Mode d'emploi de PowerUnit

HOUEKPETODJI Mahugnon Honoré (homahugnon@gmail.com)
https://www.linkedin.com/in/mahugnon-honore-4948b6114/

https://www.overleaf.com/project/5f5f7a9e37b8f8000181acd7
17 Septembre, 2020

Table des matières

0.1 Généralités sur les tests Unitaires

Définition 0.1.1. Tester un projet est un processus manuel ou automatisé qui vise à vérifier qu'un système satisfait les propriétés requises par ses spécifications, ou à détecter les différences entre les résultats produits par le système et ceux attendus par les spécifications

0.1.1 Tests

— préviennent des erreurs introduites par les développeurs ;
— préviennent des échecs lors d'exécutions ;
— préviennent des imperfections sur des parties du système susceptibles de causer des échecs d'exécutions ;
— représentent la **confiance** sur l'état de santé du système ;
— se construisent **progressivement** :
  • pas besoin d'écrire tous les tests d'un coups
  • à chaque nouveau **dysfonctionnement ou ticket, écrire des tests** ;
— c'est d'ailleurs meilleur de les écrire avant d'implémenter la fonctionnalité
  • agissent comme les premiers **clients** et une meilleure interface ;
— représentent une documentation active et synchrone des fonctionnalités présentes dans le système.

Dans ce guide, nous allons nous concentrer sur les tests unitaires

0.1.2 Tests unitaires

Vocabulaire

Définition 0.1.2. Un cas de test

- est généralement associé à la réussite d'un scénario de cas d'utilisation. Les développeurs ont souvent des scénarios de test à l'esprit, mais ils les réalisent de différentes manières (1) instructions d'affichage ; (2) le débuguage ou les fichier de traces ; etc.
- est un ensemble d'entrées de test, de conditions d'exécution et de résultats attendus, développé pour tester un chemin d'exécution particulier. Généralement, le cas est une méthode unique.

Définition 0.1.3. Une suite de test est une liste de cas de tests liés. La suite peut contenir des routines communes d'initialisation et de nettoyage spécifiques aux cas de tests qu'il contient. Généralement, la suite de test est une classe.

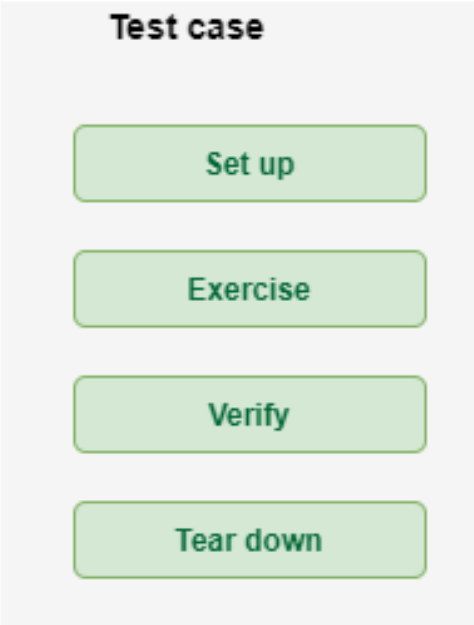

FIGURE 1 – Test case

Présentation de cas de test unitaire

Un cas de test répond au principe **Essaie, vérifie, si ça marche**. La Figure 1 montre les quatre étapes d'un cas de test unitaire. On distingue :

— **Set up.** consiste à préparer les ressources pour le tests ;

— **Exercise.** consiste à exercer la fonctionnalité à tester du système en tests sur les ressources précédemment préparées ;

— **Verify.** Consiste à vérifier que le résultat de l'exercice correspond bien au résultat attendu du système en test ;

— **Tear down.** Consiste à libérer les ressources utiliser durant le test

Caractéristiques d'un bon cas test unitaire

— Répétable

— Pas d'intervention humaine

— S'auto-décrit

— Change moins souvent que le système

— Raconte une histoire

0.2 Power unit

0.2.1 Présentation

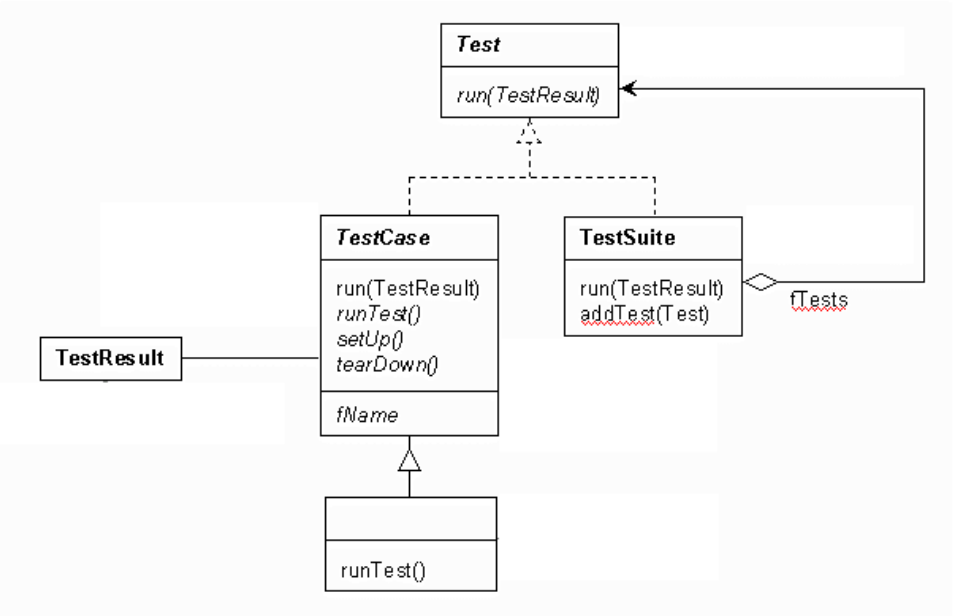

FIGURE 2 – Power unit

Power unit est le framework qui permet d'écrire les tests unitaire en Powerbuilder. Il est inspirer de JUnit. La Figure 2 presente le diagramme de classe de Power Unit. La classe *TestCase* est la classe de base pour écrire les tests unitaire en Powerbuilder. Ainsi tous les tests héritent de la classe *TestCase*.

0.3 Ecriture de tests avec Power Unit

Dans le cadre de ce apprentissage, je vais reprendre l'exemple de la documentation de Power Unit. Le programme que nous allons parcourir résoudra le problème de la représentation de l'arithmétique avec des monnaies multiples. L'arithmétique entre les même monnaies est triviale ; vous pouvez simplement additionner les deux montants. Les choses deviennent plus intéressantes une fois que des monnaies multiples sont introduites. Vous ne pouvez pas vous contenter de convertir une monnaie en une autre pour faire de l'arithmétique puisqu'il n'y a pas de taux de conversion unique.

0.3.1 Préparation de l'environnement

Pour suivre ce apprentissage, vous avez besoin :

— d'avoir l'IDE **Powerbuilder** installer sur votre machine
— de télécharger les bibliothèques de base (***powerunit.pbl, powerunitfunc.pbl***) de Power Unit ici
— de télécharger les bibliothèques de l'interface graphique de Power Unit ***powerUnit.zip*** ici
— Créer un répertoire qui va accueillir les objets durant ce apprentissage
— Créer un nouvel espace de travail nommé *PBCookBook*
— Copier les bibliothèques (***powerunit.pbl, powerunitfunc.pbl*** dans l'espace de travail.
— Créer une *target* nommée *PBCookBook* dans l'espace de travail.
— Ajouter les bibliothèques (***powerunit.pbl, powerunitfunc.pbl*** téléchargée précédemment à la target .
— Creér une nouvelle bibliothèque PBCookBookTests pour acceuillir les tests.

0.3.2 Creation d'un objet à tester

Nous allons commencer par définir simplement un objet qui représentera une valeur monétaire dans une monnaie unique. Mais d'abord, créons un objet de base que nous expliquerons plus loin dans ce mode opératoire :

— Créer un nouvel objet de classe *custom class*
— Fermez et enregistrez-le sous le nom n_cst_MoneyBase dans le PBCookBook.PBL

Maintenant, créons l'objet monnaie *money* :

— Héritez un nouvel objet de l'objet n_cst_MoneyBase
— Enregistrez le sous le nom de n_cst_Money dans la bibliothèque PBCookBook en lui donnant un commentaire approprié
— Définir deux attributs privés pour la somme et la monnaie *money*

```

        Private Decimal idc_Amount;
        Private String  is_Currency;
```

Listing 1 – variable d'instance de n_cst_money

— Créer une méthode d'initialisation pour ces attributs lors de leur création

```

    /*  of_Initialize ( decimal adc_Amount
   , string as_Currency )  */
    idc_Amount = adc_Amount;
    is_Currency = as_Currency;
    return;
```

Listing 2 – methode d'initialisation de n_cst_money

— Créer des accesseurs pour les attributs

```
1
2     /*  of_getAmount (): decimal  */
3 return  idc_Amount;
```

Listing 3 – accesseur pour recuperer la somme de l'objet *money*

```
1     /*  of_getCurrency ( ): String    */
2     return  is_Currency;
```

Listing 4 – accesseur pour recuperer la monnaie de l'objet *money*

— Créez une méthode simple pour ajouter les valeurs de deux objets *money* de la même monnaie et obtenir ainsi un nouvel objet *money*.

```
1   /*  of_add(n_cst_money anv_Money):
     n_cst_Money */
2 n_cst_Money lnv_Money;
3
4 lnv_Money = CREATE n_cst_Money;
5 lnv_Money . of_Initialize ( THIS.
    of_getAmount() + anv_Money.of_getAmount
    (), THIS.of_getCurrency( ));
6 return  lnv_Money;
```

Listing 5 – Simple addition de deux montants de la même monnaie

— Avant de pouvoir procéder, nous avons besoins comparer deux objets *money*. Ajoutez une nouvelle méthode of_Equals() à votre objet *n_cst_Money*. Deux objets Monnaie sont considérés comme égaux s'ils ont la même monnaie et la même valeur

```
1
2  /* of_equals(n_cst_Money anv_Money):
     Boolean  */
3
4  IF IsValid ( anv_Money ) THEN
5      return (( anv_Money.of_getAmount() =
     idc_Amount ) &
6           AND ( anv_Money.of_getCurrency()
     = is_Currency ));
7  ELSE
8      return false;
9  END IF;
```

Listing 6 – Comparaison de deux objets *money*

0.3.3 Première entrée de jeu

Création d'un cas de test

Maintenant, au lieu de continuer à coder, arrêtons nous et obtenons un retour d'information immédiat en pratiquant *"coder un peu, tester un peu, coder un peu, tester un peu"*. Un test est mis en oeuvre en créant une sous-classe de l'objet TestCase. Nous allons donc créer une classe *n_cst_MoneyTest*, que nous allons placer dans bibliothèque *PBCookBookTests*. Notez que "e" représente le code de la monnaie pour l'euro.

— Assurez-vous que le *PBUnit.PBL* figure dans la liste des bibliothèques de votre cible PBCookBook. Cela vous donne accès à toutes les classes d'objets de test PBUnit.

— Héritez de la classe TestObject dans PBUnit.PBL et enregistrez-la sous le nom de *n_cst_MoneyTest*.

— Écrivons un scénario de test pour nous assurer que la méthode *of_Equals* fonctionne lorsque les objets *money* sont le même objet :

```
1
2       /* EVENT TestEquals()  */
3
4  n_cst_Money lnv_euro12;
5  n_cst_Money lnv_euro14;
6  n_cst_Money lnv_euro12b;
7
8  lnv_euro12 = CREATE n_cst_Money;
9  lnv_euro14 = CREATE n_cst_Money;
10 lnv_euro12b = CREATE n_cst_Money;
11
12 lnv_euro12.of_Initialize( 12.00, "e" );
13 lnv_euro14.of_Initialize( 14.00, "e" );
14 lnv_euro12b.of_Initialize( 12.00, "e" );
15
16 THIS.Assert ( lnv_euro12.of_Equals (
      lnv_euro12 ) );
17 THIS.Assert ( NOT lnv_euro12.of_Equals (
      lnv_euro14 ) );
18 THIS.Assert ( lnv_euro12.of_Equals (
      lnv_euro12b ) );
19 return
```

Listing 7 – Test d'égalité de deux *money*

— Ajoutez une méthode, *testSimpleAdd()* qui testera notre méthode *of_add()*.

```
1     /*  EVENT testSimpleAdd() */
2     //    #1
3     n_cst_money lnv_euro12;
4     n_cst_money lnv_euro14;
5     n_cst_money lnv_ExpectedResult;
6     n_cst_money lnv_ActualResult;
7
8     lnv_euro12 = CREATE n_cst_Money;
9     lnv_euro12.of_Initialize ( 12.00, "e"
      );
10
11    lnv_euro14 = CREATE n_cst_Money;
12    lnv_euro14.of_Initialize ( 14.00, "e"
      );
13
14    lnv_ExpectedResult = CREATE
      n_cst_Money;
15    lnv_ExpectedResult.of_Initialize (
      12.00+14.00, "e" );
16    //    #2
17    lnv_ActualResult = lnv_euro12.of_Add (
      lnv_euro14 );
18    //    #3
19    THIS.Assert ( ( lnv_ExpectedResult.
      of_Equals ( lnv_ActualResult ) ), true
      );
20    return;
```

Listing 8 – Test d'addition de deux *money*

Maintenant que nous avons créé nos deux cas de test, nous remarquons qu'il y a un peu de duplication de code pour la mise en place de ces deux tests. Il serait bon de réutiliser une partie de ce code de mise en place. Pour ce faire, il suffit de déclarer les objets ressources comme variables d'instance et de les initialiser en redéfinissant l'événement ***setUp***. L'opération symétrique de ***setUp*** est le ***tearDown*** que vous pouvez redéfinir pour liberer les ressources à la fin d'un test. Chaque test a ces propre ressources ou son contexte d'execution. *PBUnit* execute les évènements ***setUp*** et ***tearDown*** pour chaque test pour éviter les éffets de bord au niveau des tests.

```
1     Private:
2     n_cst_Money inv_euro12;
3     n_cst_Money inv_euro14;
4
5     /*  EVENT setUp ()    */
```

```
6     inv_euro12 = CREATE n_cst_Money;
7     inv_euro14 = CREATE n_cst_Money;
8     inv_euro12.of_Initialize( 12.00, "e" )
      ;
9     inv_euro14.of_Initialize( 14.00, "e" )
      ;
10
11    /*  EVENT tearDown()  */
12    IF IsValid ( inv_euro12 ) THEN
      DESTROY inv_euro12;
13    IF IsValid ( inv_euro14 ) THEN
      DESTROY inv_euro14;
```

Listing 9 – Setup et TearDown

Nous pouvons réécrire les deux méthodes de cas de test, en supprimant le code de configuration commun :

```
1     /*  EVENT TestEqual   */
2     n_cst_Money lnv_euro12b;
3
4     lnv_euro12b = CREATE n_cst_Money;
5     lnv_euro12b.of_Initialize( 12.00, "e"
      );
6
7     THIS.Assert ( inv_euro12.of_Equals (
      inv_euro12 ) );
8     THIS.Assert ( NOT inv_euro12.of_Equals
      ( inv_euro14 ) );
9     THIS.Assert ( inv_euro12.of_Equals (
      lnv_euro12b ) );
10    return
```

Listing 10 – Test d'egalité sans la logique du d'initialisation

```
1     /*  EVENT testSimpleAdd   */
2     n_cst_money lnv_ExpectedResult;
3     n_cst_money lnv_ActualResult;
4
5     lnv_ExpectedResult = CREATE
      n_cst_Money;
6     lnv_ExpectedResult.of_Initialize (
      12.00+14.00, "e" );
7     //    #2
8     lnv_ActualResult = inv_euro12.of_Add (
      inv_euro14 );
```

```
9     //     #3
10    THIS.Assert ( ( lnv_ExpectedResult.
      of_Equals ( lnv_ActualResult ) ) );
11
12    return;
```

Listing 11 – Test d'ajout simple sans la logique du d'initialisation

0.3.4 Exécution des tests

Il existe deux méthodes pour exécuter les tests que vous avez programmés dans votre objet TestCase : Statique et dynamique. Avec la méthode statique, vous surchargerez l'événement runTest dans le n_cst_MoneyTest et exécuterez chacun des tests que vous voulez exécuter.

```
1     /*  EVENT runTest ()  */
2     THIS. EVENT testEqual ();
3     THIS. EVENT testSimpleAdd ();
4     Return;
```

Listing 12 – Redefinition de l'evènement *runTest*

La méthode dynamique est bien plus pratique que la méthode statique, même si elle permet d'obtenir le résultat souhaité, à savoir l'exécution des deux tests. Avec la méthode statique, vous devez vous souvenir de modifier la méthode *runTest()* pour exécuter tout nouveau test que vous ajoutez à l'objet. Avec la méthode dynamique, au lieu de déclarer les tests comme des fonctions d'objet, vous les déclarez comme des ***EVENTS*** d'objet qui ne prennent aucun argument et dont le nom commence par ***"test"***. C'est pourquoi nous avons utilisé des événements et nommé nos méthodes de test *"testequal"* et *"testsimpleadd"*. En ne remplaçant pas l'événement runtest dans notre objet *n_cst_MoneyTest*, tous les événements dont le nom commence par ***"test"*** seront automatiquement exécutés.

0.3.5 Explication de l'interface graphique de PBUnit

Cette partie suppose que vous avez déjà télécharger l'interface de Power Unit. Si ce n'est pas encore fait, veillez le télécharger ici .

— Extraire le fichier télécharger vers "bureau/powerUnit"

— Double cliquer sur ***powerunitgui.exe***. La fenêtre de la Figure 3 s'ouvre L'interface de Power Unit est composée de 3 partie principales. La partie (1) presente la liste des tests groupées par biblothèques contenues dans la *target* sélectionnée. Parlant de *target*, sélectionnons notre target *PBCookBook.pbt*.

— Cliquer sur *Open Target ...* comme indiquer sur la Figure 4 ou bien faite *Ctrl*+*O*.

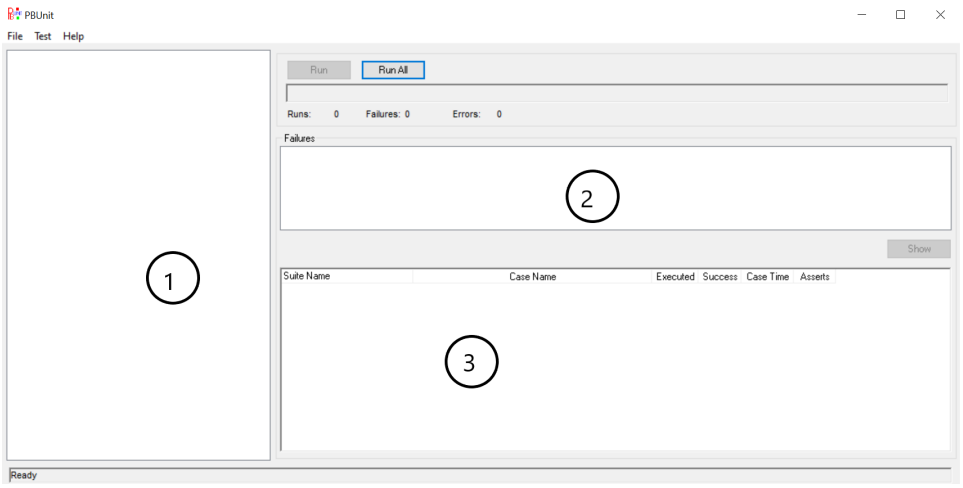

FIGURE 3 – Power unit

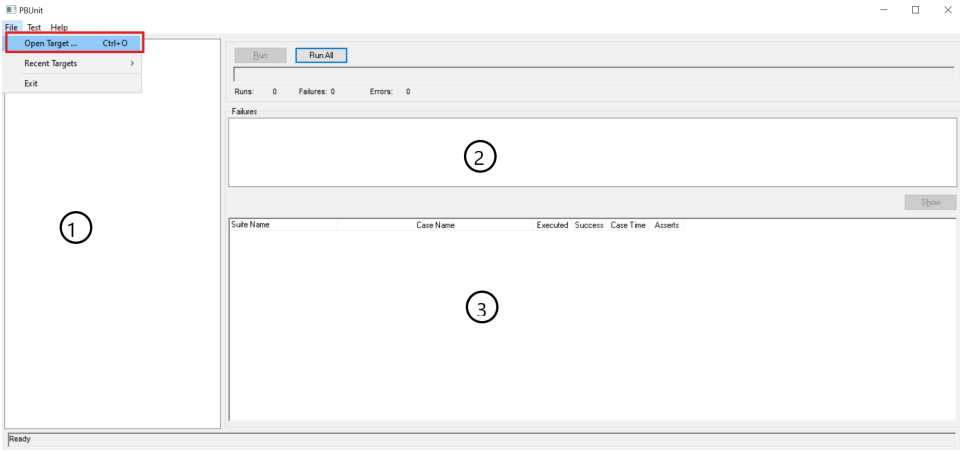

FIGURE 4 – Choisir une target

— Naviguer vers l'emplacement de la target qui contient les tests (dans ce cas *PBCookBook*), puis selectionner la target *nom_target.pbt* (*PBCookBook.pbt* dans ce cas).

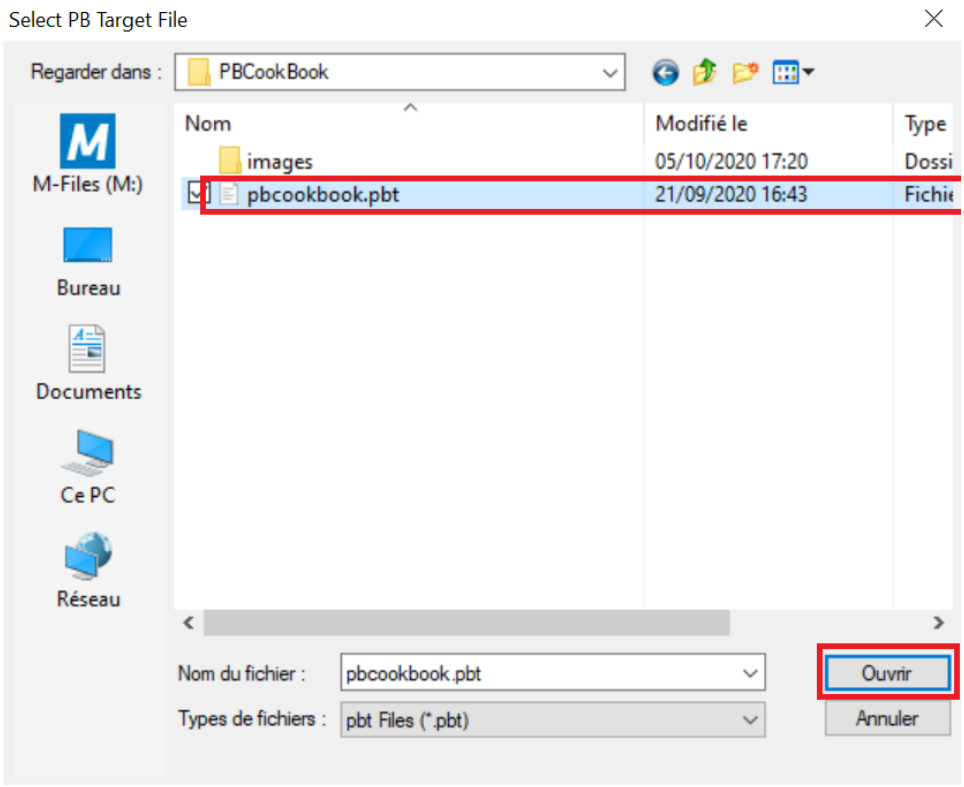

FIGURE 5 – Selectionner la target à son emplacement