Le Framework SUnit

Dr. Ducasse

ducasse@iam.unibe.ch
http://www.iam.unibe.ch/~ducasse/

Pour vous souhaiter la bonne année, nous vous présentons SUnit, un framework qui vous permettra d'écrire des tests qui assureront la pérennité de vos applications. Le mois dernier nous avons montré l'excellent Refactoring Browser et les deux outils qui l'accompagnent SmallLint et le RewriteTool [RB]. Le Refactoring Browser permet de transformer du code de manière *sécurisée*, c'est-à-dire sans en changer le comportement. Ceci qui permet une grande rapidité de développement et en particulier de modification du code. Pour commencer la nouvelle année, nous voulons finir la série d'articles liée aux outils de développement qui ont radicalement eu un impact sur notre façon de programmer. Nous allons présenter SUnit un framework minimal qui permet la définition et l'exécution de tests unitaires.

L'intérêt pour SUnit n'est pas limité à Squeak ni même à Smalltalk, en effet, de très nombreux programmeurs ont reconnus la valeur de SUnit et l'ont porté dans leur langage de prédilection. Ainsi on trouve SUnit pour pratiquement tous les langages allant de Oracle à Perl en passant par Python, Java ou C++ et bien d'autres [SUnit].

Transformer du code à l'aide de refactorings permet d'effectuer des transformations sécurisées. Le problème est que l'on ne peut pas tout faire avec exclusivement des refactorings et que le code doit aussi être manuellement édité. Une des solutions est la définition de tests appelés unitaires. Les phases de tests sont souvent mal aimées dans les phases de développement, elles sont souvent comprimées ou considérées à tort comme une perte de temps. Les tests ont plusieurs rôles: premièrement, ils servent de documentation toujours synchronisée avec le code auquel ils se rattachent, un test peut représenter un cas d'usage de la classe. Deuxièmement, ils représentent de manière *tangible* la confiance que le système peut évoluer en minimisant les risques. Les tests vont permettre de trouver extrêmement rapidement des bugs. C'est une évidence mais c'est absolument simple et vrai. Finalement, le fait d'écrire les tests en même temps que les classes auxquels ils se rattachent force le développeur à être son propre client et donc assure que l'interface sera plus lisible et claire.

eXtreme Programming une nouvelle méthodologie de développement pour des petits groupes des développeurs (moins de 10) remet le test au centre du processus développement en allant même jusqu'à demander que les tests soient écrits avant le code et que les tests servent à décrire les cas d'usage et autres fonctionnalités [Beck].

La culture du test a toujours fait partie de la philosophie de développement Smalltalk dans laquelle on écrit une méthode, la compile et la teste en écrivant un petit script soit dans un workspace soit dans les commentaires soit comme méthode de classe dans la catégorie exemples. Cependant, cette façon de faire ne permet pas de répéter les tests de manière automatique et de les répertorier. De plus, le résultat du test est souvent implicite laissant au lecteur le soin d'interpréter si le test est passé ou non. SUnit est un ensemble de classe, 4 principales, qui permet de décrire et d'exécuter des tests unitaires. Squeak 3.1 contient SUnit 3.0, la dernière version de SUnit.

1. Quels Tests

Dans le cadre du développement itératif d'applications transformer du code pour l'adapter à de nouveaux besoins est une réelle nécessité. Aussi utiliser d'une part des outils comme le Refactoring Browser couplé à l'utilisation de tests permet de minimiser les risques d'introduire des erreurs.

Il est clair que l'on ne peut pas tester tous les aspects d'une application. Couvrir toute une application est simplement impossible. Il est fort possible que des tests soient manquants et que certains bugs apparaissent après changement sans avoir été détectés par des tests. Ceci n'est pas un problème à partir du moment où si l'on découvre un bug non couvert on ajoute au jeu de tests un nouveau test couvrant précisément ce bug.

Ecrire de bons tests est une technique qui s'apprend. Un test unitaire doit avoir plusieurs propriétés afin d'apporter le maximum de bénéfice:

- Répétables. Il doit être répétable à souhait.
- Sans intervention humaine. Les tests doivent pouvoir être répétés la nuit par exemple et sans nécessiter d'interaction.
- Raconter une histoire. Un test doit couvrir un aspect du système. La bonne granularité est de décrire une fonctionnalité d'une classe à la fois.
- Avoir une fréquence de changement plus lente que celui de l'application. Si à chaque changement d'une application, on doit changer trop de tests, leur conception est maladroite. Un test doit être exprimé en utilisant autant que possible l'interface de la classe.

Le nombre de tests d'un jeu de tests doit être proportionnel aux fonctionnalités couvertes. Ainsi il est important que si dix tests sont invalidés, le bug découvert ne soit pas le même que si cent tests été étaient invalidés.

eXtreme Programming propose d'écrire les tests avant même de programmer. Ceci peut paraître contre nature mais pour avoir essayé, les résultats suivants sont à remarquer : tester en premier aide (1) à conceptualiser les responsabilités de la classe, (2) exprimer l'interface de la classe car lors de l'écriture du test, on est son premier client.

2. SUnit par l'exemple

Avant d'entrer dans le détail de SUnit, nous montrons un exemple. Pour définir une série de tests en SUnit on va tout d'abord créer une souclasse la classe `TestCase`. L'idée est que les méthodes de cette classe vont implémenter plusieurs tests, les variables d'instance représentent les objets et/ou le contexte dans lequel ces tests vont être exécutés. Pour comprendre, regardons comment quelques fonctionnalités de la classe `Set` qui est fourni sont spécifiés. Vous pourrez ainsi regarder directement le code pour approfondir ces notions. Notez que SUnit est lui-aussi tester à l'aide de lui-même.

Tout d'abord, on crée une souclasse de `TestCase` nommée `ExampleSetTest`. Cette classe définit deux variables d'instances qui représentent deux ensembles, instances de la classe `Set`.

```
TestCase subclass: #ExampleSetTest
     instanceVariableNames: 'full empty'
     classVariableNames: ''
     poolDictionaries: ''
     category: 'SUnit-Tests'
```

Ensuite on définit la méthode `setUp` qui sera invoquée *avant toute exécution* d'une méthode de test. Cette méthode met donc on place le contexte du test, appelé *fixture* dans le jargon SUnit. Ici on définit un ensemble vide et un ensemble contenant deux éléments.

```
ExampleSetTest>>setUp
  empty := Set new.
  full := Set with: 5 with: #abc
```

On définit plusieurs tests. Tout d'abord, un test qui vérifie que la méthode `includes:` qui rend vrai si un ensemble contient un élément spécifié en paramètre, fonctionne correctement. Pour cela, on vérifie que l'ensemble `full` contient bien les éléments que l'on a mis durant l'invocation de la méthode `setUp`. On teste aussi que le calcul du nombre d'éléments est aussi correct et que la méthode `remove:` fait bien ce qu'elle doit faire. Notez que tous ces tests présuppose que la méthode `setUp` ait bien été exécutée avant leur propre exécution.

```
ExampleSetTest>>testIncludes
  self assert: (full includes: 5).
  self assert: (full includes: #abc)

ExampleSetTest>>testOccurrences
  self assert: (empty occurrencesOf: 0) = 0.
  self assert: (full occurrencesOf: 5) = 1.
  full add: 5.
  self assert: (full occurrencesOf: 5) = 1

ExampleSetTest>>testRemove
  full remove: 5.
  self assert: (full includes: #abc).
  self deny: (full includes: 5)
```

La méthode `assert:` est une méthode définie sur la classe `TestCase` qui attend un booléen comme argument résultat de l'exécution de l'expression testée. Si l'argument est vrai alors l'exécution de l'expression testée est correcte, on dit que le test est validé, sinon elle est fausse et il s'agit d'un échec du test (nommé failures dans le jargon de SUnit). En fait, SUnit distingue deux sortes d'erreurs : les échecs d'un test (failures), i.e., un test n'est pas valide et les erreurs (errors) qui sont par définition les erreurs que le test n'a pas prévu comme une division par zéro non prévue ou un accès hors limite. La méthode `deny: expression` est une abréviation pour `assert: expression not`.

La méthode `should: boolean raise:` exception est utilisée pour spécifier quelle exception doit être levée. Cette méthode est utile pour tester que le code lève correctement des exceptions. Dans le test `#testIllegal`, la première ligne spécifie qu'une exception de type error doit être levée lorsque l'expression est exécutée. Ici l'exception utilisée est une exception de SUnit définit sur la classe TestResult car ce test servant d'exemple doit être indépendant des versions de Smalltalk. Bien sûr on pourrait spécifier n'importe quelle type d'exception. Lorsque le test sera exécuter si une exception de la bonne classe est levée le test sera validé.

```
ExampleSetTest>>testIllegal
  self should: [empty at: 5] raise: TestResult error.
  self should: [empty at: 5 put: #abc] raise: TestResult error
```

Exécution. Pour exécuter le test nommé `#testRemove:`, exécutez l'expression `(ExampleSetTest selector: #testRemove) run` ou `ExampleSetTest run: #testRemove.` Pour invoquer l'interface de SUnit exécutez : `TestRunner open`. Vous pouvez sélectionner un jeu de test (ici TestNewParagraphFix) et presser Run. Par défaut, l'interface exécute toutes les méthodes commençant par la chaîne 'test'.

Si vous voulez débuggez un test, utilisez les méthodes `debug: ExampleSetTest debug: #testRemove` ou `(ExampleSetTest selector: #testRemove) debug`

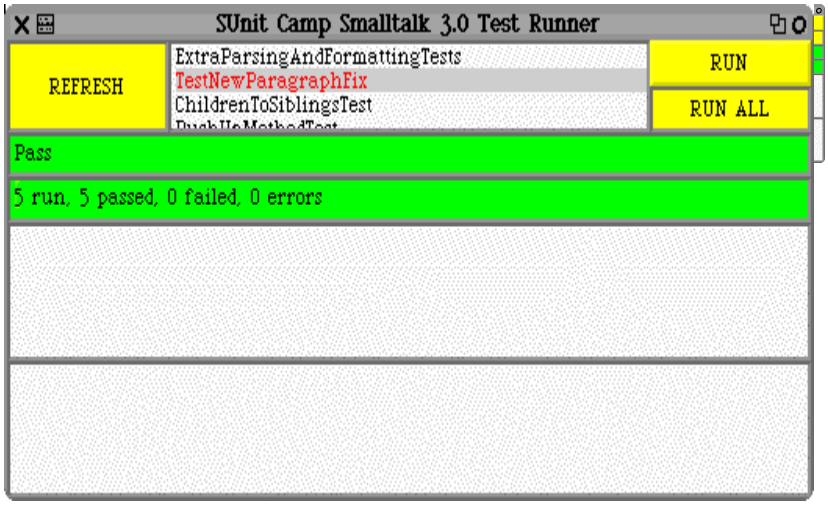

Figure 1 L'interface utilisateur de SUnit. Ici un jeu de test à été exécuté et tous les tests sont passés.

3. SUnit

Squeak 3.1 contient la version 3.0 de SUnit. Cette version introduit la notion de ressources qui sont nécessaires quand on veut tester des systèmes ayant de très longues phases d'initialisation et que de telles phases ne doivent pas être exécutées avant l'exécution de chaque test mais avant un jeu ou plusieurs jeux de tests.

SUnit est constitué de quatre principales classes `TestCase`, `TestSuite`, `TestResult` et `TestResource` comme illustré dans la Figure 1 (J'ai utilisé les Connectors pour représenter le diagramme présenté dans la Figure 1. et je vous suggère d'esssayer : prenez le code à http://bike-nomad.com/squeak).

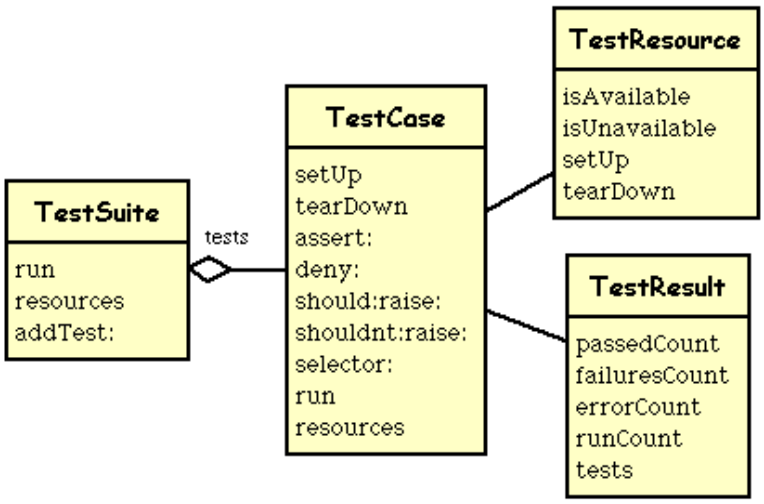

Figure 2 Les quatre classes définissant le noyau de SUnit avec leu principales méthodes.

La classe `TestCase` représente un test ou un jeu de tests dont le contexte d'exécution est partagé entre ces tests. Cette classe permet de définir le contexte dans lequel les tests vont être exécutés par la spécialisation de la méthode `setUp`. Cette méthode est exécutée avant l'exécution de chaque test. La méthode `tearDown` est la méthode symétrique de la méthode `setUp`, elle est invoquée après toute exécution d'un test, elle permet de libérer certaines ressources.

La classe `TestSuite` représente une collection de tests. Une instance de `TestSuite` est composée d'instance de `TestCase` et de `TestSuite`. Elle forme avec `TestCase` un composite pattern.

La classe `TestResult` représente les résultats de l'exécution d'un jeu de test, c'est -à-dire le nombre de tests passés, invalidés et d'erreurs rencontrés lors de l'exécution des tests.

La classe `TestResource` représente une ressource qui est utilisée par un test. Une ressource décrit un ensemble de valeurs, comme par exemple une base de données qui doit être disponible pour qu'un test puisse être exécuté. L'idée d'une ressource est qu'elle n'est pas mise en place avant l'exécution de chaque tests mais avant un ensemble de tests, une instance de `TestSuite`. Une ressource peut être définie sur pour un jeu de test, en redéfinissant la méthode de classe `resources` de la souclasse de la classe `TestCase`. Par défaut, une instance de `TestSuite` considère que ses ressources sont toutes les ressources définies dans les jeux de tests qui la composent.

Voici schématiquement comment on peut déclarer une ressource `MyResource` et l'associer avec un test `MyTestCase`. Comme pour un `TestCase`, on définit par la méthode `setUp` les actions qui vont être effectuées pour mettre en place la ressource. Sur la classe `MyTestCase` on spécialise la méthode de classe `resource` pour qu'elle rende une collection contenant le nom des classes des ressources nécessaires.

```
TestResource subclass: #MyTestResource
      instanceVariableNames: ''

  ....
```

```
TestResource>>setUp
  ici on met en place la ressource

MyTestCase class>>resources
  ^ #(MyTestResource)
```

4. Références et liens

[Beck] Kent Beck, Extreme Programming Explained: Embrace Change, Addison-Wesley, 1999.

[FBBOR] Martin Fowler, Kent Beck, John Brant, William Opdyke and Don Roberts, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.

[RB] http://www.refactory.com/RefactoringBrowser/, http://st-www.cs.uiuc.edu/users/brant/Refactory/

[RBJ1] D. Roberts, J. Brant and R. Johnson, “Why every Smalltalker should use the Refactoring Browser, Smalltalk Report, SIGS Press, http://st-www.cs.uiuc.edu/users/droberts/homePage.html#refactoring

[RBJ2]D. Roberts, J. Brant and R. Johnson, “A Refactoring Tool for Smalltalk”, TAPOS, vol. 3, no. 4, 1997, pp. 253-263, http://st-www.cs.uiuc.edu/~droberts/tapos/TAPOS.htm

[SUnit] http://www.xprogramming.com/software.htm