TD9 : Conception de jeu de tests

L'objectif de ce TD est d'apprendre à concevoir des jeux de tests de conformité pertinents.

Le test logiciel est la méthode la plus populaire pour vérifier un logiciel. Cette méthode représente environ **40-60% du prix final**. Tester, c'est exécuter le programme dans l'intention d'y trouver des anomalies ou des défauts

Plus précisément : une **spécification** est un ensemble explicite d'exigences à satisfaire par un logiciel, API, module, ….
Le **test** de conformité est l'exécution ou l'évaluation d'un système ou d'une composant par des moyens automatiques ou manuels, pour vérifier qu'il répond à ses spécifications ou identifier les différences entre les résultats attendus et les résultats obtenus.

Deux familles de tests utilisées simultanément dans l'industrie, car elles sont complémentaires :
Tests structurels (boîte de verre) : les données de tests sont produites après une analyse du code ; ces tests détectent principalement les erreurs commises.
Tests fonctionnels (boîte noire) : test de conformité par rapport à la spécification. Ces tests détectent principalement les erreurs d'omission par rapport à la spécification.

Le test est une activité intellectuelle :

- déterminer des scenarios plausibles pouvant mettre un logiciel en défaut ;
- concevoir et construire des bancs de tests permettant de vérifier les fonctionnalités et le respect des contraintes.

Quelques principes de base pour réaliser des tests.

- Un.e programmeur.e ne doit pas concevoir/écrire les bancs de tests ses propres programmes (tout simplement pour ne pas être juge et partie).
- Ne pas effectuer des tests avec l'hypothèse qu'aucune erreur ne va être trouvée.
- La définition des sorties ou résultats attendus doit être effectuée avant l'exécution d'un test ; car il y a des chances non négligeables de prendre un résultat erroné mais semblant cohérent pour un résultat correct.
- Inspecter minutieusement les résultats (les traces) de chaque test.
- Les jeux de tests doivent être écrits pour des entrées invalides ou incohérentes.

Conception des jeux de tests d'une méthode

De manière générale, les actions que **doit réaliser** une méthode sont de trois types :

- changement de l'état du système : mise à jour des attributs et des associations,
- changement des données pérennes (fichiers, base de données),
- valeur retournée,
- les effets de bords (appel à des méthodes, impression sur la sortie standard, ou sortie d'erreur, écriture dans un » journal »).

Les actions **réalisées par** une méthode dépendent de 3 choses :

- les valeurs de ses paramètres,
- l'état de l'objet (valeurs des attributs, des associations) voir l'état des objets voisins,
- Les données pérennes (fichiers, base de données).

La conception de jeu de tests pour une **méthode** consiste à élaborer un jeu de tests en vue de vérifier que la méthode réalise bien toutes les actions qu'elle doit réaliser et ceci dans les divers cas d'usage (valeur de ses paramètres, état du système et des données pérennes, …).

Spécification du Verrou (version préliminaire)
Un verrou peut être « verrouillé » ou « déverrouillé ». La méthode `estVerrouillé` retourne `true` si le verrou est verrouillé sinon `false`.
Déverrouiller (respectivement verrouiller) consiste à modifier l'état du verrou, si nécessaire, pour qu'il soit dans l'état « déverrouillé » (respectivement « verrouillé »).

```
testerVerrouBasique {
      VerouBasique verrou = new VerrouBasique();
      assertEqual(false, verrou.estVerrouillé());
      verrou.verrouiller() ;
      assertEqual(true, verrou.estVerrouillé());
      verrou.deverrouiller() ;
      assertEqual(false, verrou.estVerrouillé());
}
```

VerrouBasique
-verrouillé: boolean = false
+verrouiller () : void +déverrouiller () : void +estVerrouillé () : boolean +VerrouBasique ()

Question 1

1. Proposer une implémentation en Java de la classe VerrouBasique « passant » les tests, Cette classe ne doit pas être conforme à la spécification.
   Indice : verrouiller et déverrouiller sont des opérations « idempotentes ».

Comment procéder ?

- Déterminer les divers cas d'usage d'une méthode et quels sont les éléments (attributs, associations, …) qu'il faut tester car ils peuvent ou doivent être modifiés ;
- Puis élaborer un ensemble de scenarii permettant de tester **tous les cas d'usage** et écrire le code de la fonction qui testera un appel à la méthode à tester.

Les cas d'usage de la méthode `verrouiller` :

1. Le verrou est déverrouillé avant l'exécution de la méthode `verrouiller`
2. Le verrou est verrouillé avant l'exécution de la méthode `verrouiller`

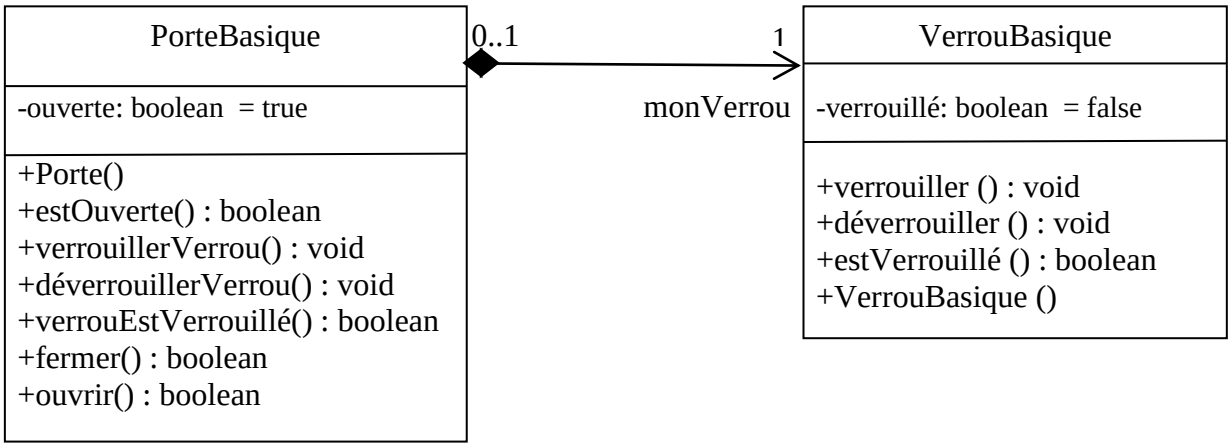
Les cas d'usage de la méthode `deverrouiller` :

3. Le verrou est déverrouillé avant l'exécution de la méthode `deverrouiller`
4. Le verrou est verrouillé avant l'exécution de la méthode `deverrouiller`

Les scénarii :

```
testerVerrouiller() {
      VerouBasique verrou = new VerrouBasique();
      assertEqual(false, verrou.estVerrouillé());
      verrou.verrouiller() ;                          // cas 1
      assertEqual(true, verrou.estVerrouillé());
      verrou.verrouiller() ;                          // cas 2
      assertEqual(true, verrou.estVerrouillé());
}
testerDeVerrouiller() {
      VerouBasique verrou = new VerrouBasique();
      assertEqual(false, verrou.estVerrouillé());
      verrou.deverrouiller() ;                        // cas 3
      assertEqual(false, verrou.estVerrouillé());
      verrou.verrouiller() ;
      verrou.deverrouiller() ;                        // cas 4
      assertEqual(false, verrou.estVerrouillé());
}
```

Porte Basique. Une porte basique a un verrou. Le verrou d'une porte doit être déverrouillé pour ouvrir ou fermer la porte ; sinon, le pêne ou loquet des verrous en position verrouillés empêcheront la fermeture ou l'ouverture de la porte.
La méthode « fermer » retourne `true` si la porte est fermée à la fin de l'exécution de cette méthode sinon elle retourne `false`. De manière similaire, la méthode « ouvrir » retourne `true` si la porte est ouverte à la fin de l'exécution de cette méthode sinon elle retourne `false`.



Hypothèse : les méthodes `estOuverte`, `verrouillerVerrou`, `déverrouillerVerrou` et `verrouEstVerrouillé` de la classe `PorteBasique`.
La méthode `verrouEstVerrouillé` retourne l'état du verrou de la porte. La méthode `verrouillerVerrou` « déverrouille » le verrou de la porte (quel que soit l'état de la porte). …

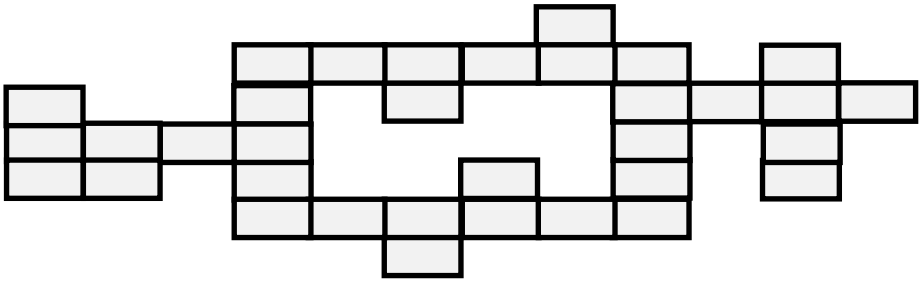
Questions 2 Tester la méthode `ouvrir` de la classe `PorteBasique` :

- ✓ listez les cas d'usage ;
- ✓ complétez le code de la méthode `ouvrirTest` définie ci-dessous.
- ✓ proposez des scenarii testant tous les cas d'usage – utiliser la méthode `ouvrirTest`.

```
ouvrirTest(PorteBasique porte) {

   // partie 1 : stocker dans des variables l'état de la porte
   //     et de son environnement avant l'exécution de la méthode ouvrir.
         boolean porteOuverteAvant = porte.estOuverte() ;
         boolean verrouVerrouilléAvant = porte.VerrouEstVerrouillé() ;

   // partie 2 : exécuter la méthode ouvrir sur la porte
         boolean resultatObtenu = porte.ouvrir() ;

   // partie 3 : pour chaque cas d'usage, définir les tests à réaliser

}
```

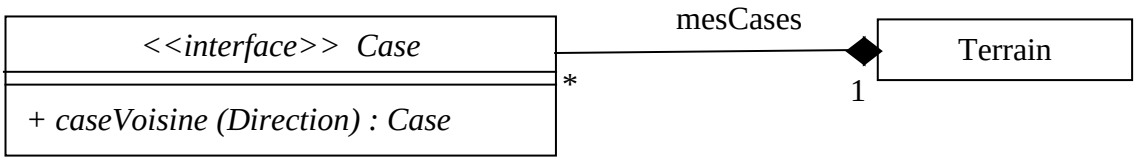
Conception d'un jeu de robots sur un terrain

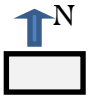
Info 1: Terrain irrégulier composé de Cases de même taille, pouvant comporter des « vides ».
L'appel `rec.caseVoisine(dir)` retourne la Case voisine de la case `rec` dans la direction `dir`, si cette Case existe dans le terrain ; sinon `null` est retourné.



Hypothèse : les méthodes permettant de construire un terrain ont déjà été testées, ainsi que les implémentations de l'interface fonctionnelle Case

Ebauche du diagramme de classes :

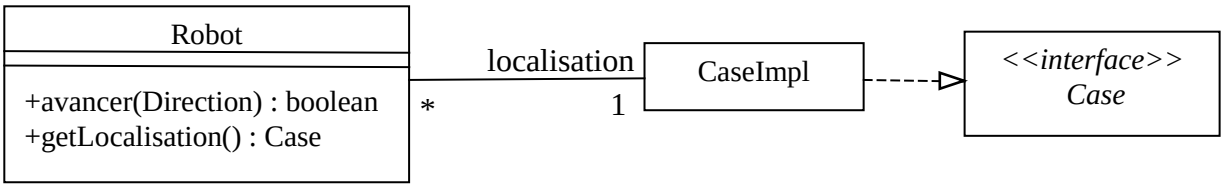





```
public enum Direction { NORD, EST,  SUD,  OUEST; }
```

Info 2: un robot est sur une Case, il peut se déplacer dans les 4 directions sur le terrain.

Ebauche 2 du diagramme de classes :



L'appel `rob.avancer(dir)` avance le robot `rob` dans la direction indique dans `dir` si cela est possible et retourne `true` sinon l'appel retourne `false`.

Question 3 Tester la méthode `avancer(Direction)`:

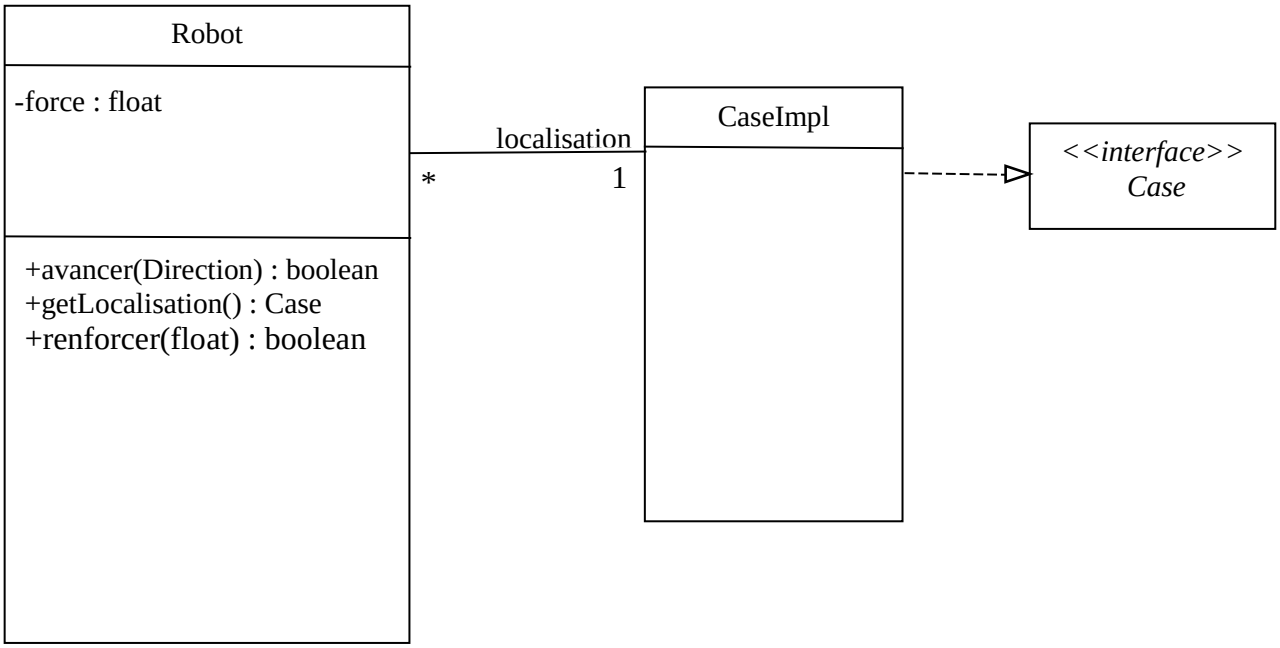
- ✓ listez les cas d'usage;
- ✓ écrivez le code de la méthode `avancerTest(Robot, Direction);`
- ✓ proposez des scenarii testant tous les cas d'usage : dessinez un terrain, nommer les cases, écrivez le code des tests – utiliser la méthode `avancerTest`.

Info 3 : certaines cases contiennent un trésor caché (des pièces d'argent) ; le premier robot qui passe sur la case gagne le trésor.

Questions 4

- Complétez l'ébauche 3 du diagramme de classes.
- listez les cas d'usage correspondant à la récupération d'un trésor caché.
- écrivez le code de la méthode `tresorTest(Robot, Direction, float);`
- proposez des scenarii testant tous les cas d'usage : dessinez un terrain, nommer les cases, écrivez le code des tests – utiliser la méthode `tresorTest`.

Ebauche 3 du diagramme de classes :



Info 4 : La force d'un robot est définie par la valeur de l'attribut force (plus la valeur est grande plus le robot est fort). Un robot peut augmenter sa force (méthode `renforcer`) mais cela a un coût `x*100` pour une augmentation de la force de `x`. Ce montant est prélevé sur la fortune du robot. Si le robot n'est pas assez riche alors sa force n'est pas augmentée et la méthode retourne `false`.

Questions 5

- Complétez l'ébauche 3 du diagramme de classes.
- Tester la méthode `renforcer` :
  - ✓ listez les cas d'usage ;
  - ✓ écrivez le code de la méthode `renforcerTest` après avoir défini sa signature ;
  - ✓ proposez des scenarii testant tous les cas d'usage : dessiner un terrain, nommer les cases, écrivez le code des tests – utiliser la méthode `renforcerTest`.

Info 5 : Lorsqu'un robot arrive sur une Case contenant un autre robot, le plus fort vole 10% de la fortune du plus faible.

Questions 6

- Vérifiez que l'implémentation de la méthode `avancer(Direction)` satisfait les spécifications données dans le texte de l'info 5.
- La spécification du « vol » est-elle complète (assez précise) ?