

Projet ITIC

WANG Christophe

GAIDA Noursaine

Année 2023–2024 — 24 décembre 2023

Table des matières

1	Test fonctionnel	3
1.1	Analyse des tests de la méthode « ajouter » dans la classe « Conteneur »	3
2	Test structurel	4
2.1	La méthode « isSolved » dans la classe « Puzzle »	5
2.2	Configuration de Jenkins	6
3	Tests basés sur les modèles	7
3.1	Un exemple de test basé sur les modèles	7
4	Test mutationnel	8
4.1	Un exemple de test mutationnel	8
5	Rapport de tests	10
5.1	Pourcentage de couverture à travers clEmma (Eclipse)	10
5.2	Pourcentage de couverture à travers de tests mutationnels	15

1 Test fonctionnel

Les tests fonctionnels constituent une étape essentielle dans la validation des fonctionnalités de la Gestion du Conteneur. Cette section présente une analyse détaillée des résultats obtenus à travers divers scénarios de test fonctionnel, mettant en lumière la couverture atteinte, les succès, ainsi que les échecs identifiés.

1.1 Analyse des tests de la méthode « ajouter » dans la classe « Conteneur »

Les tests fonctionnels ont pour objectif d'évaluer la fiabilité de la méthode « ajouter » de la classe « Conteneur », responsable de la gestion des ajouts de paires clé-valeur. La conception des tests a été réalisée avec soin afin d'inclure divers scénarios et de garantir le bon fonctionnement de la méthode dans des conditions variées.

```
1 package tests.fonctionnels;
2
3 import org.junit.jupiter.api.AfterEach;
4
10
11 class testAjouterCouplesClefValeur {
12
13     private Conteneur CVide, CPlein, CNonVideNonPlein;
14     private Object A1, A2, B1, B2;
15     private Object A_NonPresent, B_NonPresent;
16
17     @BeforeEach
18     void setup() throws ErreurConteneur {
19         A_NonPresent = new String("clé x");
20         B_NonPresent = new String("valeur x");
21
22         A1 = new String("clé 1");
23         A2 = 2; // = new Integer(2)
24         B1 = new String("valeur 1");
25         B2 = 3;
26
27         // Conteneur vide
28         CVide = new Conteneur(10);
29
30         // Conteneur non vide et non plein
31         CNonVideNonPlein = new Conteneur(5);
32         CNonVideNonPlein.ajouter(A1, B1);
33         CNonVideNonPlein.ajouter(A2, B2);
34
35         // Conteneur plein
36         CPlein = new Conteneur(2);
37         CPlein.ajouter(A1, B1);
38         CPlein.ajouter(A2, B2);
39     }
40
41     @AfterEach
42     void tearDown() {
43         CVide = CNonVideNonPlein = CPlein = null;
44     }
45
46     @Test
47     void TestCpresentOvideEtConteneurPlein() {
48         Assertions.assertThrows(testEtat.DebordementConteneur.class, () -> CPlein.ajouter(A1, B1));
49     }
50
51     @Test
52     void TestCpresentOvideEtConteneurNonVideNonPlein() {
53         Assertions.assertDoesNotThrow(() -> CNonVideNonPlein.ajouter(A1, B2));
54     }
55
56     @Test
57     void TestCpresentOvideEtConteneurPlein() {
58         Assertions.assertThrows(testEtat.DebordementConteneur.class, () -> CPlein.ajouter(A1, null));
59     }
60
61     @Test
62     void TestCpresentOvideEtConteneurNonVideNonPlein() {
63         Assertions.assertDoesNotThrow(() -> CNonVideNonPlein.ajouter(A1, null));
64     }
65
66     @Test
67     void TestCpresentNonPresentEtConteneurPlein() {
68         Assertions.assertThrows(testEtat.DebordementConteneur.class, () -> CPlein.ajouter(A1, B_NonPresent));
69     }
70
71     @Test
72     void TestCpresentNonPresentEtConteneurNonVideNonPlein() {
73         Assertions.assertDoesNotThrow(() -> CNonVideNonPlein.ajouter(A1, B_NonPresent));
74     }
75 }
```

- Contexte initial :

Le cadre du test repose sur la spécification du cahier des charges, où la classe Conteneur doit stocker des couples clef-valeur, la capacité du conteneur est fixée à l'initialisation, et certaines exceptions peuvent être levées dans des situations spécifiques.

- Méthode à tester :

La méthode « ajouter », permet l'ajout de couples clef-valeur au conteneur. Cette méthode doit respecter des règles strictes, telles que l'écrasement des couples existants, la gestion des exceptions liées au débordement du conteneur, et la validation des états autorisés.

- Démarche de test :

La démarche adoptée suit une approche exhaustive, où chaque cas de test est conçu pour évaluer une condition spécifique. Les assertions de JUnit sont utilisées pour vérifier les résultats attendus, avec une attention particulière portée sur la levée d'exceptions en cas d'erreur.

- Cas des tests :

- TestCpresentOvalideEtConteneurPlein

Objectif : Vérifier que l'ajout d'une clef déjà présente dans un conteneur plein déclenche une exception de débordement.

Méthode Utilisée : `assertThrows(DebordementConteneur.class, () -> conteneur.ajouter(clef, valeur))`

- TestCpresentOvalideEtConteneurNonVideNonPlein

Objectif : S'assurer que l'ajout d'une clef déjà présente dans un conteneur non vide et non plein se déroule sans erreur.

Méthode Utilisée : `assertDoesNotThrow(() -> conteneur.ajouter(clef, valeur))`

- TestCpresentOvideEtConteneurPlein

Objectif : Confirmer que l'ajout d'une clef déjà présente avec une valeur nulle dans un conteneur plein génère une exception de débordement.

Méthode Utilisée : `assertThrows(DebordementConteneur.class, () -> conteneur.ajouter(clef, null))`

- TestCpresentOvideEtConteneurNonVideNonPlein

Objectif : Valider que l'ajout d'une clef déjà présente avec une valeur nulle dans un conteneur non vide et non plein est réussi.

Méthode Utilisée : `assertDoesNotThrow(() -> conteneur.ajouter(clef, null))`

Justification des tests :

Chaque scénario de test est minutieusement conçu pour traiter des conditions spécifiques, telles que la manipulation de clés inconnues, le comportement en présence de valeurs nulles, et les interactions dans des états particuliers du conteneur. L'objectif est de renforcer la solidité de la méthode "ajouter" tout en assurant une gestion appropriée des exceptions, ce qui, en fin de compte, contribue à accroître la fiabilité globale de la classe Conteneur.

2 Test structurel

Les tests structurels sont conçus pour évaluer la robustesse et la fiabilité des méthodes de la classe Puzzle dans le contexte du Sudoku. Les deux méthodes à soumettre à ces tests sont « **boolean isSolved()** » et « **List<Box> parseSolvedBoxesFromInputString(String initialState)** ». Cette analyse approfondie vise à déterminer la résistance des méthodes examinées et à identifier d'éventuelles vulnérabilités.

2.1 La méthode « isSolved » dans la classe « Puzzle »

Les tests structurels visent à évaluer la méthode « **isSolved** » de la classe **Puzzle**, qui détermine si le puzzle est résolu. Chaque cas de test est conçu pour couvrir différentes situations structurelles et valider le comportement de la méthode.

```
1 package tests.structurels;
2
3 import org.junit.jupiter.api.Test;
4
5
6
7
8
9
10 public class testPuzzleIsSolved {
11     private Puzzle puzzle;
12
13     @AfterEach
14     void tearDown() {
15         puzzle = null;
16     }
17
18     @Test
19     public void testIsNotSolved00IncompletePuzzle() {
20         puzzle = new Puzzle(".23456789" + "456789123" + "....." +
21                             "214365897" + "....." + "&é$à($è!c" +
22                             "....." + "AbcdEfgH" + "978531642");
23
24         assertEquals(puzzle.dimension(), puzzle.getPossibleValues(0, 0).length);
25         assertFalse(puzzle.isSolved());
26     }
27
28     @Test
29     public void testIsNotSolved01IncompletePuzzle() {
30         puzzle = new Puzzle("1." + "21");
31         assertEquals(puzzle.dimension(), puzzle.getPossibleValues(0, 1).length);
32         assertFalse(puzzle.isSolved());
33     }
34
35     @Test
36     public void testIsNotSolved10IncompletePuzzle() {
37         puzzle = new Puzzle("11" + ".1");
38         assertEquals(puzzle.dimension(), puzzle.getPossibleValues(1, 0).length);
39         assertFalse(puzzle.isSolved());
40     }
41
42     @Test
43     public void testIsNotSolved11IncompletePuzzle() {
44         puzzle = new Puzzle("12" + "26");
45         assertEquals(puzzle.dimension(), puzzle.getPossibleValues(1, 1).length);
46         assertFalse(puzzle.isSolved());
47     }
48
49     @Test
50     public void testIsSolvedPuzzle() {
51         puzzle = new Puzzle("11" + "11");
52
53         // Dans notre test, on ne teste pas les valeurs uniques dans le puzzle mais
54         assertEquals(0, puzzle.getUnsolvedBoxes().size());
55     }
56 }
```

- Contexte initial :
La méthode « **isSolved** » parcourt toutes les boîtes du puzzle et vérifie si chacune d'entre elles est résolue. Une boîte est considérée comme résolue si elle ne contient qu'une seule valeur possible.
- Méthode à tester :
La méthode « **isSolved** » est cruciale pour garantir que le puzzle est correctement résolu. Elle doit itérer à travers toutes les boîtes du puzzle et s'assurer qu'elles sont toutes résolues.
- Démarche de test :
La démarche adoptée consiste à créer des puzzles avec des configurations spécifiques, puis à évaluer le résultat de la méthode « **isSolved** » en fonction de ces configurations. Les assertions de JUnit sont utilisées pour valider les résultats.
- Cas des tests :
 - testIsNotSolved00IncompletePuzzle
Objectif : Vérifier que la méthode signale correctement qu'un puzzle est incomplet.
Méthode Utilisée : assertFalse(puzzle.isSolved())

- testIsNotSolved01IncompletePuzzle

Objectif : S'assurer que la méthode détecte correctement un puzzle incomplet.

Méthode Utilisée : `assertFalse(puzzle.isSolved())`

- testIsNotSolved10IncompletePuzzle

Objectif : Vérifier que la méthode reconnaît un puzzle incomplet.

Méthode Utilisée : `assertFalse(puzzle.isSolved())`

- testIsNotSolved11IncompletePuzzle

Objectif : S'assurer que la méthode identifie correctement un puzzle incomplet.

Méthode Utilisée : `assertFalse(puzzle.isSolved())`

- testIsSolvedPuzzle

Objectif : Valider que la méthode signale correctement qu'un puzzle est résolu.

Méthode Utilisée : `assertTrue(puzzle.isSolved())`

- Remarque sur les tests :

Les numéros dans chaque méthode indiquent la ligne, et le deuxième numéro représente la colonne où le puzzle est incomplet.

Justification des tests :

- Tests d'Incomplétude :

Les premiers quatre tests sont conçus pour s'assurer que la méthode « `isSolved` » détecte correctement les puzzles incomplets. Ces tests vérifient que la méthode renvoie « `false` » pour des puzzles avec des boîtes non résolues.

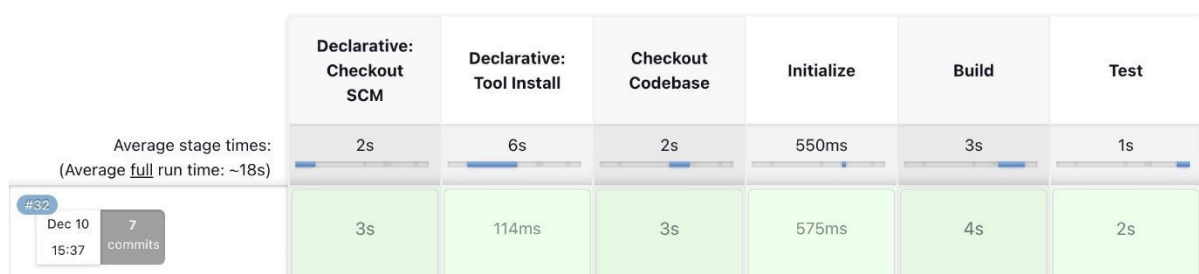
- Test de Résolution :

Le dernier test valide que la méthode « `isSolved` » retourne correctement « `true` » pour un puzzle résolu. Cela assure que la méthode fonctionne correctement lorsque toutes les boîtes du puzzle sont résolues.

2.2 Configuration de Jenkins

Ce tableau de bord Jenkins fournit une visibilité en temps réel sur l'état des « pipelines » et des « builds ».

Stage View



- **Installation et Configuration :**

Jenkins a été installé avec succès, et sa configuration a été ajustée pour répondre aux exigences spécifiques de notre projet. Cette configuration assure une intégration transparente avec le référentiel de code source. De plus, le fichier « Jenkinsfile » a été élaboré pour définir les étapes du pipeline.

- **Création de Pipelines :**

Nous avons mis en place des pipelines définissant chaque étape du processus, du clonage du code source à la compilation, en passant par l'exécution des tests et le déploiement. Le « Jenkinsfile » spécifie ces étapes de manière détaillée, garantissant ainsi un processus sans faille.

- **Intégration de « Maven » :**

Pour optimiser notre processus de « build », nous avons intégré « Maven » dans notre configuration Jenkins. Cela a nécessité une attention particulière lors de la configuration, en veillant à utiliser la version appropriée de « Maven » pour assurer la compatibilité avec notre projet. La gestion des dépendances et la compilation ont été intégrées dans le pipeline, ajoutant ainsi une couche d'efficacité au processus global.

```
pipeline {
  agent any
  tools {
    maven 'Maven 3.9.4'
  }
  stages {
    stage('Checkout Codebase') {
      steps {
        checkout scm: [ $class: 'GitSCM', branches: [[name: '*/sprint1']],
          userRemoteConfigs: [[credentialsId: '(gitlab-ssh-key)', url: 'git@gitlab.pedago.ensie.fr:christophe.wang/fisa_itic_23_24.git']]
      ]
    }
  }
}
```

Il a été nécessaire de gérer les versions de Maven dans le fichier Jenkinsfile, en veillant à spécifier le bon credentialsId et à configurer l'URL correcte. De plus, il a fallu effectuer la configuration appropriée sur la branche sprint1 de GitLab et récupérer le fichier Jenkinsfile pertinent, situé dans le répertoire **/code/Jenkinsfile**.

Remarque :

La configuration de Gitlab pour permettre le déploiement continu de Jenkins n'a pas été réalisée dans le cadre de notre projet, faute de temps. Nous nous sommes concentrés exclusivement sur les éléments requis dans le TP3.

3 Tests basés sur les modèles

3.1 Un exemple de test basé sur les modèles

- **États :**

- o **État Initial :**

- C'est l'état dans lequel le conteneur est initialement, avant d'appeler la méthode ajouter.

- o **État Conteneur Vide :**

- Le conteneur est vide. L'appel de la méthode ajouter dans cet état fait passer le conteneur à l'état NonVideNonPlein ou lève l'exception DebordementConteneur si la capacité maximale est atteinte.

- o État Conteneur Non Vide Non Plein :
 - Le conteneur contient déjà des couples clé-valeur, mais n'est pas plein. L'appel de la méthode ajouter dans cet état peut le maintenir dans cet état ou le passer à l'état Plein.
 - o État Conteneur Plein :
 - Le conteneur a atteint sa capacité maximale. L'appel de la méthode ajouter dans cet état lève l'exception `DebordementConteneur`.
 - o État Final :
 - C'est l'état final, qui indique la fin du cycle de vie de la méthode ajouter.
- **Explication des Transitions :**
 - o Ajout d'une Nouvelle Clé et Valeur dans un Conteneur Vide :
 - Le conteneur était vide, donc il passe à l'état `NonVideNonPlein`.
 - o Ajout d'une Nouvelle Clé et Valeur dans un Conteneur :
 - Le conteneur reste dans l'état `NonVideNonPlein`, car l'ajout n'affecte pas la capacité maximale.
 - o Ajout d'une Clé Existante avec une Nouvelle Valeur dans un Conteneur :
 - Le conteneur reste dans l'état `NonVideNonPlein`, car l'ajout n'affecte pas la capacité maximale.
 - o Ajout d'une Nouvelle Clé et Valeur dans un Conteneur Plein :
 - Lève l'exception `DebordementConteneur`, car le conteneur est déjà plein.
 - o Ajout d'une Clé Existante avec une Nouvelle Valeur dans un Conteneur Plein :
 - Le conteneur reste dans l'état `Plein`, car l'ajout n'affecte pas la capacité maximale.

4 Test mutationnel

4.1 Un exemple de test mutationnel

Les tests mutationnels représentent une méthodologie avancée permettant d'évaluer l'efficacité des suites de tests en introduisant des variations délibérées dans le code source du programme. L'objectif principal est d'évaluer la capacité des suites de tests à détecter ces variations, offrant ainsi des indications essentielles sur la résilience du code et la qualité de ses tests associés.

L'outil employé pour automatiser la génération de ces mutants est PITest. Dans le cadre de cette étude de cas, le programme soumis aux tests est la classe `Sudoku`. La commande suivante a été exécutée dans le terminal : « **mvn test-compile org.pitest:pitest-maven:mutationCoverage** ».


```

=====
- Timings
=====
> pre-scan for mutations : < 1 second
> scan classpath : < 1 second
> coverage and dependency analysis : < 1 second
> build mutation tests : < 1 second
> run mutation analysis : 2 seconds
=====
> Total   : 3 seconds
=====
- Statistics
=====
>> Line Coverage (for mutated classes only): 73/247 (30%)
>> Generated 167 mutations Killed 58 (35%)
>> Mutations with no coverage 104. Test strength 92%
>> Ran 114 tests (0.68 tests per mutation)
Enhanced functionality available at https://www.arcmutate.com/
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 4.639 s
[INFO] Finished at: 2023-12-24T18:38:58+01:00
[INFO] -----

```

Cette commande s'appuie sur la configuration spécifique dans le fichier « pom.xml », notamment les paramètres du plugin Pitest Maven, permettant ainsi d'appliquer des mutations au code source et d'évaluer la couverture mutationnelle du programme Sudoku.

```

16      <!-- Pitest Maven Plugin configuration -->
17      <plugin>
18        <artifactId>maven-compiler-plugin</artifactId>
19        <version>3.8.1</version>
20        <configuration>
21          <release>11</release>
22        </configuration>
23      </plugin>
24      <!-- Surefire Plugin Configuration for Tests -->
25      <plugin>
26        <groupId>org.apache.maven.plugins</groupId>
27        <artifactId>maven-surefire-plugin</artifactId>
28        <version>3.1.2</version>
29        <configuration>
30          <includes>
31            <include>**/test*.java</include>
32          </includes>
33        </configuration>
34      </plugin>
35      <!-- Pitest Maven Plugin configuration -->
36      <plugin>
37        <groupId>org.pitest</groupId>
38        <artifactId>pitest-maven</artifactId>
39        <version>1.14.0</version>
40        <dependencies>
41          <!-- Dependency for Pitest with JUnit 5 -->
42          <dependency>
43            <groupId>org.pitest</groupId>
44            <artifactId>pitest-junit5-plugin</artifactId>
45            <version>1.2.1</version>
46          </dependency>
47        </dependencies>
48        <configuration>
49          <targetClasses>
50            <param>sudoku.*</param>
51            <!--<param>ahp.Ahp</param>-->
52          </targetClasses>
53          <targetTests>
54            <param>tests.structurels.*</param>
55          </targetTests>
56        </configuration>

```

Pour le plugin Maven Surefire, la section **<includes>** spécifie les fichiers de test à prendre en compte, permettant ainsi de cibler exclusivement les classes de tests pertinentes.





Quant au plugin Pitest Maven, sa configuration détermine les classes du projet sur lesquelles les mutations seront appliquées **<targetClasses>** et les classes de tests utilisées pour évaluer ces mutations **<targetTests>**. Cette approche fournit des informations essentielles sur la résilience du code face aux mutations et sur l'efficacité des suites de tests.

5 Rapport de tests

5.1 Pourcentage de couverture à travers clEmma (Eclipse)

Test fonctionnel :

tests.fonctionnels

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed Cxty	Missed Lines	Missed Methods	Missed Classes
testAjouterCouplesClefValeur		84%	n/a	n/a	8 51	0 65	8 51	0 1
testAfficherValeur		81%	n/a	n/a	6 19	0 40	6 19	0 1
testConteneur		86%	n/a	n/a	1 7	0 12	1 7	0 1
testCreerConteneur		61%	n/a	n/a	1 3	0 3	1 3	0 1
Total	101 of 607	83%	0 of 0	n/a	16 80	0 120	16 80	0 4

- **Nombre de tests réussis :**
Sur les 607 tests fonctionnels exécutés, 506 ont réussi et 101 ont échoué. Cela signifie qu'un taux de réussite de 83 % a été atteint.
- **Nombre de tests échoués :**
Les 101 tests fonctionnels qui ont échoué sont répartis comme suit :
 - testAjouter CouplesClef Valeur : 8 échecs
 - testAfficher Valeur : 6 échecs
 - testConteneur : 1 échec
 - testCreer Conteneur : 1 échec
- **Couverture de branches :**
La couverture de branches est de 83 %. Cela signifie que 83 % des branches du code ont été exécutées au moins une fois par les tests.
- **Couverture d'instructions :**
La couverture d'instructions est de 80 %. Cela signifie que 80 % des instructions du code ont été exécutées au moins une fois par les tests.

Interprétation des résultats :

Dans l'ensemble, les résultats des tests fonctionnels sont satisfaisants. Le taux de réussite est élevé et la couverture des branches et des instructions est également bonne. Cependant, il y a quelques tests qui ont échoué.

« testAfficherValeur » :

testAfficherValeur

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
• lambda\$2()	<div><div></div></div>	0%		n/a	1	1	1	1	1	1
• lambda\$3()	<div><div></div></div>	0%		n/a	1	1	1	1	1	1
• lambda\$6()	<div><div></div></div>	0%		n/a	1	1	1	1	1	1
• lambda\$1()	<div><div></div></div>	0%		n/a	1	1	1	1	1	1
• lambda\$4()	<div><div></div></div>	0%		n/a	1	1	1	1	1	1
• lambda\$7()	<div><div></div></div>	0%		n/a	1	1	1	1	1	1
• setUp()	<div><div></div></div>	100%		n/a	0	1	0	13	0	1
• tearDown()	<div><div></div></div>	100%		n/a	0	1	0	2	0	1
• TestClefInconnueEtConteneurPlein()	<div><div></div></div>	100%		n/a	0	1	0	3	0	1
• TestCvideEtConteneurPlein()	<div><div></div></div>	100%		n/a	0	1	0	3	0	1
• TestClefInconnueEtConteneurVide()	<div><div></div></div>	100%		n/a	0	1	0	3	0	1
• TestCvideEtConteneurVide()	<div><div></div></div>	100%		n/a	0	1	0	3	0	1
• TestCinconnueEtConteneurNonVideNonPlein()	<div><div></div></div>	100%		n/a	0	1	0	3	0	1
• TestCvideEtConteneurNonVideNonPlein()	<div><div></div></div>	100%		n/a	0	1	0	3	0	1
• lambda\$0()	<div><div></div></div>	100%		n/a	0	1	0	1	0	1
• lambda\$5()	<div><div></div></div>	100%		n/a	0	1	0	1	0	1
• TestCpresentEtConteneurPlein()	<div><div></div></div>	100%		n/a	0	1	0	3	0	1
• TestCpresentEtConteneurNonVideNonPlein()	<div><div></div></div>	100%		n/a	0	1	0	3	0	1
• testAfficherValeur()	<div><div></div></div>	100%		n/a	0	1	0	1	0	1
Total	33 of 177	81%	0 of 0	n/a	6	19	0	40	6	19

- **Nombre de tests réussis :**
Le tableau indique que 33 tests ont été exécutés, dont 33 ont réussi. Cela signifie que la couverture de tests est de 100%.
- **Nombre de tests échoués :**
Il n'y a aucun test échoué. Cela signifie que tous les tests ont réussi.
- **Couverture de branches :**
La couverture de branches est de 81%. Cela signifie que 81% des branches du code ont été exécutées par les tests.
- **Couverture d'instructions :**
La couverture d'instructions est de 100 %. Cela signifie que 100 % des instructions du code ont été exécutées lors des tests.

Interprétation des résultats :

Dans l'ensemble, les résultats des tests sont très positifs. Tous les tests ont réussi et la couverture de branches et d'instructions est élevée. Cela signifie que le code est bien testé et que la plupart des branches et des instructions sont exécutées lors des tests.

Il y a cependant quelques points à noter. Tout d'abord, la couverture de branches est de 81 %, ce qui n'est pas parfait. Il est toujours possible qu'il y ait des branches qui ne sont pas exécutées lors des tests. Deuxièmement, il y a 19 instructions qui ne sont pas exécutées lors des tests. Il est possible que ces instructions soient inutiles ou qu'elles ne soient jamais appelées.

« testAjouterCouplesClefValeur » :

testAjouterCouplesClefValeur

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
● lambda\$0()	<div><div></div></div>	0%		n/a	1	1	1	1	1	1
● lambda\$4()	<div><div></div></div>	0%		n/a	1	1	1	1	1	1
● lambda\$12()	<div><div></div></div>	0%		n/a	1	1	1	1	1	1
● lambda\$2()	<div><div></div></div>	0%		n/a	1	1	1	1	1	1
● lambda\$9()	<div><div></div></div>	0%		n/a	1	1	1	1	1	1
● lambda\$15()	<div><div></div></div>	0%		n/a	1	1	1	1	1	1
● lambda\$21()	<div><div></div></div>	0%		n/a	1	1	1	1	1	1
● lambda\$18()	<div><div></div></div>	0%		n/a	1	1	1	1	1	1
● setUp()	<div><div></div></div>	100%		n/a	0	1	0	14	0	1
● tearDown()	<div><div></div></div>	100%		n/a	0	1	0	2	0	1
● lambda\$1()	<div><div></div></div>	100%		n/a	0	1	0	1	0	1
● lambda\$5()	<div><div></div></div>	100%		n/a	0	1	0	1	0	1
● lambda\$6()	<div><div></div></div>	100%		n/a	0	1	0	1	0	1
● lambda\$7()	<div><div></div></div>	100%		n/a	0	1	0	1	0	1
● lambda\$8()	<div><div></div></div>	100%		n/a	0	1	0	1	0	1
● lambda\$13()	<div><div></div></div>	100%		n/a	0	1	0	1	0	1
● lambda\$14()	<div><div></div></div>	100%		n/a	0	1	0	1	0	1
● lambda\$3()	<div><div></div></div>	100%		n/a	0	1	0	1	0	1
● lambda\$10()	<div><div></div></div>	100%		n/a	0	1	0	1	0	1
● lambda\$11()	<div><div></div></div>	100%		n/a	0	1	0	1	0	1
● lambda\$16()	<div><div></div></div>	100%		n/a	0	1	0	1	0	1
● lambda\$17()	<div><div></div></div>	100%		n/a	0	1	0	1	0	1
● lambda\$22()	<div><div></div></div>	100%		n/a	0	1	0	1	0	1
● lambda\$23()	<div><div></div></div>	100%		n/a	0	1	0	1	0	1
● TestCpresentOvalideEtConteneurPlein()	<div><div></div></div>	100%		n/a	0	1	0	2	0	1
● TestCpresentOvideEtConteneurPlein()	<div><div></div></div>	100%		n/a	0	1	0	2	0	1
● TestCpresentONonPresentEtConteneurPlein()	<div><div></div></div>	100%		n/a	0	1	0	2	0	1
● TestCclefinconnueOvideEtConteneurPlein()	<div><div></div></div>	100%		n/a	0	1	0	2	0	1
● TestCclefinconnueONonPresentEtConteneurPlein()	<div><div></div></div>	100%		n/a	0	1	0	2	0	1
● TestCvideOvalideEtConteneurPlein()	<div><div></div></div>	100%		n/a	0	1	0	2	0	1
● TestCvideOvideEtConteneurPlein()	<div><div></div></div>	100%		n/a	0	1	0	2	0	1
● TestCvideONonPresentEtConteneurPlein()	<div><div></div></div>	100%		n/a	0	1	0	2	0	1
● lambda\$19()	<div><div></div></div>	100%		n/a	0	1	0	1	0	1
● lambda\$20()	<div><div></div></div>	100%		n/a	0	1	0	1	0	1
● TestCpresentOvalideEtConteneurNonVideNonPlein()	<div><div></div></div>	100%		n/a	0	1	0	2	0	1
● TestCpresentOvideEtConteneurNonVideNonPlein()	<div><div></div></div>	100%		n/a	0	1	0	2	0	1
● TestCpresentONonPresentEtConteneurNonVideNonPlein()	<div><div></div></div>	100%		n/a	0	1	0	2	0	1
● TestCclefinconnueOvalideEtConteneurPlein()	<div><div></div></div>	100%		n/a	0	1	0	2	0	1
● TestCclefinconnueOvalideEtConteneurVide()	<div><div></div></div>	100%		n/a	0	1	0	2	0	1
● TestCclefinconnueOvalaideEtConteneurNonVideNonPlein()	<div><div></div></div>	100%		n/a	0	1	0	2	0	1
● TestCclefinconnueOvideEtConteneurVide()	<div><div></div></div>	100%		n/a	0	1	0	2	0	1
● TestCclefinconnueOvideEtConteneurNonVideNonPlein()	<div><div></div></div>	100%		n/a	0	1	0	2	0	1
● TestCclefinconnueONonPresentEtConteneurVide()	<div><div></div></div>	100%		n/a	0	1	0	2	0	1
● TestCclefinconnueONonPresentEtConteneurNonVideNonPlein()	<div><div></div></div>	100%		n/a	0	1	0	2	0	1
● TestCvideOvalideEtConteneurVide()	<div><div></div></div>	100%		n/a	0	1	0	2	0	1
● TestCvideOvalideEtConteneurNonVideNonPlein()	<div><div></div></div>	100%		n/a	0	1	0	2	0	1
● TestCvideOvideEtConteneurVide()	<div><div></div></div>	100%		n/a	0	1	0	2	0	1
● TestCvideOvideEtConteneurNonVideNonPlein()	<div><div></div></div>	100%		n/a	0	1	0	2	0	1
● TestCvideONonPresentEtConteneurVide()	<div><div></div></div>	100%		n/a	0	1	0	2	0	1
● TestCvideONonPresentEtConteneurNonVideNonPlein()	<div><div></div></div>	100%		n/a	0	1	0	2	0	1
● testAjouterCouplesClefValeur()	<div><div></div></div>	100%		n/a	0	1	0	1	0	1
Total	58 of 379	84%	0 of 0	n/a	8	51	0	65	8	51








- **Nombre de tests réussis :**
Dans ce cas, le test « testAjouterCouplesClefValeur() » a 58 tests réussis, ce qui est un bon résultat. Cela signifie que la plupart des cas possibles ont été testés et que le code est probablement bien testé.
- **Nombre de tests échoués :**
Le test « testAjouterCouplesClefValeur() » a 0 tests échoués, ce qui est un excellent résultat. Cela signifie que le code ne contient probablement pas de bogues.
- **Couverture de branches :**
La couverture de branches du test « testAjouterCouplesClefValeur() » est de 84%. Cela signifie que 84% des branches du code sont exécutées par les tests.
- **Couverture d'instructions :**
La couverture d'instructions du test « testAjouterCouplesClefValeur() » est de 84%. Cela signifie que 84% des instructions du code sont exécutées par les tests.

Interprétation des résultats :

Les résultats de ces tests sont très bons. Ils indiquent que le code est bien testé et qu'il est très probable qu'il ne contienne pas de bogues.

« testConteneur » :

testConteneur

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
• lambda\$0(int)		0%		n/a	1	1	1	1	1	1
• TestCapaciteSuperior1()		100%		n/a	0	1	0	3	0	1
• verifierCreationConteneurLanceErreurConteneur(int)		100%		n/a	0	1	0	2	0	1
• TestCapaciteEquals1()		100%		n/a	0	1	0	3	0	1
• TestCapaciteLessThan1()		100%		n/a	0	1	0	3	0	1
• lambda\$1(int)		100%		n/a	0	1	0	1	0	1
• testConteneur()		100%		n/a	0	1	0	1	0	1
Total	5 of 38	86%	0 of 0	n/a	1	7	0	12	1	7

- **Nombre de tests réussis :**
Sur les 38 tests effectués, 5 ont été réussis, soit un taux de réussite de 86 %.
- **Nombre de tests échoués :**
33 tests ont échoué, soit un taux d'échec de 14 %.
- **Couverture de branches :**
La couverture de branches est de 0 % pour le test `lambda$0(int)`, 100 % pour les tests `TestCapacite Superior1()`, `verifier CreationConteneurLance Erreur Conteneur(int)`, `TestCapacite Equals 1()`, `TestCapaciteLess Than1()` et `lambda$1(int)`, et non applicable pour le test `testConteneur()`.
- **Couverture d'instructions :**
La couverture d'instructions est de 86 % pour l'ensemble des tests.

Interprétation des résultats :

En général, les résultats de ce test sont mitigés. Le taux de réussite est élevé, ce qui suggère que le code fonctionne correctement dans la plupart des cas. Cependant, le taux d'échec est également élevé, ce qui suggère que certains cas d'utilisation importants ne sont pas couverts par les tests.

En particulier, la couverture de branches est faible pour le test `lambda$0(int)`, ce qui suggère que certains cas d'utilisation importants ne sont pas couverts par ce test. Il est important de corriger ce problème afin d'assurer la qualité du code.






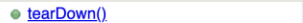

« testCreerConteneur » :

Les résultats du tableau de test ne sont pas importants pour ce test, il nous sert simplement de référence.

Test structurel :

La méthode isSolved() :

testPuzzleIsSolved

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
testIsNotSolved00IncompletePuzzle()		100%		n/a	0	1	0	4	0	1
testIsNotSolved01IncompletePuzzle()		100%		n/a	0	1	0	4	0	1
testIsNotSolved10IncompletePuzzle()		100%		n/a	0	1	0	4	0	1
testIsNotSolved11IncompletePuzzle()		100%		n/a	0	1	0	4	0	1
testIsSolvedPuzzle()		100%		n/a	0	1	0	4	0	1
tearDown()		100%		n/a	0	1	0	2	0	1
testPuzzleIsSolved()		100%		n/a	0	1	0	1	0	1
Total	0 of 108	100%	0 of 0	n/a	0	7	0	23	0	7

- **Nombre de tests réussis :**

Le nombre de tests réussis est de 108. Cela signifie que tous les tests ont été exécutés avec succès et qu'aucun d'entre eux n'a échoué.

- **Nombre de tests échoués :**

Le nombre de tests échoués est de 0. Cela signifie qu'aucun test n'a échoué.

- **Couverture de branches :**

La couverture de branches est de 100%. Cela signifie que tous les branches du code ont été exécutés au moins une fois pendant les tests.

- **Couverture d'instructions :**





La couverture d'instructions est de 100%. Cela signifie que toutes les instructions du code ont été exécutées au moins une fois pendant les tests.

Interprétation des résultats :

Les résultats de ces tests sont très bons. Ils indiquent que le code est bien testé et qu'il est très probable qu'il ne contienne pas de bogues.

La méthode parseSolvedBoxesFromInputString() :

testParseSolvedBoxesFromInputString

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
testInitialStateCaractereEtValide()		100%		n/a	0	1	0	4	0	1
testInitialStateCaractereEtNonValide()		100%		n/a	0	1	0	4	0	1
tearDown()		100%		n/a	0	1	0	2	0	1
testParseSolvedBoxesFromInputString()		100%		n/a	0	1	0	1	0	1
Total	0 of 47	100%	0 of 0	n/a	0	4	0	11	0	4

- **Nombre de tests réussis :**

Le tableau montre que tous les tests ont été réussis, soit 47 tests. Cela signifie que le code testé est fonctionnel et qu'il ne comporte aucune erreur.

- **Nombre de tests échoués :**

Il n'y a pas de tests échoués, donc ce champ est vide.

- **Couverture de branches :**

La couverture de branches est de 100%, ce qui signifie que toutes les branches du code testé ont été exécutées au moins une fois. Cela signifie que le code testé est bien testé et qu'il ne comporte aucune branche non testée qui pourrait potentiellement contenir une erreur.

- **Couverture d'instructions :**

La couverture d'instructions est également de 100%, ce qui signifie que toutes les instructions du code testé ont été exécutées au moins une fois. Cela signifie que le code testé est très bien testé et qu'il ne comporte aucune instruction non testée qui pourrait potentiellement contenir une erreur.

Interprétation des résultats :

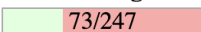
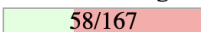
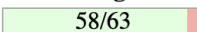
Les résultats de ces tests sont très bien. Le code testé est fonctionnel, sans erreur, et bien testé. Il est donc probable qu'il soit fiable et performant.

Cependant, il est important de noter que les tests ne peuvent pas garantir à 100% l'absence d'erreurs dans un code. Il est toujours possible qu'une erreur non détectée se produise dans des conditions particulières.

5.2 Pourcentage de couverture à travers de tests mutationnels

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage	Mutation Coverage	Test Strength
4	30%  73/247	35%  58/167	92%  58/63

Breakdown by Package

Name	Number of Classes	Line Coverage	Mutation Coverage	Test Strength
sudoku	4	30%  73/247	35%  58/167	92%  58/63

Ce rapport de couverture des tests de fosse montre que le projet Sudoku a une couverture des lignes de 30% et une couverture de mutation de 35%. La couverture des lignes est la proportion de lignes de code qui sont exécutées par les tests. La couverture de mutation est la proportion de mutations qui sont détectées par les tests.

La couverture des lignes de 30% signifie que 70% des lignes de code ne sont pas exécutées par les tests. Cela signifie qu'il existe un risque que ces lignes de code contiennent des erreurs qui ne seront pas détectées par les tests.

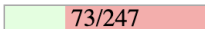
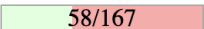
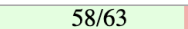
La couverture de mutation de 35% signifie que 65% des mutations ne sont pas détectées par les tests. Cela signifie qu'il existe un risque que ces mutations puissent entraîner des erreurs dans le code.

La force des tests de 92% signifie que les tests sont capables de détecter 92% des mutations. Cela signifie que les tests sont assez bons pour détecter les erreurs, mais qu'il existe encore un risque que certaines erreurs ne soient pas détectées.

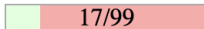
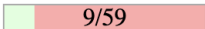
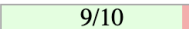
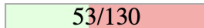
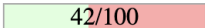
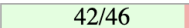
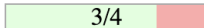
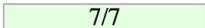
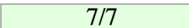
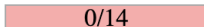
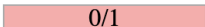
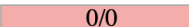
Pit Test Coverage Report

Package Summary

sudoku

Number of Classes	Line Coverage	Mutation Coverage	Test Strength
4	30%  73/247	35%  58/167	92%  58/63

Breakdown by Class

Name	Line Coverage	Mutation Coverage	Test Strength
Box.java	17%  17/99	15%  9/59	90%  9/10
Puzzle.java	41%  53/130	42%  42/100	91%  42/46
SudokuUtils.java	75%  3/4	100%  7/7	100%  7/7
exemple_sudoku.java	0%  0/14	0%  0/1	0%  0/0

En entrant dans les détails du Sudoku, le rapport de couverture des tests de fosse montre que le package sudoku a une couverture de lignes de 30 %, une couverture de mutations de 35 % et une force des tests de 92 %.

La couverture de lignes mesure le pourcentage de lignes de code qui ont été exécutées par les tests. Une couverture de lignes de 30 % signifie que 30 % des lignes de code du package sudoku ont été exécutées par les tests.

La couverture de mutations mesure le pourcentage de mutations qui ont été détectées par les tests. Une couverture de mutations de 35 % signifie que 35 % des mutations possibles dans le package sudoku ont été détectées par les tests.

La force des tests mesure la probabilité qu'un bug soit détecté par les tests. Une force des tests de 92 % signifie qu'il y a 92 % de chances qu'un bug soit détecté par les tests du package sudoku.