TP – Tutoriel BlueJ

L’Hôtel des Murmures

Djena Haddar – Nathan Halioua
MIDO – Agilité – Tests unitaires

Introduction

L’Hôtel des Murmures accueille depuis des années des voyageurs un peu particuliers. Certaines chambres sont hantées, d’autres changent mystérieusement de prix, et les repas servis la nuit semblent parfois venir d’un autre monde.

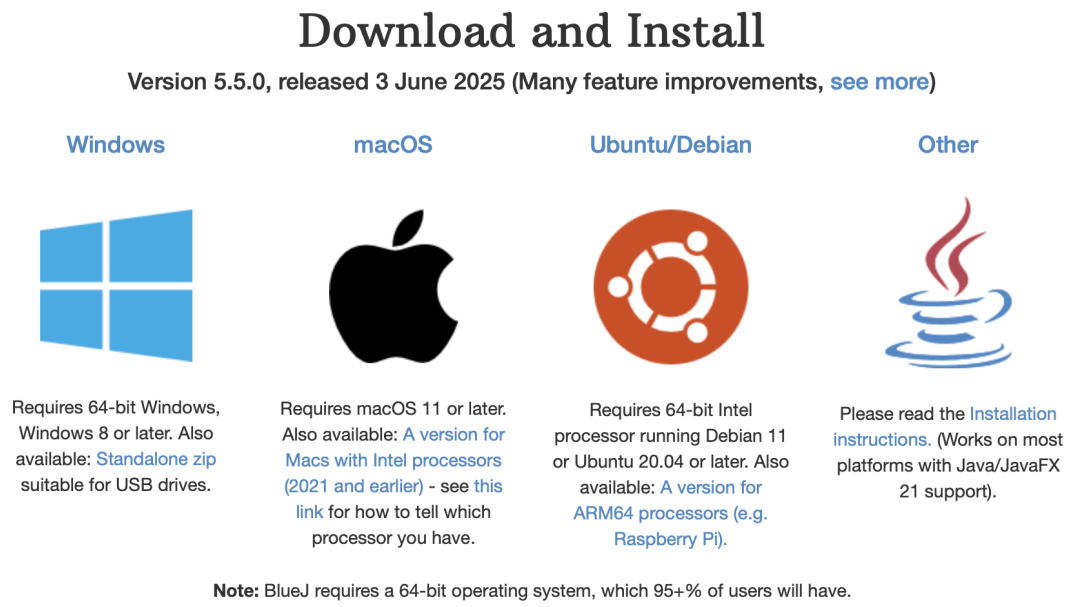
Ce tutoriel raconte la vie de cet hôtel à travers la programmation orientée objet. L’objectif n’est pas d’expliquer des concepts théoriques, mais de les faire apparaître en manipulant des objets, en observant leur comportement et en les testant.

La classe centrale de notre histoire est la classe **Hotel**. Elle donnera le ton à tout le projet.

I. Première itération – Prise en main avec BlueJ

1. Téléchargement et installation de BlueJ

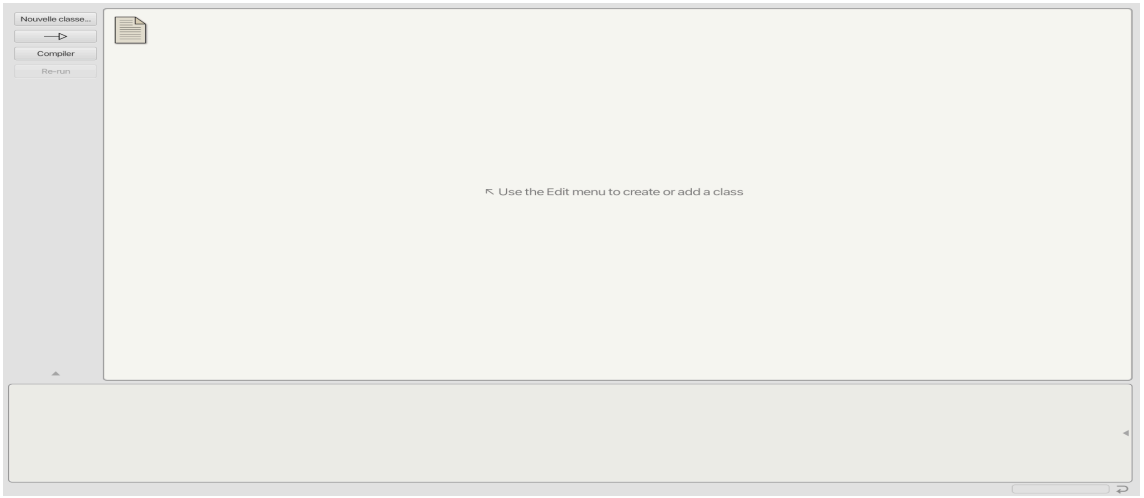
BlueJ est téléchargé depuis le site officiel (http://www.bluej.org) puis installé sur la machine.



---

2. Création du projet

Un nouveau projet BlueJ est créé sous le nom `HotelDesMurmures`.



---

3. Création de la classe fétiche

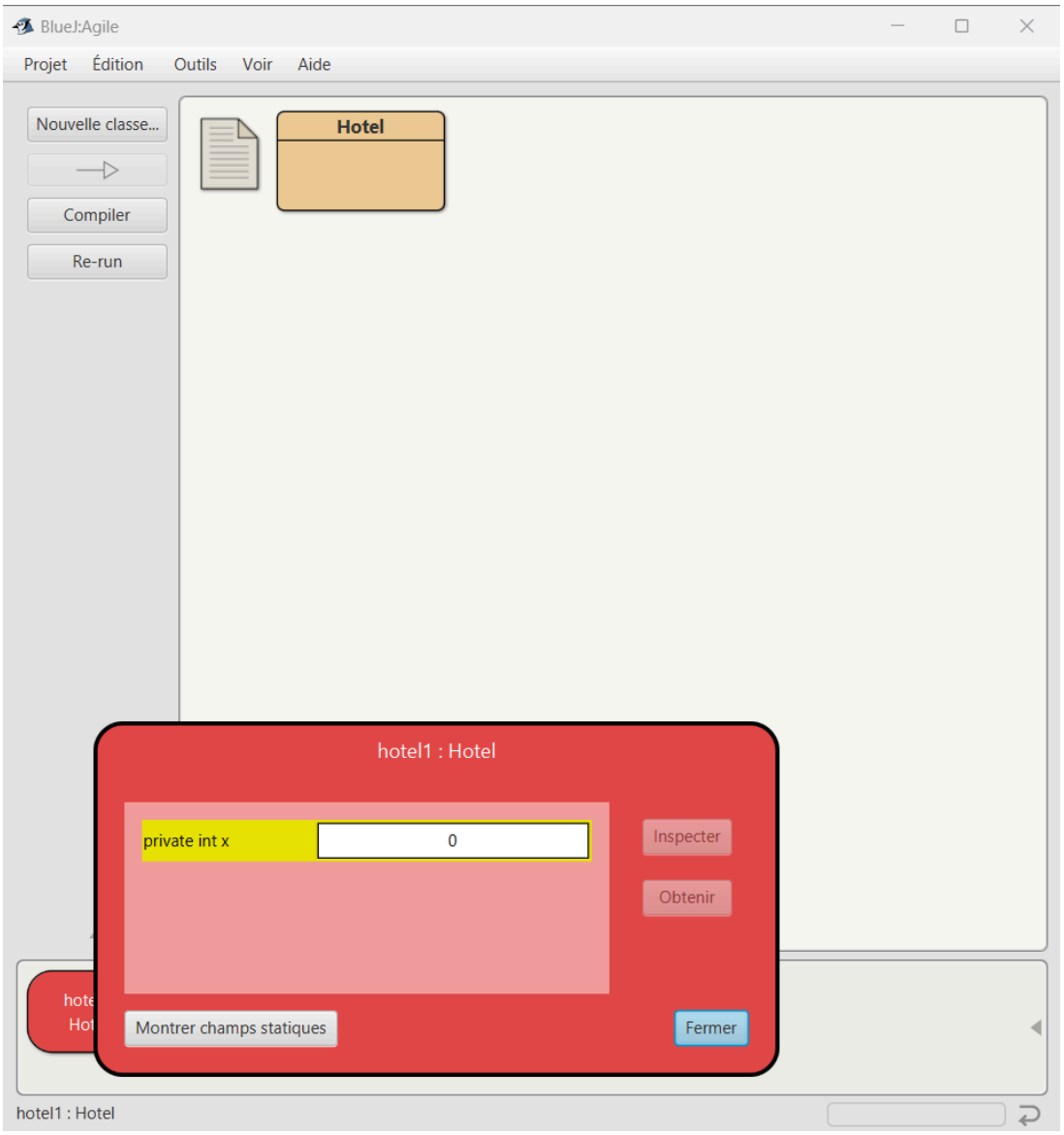
Une première classe nommée `Hotel` est créée. Elle représente l'hôtel hanté.

La classe est ensuite compilée.

---

4. Instanciation de la classe Hotel

Une instance de `Hotel` est créée de manière interactive dans BlueJ. L'objet apparaît sur le banc d'objets.



---

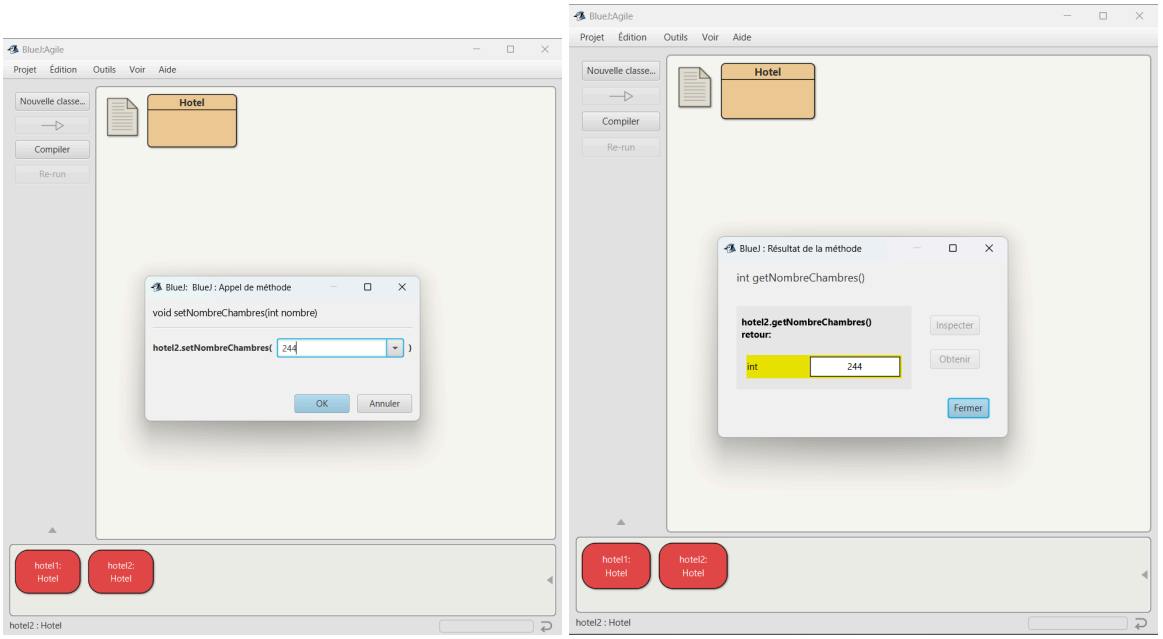
5. Ajout d’attributs et d’une méthode

La classe `Hotel` est enrichie avec :

- `nombreDeChambres`
- `tarifParNuit`

Des accesseurs sont ajoutés, ainsi qu’une méthode `renoverHotel()`.

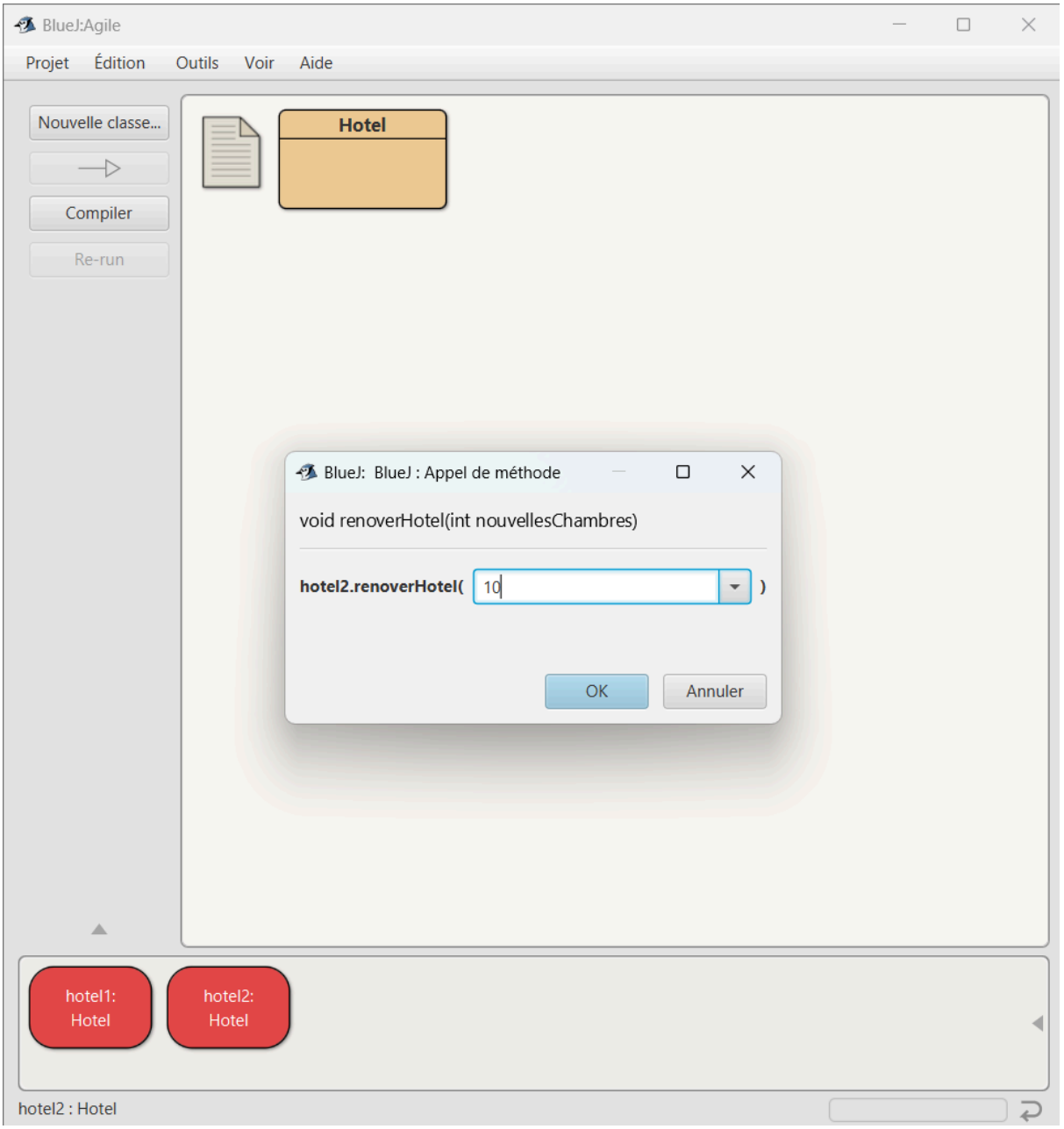
Cette méthode augmente le tarif d’une nuit de 2 % par chambre, pour représenter les rénovations (ou les perturbations liées aux fantômes).

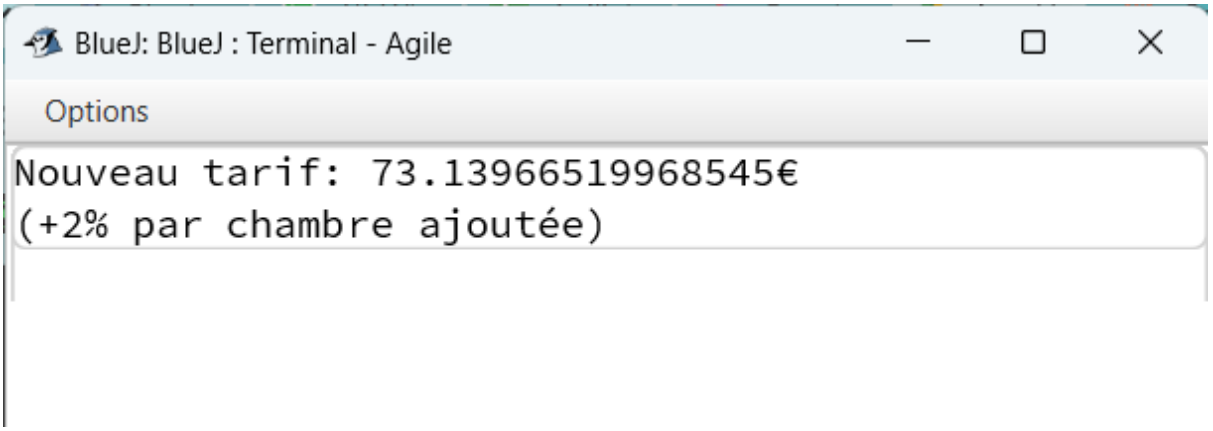


---

6. Observation du comportement de l’objet

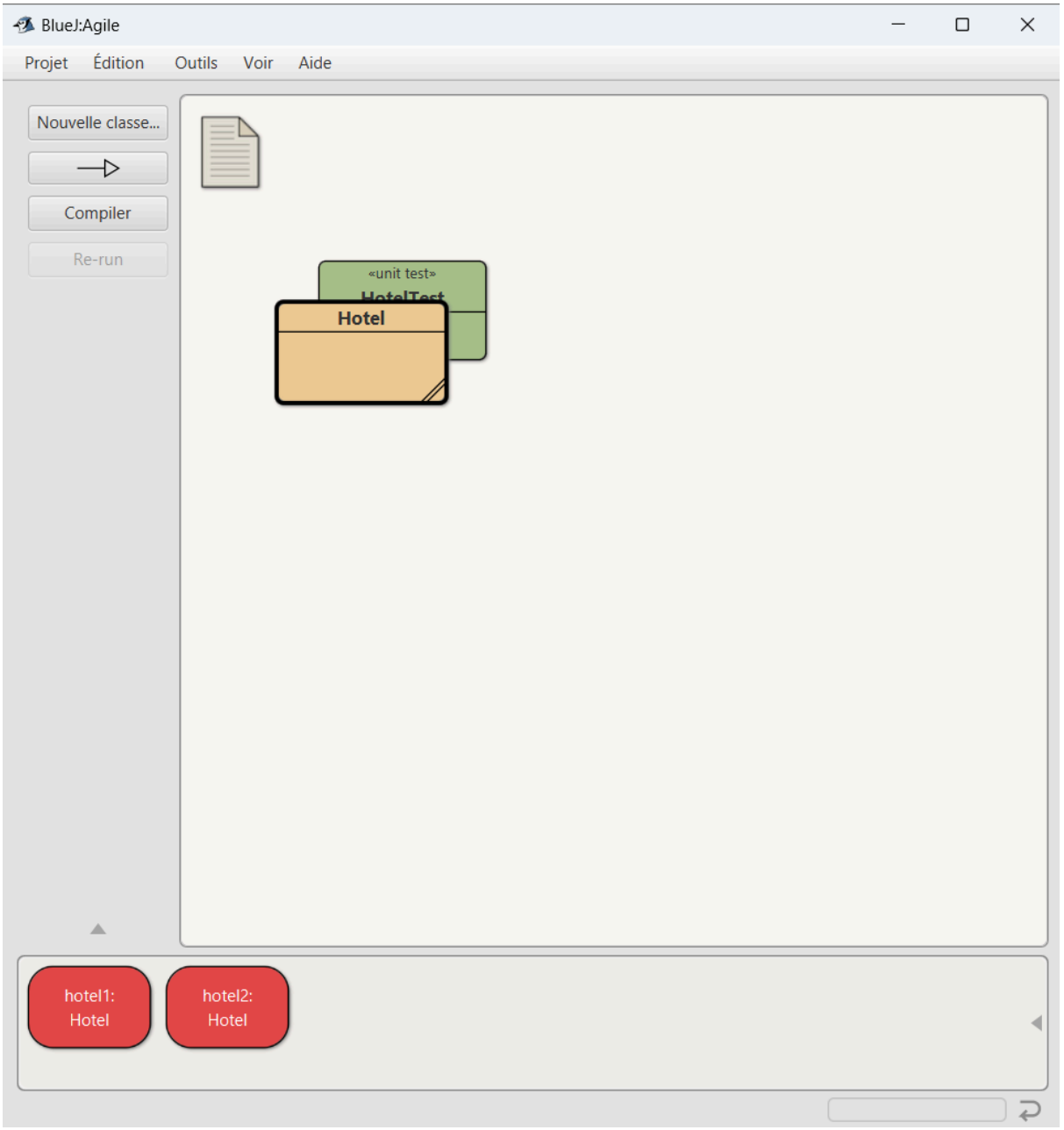
Une nouvelle instance de `Hotel` est créée. La méthode `renoverHotel()` est exécutée de façon interactive et l’évolution du tarif est observée directement sur l’objet.


BlueJ:Agile
Projet
Édition
Outils
Voir
Aide
Nouvelle classe...
Compiler
Re-run
Hotel
BlueJ: BlueJ : Appel de méthode
void renoverHotel(int nouvellesChambres)
hotel2.renoverHotel(10)
OK
Annuler
hotel1: Hotel
hotel2: Hotel
hotel2 : Hotel

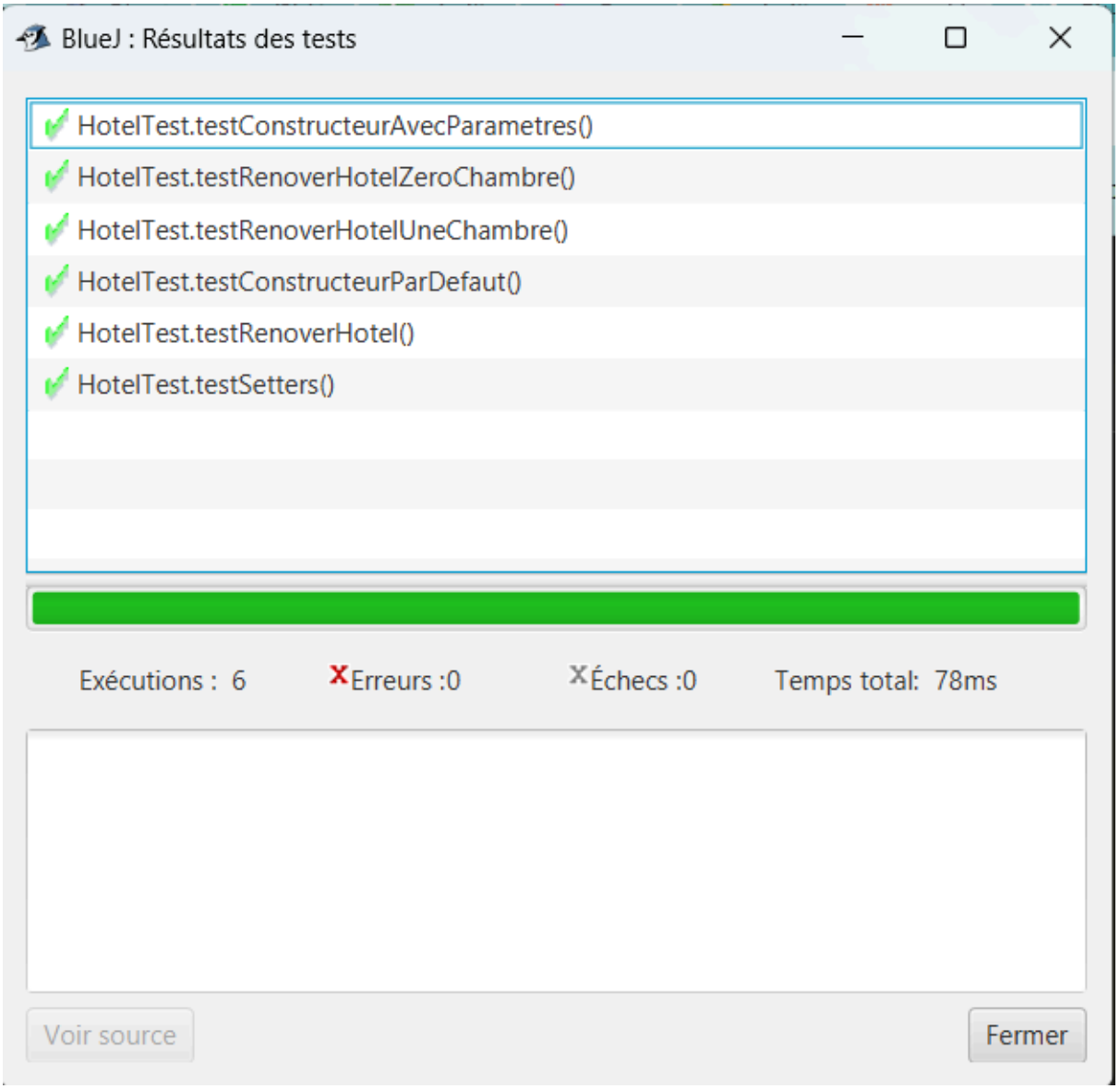

BlueJ: BlueJ : Terminal - Agile
Options
Nouveau tarif: 73.13966519968545€
(+2% par chambre ajoutée)

7. Test unitaire de la classe Hotel

Une classe de test `HotelTest` est créée. Un test vérifie que la méthode `renoverHotel()` modifie correctement le tarif.



Le test passe et la barre de résultat est verte.



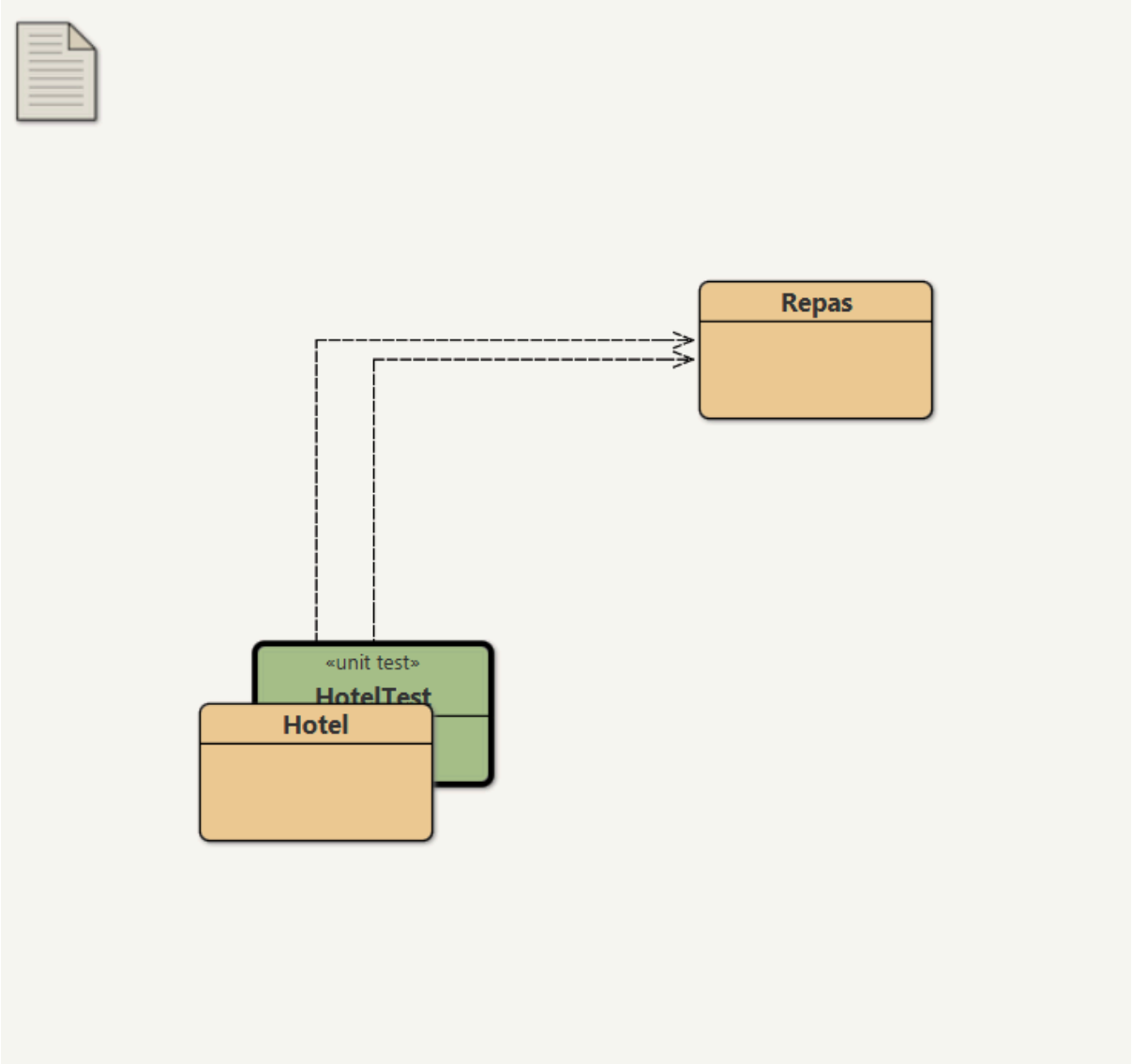
8. Ajout d'une seconde classe : Repas

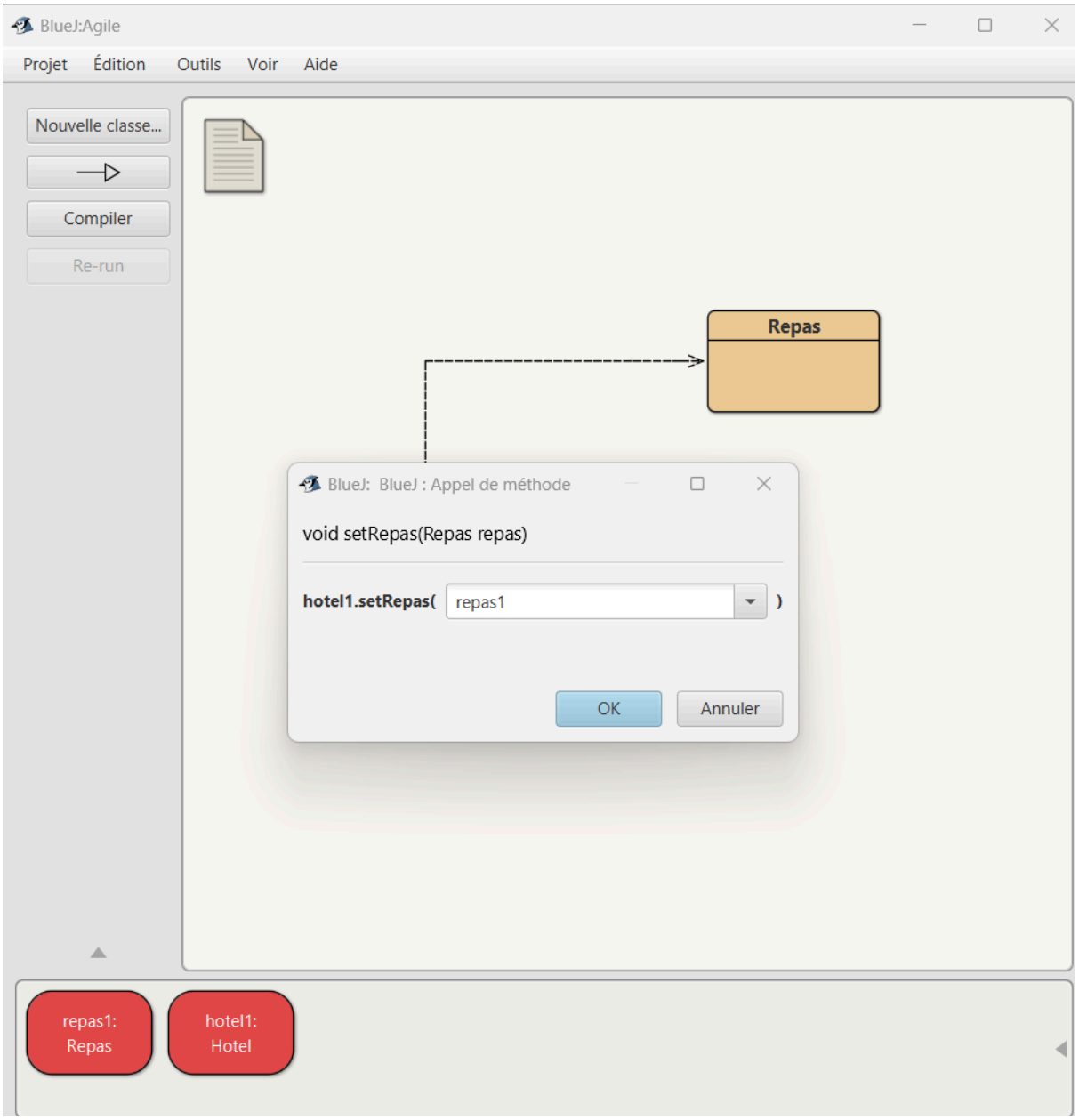
Une seconde classe `Repas` est créée. Elle représente les repas proposés par l'hôtel.

Attributs :

- `typeDeRepas`
- `prixDuRepas`

Cette classe est associée de manière unidirectionnelle à la classe `Hotel` (multiplicité 0..1 à 0..1).



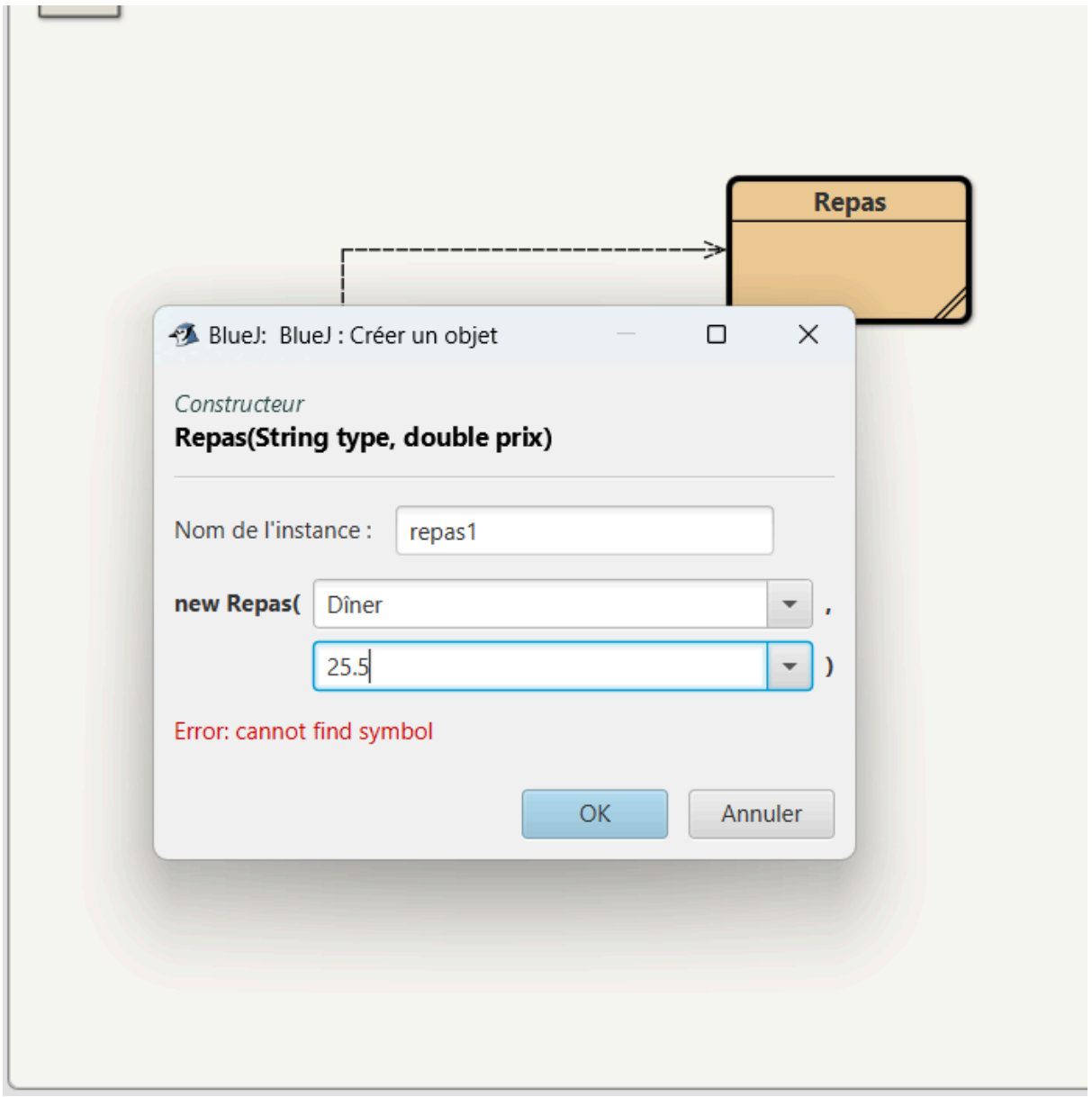


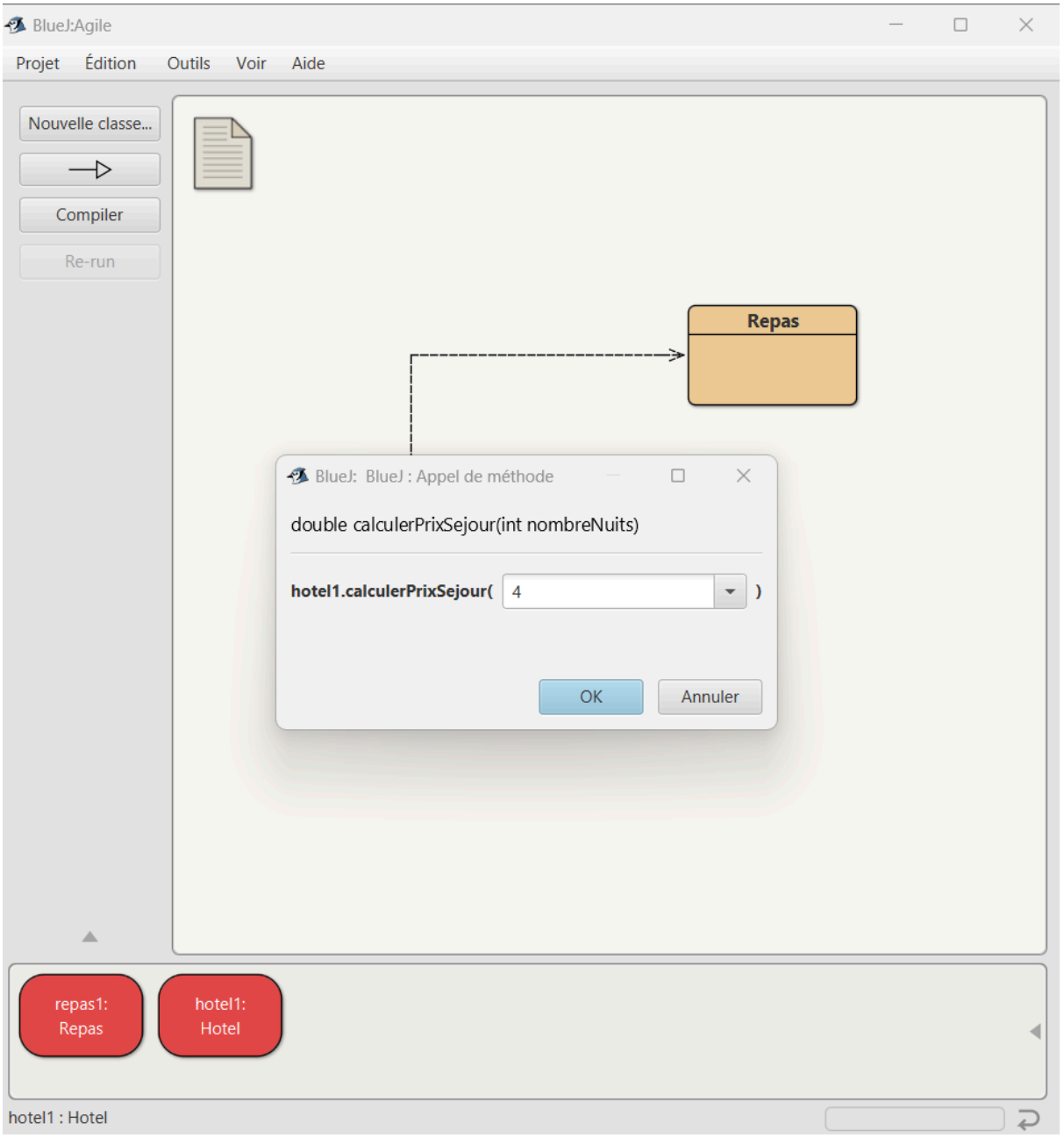
---

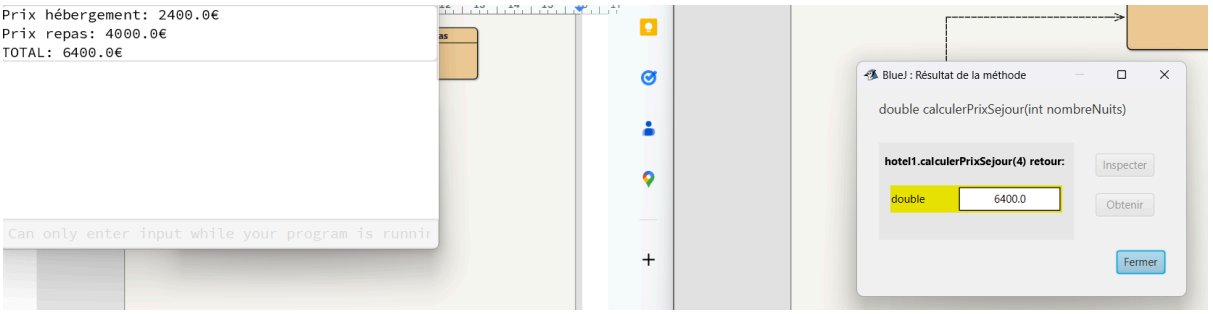
9. Collaboration entre Hotel et Repas

Une méthode `calculerPrixSejour(Repas repas)` est ajoutée à la classe `Hotel`.

Elle calcule le prix total d'un séjour en additionnant le tarif de la nuit et le prix du repas.


Repas
BlueJ: BlueJ : Créer un objet
Constructeur
Repas(String type, double prix)
Nom de l'instance :
repas1
new Repas(
Dîner
,
25.5
)
Error: cannot find symbol
OK
Annuler

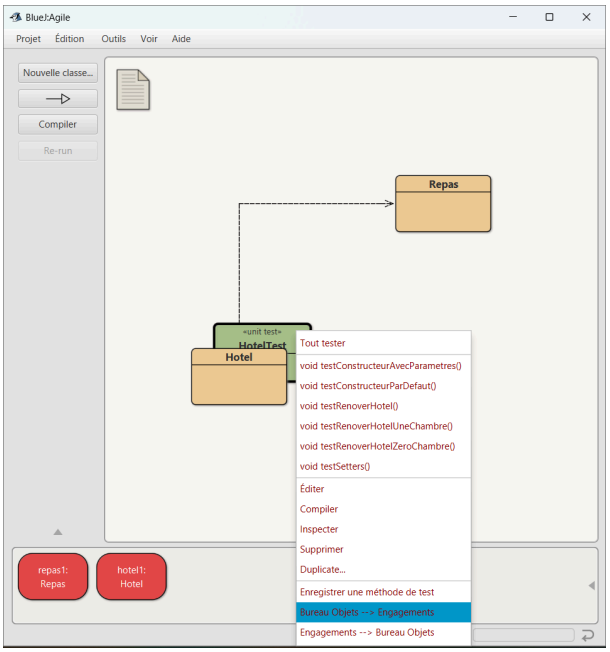

BlueJ:Agile
Projet
Édition
Outils
Voir
Aide
Nouvelle classe...
Compiler
Re-run
Repas
BlueJ: BlueJ : Appel de méthode
double calculerPrixSejour(int nombreNuits)
hotel1.calculerPrixSejour(
4
)
OK
Annuler
repas1:
Repas
hotel1:
Hotel
hotel1 : Hotel

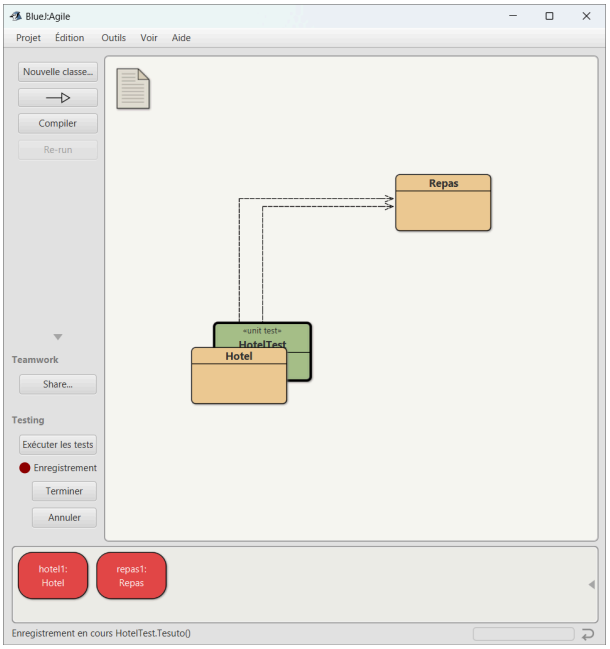

Prix hébergement: 2400.0€
Prix repas: 4000.0€
TOTAL: 6400.0€
Can only enter input while your program is runnir
BlueJ : Résultat de la méthode
double calculerPrixSejour(int nombreNuits)
hotel1.calculerPrixSejour(4) retour:
double
6400.0
Inspecter
Obtenir
Fermer

10. Fixture de test

Dans la classe de test, une fixture est mise en place à l'aide d'une méthode `setUp` :

- création d'un objet `Hotel`
- création d'un objet `Repas`
- association des deux objets

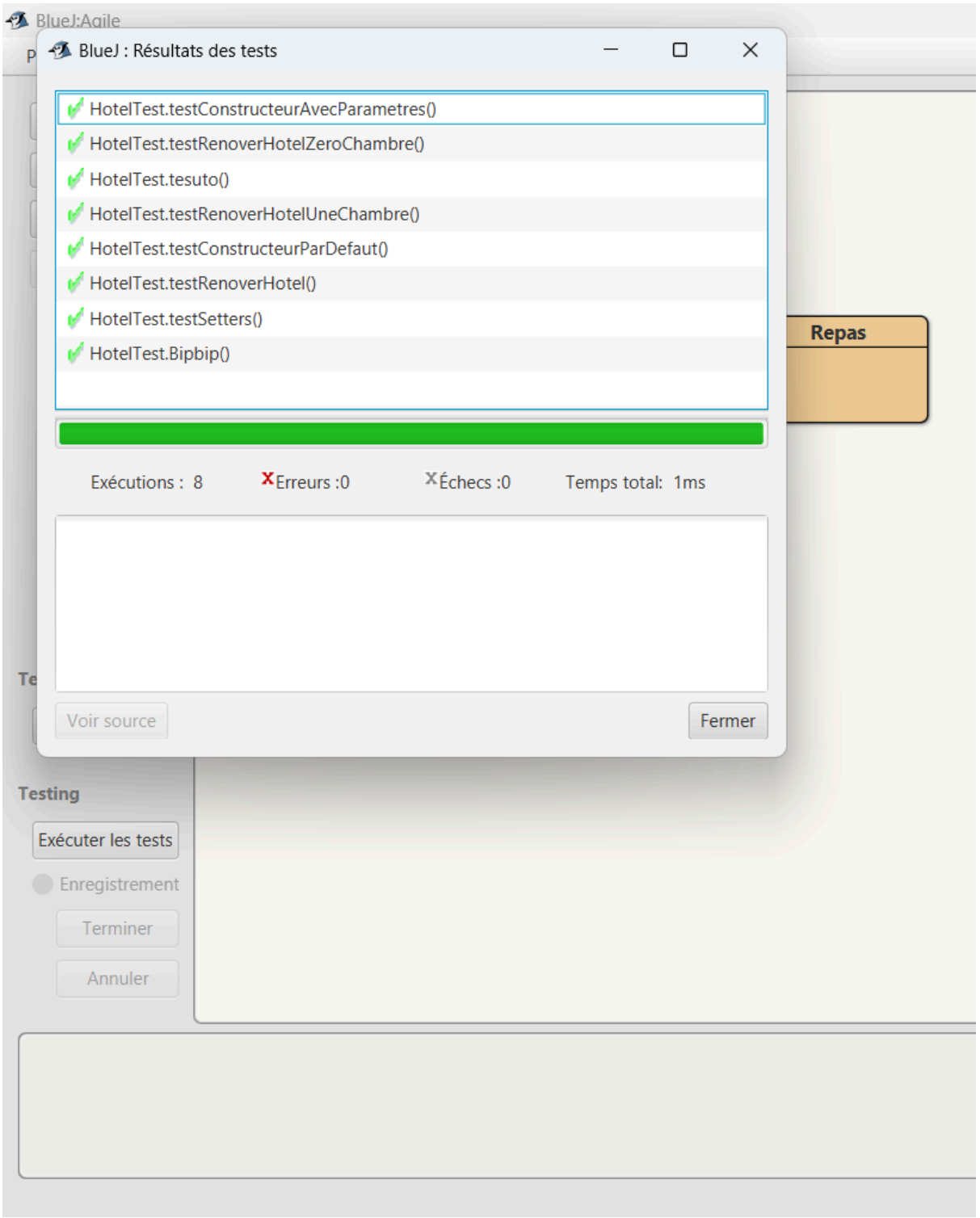




11. Test utilisant la fixture

Un test est écrit en utilisant la fixture **(méthode de Test BipBip)** afin de vérifier le calcul du prix du séjour.

Le test est exécuté avec succès et la barre est verte.



II. Seconde itération (complément)

14. Association bidirectionnelle (0..1 ↔ *)

La relation entre `Hotel` et `Repas` a été étendue afin qu'un hôtel puisse proposer plusieurs repas, tandis qu'un repas reste associé à au plus un hôtel. Cette association est bidirectionnelle et correctement encapsulée.

La classe `Hotel` maintient une liste privée `listeRepasProposes`. L'ajout d'un repas se fait exclusivement via la méthode `ajouterRepas`, qui garantit la cohérence de la relation en mettant à jour les deux côtés de l'association. La liste des repas n'est jamais exposée directement, mais uniquement via une vue non modifiable.

Des tests unitaires vérifient que l'ajout d'un repas met bien à jour l'hôtel côté `Repas`, et qu'aucune incohérence ne peut apparaître.

---

15. Refactoring

Deux techniques de refactoring ont été appliquées.

Un **rename** a été effectué sur l'attribut contenant les repas, renommé en `listeRepasProposes` afin de mieux refléter la multiplicité de l'association et d'améliorer la lisibilité du code.

```java
public class Hotel {

    private int nombreChambres;
    private double tarifNuit;

    /**
     * REFACTORING "RENAME" (Point 15) :
     * Renommé de 'repas' à 'listeRepasProposes' pour refléter l'association "0..1 à *"
     */
    private final List<Repas> listeRepasProposes;

    public Hotel(int nombreChambres, double tarifNuit) {
        this.nombreChambres = nombreChambres;
        this.tarifNuit = tarifNuit;
        this.listeRepasProposes = new ArrayList<>();
    }
```

Un **extract method** a été utilisé dans le calcul du prix du séjour. La logique de calcul du coût total des repas a été extraite dans une méthode dédiée, ce qui rend la méthode principale plus simple à lire et plus facile à maintenir.

```
public double calculerPrixSejour(int nombreNuits) {
    if (nombreNuits < 0) {
        throw new IllegalArgumentException("La durée du séjour ne peut pas être négative."
    }

    // REFACTORING "EXTRACT METHOD" (Point 15) :
    // La logique de somme des repas est extraite pour rendre cette méthode plus lisible.
    double prixTotalRepas = calculerSommePrixTousLesRepas();

    return (this.tarifNuit + prixTotalRepas) * nombreNuits;
}

/**
 * Méthode extraite via Extract Method.
 * Isole le calcul complexe pour faciliter la maintenance.
 */
private double calculerSommePrixTousLesRepas() {
    double somme = 0;
    for (Repas r : listeRepasProposes) {
        somme += r.getPrix();
    }
    return somme;
}
```

Après chaque refactoring, l'ensemble des tests unitaires a été exécuté avec succès.

16. Lecture de l'article « Test Infected » et amélioration adaptée

Nous avons consulté le site officiel de JUnit et lu l'article intitulé **« Test Infected »**, qui encourage à écrire des tests **avant**ou **dès que possible**, afin que le code soit continuellement guidé par les besoins de vérification.

Inspirés par cet article, nous avons adopté une pratique équivalente dans notre projet : **chaque nouvelle fonctionnalité est abordée en commençant par un test.**

Par exemple, avant d'implémenter l'association bidirectionnelle entre `Hotel` et `Repas`, nous avons d'abord écrit un test décrivant le comportement attendu (ajout d'un repas, vérification de la cohérence, cas d'erreur). Le code a ensuite été développé pour satisfaire ce test, ce qui nous a permis d'assurer la fiabilité de l'évolution dès la première implémentation.

---

17. Exécution des tests en ligne de commande

Les tests unitaires JUnit ont été exécutés depuis la ligne de commande pour vérifier que le projet fonctionne de manière autonome, en dehors de BlueJ.

À partir du dossier du projet, nous avons compilé les classes et exécuté les tests avec les commandes suivantes :

```
javac -cp .:junit-platform-console-standalone.jar
*.java
```

```
java -jar junit-platform-console-standalone.jar
--class-path . --scan-class-path
```

L'exécution a montré que tous les tests passent, ce qui confirme que les fonctionnalités sont stables et indépendantes de l'IDE.

18. Loi de Murphy associée

Loi de Murphy : *"Tout ce qui est susceptible de mal tourner finira par mal tourner."*

Dans ce projet, cette loi s'est manifestée lorsqu'un repas a été accidentellement associé à deux hôtels différents à cause d'une méthode d'ajout incorrectement implémentée. Sans tests unitaires, cette incohérence aurait pu rester invisible et provoquer des erreurs lors de l'utilisation.

Grâce aux tests que nous avions écrits en amont, le problème a été détecté immédiatement, ce qui nous a permis de corriger l'implémentation de l'association bidirectionnelle avant que d'autres étapes ne reposent sur ce comportement.