Durée totale : **2h**. Tous documents autorisés. Nombre de pages : **2**
barème indicatif : test = 3/4, model-checking = 1/4
Les parties test et model-checking sont à rendre sur 2 copies séparées.

Exercice 1 : Test

Les 2 **questions** sont **indépendantes**. La réponse attendue pour chaque question, sur une copie propre et lisible, est une **classe de test JUnit4 exclusivement**. Les cas de test doivent être conformes à l'esprit TDD : simples, et suffisamment clairs et exhaustifs pour illustrer les fonctionnements des classes testées. Si vous choisissez d'utiliser des mocks, vous avez le choix entre vous servir de Mockito ou écrire les mocks à la main.

Le but de cet exercice est de tester (en utilisant JUnit) un framework de test appelé MyJUnit, évidemment inspiré de JUnit mais suffisamment simplifié pour garantir l'indépendance des questions.

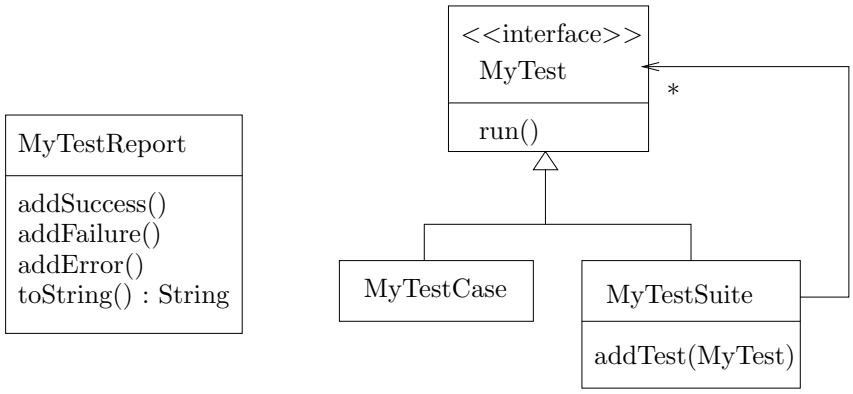


FIG. 1 – Diagramme UML du paquetage `myjunit`

1.1 : Test de `MyTestReport`

La classe `MyTestReport`[1] collecte grâce aux méthodes `addSuccess`, `addFailure` et `addError` les verdicts (*success*, *failure*, *error*) des tests qui sont exécutés. La méthode `toString` produit une chaîne décrivant les verdicts dans l'ordre dans lequel ils auront été collectés, avec les conventions suivantes :

- *failure* est représenté par `F` ;
- *success* est représenté par `.` (un point) ;
- *error* est représenté par `E`.

Par exemple, dans le cas de deux *success* et d'une *error* collectés successivement, la chaîne produite sera `"..E"`.

Q 1.1 : Donner une classe de test JUnit4 qui teste la méthode `toString` de la classe `MyTestReport`. Il est inutile de donner les clauses `import`. □

1.2 : Test de la classe `MyTestSuite`

La classe `MyTestSuite` représente une collection de tests de type `MyTest`. On lui ajoute un nouveau test par la méthode `addTest(MyTest)` (il est possible d'ajouter plusieurs fois le même test). La méthode `run` de `MyTestSuite` est chargée d'appeler la méthode `run` de chacun des tests qui la constituent, peu importe dans quel ordre. Si la suite de test ne contient aucun test, rien ne se passe. Il est bien sûr possible de créer récursivement des suites de test à partir d'autres suites.

Q 1. 2 : En adoptant un test *en isolation* de la classe `MyTestSuite`, donner une classe de test JUnit4 qui teste la méthode `run`. Il est inutile de donner les clauses `import`. □

[1] Cette classe est très simplifiée par rapport à son homologue JUnit, qui associe notamment chaque verdict à un objet `Test`.

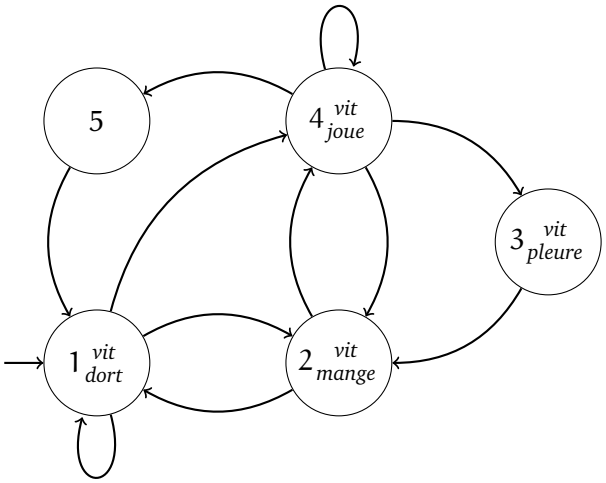
Model-checking d'un tamagotchi

On s'intéresse ici à la spécification et la vérification d'un Tamagotchi[1].

On identifie les propositions atomiques suivantes sur les états du tamagotchi :

– *vit* ;
– *joue* ;
– *mange* ;
– *dort* ;
– *pleure*.

Dans un accès de paresse, on se donne la structure de Kripke suivante, très simple, comme modèle de Tamagotchi (dans chaque état on indique uniquement les propositions atomiques vraies) :



Exprimer chacune des propriétés suivantes en LTL puis indiquer (en justifiant) si le modèle la vérifie.

1. Rien n'est immortel : le tamagotchi meurt à un moment donné[2].

2. Vive la sieste : juste après (à l'état suivant) avoir mangé le tamagotchi s'endort.

3. Jouer réveille : l'excitation du jeu empêche de s'endormir juste après.

4. Quand on est mort c'est pour longtemps : le tamagotchi n'a pas le droit de ressusciter.

5. Jouer fatigue : si le tamagotchi joue trop longtemps (3 états d'affilée), il se met à pleurer de fatigue.

[1] Wikipédia en propose la définition suivante : le Tamagotchi est un animal de compagnie virtuel, japonais. Le jeu consiste à simuler l'éducation d'un animal à l'aide d'une petite console miniature, de la taille d'une montre, dotée d'un programme informatique.

[2] Indice : comment peut-on exprimer que le tamagotchi est mort avec les propositions atomiques données ?