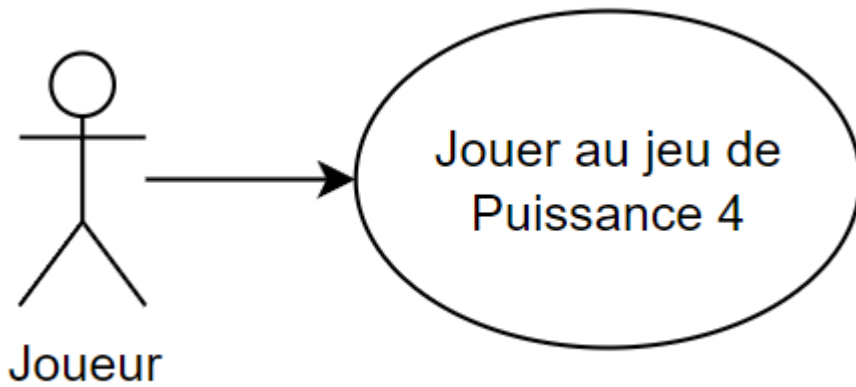


Diagramme de cas d'utilisation



Scénario

UC : Jouer au Puissance 4

Périmètre : Le jeu

Niveau : But utilisateur

Acteur principal : Joueur

Précondition : Aucune

Postconditions : Aucune

Scénario nominal

1. L'utilisateur lance le programme
2. Le système demande le nom du joueur 1
3. Le joueur 1 répond
4. Le système enregistre le nom du joueur 1
5. Le système demande au joueur 2
6. Le joueur 2 répond
7. Le système enregistre le nom du joueur 2
8. Le système affiche le plateau ainsi que le nom du joueur qui doit jouer
9. Le joueur choisi dans quelle colonne il souhaite mettre un jeton
10. Le système vérifie si la partie est gagnée
11. Le système affiche le vainqueur et demande si l'utilisateur souhaite refaire une partie ("y" or "n")
12. L'utilisateur répond "y" ou "n"
13. Le système retourne au point 8 du scénario nominal et incrémente le nombre de partie

Extensions

3.a L'utilisateur répond à la question avec une chaîne vide

1. Retour au point 2 du scénario nominal

6.a L'utilisateur répond à la question avec une chaîne vide

1. Retour au point 5 du scénario nominal

9.a L'utilisateur choisit un numéro de colonne $n \notin \{x \in \mathbb{N}, tq\ 1 \leq x \leq 7\}$

1. Le système affiche un message d'erreur
2. Retour au point 8

9.b L'utilisateur choisit un numéro de colonne valide, mais la colonne est pleine

1. Le système affiche un message d'erreur
2. Retour au point 8

10.a La partie n'est pas finie

1. Le système change le joueur qui doit jouer
2. Retour au point 8 du scénario nominal

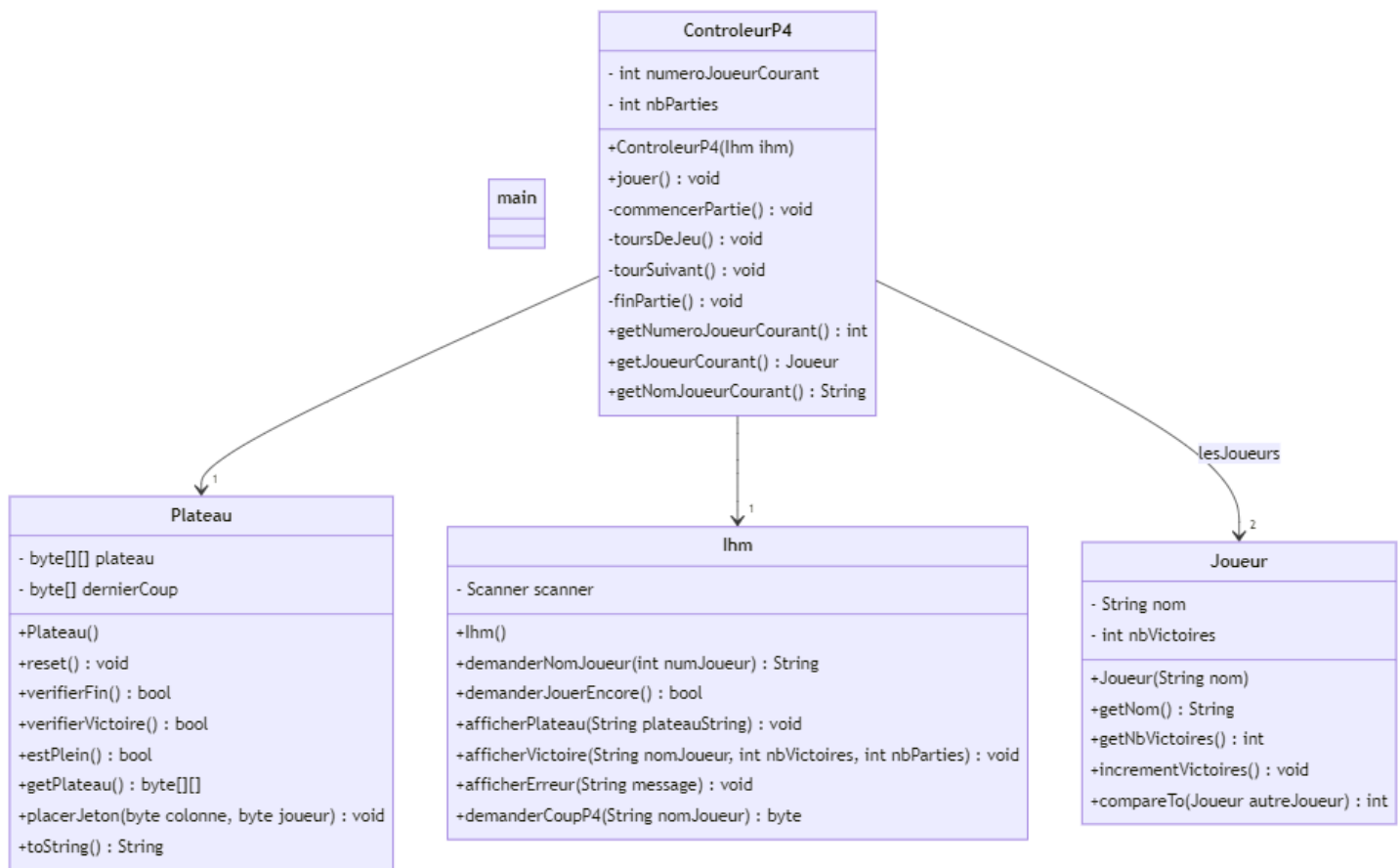
12.a l'utilisateur répond autre chose que 'y' ou 'n'

1. Le système affiche un message d'erreur
2. Retour au point 11 du scénario nominal

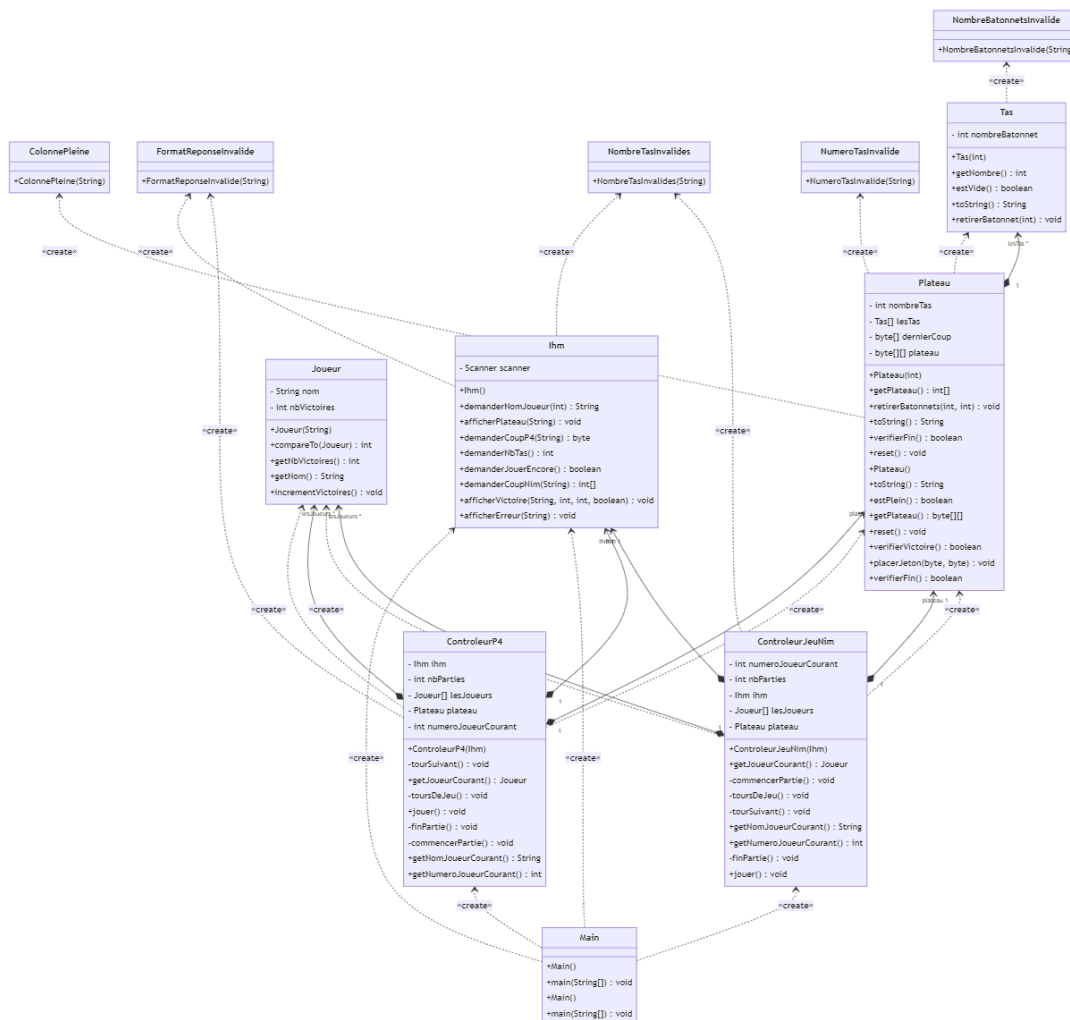
12.b L'utilisateur répond 'n'

1. Le système affiche le gagnant et le nombre de victoire du gagnant
 - a. - Il n'y a pas de gagnant
 - i. Le système affiche ex aequo et le nombre de victoire des joueurs (il est identique pour les deux)
 - ii. Retour au point 2 de l'extension 12.b
2. Fin du programme

Modèle de Conception



Version intellij



Répartition des tâches

14/03/2024 (Publication du sujet)

Nathan, Dany, Agathe : review du sujet et brainstorming (check de la victoire etc)

16/03/2024

Nathan : Diagramme de classe, Cas d'utilisation

Dany : review du Diagramme

Dany : Développement

Agathe : Développement

Dany : Review, documentation, fix


Dany : mise en place de test avec JUnit

Utilisation de l'IA


Nathan : Aucun Usage




Agathe : Aucun Usage

Dany : Copilot (screen dans img)

 danydudiot

quand j'utilise Scanner, par moment il capte des inputs fantome, comment faire pour les éviter

 GitHub Copilot



L'utilisation de la classe `Scanner` en Java peut parfois conduire à des problèmes d'inputs "fantômes". Cela se produit généralement lorsque vous mélangez des appels à `nextLine()` avec d'autres méthodes de `Scanner` comme `nextInt()`, `nextDouble()`, etc.

Ces méthodes ne consomment pas le caractère de fin de ligne (c'est-à-dire `\n`) dans l'input. Donc, si un utilisateur entre un nombre puis appuie sur Entrée, `nextInt()` lira le nombre, mais le caractère de fin de ligne restera dans le buffer d'input. La prochaine fois que vous appellerez `nextLine()`, il lira ce caractère de fin de ligne restant et retournera immédiatement une chaîne vide, ce qui peut sembler être un input "fantôme".

Pour éviter cela, vous pouvez ajouter un appel supplémentaire à `nextLine()` après chaque appel à `nextInt()`, `nextDouble()`, etc. pour consommer le caractère de fin de ligne restant. Voici comment vous pouvez modifier votre méthode `demandeNbTas()` pour éviter les inputs "fantômes" :

```
public int demandeNbTas() throws NombreTasInvalides {
    System.out.println("Avec combien de tas voulez-vous jouer ?\nEntrez un entier >= 1 : ");
    if (scanner.hasNextInt()) {
        int nbTas = scanner.nextInt();
        scanner.nextLine(); // consomme le caractère de fin de ligne restant
        return nbTas;
    }
    scanner.next();
    throw new NombreTasInvalides("Format de réponse invalide.");
}
```

Appliquez cette modification à toutes les méthodes où vous mélangez `nextLine()` avec d'autres méthodes de `Scanner`.

> Using 1 Reference

💡 How can I handle multiple types of inputs (int, double, etc.) without encountering "phantom" inputs?

Utilisation de l'IA pour les tests

[illegible]

GitHub Copilot

Pour résoudre ce problème sans modifier les classes `ContrôleurJeuNim` et `Ihm`, vous pouvez utiliser une approche de test qui simule les entrées utilisateur. Vous pouvez utiliser `System.setIn()` pour rediriger l'entrée standard vers une nouvelle `InputStream` qui contient les entrées de test.

Dépendant, comme vous utilisez deux instances de `Scanner` dans la classe `Ihm`, vous devez vous assurer que toutes les entrées de test pour une instance de `Scanner` sont fues avant de passer à l'autre. Vous pouvez le faire en ajoutant une nouvelle ligne `\n` à la fin de chaque entrée de test.

Voici comment vous pouvez le faire dans votre classe de test `TestContrôleurJeuNim` :

```

@JTest
public void testJouer() {
    // Préparez vos entrées de test
    String input = "3\n" // entrée pour demanderNim
        + "Jouer\n" // entrée pour demanderNimJouer pour le joueur 1
        + "Jouer\n" // entrée pour demanderNimJouer pour le joueur 2
        + "3 2\n" // entrée pour demanderCouplet pour le joueur 1
        + "3 2\n" // entrée pour demanderCouplet pour le joueur 2
        + "n\n"; // entrée pour demanderJouerEncore

    // Rediriger l'entrée standard vers une nouvelle InputStream qui contient vos entrées de test
    InputStream in = new ByteArrayInputStream(input.getBytes());
    System.setIn(in);

    // Créez une nouvelle instance de ContrôleurJeuNim et appelez la méthode jouer
    ContrôleurJeuNim contrôleur = new ContrôleurJeuNim();
    contrôleur.jouer();

    // Vérifiez que le jeu s'est déroulé comme prévu
    // ...
}

```

Dans cet exemple, chaque entrée de test est suivie d'une nouvelle ligne `\n`, ce qui signifie que chaque appel à `nextLine()` sur une instance de `Scanner` lira une entrée de test complète.

• Using | Reference

• Pin Java

The image shows a GitHub Copilot chat window with a user's question and the AI's response. The user asks how to test a method that uses a Scanner when the Scanner is initialized from the console. The AI provides a code example for a test class and explains the steps to run the test.

User Question:

comment faire des test quand la méthode utilise des Scanner ?

AI Response:

Tester des méthodes qui utilisent Scanner peut être un peu délicat car Scanner lit l'entrée de l'utilisateur à partir de la console. Cependant, vous pouvez contourner cela en utilisant `ByteArrayInputStream` pour simuler l'entrée de l'utilisateur.

Voici comment vous pouvez le faire :

```
package test;

import modele.Joueur;
import org.junit.Test;
import vue.Ihm;
import exception.NombreTasInvalides;
import java.io.ByteArrayInputStream;
import java.io.InputStream;

import static org.junit.Assert.*;

public class IhmTest {

    @Test
    public void testDemanderNbTas() {
        String input = "3";
        InputStream in = new ByteArrayInputStream(input.getBytes());
        System.setIn(in);

        Ihm ihm = new Ihm();
        try {
            int nbTas = ihm.demanderNbTas();
            assertEquals(3, nbTas);
        } catch (NombreTasInvalides e) {
            fail("Exception NombreTasInvalides levée");
        }
    }
}
```

Dans cet exemple, nous simulons l'entrée de l'utilisateur "3" pour la méthode `demanderNbTas()`. Nous utilisons `System.setIn(in);` pour rediriger l'entrée standard vers notre `ByteArrayInputStream`. Ensuite, nous appelons la méthode `demanderNbTas()` et vérifions si elle renvoie la valeur attendue.

Notez que cette méthode modifie l'entrée standard globale, ce qui peut affecter d'autres tests ou parties de votre programme qui utilisent `System.in`. Assurez-vous de réinitialiser `System.in` à sa valeur d'origine après avoir terminé le test.

> Using 3 References

User Question:

comment faire pour lancer les test dans TestTas.java depuis Test.java ?

AI Response:

Pour lancer les tests de `TestTas.java` depuis `Test.java`, vous pouvez utiliser la classe `org.junit.runner.JUnit4` de JUnit. Cette classe a une méthode `runClasses()` qui peut exécuter des tests de plusieurs classes.

Voici comment vous pouvez le faire :

```
package test;

import org.junit.runner.RunWith;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;
import test.modele.TestTas;

@RunWith(JUnit4.class)
public class Test {

    public static void main(String[] args) {
        Result result = JUnit4.runClasses(TestTas.class);


        for (Failure failure : result.getFailures()) {
            System.out.println(failure.toString());
        }

        System.out.println(result.wasSuccessful());
    }
}
```

Dans cet exemple, nous exécutons les tests de la classe `TestTas` avec `JUnit4.runClasses(TestTas.class)`. Nous parcourons ensuite tous les échecs de test et les imprimons. Enfin, nous imprimons si tous les tests ont réussi ou non.

> Using 1 Reference

 danydudiot



comment faire pour que si j'appel 2 deux un scanner j'ai "3" puis "a" ?

 GitHub Copilot

Pour obtenir les valeurs "3" puis "a" à partir de votre tableau `input` lors de l'appel de `Scanner`, vous devez convertir votre tableau en une seule chaîne, avec chaque élément séparé par une nouvelle ligne (`\n`). Ensuite, vous pouvez utiliser cette chaîne pour créer un nouveau `ByteArrayInputStream`. Voici comment vous pouvez le faire :

```
// Convertir le tableau en une seule chaîne, avec chaque élément séparé par une nouvelle ligne
String inputString = String.join("\n", input);


// Créer un nouveau ByteArrayInputStream à partir de la chaîne
InputStream in = new ByteArrayInputStream(inputString.getBytes());

// Utiliser 'in' pour créer un nouveau Scanner
Scanner scanner = new Scanner(in);


// Maintenant, lorsque vous appelez scanner.nextLine(), vous obtiendrez d'abord "3", puis "a"
String first = scanner.nextLine(); // Retourne "3"
String second = scanner.nextLine(); // Retourne "a"
```



Dans votre cas, vous pouvez appliquer cette modification dans votre méthode de test `testDemanderNbTas()`.

> Using 2 References

 danydudiot

comment récupérer le contenu du printstream

 GitHub Copilot

Pour récupérer le contenu d'un `PrintStream`, vous devez d'abord le connecter à un `ByteArrayOutputStream`. Ensuite, vous pouvez utiliser la méthode `toString` de `ByteArrayOutputStream` pour obtenir une représentation `String` du contenu.

Voici comment vous pouvez le faire :

```
import java.io.ByteArrayOutputStream;
import java.io.PrintStream;


public class Main {
    public static void main(String[] args) {
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        PrintStream printStream = new PrintStream(baos);
        System.setOut(printStream);


        System.out.println("Hello, world!");

        String output = baos.toString(); // récupère le contenu du PrintStream
        System.err.println(output); // affiche le contenu sur la console d'erreur (pour éviter la redirection)
    }
}
```


Dans cet exemple, nous redirigeons `System.out` vers un `PrintStream` qui est connecté à un `ByteArrayOutputStream`. Ensuite, nous imprimons "Hello, world!" sur `System.out`, qui est maintenant redirigé vers notre `ByteArrayOutputStream`. Enfin, nous utilisons `baos.toString()` pour obtenir une `String` représentant le contenu de `System.out`, et nous l'affichons sur la console d'erreur (pour éviter la redirection de `System.out`).




> Using 1 Reference

 danydudiot



comment je peut simplifier ca


 GitHub Copilot


  

Vous pouvez simplifier la création et l'initialisation du tableau en utilisant une déclaration de tableau littéral. Voici comment vous pouvez le faire :


```
int[] resultatAttendu = {1, 3, 5};
```




> Using 4 References

 danydudiot



comment faire pour rediriger normalement l'entrer et la sortie

 GitHub Copilot

Pour rediriger l'entrée et la sortie standard à leur état normal, vous pouvez stocker les flux d'entrée et de sortie standard originaux avant de les modifier. Ensuite, vous pouvez les rétablir après avoir terminé avec les flux modifiés. Voici comment vous pouvez le faire :

```
// Stocker les flux d'entrée et de sortie standard originaux
InputStream originalIn = System.in;
PrintStream originalOut = System.out;

// Modifier les flux d'entrée et de sortie
String input = "3";
InputStream in = new ByteArrayInputStream(input.getBytes());
PrintStream out = new PrintStream(new ByteArrayOutputStream());
System.setIn(in);
System.setOut(out);

// Utiliser les flux modifiés
// ...

// Rétablir les flux d'entrée et de sortie standard originaux
System.setIn(originalIn);
System.setOut(originalOut);
```

Dans votre cas, vous pouvez appliquer cette modification dans votre méthode de test `testDemanderNbTas()`.

> Using 1 Reference

danydudiot

comment faire des tests de methode en java

GitHub Copilot

Pour faire des tests de méthode en Java, vous pouvez utiliser JUnit, un framework populaire pour tester le code Java. Voici comment vous pouvez le faire :

1. Ajoutez la dépendance JUnit à votre projet. Si vous utilisez Maven, vous pouvez ajouter ceci à votre fichier `pom.xml` :

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

2. Créez une nouvelle classe de test dans votre répertoire de test. Par convention, le nom de la classe de test est le nom de la classe que vous testez, suivi de "Test".

3. Dans cette classe de test, écrivez une méthode pour chaque méthode que vous voulez tester dans votre classe. Chaque méthode de test doit être annotée avec `@Test`.

Voici un exemple de comment vous pouvez tester une méthode dans votre classe Test :

```
package test;

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

public class TestTest {
    @Test
    public void test() {
        Test testInstance = new Test();
        // Remplacez "yourMethod" et "expectedResult" par la méthode que vous testez et le résultat attendu
        // Par exemple, si vous testez une méthode qui additionne deux nombres, vous pouvez faire :
        // int result = testInstance.add(2, 3);
        // assertEquals(5, result);
        // Votre méthode de test pourrait ressembler à ceci :
        // int result = testInstance.yourMethod();
        // assertEquals(expectedResult, result);
    }
}
```

N'oubliez pas de remplacer "yourMethod" et "expectedResult" par la méthode que vous testez et le résultat attendu.

> Using 1 Reference

danydudiot

je n'utilise pas maven

GitHub Copilot

Si vous n'utilisez pas Maven, vous pouvez télécharger la bibliothèque JUnit directement depuis le site officiel et l'ajouter à votre projet. Ensuite, vous pouvez suivre les étapes 2 et 3 que j'ai mentionnées précédemment pour créer et exécuter vos tests.

Voici comment vous pouvez tester une méthode dans votre classe Test :

```
package test;

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

public class TestTest {

    @Test
    public void testYourMethod() {
        Test testInstance = new Test();
        // Remplacez "yourMethod" et "expectedResult" par la méthode que vous testez et le résultat attendu
        // Par exemple, si vous testez une méthode qui additionne deux nombres, vous pouvez faire :
        // int result = testInstance.add(2, 3);
        // assertEquals(5, result);
        // Votre méthode de test pourrait ressembler à ceci :
        // int result = testInstance.yourMethod();
        // assertEquals(expectedResult, result);
    }
}
```

N'oubliez pas de remplacer "yourMethod" et "expectedResult" par la méthode que vous testez et le résultat attendu.

> Using 1 Reference

danydudiot

peut tu me donner un exemple avec cette classe ?

GitHub Copilot

Bien sûr, je peux vous aider à créer un test pour la classe `Joueur`. Nous allons tester les méthodes `getNom`, `getNbVictoires`, `incrementVictoires` et `compareTo`.

Voici comment vous pouvez le faire :

```
package test;

import modele.Joueur;
import org.junit.Test;
import static org.junit.Assert.*;

public class JoueurTest {

    @Test
    public void testGetNom() {
        Joueur joueur = new Joueur("Test");
        assertEquals("Test", joueur.getNom());
    }

    @Test
    public void testGetNbVictoires() {
        Joueur joueur = new Joueur("Test");
        assertEquals(0, joueur.getNbVictoires());
    }

    @Test
    public void testIncrementVictoires() {
        Joueur joueur = new Joueur("Test");
        joueur.incrementVictoires();
        assertEquals(1, joueur.getNbVictoires());
    }

    @Test
    public void testCompareTo() {
        Joueur joueur1 = new Joueur("Test1");
        Joueur joueur2 = new Joueur("Test2");
        joueur1.incrementVictoires();
        assertTrue(joueur1.compareTo(joueur2) > 0);
    }
}
```

Dans cet exemple, nous créons un nouvel objet `Joueur` pour chaque test. Nous utilisons ensuite les méthodes `assertEquals` et `assertTrue` de JUnit pour vérifier que les méthodes de la classe `Joueur` fonctionnent comme prévu.

> Using 3 References