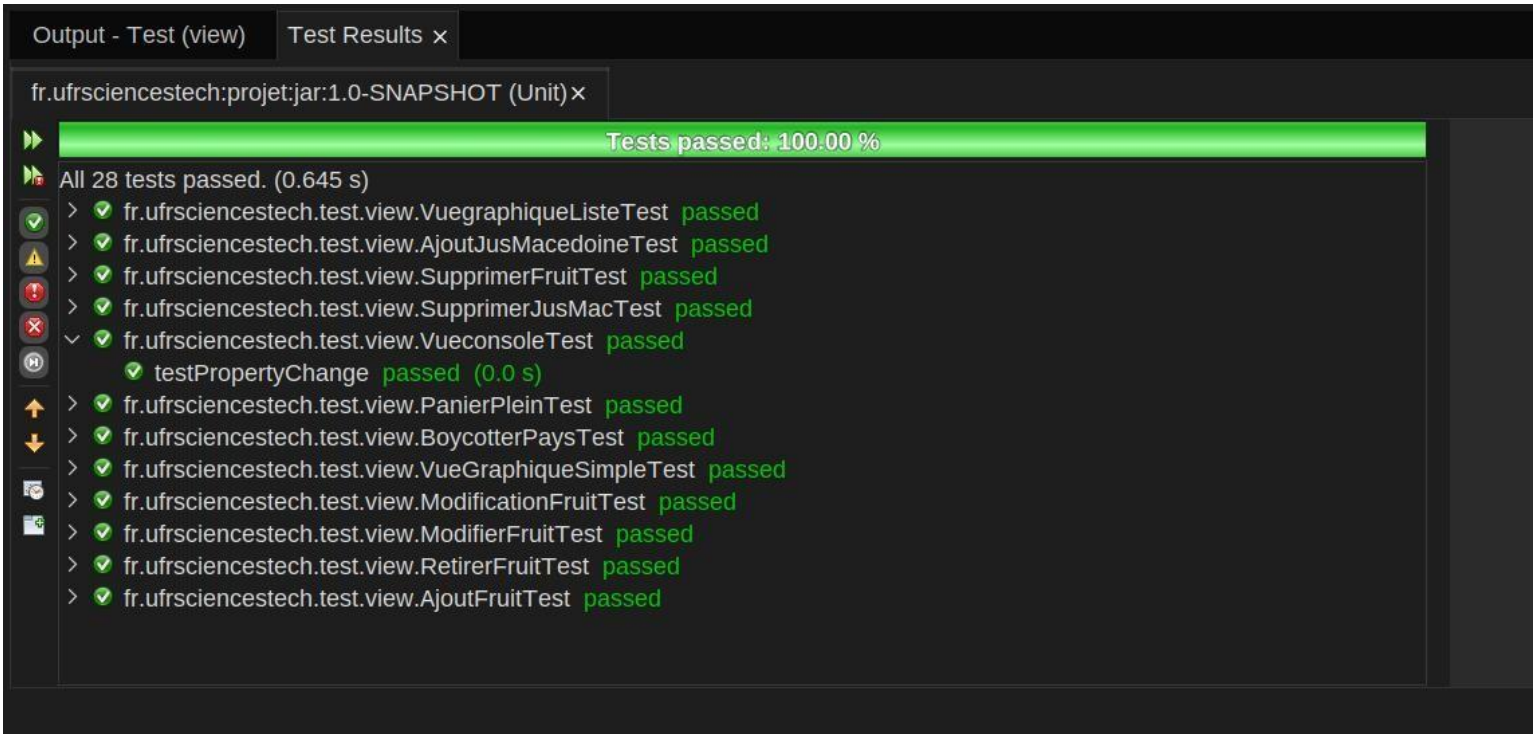
Suppression de jus et de macédoines. Gestion des sélections non valides.

- Classe ModifierFruit et ModificationduFruit

Ces classes permettent de modifier les attributs d'un fruit. Les tests ont inclus la modification des valeurs et la validation de la mise à jour des données.

- Cas de Test :

Modification des attributs d'un fruit. Annulation de la modification.



5. SCRUM

a) Introduction SCRUM

Ce projet a été réalisé en suivant le cadre de travail SCRUM, une méthodologie agile de gestion de projet. Scrum est un cadre de développement agile beaucoup utilisé dans le domaine du développement logiciel et aussi de manière plus générale dans le domaine de l'informatique en entreprise.

Scrum se concentre sur la collaboration, la flexibilité et la livraison continue de produits de qualité. Il divise le processus de développement en cycles itératifs appelés "itérations" ou "sprints", d'une durée de deux à quatre semaines.

- Chaque sprint commence par une réunion de planification appeler Sprint planning meeting, au cours de laquelle l'équipe Scrum définit les objectifs à atteindre pendant le sprint.
  Pendant le sprint, l'équipe travaille sur les **fonctionnalités prioritaires** du produit, généralement définies dans un document appelé backlog de produit.

- L'équipe se réunit tous les jours lors d'une réunion quotidienne de courte durée appelée Daily scrum meeting pour discuter de l'avancement, des obstacles éventuels et des plans pour la journée.

- À la fin de chaque sprint, une démonstration est organisée pour présenter les fonctionnalités terminées aux parties prenantes. Ensuite, l'équipe participe à une réunion de rétrospective où elle évalue son propre travail, la durée qu'il estime être adéquate pour compléter son travail et identifie des moyens d'améliorer le processus pour les sprints futurs.

L'une des caractéristiques clés de Scrum est son approche **collaborative**. L'équipe Scrum est autonome et responsable de la planification, de l'exécution et de l'amélioration continue du processus de développement.

b) Notre cas de travail

Pour notre projet, nous avons décidé de suivre au maximum la méthode de travail SCRUM. Cependant, en raison du délai de rendu très court et du fait que tous les membres du groupe n'avaient pas les mêmes disponibilités, certains étant des étudiants salariés et d'autres ayant d'autres engagements, nous avons dû adapter la méthode SCRUM afin de la rendre flexible pour tous les membres de l'équipe sans pour autant compromettre son efficacité.

Pour pallier ce défi, nous avons mis en place des ajustements dans notre approche SCRUM. Par exemple, au lieu de tenir des réunions quotidiennes physiques, nous avons opté pour des mises à jour régulières via des outils de communication en ligne qu'un serveur Discord. Cela a permis aux membres de l'équipe de partager leurs progrès, leurs obstacles et leurs plans pour la journée, même en dehors des heures de travail habituelle.

De plus, au lieu de planifier des sprints stricts de deux semaines, nous avons adopté une approche plus souple en décomposant les tâches en éléments plus petits et en les répartissant selon la disponibilité de chaque membre et en réduisant la durée d'un sprint de 2 semaine a 1 semaine.
Certains membres étaient plus disponibles en semaine, tandis que d'autres avaient plus de temps libre en week-end. Nous avons donc organisé nos tâches en conséquence, en veillant à ce que chacun puisse contribuer de manière significative, même avec des plages horaires limitées.

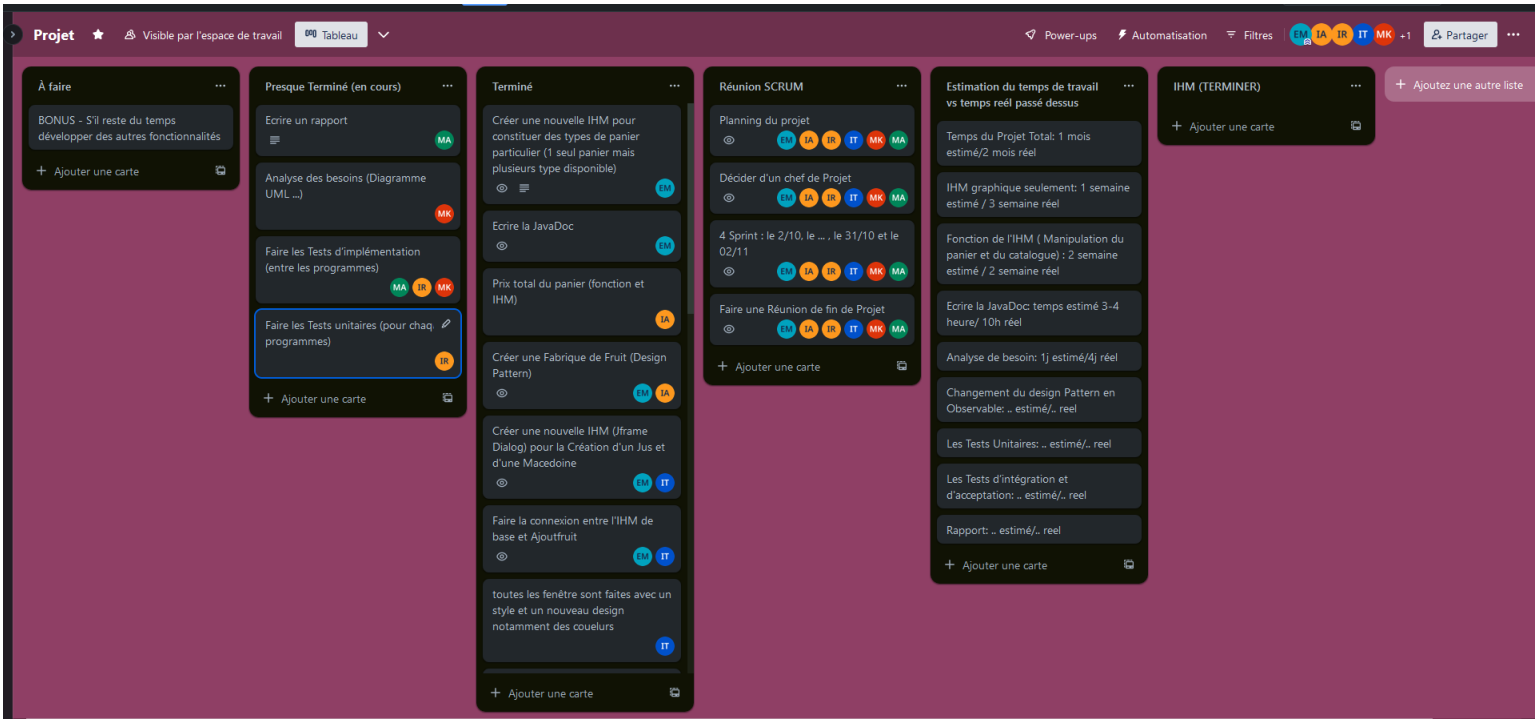
Malgré ces adaptations, nous avons maintenu l'intégrité de la méthode SCRUM comme les réunions de planification de sprint, les revues et les rétrospectives bien que moins nombreuse et effectué parfois très tard dans le soir ces événements étaient essentiels pour évaluer notre progression, identifier les points d'amélioration. En fin de compte, notre approche flexible de SCRUM nous a permis de surmonter les contraintes de temps et de disponibilité, garantissant ainsi le succès de notre projet dans les limites fixées par le délai de rendu.

Malgré ces adaptations, nous avons maintenu l'intégrité des méthodes SCRUM en utilisant des outils collaboratifs tels que GitHub et Trello. GitHub a été essentiel pour la gestion du code source et le contrôle de version. Chaque membre de l'équipe pouvait travailler sur sa propre branche (branch) et soumettre des demandes de tirage (pull requests) pour fusionner son travail avec la branche principale. Cela nous a permis de travailler simultanément sur différentes parties du projet tout en garantissant l'intégrité du code. Les commentaires JavaDoc et les révisions de code facilitaient la collaboration et assuraient la qualité du code produit.

Lien de notre Github : https://github.com/fe874388/Projet_Panier

En parallèle, Trello a été utilisé comme tableau en ligne pour organiser et suivre nos tâches. Nous avons créé des cartes pour chaque tâche à accomplir et les avons déplacées à travers les colonnes "À faire", "En cours" et "Terminé" en fonction de l'avancement. **Ce tableau nous a aussi servi de « blocs note » afin de comparer le temps estimé et le temps réel passé sur chaque tâche**. Cela nous a permis d'avoir une vue d'ensemble claire sur l'état d'avancement du projet, de visualiser les tâches en cours et à venir, et d'assigner des responsabilités à chaque membre de l'équipe. De plus, Trello nous a offert la flexibilité nécessaire pour ajuster notre planification en fonction des disponibilités de chacun.

Notre tableau Trello à J-7 du rendu :



Grâce à ces outils collaboratifs, nous avons pu maintenir une communication transparente au sein de l'équipe, suivre le progrès du projet en temps réel, et réagir rapidement aux changements. Les fonctionnalités de GitHub et Trello ont été intégrées dès le début et très harmonieusement à notre processus SCRUM, nous permettant ainsi de respecter les principes de cette méthodologie tout en nous adaptant aux contraintes qu'avait chaque membre. Ces outils ont joué un rôle central dans notre réussite en nous permettant de collaborer efficacement et de livrer un projet de qualité dans les délais impartis.

VI. Conclusion Générale

Ce projet de développement logiciel a revêtu une importance particulière pour notre équipe, car il a été l'occasion d'acquérir une expérience concrète et significative dans la conception d'Interfaces Homme-Machine (IHM) et de systèmes logiciels résilients. Tout au long de ce processus, nous avons eu l'opportunité d'appliquer une variété de concepts et de techniques de génie logiciel, ce qui a considérablement renforcé notre compréhension des bonnes pratiques de développement.

Les IHM que nous avons élaborées résultent d'une approche méthodique, commençant par la spécification des besoins et se déployant jusqu'à la phase de réalisation. Elles offrent un éventail de fonctionnalités avancées pour la gestion de fruits et de paniers, répondant de manière exhaustive aux attentes des utilisateurs. Ce projet a représenté un défi technique stimulant, et il a renforcé notre capacité à travailler en équipe de manière efficace, démontrant ainsi notre aptitude à relever des défis complexes.

Nous sommes particulièrement fiers du travail accompli et de l'expérience enrichissante que ce projet nous a offerte. Il a renforcé notre confiance dans nos compétences en génie logiciel et nous a permis de développer des solutions logicielles de haute qualité. Cette opportunité a été une étape cruciale dans notre parcours d'apprentissage et de croissance professionnelle.