

Découverte

Les tests unitaires en Java (IMC)

Rédigé par

David ROUMANET
Professeur BTS SIO



Changement

Date	Révision

Sommaire

A Introduction.....	1
A.1 Présentation.....	1
A.2 Prérequis.....	1
B Les tests unitaires.....	2
B.1 Explications.....	2
B.2 Fonctionnement.....	2
B.3 Méthode de développement TDD (Test Driven Development).....	2
C Ajout de tests unitaires IMCFX.....	3
C.1 Réflexion sur les tests.....	3
C.2 Ajout (vérification) de la dépendance Junit.....	4
C.3 Modification de la structure du projet.....	6
C.4 Rédaction des tests unitaires.....	6
C.5 Lancer les tests.....	7
D Amélioration de la lisibilité des tests.....	8
D.1 Explication des commandes.....	8
D.2 Modification des tests.....	9
D.3 Bonus : gestion des exceptions dans les tests.....	10
D.3.1 Comprendre les tests sur exception.....	10
D.3.2 Gestion des exceptions dans le programme.....	11
E Conclusion.....	12
E.1 Compétences acquises.....	12
E.2 Compétences potentielles.....	12
E.3 Compétences à découvrir.....	12
F Annexes.....	13
F.1 Sources.....	13
F.1.1 Le TDD : Test Driven Development.....	13
F.1.2 Les différents tests.....	13
F.1.3 Tutoriel de test avec TestFX.....	13
F.2 Comparaison des différents tests.....	13
F.3 Corrections méthodes complètes de tests.....	14

Nomenclature :

- **Assimiler** : cours pur. Explication théorique et détaillée (globalement supérieur à 4 pages).
- **Décoder** : fiche de cours, généralement inférieure à 5 pages.
- **Découvrir** : Travaux dirigés. Faisable sans matériel.
- **Explorer** : Travaux pratiques. Nécessite du matériel ou des logiciels.
- **Mission** : Projet encadré ou partie d'un projet.
- **Voyager** : Projet en autonomie totale. Environnement ouvert : Vous êtes le capitaine !

A Introduction

Dans la précédente activité de découverte, nous avons créé une petite application graphique en JavaFX qui permet de calculer un IMC. Nous allons découvrir ici, comment nous pouvons rendre le fonctionnement plus fiable et assurer le fonctionnement parfait de la méthode de calcul.

A.1 Présentation

Nous allons reprendre le projet précédent IMCFX et ajouter des tests unitaires.

La découverte porte sur cette méthode de travail qui est obligatoire dans les projets importants. Il s'agit ici, d'en comprendre l'intérêt, le fonctionnement et les limitations.

A.2 Prérequis

Avoir réalisé le projet précédent IMCFX.

B Les tests unitaires

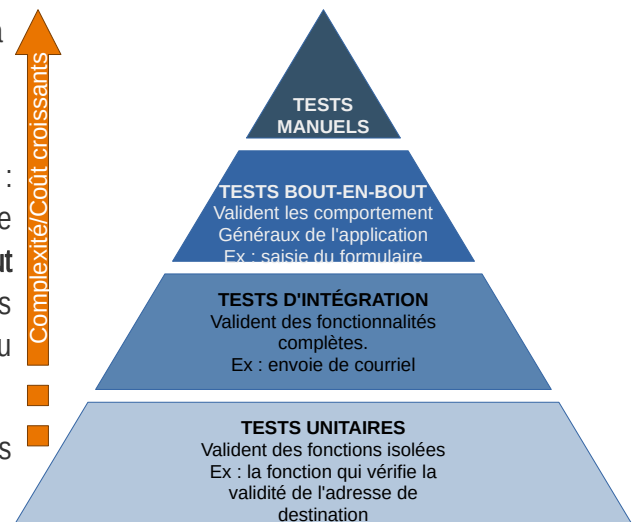
B.1 Explications

Les tests unitaires sont des codes conçus pour tester des méthodes (fonctions) dans une application. Ces tests ne sont exécutés que par les développeurs et ne sont pas visibles par les utilisateurs de l'application.

Dans une organisation, les **tests unitaires** servent à assurer que la modification de l'application n'entraîne pas des erreurs ou des régressions.

Ce sont les tests les plus faciles à mettre en œuvre : viennent ensuite les **tests d'intégration** qui testent une fonctionnalité complète. Les **tests de bout-en-bout** utilisent souvent des outils de manipulation des interfaces graphiques (outils capables de "cliquer" ou saisir un champ de formulaire).

Enfin, les **tests manuels** pour tout ce qui n'est pas automatisable.



B.2 Fonctionnement

La plupart des systèmes de tests unitaires font appel à un framework dédié, qui contient des fonctions spécialisées. Il faut donc :

- Installer ce framework
- Créer du code pour ce framework qui utilisera des fonctions de l'application
- Lancer les tests

Cette découverte ne testera qu'une seule fonction (le calcul de l'IMC) mais permettra de comprendre une limitation importante des tests unitaires : ils ne peuvent (généralement) interagir avec les interfaces graphiques.

B.3 Méthode de développement TDD (Test Driven Development)

Un courant de développement a également vu le jour grâce à l'automatisation des tests. Cette méthode consiste à créer le code de test d'abord, puis à rédiger le code de la fonction. Le développeur doit alors modifier son code jusqu'à ce que tous les tests soient passés avec succès.

L'intérêt est de penser security-by-design, en commençant par inventorier tous les cas de figures d'erreurs possibles.

Vous trouverez plus d'information en annexe.

C Ajout de tests unitaires IMCFX

Reprendre le projet vu précédemment.

Comme les tests unitaires ne peuvent pas s'appliquer sur une méthode qui récupère des éléments d'une interface graphique (car il n'est pas facile d'injecter une réponse dans un champ, par logiciel), il faut ré-écrire la fonction `onHelloButtonClick()`. Actuellement, elle est écrite comme ceci :

```
protected void onHelloButtonClick() {
    int poids = Integer.parseInt(txtPoids.getText());
    float taille = Float.parseFloat(txtTaille.getText());
    float imc = poids/(taille*taille);
    txtResultat.setText("Votre IMC est de "+imc);
}
```

Il est donc primordial d'avoir des méthodes avec des paramètres et un retour.

Remplacez-là dans votre code par le code suivant (deux méthodes) :

```
protected void onHelloButtonClick() {
    int poids = Integer.parseInt(txtPoids.getText());
    float taille = Float.parseFloat(txtTaille.getText());
    float imc = calculerIMC(poids, taille);
    txtResultat.setText("Votre IMC est de "+imc);
}

private float calculerIMC(int poids, float taille) {
    return poids / (taille * taille);
}
```

C.1 Réflexion sur les tests

Nous avons actuellement un code qui effectue l'opération suivante :

$$\text{IMC} = \text{poids} / \text{Taille}^2$$

L'intérêt de créer des tests unitaires est de prévoir tous les cas de figures possibles et déterminer les réponses appropriées.

Essayez de réfléchir aux différentes possibilités que la fonction `float calculerIMC(int poids, float taille)` peut rencontrer et compter les. Il doit y avoir environ 8 cas...

Voici une table des possibilités :

poids	taille	résultat	
Poids négatif ou zéro	*	Erreur à signaler : poids	
*	Taille négative ou zéro	Erreur à signaler : taille	
Poids : String	*	Erreur à signaler : poids	
*	Taille : String	Erreur à signaler : taille	
Poids nul	*	Erreur à signaler : poids	
*	Taille nul	Erreur à signaler : taille	
Poids hors intervalle	Taille hors intervalle	Erreur à signaler : intervalle	
Poids normal : int	Taille normale : float	calcul	

Vous pouvez constater qu'il faudrait que notre fonction `calculerIMC()` doit être améliorée :

Essayez quelques valeurs du tableau ci-dessus pour vérifier que le programme affiche des erreurs dans la console d'IntelliJ. Par chance, il ne plante pas l'interface. Par exemple, Poids = 80.5 et taille = 1.80

Nous pouvons améliorer notre fonction à partir du tableau, mais il y a toujours un risque que la fonction soit modifiée plus tard et qu'elle ne fonctionne plus. Les tests unitaires vont nous permettre de tester toutes les combinaisons à chaque compilation et éviter ce risque.

C.2 Ajout (vérification) de la dépendance Junit

Dans le fichier `pom.xml`, il faudrait théoriquement ajouter les lignes suivantes :

```

<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <junit.version>5.10.2</junit.version>
</properties>
```

et dans la partie `dependencies` :

```

<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-api</artifactId>
  <version>${junit.version}</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-engine</artifactId>
  <version>${junit.version}</version>
  <scope>test</scope>
</dependency>
```

Logiquement, IntelliJ s'est occupé de ça automatiquement, mais il est également possible de copier/coller le code depuis le dépôt officiel de Maven, par exemple :

<https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-api/5.11.4>

MVN REPOSITORY Search for groups, artifacts, categories

Home » org.junit.jupiter » junit-jupiter-api » 5.11.4

JUnit Jupiter API » 5.11.4
JUnit Jupiter is the API for writing tests using JUnit 5.

License EPL 2.0

Categories Testing Frameworks & Tools

Tags quality junit testing api

HomePage <https://junit.org/junit5/>

Date Dec 16, 2024

Files pom (3 KB) jar (211 KB) View All

Repositories Central Auxilor

Ranking #22 In MvnRepository (See Top Artifacts)
#3 In Testing Frameworks & Tools

Used By 17,506 artifacts

Note: There is a new version for this artifact

New Version

Maven Gradle Gradle (Short) Gradle (Kotlin) SBT Ivy Grape Leiningen Buildr

```
<!-- https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-api -->
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-api</artifactId>
  <version>5.11.4</version>
  <scope>test</scope>
</dependency>
```

Include comment with link to declaration

Maven

Code pom.xml

Moteur de recherche

Information de version



Il est important de noter que pour les tests unitaires, il n'est pas recommandé de changer régulièrement de version (mise à jour) sans faire de... tests au préalable. En effet, une incompatibilité de version pourrait entraîner l'échec de plusieurs tests.

Dans le code pour le fichier **pom.xml**, on note la présence de `<scope>test</scope>` qui indique que cette dépendance ne s'applique que dans le cadre de tests et non en production.

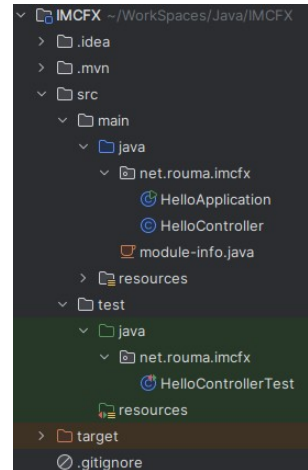
Nous pourrions aussi réécrire le numéro de version des dépendances org.javaafx comme pour Junit.

C.3 Modification de la structure du projet

Il faut maintenant ajouter un répertoire séparé dans lequel nous développerons nos tests.

Les tests seront écrits en Java, mais il est plus propre de bien séparer la partie "code utile" et "code de test". Voici la structure à adopter :

Cliquez sur le répertoire **src** puis faites un clic droit **New > Directory**. Nommez le répertoire **test**. Dans ce répertoire test, créez un package **net.rouma.imcfx** (ou le même nom que la paquetage contenu dans le répertoire java), puis à l'intérieur de ce paquetage, créez deux sous-répertoires **java** et **resources** (attention un seul 's', à l'anglaise). La couleur de ces répertoires est verte, pour indiquer une différence avec le reste du code.



Dans cette structure, il faut rédiger les tests dans le répertoire **src/test/java**.

C.4 Rédaction des tests unitaires

Nous allons créer une classe de test et donc le fichier portera son nom.

HelloControllerTest.java

```
package net.rouma.imcfx;

import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;

public class HelloControllerTest {

    @Test
    public void testCalculerIMC() {
        HelloController controller = new HelloController();

        // Test avec des valeurs normales
        assertEquals(24.80158805847168, controller.calculerIMC(70, 1.68), 0.01);

        // Test avec une taille de 0 (devrait théoriquement lancer une exception, mais ici, on le gère)
        assertEquals(Float.POSITIVE_INFINITY, controller.calculerIMC(70, 0), 0.01);

        // Test avec des valeurs limites
        assertEquals(15.943878173828125, controller.calculerIMC(45, 1.68), 0.01);
    }
}
```

Recopiez ce code dans le fichier correspondant. Il doit y avoir une erreur sur `calculerIMC()` : trouvez pourquoi.

Corrigez le code du fichier **HelloController.java** en mettant la portée de la méthode **calculerIMC()** à `protected`.



Trouvez désormais pourquoi les valeurs transmises dans `HelloTestController.java` sont soulignées en rouge.

💡 il s'agit d'un problème de type, sachant qu'un nombre à virgule est considéré – par défaut – comme étant un double et non un flottant.

C.5 Lancer les tests

Enfin, nous pouvons lancer les tests et vérifier que tout fonctionne comme prévu.

Cliquez sur l'icône verte de lancement à gauche de la méthode `HelloControllerTest`.

```
6  public class HelloControllerTest {  
7  
8     @Test  
9  public void testCalculerIMC() {  
10         HelloController controller = new HelloController();
```

Normalement, il n'y a pas d'erreur et les tests se sont exécutés normalement.

D Amélioration de la lisibilité des tests

Dans le chapitre précédent, nous avons créé un fichier contenant une méthode de test qui contient plusieurs assertions. Cette partie doit être nébuleuse pour vous et il est souhaitable de prendre le temps d'expliquer ce que fait le programme de test.

D.1 Explication des commandes

Voici les lignes du fichier HelloControllerTest.java.



Par convention, un fichier testeur pour toujours le même nom que la classe testée, suivi du mot Test à la fin.

HelloControllerTest.java

```
@Test
public void testCalculerIMC() {
    HelloController controller = new HelloController();

    // Test avec des valeurs normales
    assertEquals(24.80158805847168f, controller.calculerIMC(70, 1.68f), 0.01);

    // Test avec une taille de 0 (devrait théoriquement lancer une exception, mais ici, on le gère)
    assertEquals(Float.POSITIVE_INFINITY, controller.calculerIMC(70, 0), 0.01);

    // Test avec des valeurs limites
    assertEquals(15.943878173828125f, controller.calculerIMC(45, 1.68f), 0.01);
}
```

La ligne `HelloController controller = new HelloController();` instancie le contrôleur qui contient le code à tester, avec que le contrôleur de test puisse accéder aux attributs et méthodes de cette classe. Les attributs ne sont ici pas utilisés, seule la méthode `calculerIMC()` est intéressante.

Les méthodes `assertEquals(resultatAttendu, fonctionATester, ecartDeTolerance)` est une méthode de test fournie par la dépendance JUnit.

- **AssertEquals** vérifie une égalité (si c'est égal, sera vraie, sinon sera fausse). Il existe d'autres méthodes, comme `assertTrue(fonctionATester)` ou `assertNull(fonctionATester)` et bien d'autres...
- **resultatAttendu** est la valeur que la fonction à tester devrait répondre, si elle fonctionne normalement.
- **fonctionATester()** est la fonction que l'on veut tester, ici `calculerIMC()` à laquelle on passe des paramètres et dont on connaît le résultat théorique.
- **ecartDeTolerance** est un paramètre propre à la méthode `AssertEquals`, pour accepter un petit écart de calcul (notamment avec des nombres doubles ou des nombres flottants).

Donc, la fonction `testCalculerIMC()` effectue ici trois tests simultanément.

D.2 Modification des tests

Nous allons modifier le code de test, en séparant les trois tests de la méthode `testCalculerIMC()` et au contraire créer trois méthodes séparées, mais plus lisibles.

Remplacez le code de la classe par celui-ci :

```
public class HelloControllerTest {  
  
    @Test  
    public void testCalculerIMC_NormalValues() {  
        HelloController controller = new HelloController();  
        assertEquals(24.80158805847168f, controller.calculerIMC(70, 1.68f), 0.01);  
    }  
  
    @Test  
    public void testCalculerIMC_ZeroTaille() {  
        HelloController controller = new HelloController();  
        assertEquals(Float.POSITIVE_INFINITY, controller.calculerIMC(70, 0), 0.01);  
    }  
}
```

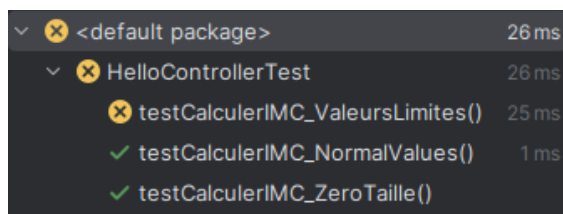
Ajouter la méthode manquante `testCalculerIMC_Valeurs_Limites()`

💡 N'oubliez pas le `@Test`

💡 Astuce, essayez un poids de 20 kg pour 1.80f et une valeur attendue de 0.

Lancez tous les tests en faisant un clic droit sur le répertoire `java` (zone en vert) et choisir **Run 'all tests'**.

Les tests se lancent et il doit y avoir un test en erreur, comme sur l'image ci-après :



On constate que le test sur la valeur de taille à zéro renvoie bien l'infini, comme attendu, que les valeurs normales fonctionnent aussi, mais que le test des valeurs limites ne renvoie pas la valeur zéro.

Modifiez le code source de la méthode `calculerIMC()` pour qu'elle renvoie 0 si les valeurs de poids sortent des limites suivantes :

Poids : entre 20 et 200 kg

Taille : entre 0,60 et 2,10 m

Supprimez le test `testCalculerIMC_ZeroTaille()` et Testez votre code, jusqu'à ce qu'il fonctionne

En effet, le test d'une valeur ou la taille serait de zéro devient inutile, puisqu'on teste des limites au-delà de zéro. Vous pouvez vous aider de la solution donnée en annexe, elle vous montre comment ajouter une boîte de dialogue pour afficher une erreur à l'utilisateur.

D.3 Bonus : gestion des exceptions dans les tests

Il reste la gestion des erreurs de saisie : formulaire vide ou contenant des caractères non numériques.

Nous pourrions utiliser une exception try... catch dans la fonction calculerIMC et ainsi, intercepter l'erreur.

D.3.1 Comprendre les tests sur exception

Nous pouvons aussi laisser l'erreur générer une exception : les tests unitaires peuvent les gérer pour valider ce fonctionnement... cependant, l'application affichera une exception dans la console et n'affichera rien : c'est donc une mauvaise solution technique (il est préférable de gérer l'exception dans calculerIMC()) mais qui permet de voir les tests unitaires associés aux exceptions :

■ Ajoutez les tests suivants dans la classe HelloControllerTest :

```
@Test
public void testCalculerIMC_EmptyParameters() {
    HelloController controller = new HelloController();
    // Simulez des champs vides en modifiant la méthode pour gérer les chaînes vides
    assertThrows(NumberFormatException.class, () -> {
        controller.calculerIMC(Integer.parseInt(""), Float.parseFloat(""));
    });
}

@Test
public void testCalculerIMC_NonNumericParameters() {
    HelloController controller = new HelloController();
    // Simulez des entrées non numériques
    assertThrows(NumberFormatException.class, () -> {
        controller.calculerIMC(Integer.parseInt("abc"), Float.parseFloat("def"));
    });
}
```

■ Il faut également ajouter dans l'entête, l'importation de la bonne bibliothèque :

```
import static org.junit.jupiter.api.Assertions.assertThrows;
```

Ainsi, lors des tests unitaires, la méthode calculerIMC() génère bien une exception lorsqu'on saisit du texte ou des chaînes vides.

La gestion des exceptions n'existe pas dans le programme.

D.3.2 Gestion des exceptions dans le programme

Pour coder de manière plus propre, il est recommandé d'intercepter les erreurs lorsqu'elles surviennent.

Recopiez ce code utilisant les exceptions et mots-clés try et catch :

```
HelloController.java
```

```
@FXML
protected void onHelloButtonClick() {
    try {
        int poids = Integer.parseInt(txtPoids.getText());
        float taille = Float.parseFloat(txtTaille.getText());
        float imc = calculerIMC(poids, taille);
        if (imc > 0) {
            txtResultat.setText("Votre IMC est de " + imc);
        } else {
            txtResultat.setText("");
            // Cadeau, voici comment afficher une boite d'alerte
            Alert alert = new Alert(AlertType.ERROR, "Erreur dans la taille ou le poids",
                ButtonType.OK);
            alert.showAndWait();
        }
    } catch (Exception e) {
        // Afficher un message d'erreur
        Alert alert = new Alert(AlertType.ERROR, "Veuillez entrer des valeurs numériques valides.",
            ButtonType.OK);
        alert.showAndWait();
    }
}
```

Il est probable que vous aurez à ajouter l'importation de la bibliothèque qui concerne les boites d'alerte :

```
import static javafx.scene.control.Alert.AlertType;
```

Testez votre code, il ne doit plus y avoir de message dans la console.

Désormais, les erreurs de saisies ne peuvent plus entraîner le plantage de votre application.

E Conclusion

Cette initiation aux tests unitaires doit vous permettre de pouvoir travailler sur des projets conséquents, tout en comprenant la structure de ceux-ci.

E.1 Compétences acquises

Il est peu probable que vous sachiez créer une structure de tests unitaires sans regarder à nouveau l'activité, mais vous devriez être capable :

- D'ajouter la dépendance nécessaire dans votre projet
 - Avec Java, l'usage de Maven et du dépôt maven facilite l'intégration
 - Avec IntelliJ, les dépendances sont déjà présentes (junit-jupiter-api et junit-jupiter-engine)
- De modifier la structure du projet
 - En intégrant l'arborescence de votre paquetage (ex. com.microsoft.x)
 - En ajoutant Test, puis java et resources
 - En ajoutant la classe de test (nom de classe + Test à la fin)
- De trouver de l'information sur les tests unitaires
 - Consulter la documentation officielle JUnit : <https://junit.org/junit5/docs/current/user-guide/#writing-tests>
 - De créer vos tests unitaires

E.2 Compétences potentielles

Vous êtes en capacité de créer des applications correctement structurées avec de l'entraînement.

Vous pourriez développer un code en respectant la structure proposée.

Vous sauriez lire un projet existant et probablement aider au débogage d'une fonction simple.

E.3 Compétences à découvrir

Il existe des tests utilisant TestFX, une dépendance capable d'interagir avec les composants graphiques de JavaFX : <http://testfx.github.io/TestFX/>

Cela dépasse les attentes de cette découverte, mais peut vous intéresser.

F Annexes

F.1 Sources

F.1.1 Le TDD : Test Driven Development

Aussi appelé méthode du red, green, refactor :

<https://www.ionos.fr/digitalguide/sites-internet/developpement-web/quest-ce-que-le-test-driven-development/>

<https://openclassrooms.com/fr/courses/5641591-testez-votre-application-c/5656581-decouvrez-les-principes-du-test-driven-development-tdd>

F.1.2 Les différents tests

<https://www.twilio.com/en-us/blog/unit-integration-end-to-end-testing-difference>

<https://www.globalapptesting.com/blog/unit-testing-vs-end-to-end-testing>

F.1.3 Tutoriel de test avec TestFX

Pour les plus curieux : https://www.youtube.com/playlist?list=PLKiN3faYVq8_OQKZd4C8Dh0ZR6d2zL0x-

F.2 Comparaison des différents tests

Caractéristique	Tests Unitaires	Tests d'Intégration	Tests End-to-End	Tests Manuels
Portée	Composant individuel	Plusieurs composants	Application complète	Cas d'utilisation spécifiques
Vitesse	Rapide	Moyenne	Lente	Variable
Automatisation	Oui	Oui	Oui	Non
Complexité	Simple	Moyenne	Élevée	Variable
Objectif	Vérification du code	Vérification des interactions	Vérification du comportement global	Vérification de l'expérience utilisateur

F.3 Corrections méthodes complètes de tests

Voici le code source de la classe HelloControllerIMC :

```
package net.rouma.imcfx;

import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;

public class HelloControllerTest {

    @Test
    public void testCalculerIMC_NormalValues() {
        // Teste la formule de calcul
        HelloController controller = new HelloController();
        assertEquals(24.80158805847168f, controller.calculerIMC(70, 1.68f), 0.01);
    }

    @Test
    public void testCalculerIMC_ValeursLimites() {
        HelloController controller = new HelloController();
        assertEquals(0f, controller.calculerIMC(20, 1.80f), 0.01);
        assertEquals(0f, controller.calculerIMC(200, 1.80f), 0.01);
        assertEquals(0f, controller.calculerIMC(80, 0.6f), 0.01);
        assertEquals(0f, controller.calculerIMC(80, 2.10f), 0.01);
    }
}
```

et le code pour la fonction calculerIMC()

```
@FXML
protected void onHelloButtonClick() {
    int poids = Integer.parseInt(txtPoids.getText());
    float taille = Float.parseFloat(txtTaille.getText());
    float imc = calculerIMC(poids, taille);
    if (imc > 0) {
        txtResultat.setText("Votre IMC est de " + imc);
    } else {
        txtResultat.setText("");
        // Cadeau, voici comment afficher une boite d'alerte
        Alert alert = new Alert(ERROR, "Erreur dans la taille ou le poids", ButtonType.OK);
        alert.showAndWait();
    }
}

protected float calculerIMC(int poids, float taille) {
    // Gestion des limites
    if (poids >= 200 || poids <= 20) return 0;
    if (taille >= 2.10f || taille <= 0.60f) return 0;

    // Calcul standard
    return poids / (taille * taille);
}
```