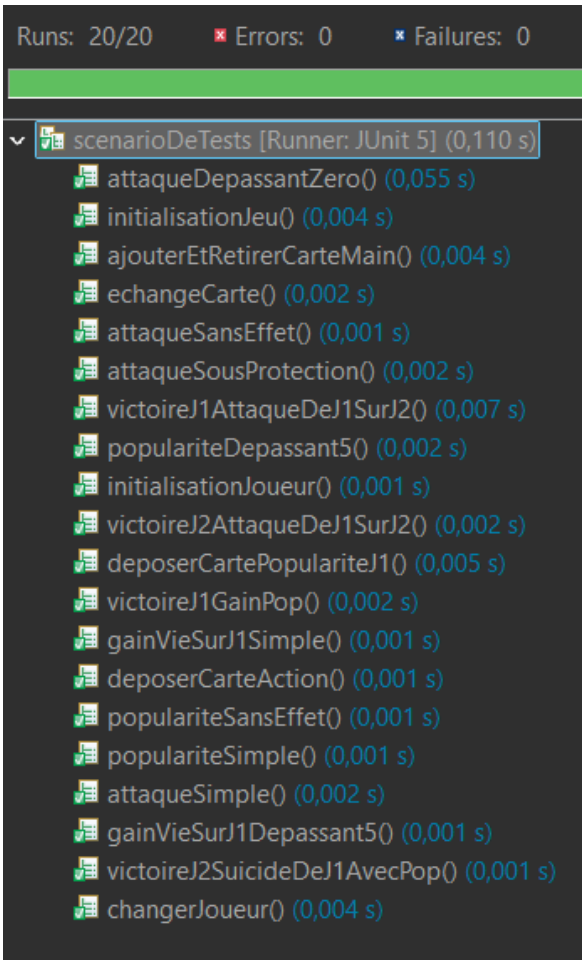
Rapport des tests de mutation

Ce document présente en premier lieu, une série de tests pertinents pour le bon fonctionnement d'une partie de jeu, puis en second lieu des mutations sur le code source et leur impact sur ce dernier en nous basant sur les résultats de la série de tests.

Série de test

Voici ci-contre l'affichage JUnit du fichier scenarioDeTest.java avec 20 tests pertinents.

Tous les tests passent jusqu'à présent avant l'arrivée des mutations.



Mutation_01 : changement d'opérateur booléen sur la class Jeu

Changement d'un *ou logique* avec un *et logique* dans la condition de la fonction giveJoueurGagnant de la class Jeu.

Code original :

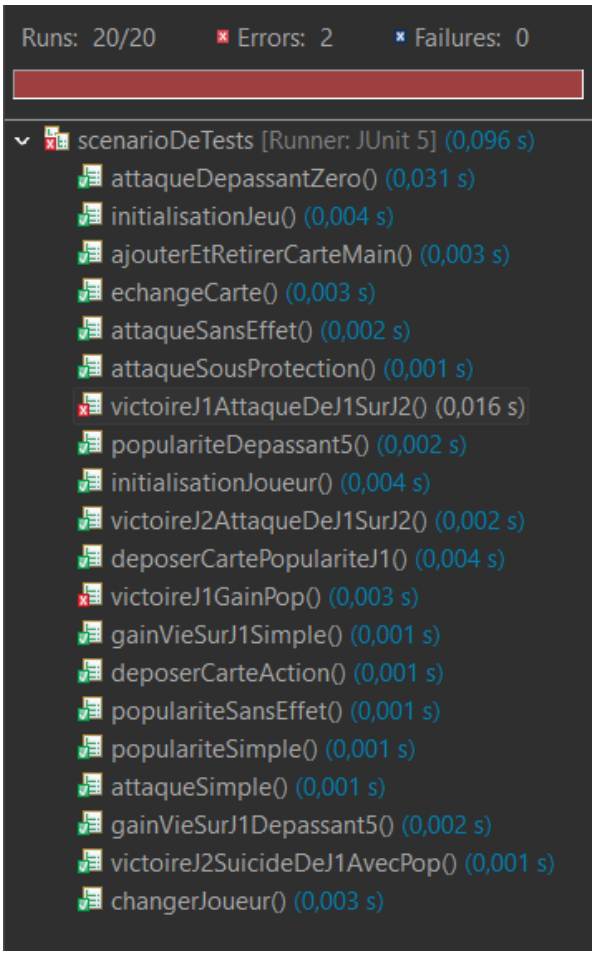
```
public Joueur giveJoueurGagnant() {
    if (joueur1.aGagne() || joueur2.aPerdu()) {
        return joueur1;
    } else if (joueur2.aGagne() || joueur1.aPerdu()) {
        return joueur2;
    }

    return null;
}
```

Code mutée :

```
public Joueur giveJoueurGagnant() {
    if (joueur1.aGagne() && joueur2.aPerdu()) {
        return joueur1;
    } else if (joueur2.aGagne() || joueur1.aPerdu()) {
        return joueur2;
    }

    return null;
}
```



Résultat obtenu :

La fonction giveJoueurGagnant renvoie donc null au lieu du joueur, cela cause une NullPointerException uniquement si le joueur 1 est censé gagner.

2 tests sur 20 ne passent pas, le mutant est donc tué.

Mutation_02 : changement d'opérateur booléen sur la class Jeu

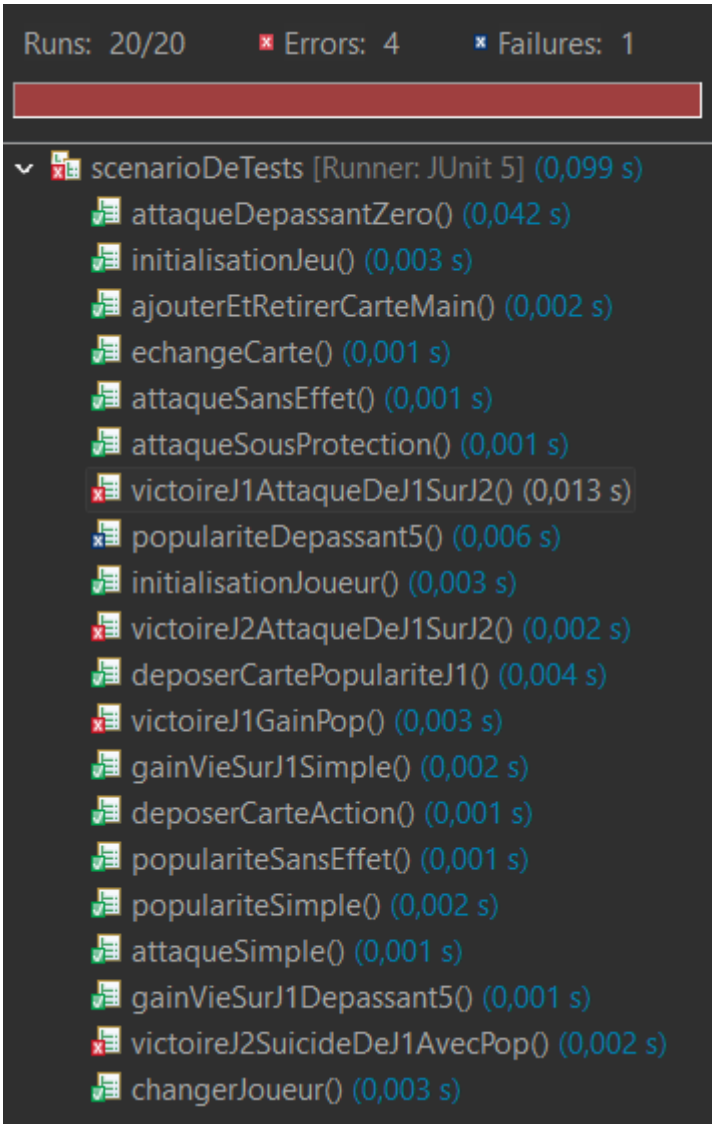
Changement des *"ou logique"* avec des *"et logique"* dans la méthode giveJoueurGagnant de ma class Jeu.

Code original :

```
68    public Joueur giveJoueurGagnant() {
69        if (joueur1.aGagne() || joueur2.aPerdu()) {
70            return joueur1;
71        } else if (joueur2.aGagne() || joueur1.aPerdu()) {
72            return joueur2;
73        }
74
75        return null;
76    }
```

Code muté :

```
68    public Joueur giveJoueurGagnant() {
69        if (joueur1.aGagne() && joueur2.aPerdu()) {
70            return joueur1;
71        } else if (joueur2.aGagne() && joueur1.aPerdu()) {
72            return joueur2;
73        }
74
75        return null;
76    }
```



Résultat obtenu :

La fonction giveJoueurGagnant renvoie donc null au lieu du joueur, cela cause une NullPointerException mais aussi pour le joueur 2 cette fois ci.

4 tests sur 20 ne passent pas, le mutant est donc tué.

Mutation_03 : changement d'opérateur booléen sur la class Jeu

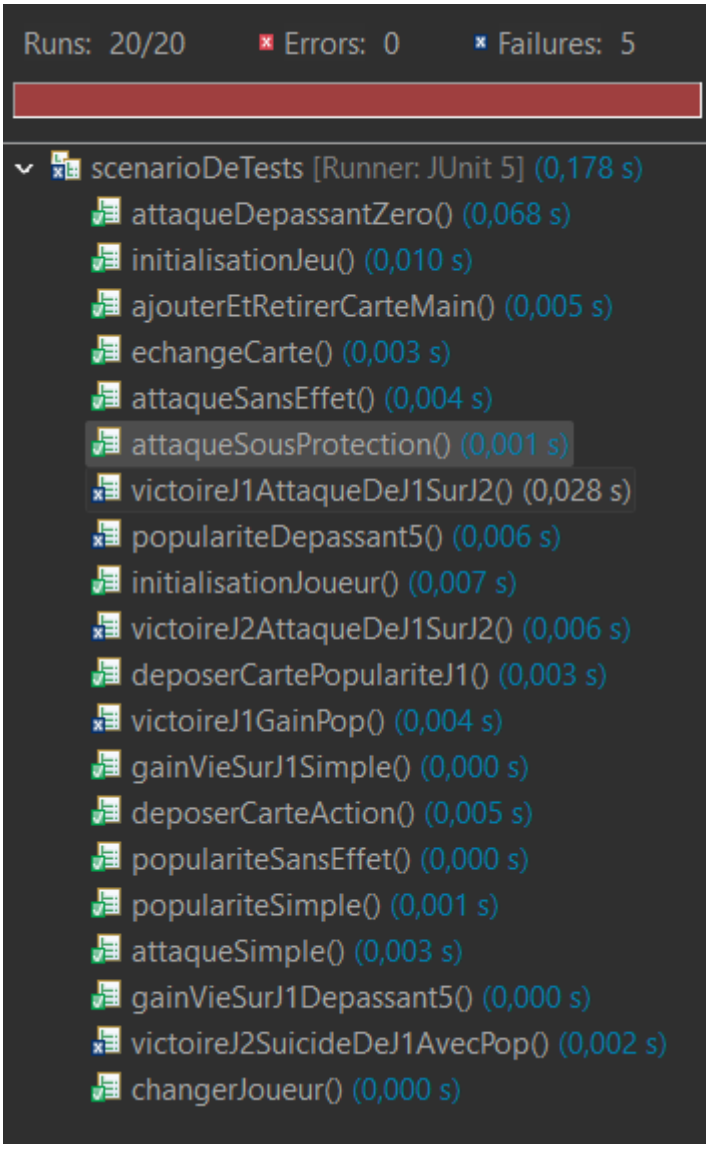
Changement des "*ou logique*" avec des "*et logique*" dans la condition de la fonction verifierFinPartie dans la class Jeu.

Code original :

```
61    public boolean verifierFinPartie() {
62        return joueur1.aGagne()
63                || joueur1.aPerdu()
64                || joueur2.aGagne()
65                || joueur2.aPerdu();
66    }
```

Code muté :

```
61    public boolean verifierFinPartie() {
62        return joueur1.aGagne()
63                && joueur1.aPerdu()
64                && joueur2.aGagne()
65                && joueur2.aPerdu();
66    }
```



Résultat obtenu :

La fonction gagnerVie renvoie donc toujours false pour savoir si la partie est finie. Une partie pourra se dérouler mais techniquement jamais se terminer au niveau logiciel, les utilisateurs peuvent tout de même jouer une partie.

Tous les tests passent sans erreur, 5 tests présentent une failure.

Le mutant est tué.

Mutation_04 : changement d'opérateur arithmétiques sur la class Joueur

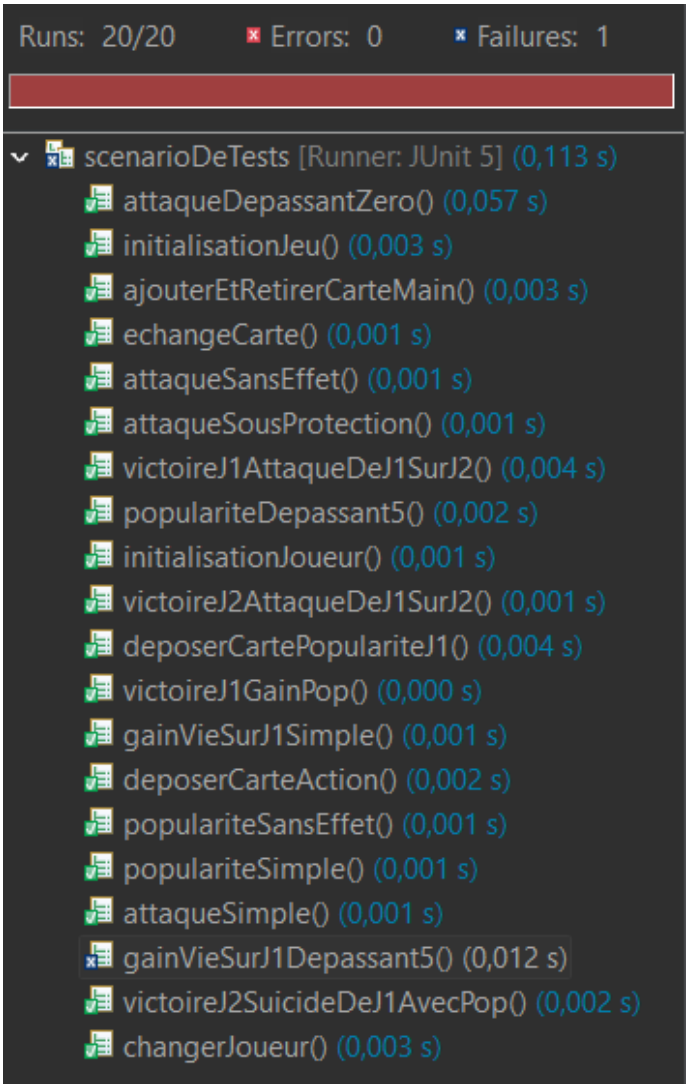
Changement d'un + en - dans la fonction gagnerVie de la class Joueur à la ligne 72.

Code original :

```
public void gagnerVie(int vie) {
    if (indiceVie + vie <= MAXVIE) {
        this.indiceVie += vie;
    }
    else {
        this.indiceVie = MAXVIE;
    }
}
```

Code muté :

```
public void gagnerVie(int vie) {
    if (indiceVie - vie <= MAXVIE) {
        this.indiceVie += vie;
    }
    else {
        this.indiceVie = MAXVIE;
    }
}
```



Résultat obtenu :

La fonction gagnerVie donne des points de vie même au lieu d'en faire perdre. Une partie pourra se dérouler mais les règles du jeu ne seront pas respectées et les joueurs ne pourront gagner uniquement par le gain de popularité.

Tous les tests passent sans erreur, un test présente une failure.

Le mutant est tué.

Mutation_05 : changement d'opérateur arithmétiques sur la class Joueur

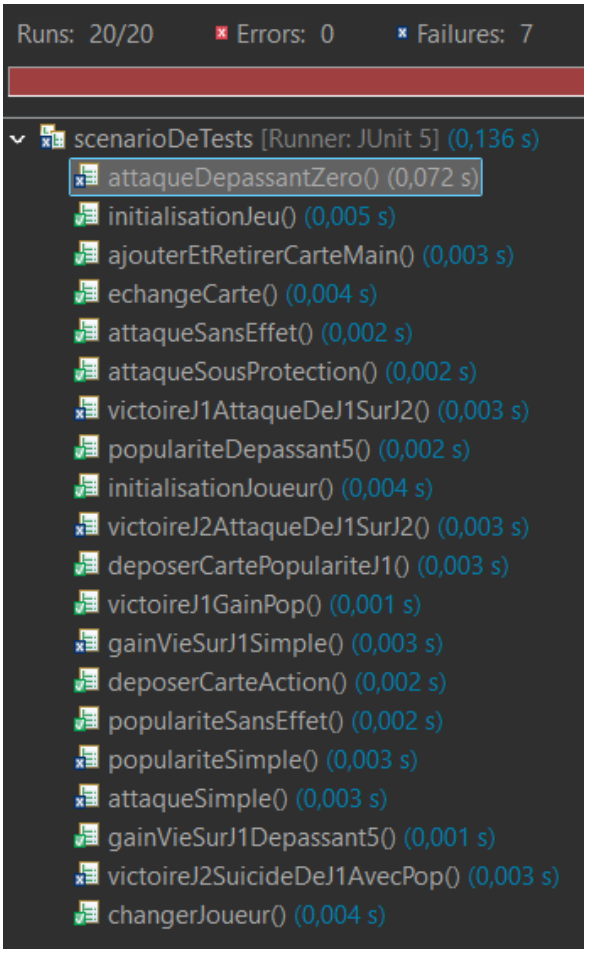
Changement d'un - en + dans la fonction gagnerVie de la class Joueur à la ligne 73.

Code original :

```
61    public void perdreVie(int vie){
62        if (indiceVie - vie >= MINVIE){
63            this.indiceVie -= vie;
64        }
65        else {
66            this.indiceVie = MINVIE;
67        }
68    }
```

Code muté :

```
61    public void perdreVie(int vie){
62        if (indiceVie - vie >= MINVIE){
63            this.indiceVie += vie;
64        }
65        else {
66            this.indiceVie = MINVIE;
67        }
68    }
```



Résultat obtenu :

La fonction perdreVie donne des points de vie même si le montant de vie final dépassera la vie max du joueur. Une partie pourra se dérouler mais les règles du jeu ne seront pas respectées.

Tous les tests passent sans erreur, un test présente une failure.

Le mutant est tué.

Mutation_06 : changement d'opérateur arithmétiques sur la class Joueur

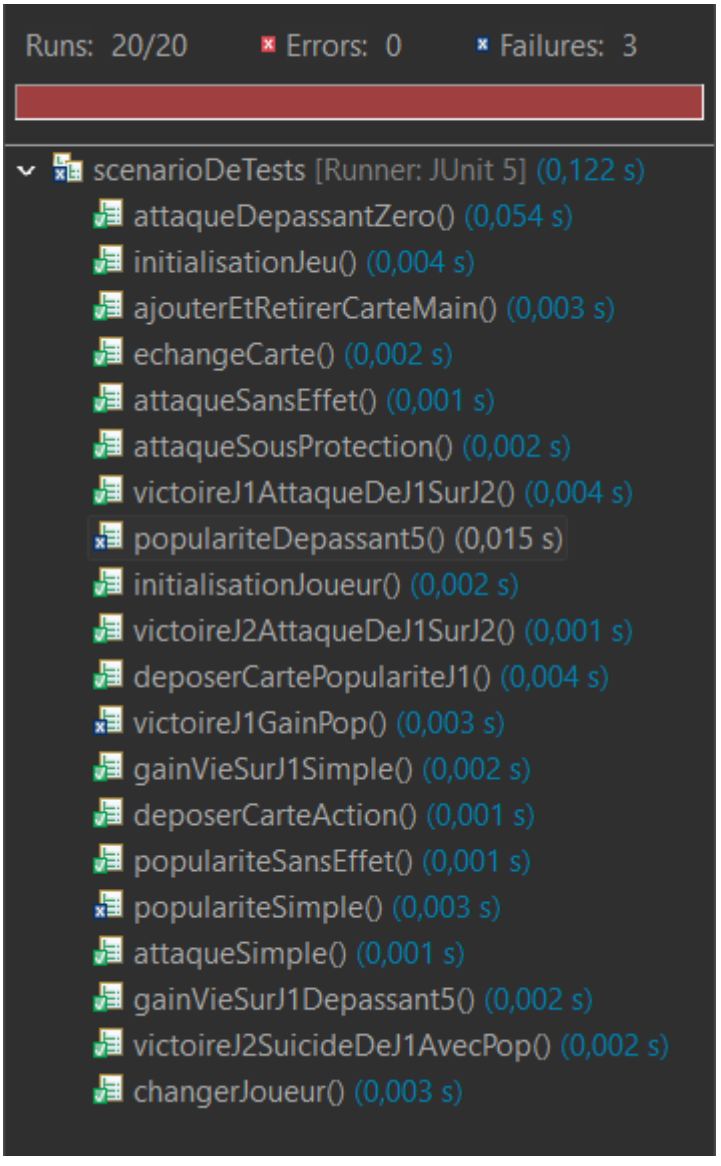
Changement d'un + en - dans la fonction gagnerPop de la class Joueur à la ligne 82.

Code original :

```
81    public void gagnerPop(int popularite){
82        if (indicePopularite + popularite <= MAXPOP) {
83            this.indicePopularite += popularite;
84        }
85        else{
86            this.indicePopularite = MAXPOP;
87        }
88    }
```

Code muté :

```
81    public void gagnerPop(int popularite){
82        if (indicePopularite - popularite <= MAXPOP) {
83            this.indicePopularite += popularite;
84        }
85        else{
86            this.indicePopularite = MAXPOP;
87        }
88    }
```



Résultat obtenu :

La fonction gagnerPop donne des points de popularité même si le montant de popularité final dépassera la popularité max du joueur. Une partie pourra se dérouler mais les règles du jeu ne seront pas respectées.

Tous les tests passent sans erreur, 3 tests présentent une failure.

Le mutant est tué.

Mutation_07 : changement d'opérateur relationnels sur la class Joueur

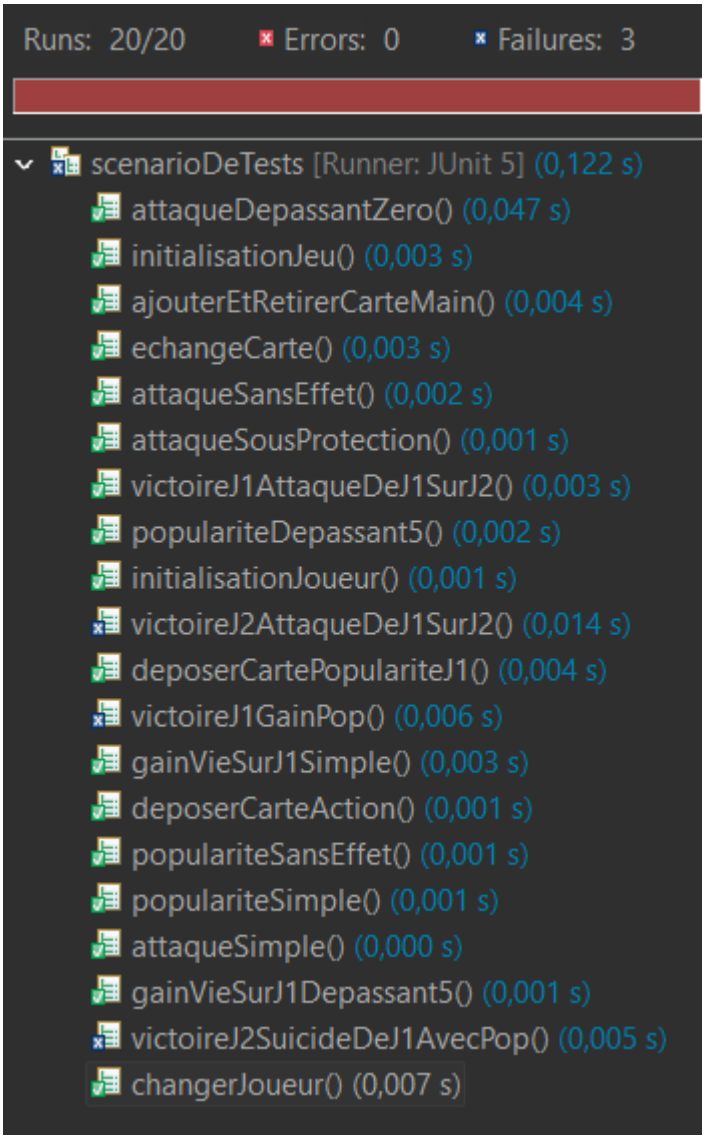
Changement de "==" par "!=" dans la fonction aGagne de la class Joueur à la ligne 92.

Code original :

```
91    public boolean aGagne(){
92        return this.indicePopularite == MAXPOP;
93    }
```

Code muté :

```
91    public boolean aGagne(){
92        return this.indicePopularite != MAXPOP;
93    }
```



Résultat obtenu :

La fonction aGagne ne renvoie donc jamais le bon booléen ce qui n'est pas un problème pour la fonction verifierFinPartie mais un problème pour la fonction giveJoueurGagnant qui donnera le mauvais joueur. Une partie pourra donc se dérouler sans soucis jusqu'au moment des résultats.

Tous les tests passent sans erreur, 3 tests présentent une failure.

Le mutant est tué.

Mutation_08 : changement d'opérateur relationnels sur la class Joueur

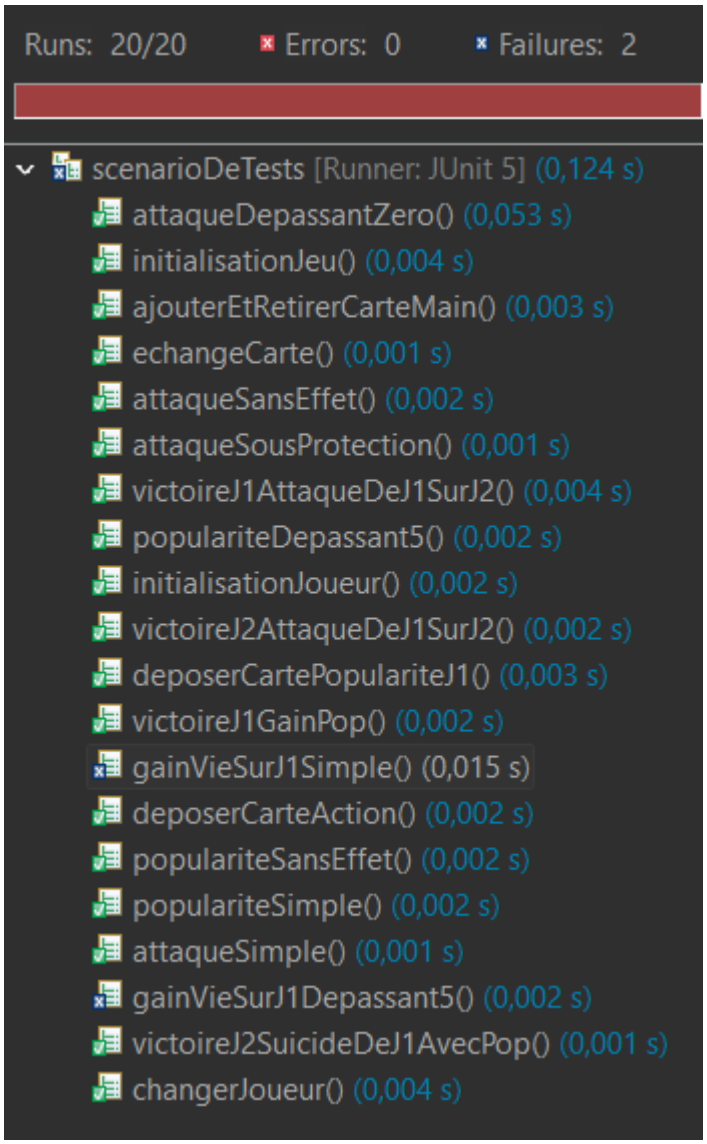
Changement de "<=" par ">" dans la fonction gagnerVie de la class Joueur à la ligne 72.

Code original :

```
71    public void gagnerVie(int vie) {
72        if (indiceVie + vie <= MAXVIE) {
73            this.indiceVie += vie;
74        }
75        else {
76            this.indiceVie = MAXVIE;
77        }
78    }
```

Code muté :

```
71    public void gagnerVie(int vie) {
72        if (indiceVie + vie > MAXVIE) {
73            this.indiceVie += vie;
74        }
75        else {
76            this.indiceVie = MAXVIE;
77        }
78    }
```



Résultat obtenu :

La fonction gagnerVie va faire dépasser le max de points de vie aux joueurs ce qui pourrait rendre des parties interminables. Une partie pourra donc se dérouler mais le principe de gain de vie ne sera pas fonctionnel et va permettre de dépasser le seuil de points de vie maximum.

Tous les tests passent sans erreur, 2 tests présentent une failure.

Le mutant est tué.

Mutation_09 : changement d'opérateur relationnels sur la class Joueur

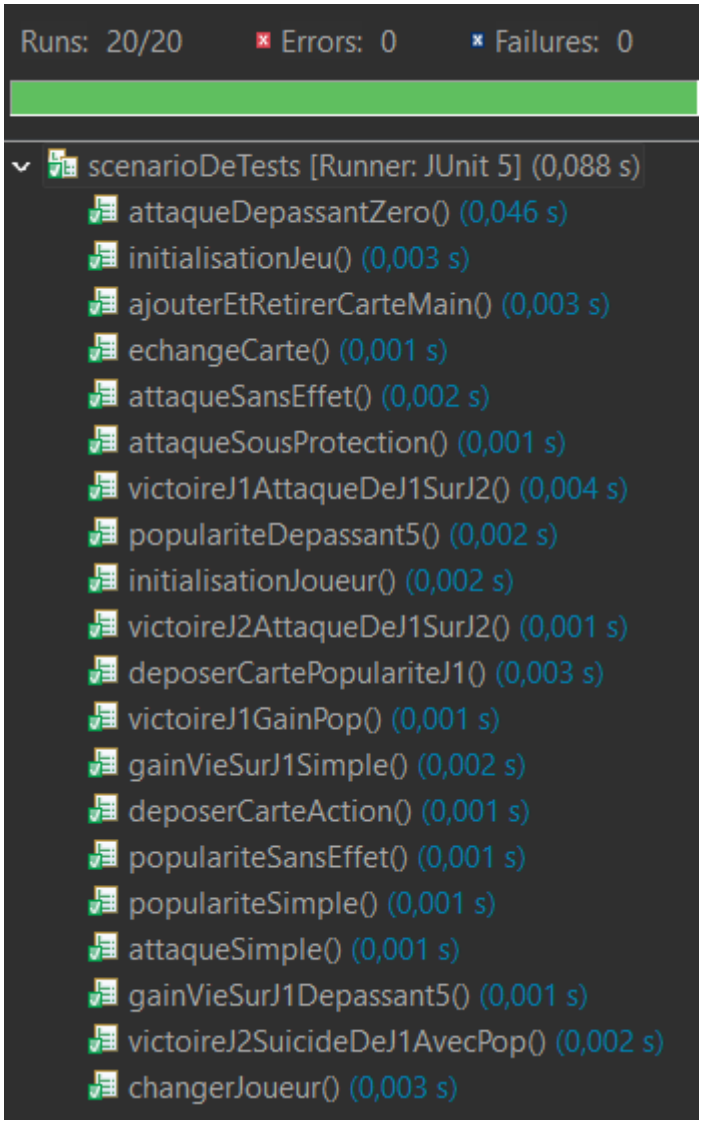
Changement de "==" par "!=" dans la condition de la ligne 53.

Le code original :

```
51    @Override
52    public boolean equals(Object o) {
53        if (this == o) return true;
54        if (o == null || getClass() != o.getClass())
55        Carte c = (Carte) o;
56
57        return this.nom.equals(c.getNom()) && this.d
58    }
```

Code muté :

```
51    @Override
52    public boolean equals(Object o) {
53        if (this != o) return true;
54        if (o == null || getClass() != o.getClass()) return false;
55        Carte c = (Carte) o;
56
57        return this.nom.equals(c.getNom()) && this.description.equ
58    }
```



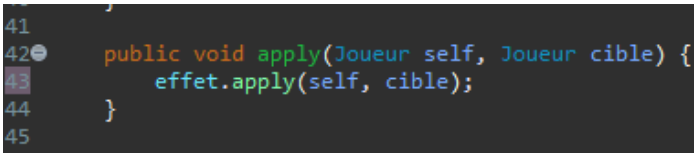
Résultat obtenu :

Tous les tests passent, le mutant survie car les tests sont incomplets pour le moment, il manque un test qui cible cette faute.

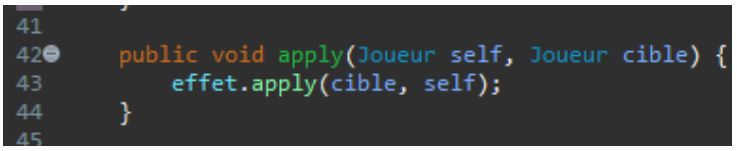
Mutation_10 : changement d'une variable sur la class Carte

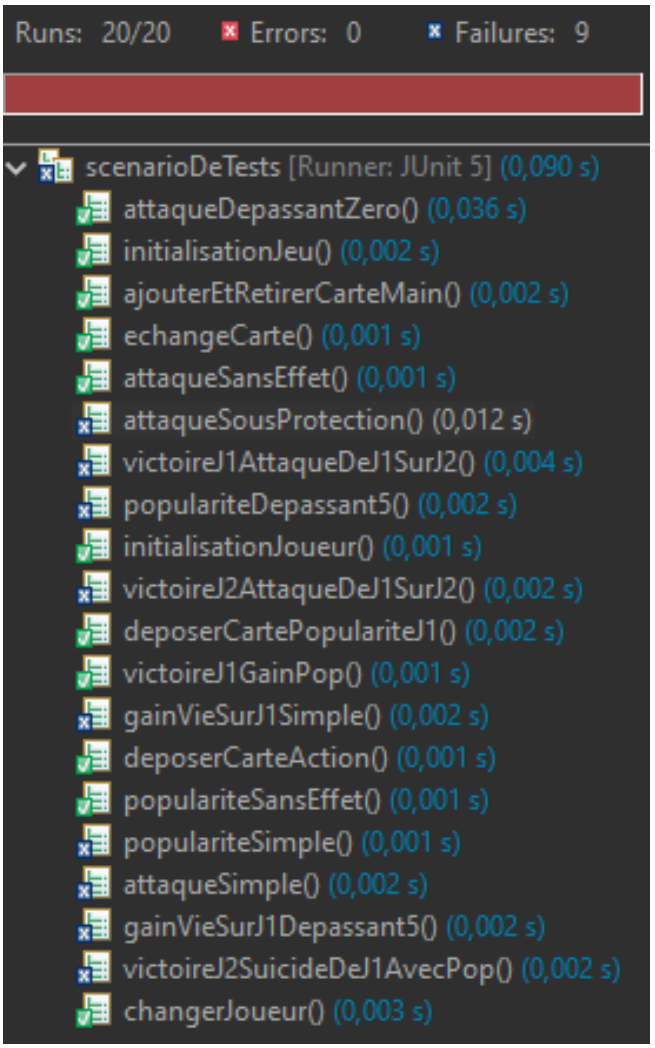
Échange de la variable "self" avec la variable "cible" dans la fonction apply de la class Carte à la ligne 43.

Code original :



Code muté :





Résultat obtenu :

La fonction apply va donc appliquer les effets d'une carte sur le lanceur plutôt que sur l'adversaire. Une partie pourra se dérouler mais elle n'aura aucun sens.

Tous les tests passent sans erreur, 9 tests présentent une failure.

Le mutant est tué.

Mutation_11 : effacement d'une instruction dans la class Joueur

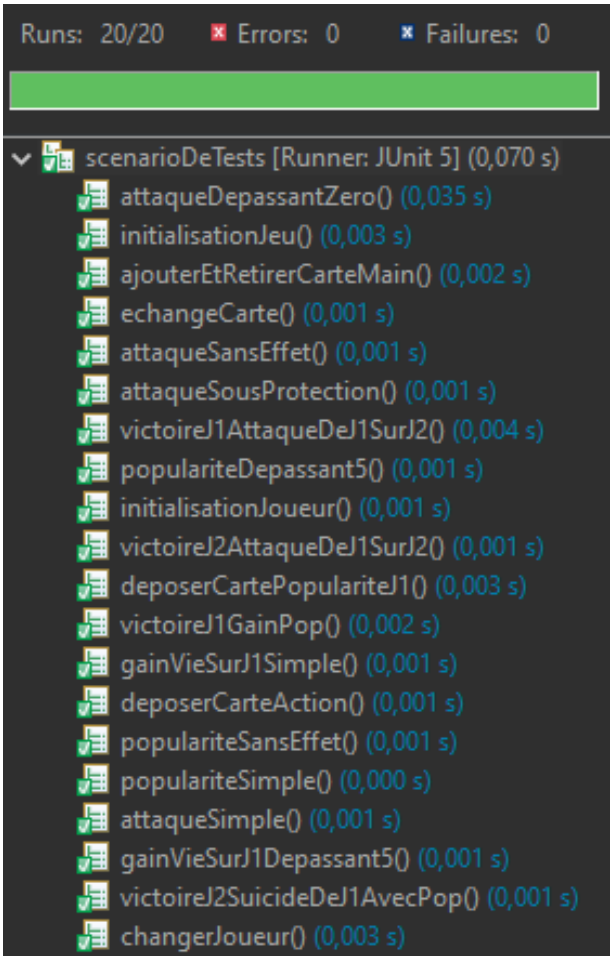
Effacement de l'instruction ligne 121 de la fonction carteHasard de la class Joueur.

Code original :

```
116
117     public Carte carteHasard(int nbCartes){
118         Random randomNum = new Random();
119         int nb = randomNum.nextInt(nbCartes);
120         Carte c = this.mainJoueur.getCartes().get(nb);
121         this.mainJoueur.getCartes().remove(nb);
122         return c;
123     }
124
```

Code muté :

```
116
117     public Carte carteHasard(int nbCartes){
118         Random randomNum = new Random();
119         int nb = randomNum.nextInt(nbCartes);
120         Carte c = this.mainJoueur.getCartes().get(nb);
121         return c;
122     }
123
```



Résultat obtenu :

La fonction carteHasard ne va donc pas enlever la carte qui devrait être donnée à l'adversaire. Une partie peut donc se dérouler sans problèmes à part l'incohérence au niveau des échanges de cartes.

Tous les tests passent sans erreur.

Le mutant survit.

Mutation_12 : changement de visibilité d'une méthode dans la class Carte

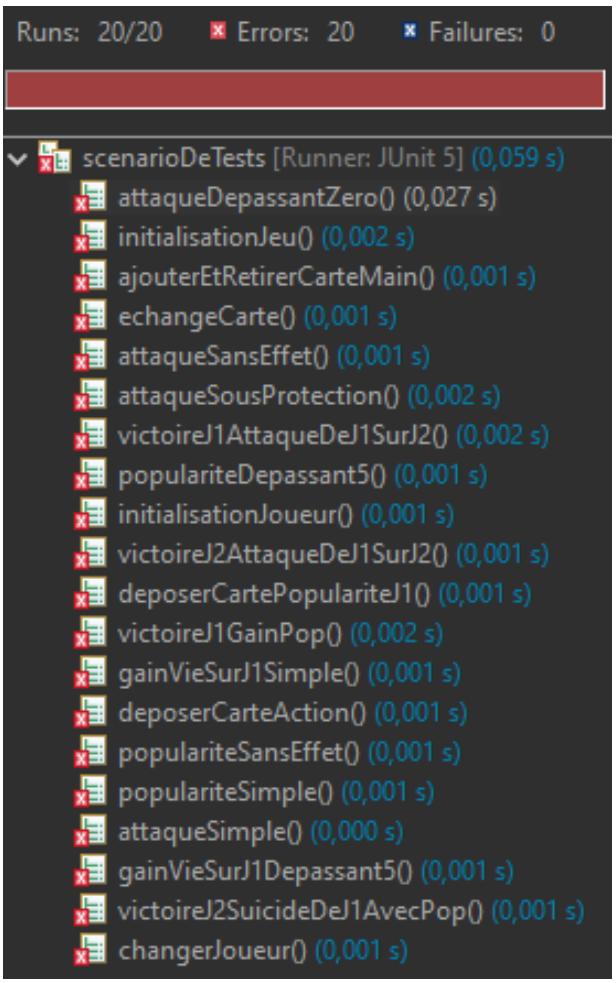
Changement de la visibilité de la méthode setEffet dans la class Carte, la passant de publique et privée.

Code original :

```
21
22      public void setEffet(EffetCarte effet) {
23          this.effet = effet;
24      }
```

Code muté :

```
21
22      private void setEffet(EffetCarte effet) {
23          this.effet = effet;
24      }
```



Résultat obtenu :

La fonction setEffet ne pourra jamais être appelée. Jouer une carte ne sera donc plus possible.

Aucun test ne passe.

Le mutant est tué.

Conclusion :

Mutant	**Impact**
Mutant_01	Fort
Mutant_02	Fort
Mutant_03	Moyen
Mutant_04	Fort
Mutant_05	Fort
Mutant_06	Moyen
Mutant_07	Moyen
Mutant_08	Fort
Mutant_09	Faible
Mutant_10	Fort
Mutant_11	Faible
Mutant_12	Très fort