Introduction aux tests unitaires

# Introduction

Cet exemple reprend l’exemple du TP précédent de gestion d’une section d’étudiants, avec pour objectif cette fois de programmer des tests unitaires permettant de déceler d’éventuelles erreurs au sein des méthodes des classes métiers. De plus, dans ce cadre qui est celui de la qualité du logiciel, ce TP introduit également la documentation des méthodes.

# Aparté sur les exceptions

Le code présenté utilise les fonctionnalités de gestion des « exceptions ». Une exception est quelque chose qui ne devrait normalement pas se produire dans le fonctionnement « nominal » du programme (sinon, on traiterait le cas avec un si-alors-sinon). La syntaxe de ce bloc est :

**Essayer** (**try**) {

faire quelque chose qui normalement devrait fonctionner

mais qui pourrait lever une exception (throw)

} **Intercepter** (**catch**) {

réagir dans le cas ou une exception se produit

}

# Préparation

Créer un dossier (tp2a), l’ouvrir avec Visual Studio Code et initialiser un projet « Haxe » (pas OpenFl) puis ajouter au dossier src les fichiers Etudiant.hx et Section.hx (voir ci-après).

## Classe Etudiant

class Etudiant {

var nom : String;

var prenom : String;

public var classe : String;

public function new(nom : String, prenom : String) {

if (setNom(nom) == null) {

**throw** "'nom' should not be null or empty !";

}

if (setPrenom(prenom) == null) {

**throw** "'prenom' should not be null or empty !";

}

}

public function setNom(nom : String) : String {

if (nom != null && nom != "") {

this.nom = nom;

}

return this.nom;

}

public function setPrenom(prenom : String) : String {

if (prenom != null && prenom != "") {

this.prenom = prenom;

}

return this.prenom;

}

public function getNom() : String {

return this.nom;

}

public function getPrenom() : String {

return this.prenom;

}

}

Remarquer que les méthodes set\* retournent la valeur de la propriété qui peut ne pas avoir été modifiée. Le constructeur utilise leurs valeurs de retour pour lever une exception si les paramètres fournis ne permettent pas la création d’un étudiant (valeur null ou chaîne vide pour le nom et / ou le prénom).

## Classe Section

class Section {

var etudiants : Array<Etudiant>;

public function new() {

etudiants = new Array();

}

public function ajoutEtudiant(e : Etudiant) : Section {

if (e != null) {

etudiants.push(e);

}

return this;

}

public function getNbEtudiants() : Int {

return etudiants.length;

}

public function getEtudiants() : Array<Etudiant> {

return etudiants.copy();

}

}

Remarquer que la méthode ajoutEtudiant retourne la section (l’objet courant) qui vient d’être manipulée. Cela permet de chaîner les opérations sur une section ; exemple (à ne pas coder) :

var s = new Section();

var e1 = new Etudiant("Damasio", "Alain");

var e2 = new Etudiant("Simmons", "Dan");

var nbEtudiants = **s.ajoutEtudiant(e1).ajoutEtudiant(e2).getNbEtudiants()**;

# Écriture manuelle de tests unitaires

## Soit le fichier Main.hx

La classe Main est la classe qui contient le point d’entrée du programme (fonction statique main). Cette classe est un cas de test (cf TestCase), elle contient des fonctions dont le nom commence par test et dont le rôle consiste à tester les classe métier ; exemple :

* La fonction testNewEtudiantNullName vérifie que la création d’un étudiant échoue lorsqu’on transmet un nom ayant une valeur nulle.
* La fonction testNewEtudiantOk vérifie la correcte initialisation des propriétés de l’étudiant.

Les fonctions de test doivent contenir au moins une assertion (assertEquals, assertTrue / False), c’est-à-dire vérifier qu’une valeur est bien égale à ce qui est attendu ; exemples :

var ch : String = "Hello";

assertEquals("HELLO", s.toUpperCase());

assertTrue(10<100);

La classe Main utilise donc les classe TestCase et TestRunner (pour exécuter des cas de tests) de la bibliothèque standard Haxe (paquetage haxe.unit – pour tests unitaires).

import haxe.unit.TestCase;

import haxe.unit.TestRunner;

class **Main extends TestCase** {

static function main() {

**var runner = new TestRunner();**

**runner.add(new Main());** //ajoute un cas de test

**runner.run();** //lance les tests

}

function new() { super(); } //constructeur du cas de test

function **test**NewEtudiantNullName() {

var e : Etudiant;

**try** { //essayer

e = new Etudiant(null, ""); //devrait lever une exception

assertFalse(true); //cette instruction ne devrait jamais être exécutée

} **catch** (e : String) { //interception de l’exception

assertFalse(false); //effectivement, la création ne devrait pas réussir

}

}

function **test**NewEtudiantOk() {

var e = new Etudiant("Simmons", "Dan");

assertEquals("Simmons", e.getNom());

assertEquals("Dan", e.getPrenom()); //le prénom de l’étudiant devrait être "Dan"

assertEquals(null, e.classe);

}

}

Ajouter des tests permettant de vérifier le bon fonctionnement :

* de la modification du prénom d’un étudiant,
* de l’ajout d’un étudiant à une section.

Penser à traiter les erreurs (nouveau prénom vide ou valeur nulle pour l’ajout d’un étudiant).

Essayer le programme de tests, et effectuer des modifications permettant de faire échouer (!) les tests.

# Génération automatique des tests unitaires et de la documentation

La solution précédente permet de déceler des erreurs au sein des classes métiers, grâce à un programme de test indépendant. Une autre approche existe et consiste à écrire les tests directement au sein des classes métiers, sous la forme de commentaire, ce qui facilite leur maintenabilité et contribue à la documentation.

Pour cela, une bibliothèque additionnelle est nécessaire ; l’installer : haxelib install haxe-doctest puis consulter sa documentation : <https://github.com/vegardit/haxe-doctest>

Pour pouvoir ultérieurement générer la documentation, installer aussi la bibliothèque « dox » (cf <https://github.com/HaxeFoundation/dox>).

## Tests unitaires

### Le programme principal

Dupliquer le dossier contenant le code réalisé au cours de l’étape précédente (cf : cp -R tp2a tp2b).

Ajouter la directive de compilation suivante au fichier build.hxml ; la directive lib permet de spécifier les bibliothèques additionnelles nécessaires à la compilation :

-lib haxe-doctest

Supprimer les fonctions test\* du fichier Main.hx et ajouter la macro suivante juste au-dessus de la déclaration de la classe cas de test :

@:build(hx.doctest.DocTestGenerator.generateDocTests())

Les macros sont des portions de code exécutées avant compilation et qui permettent (entre autres) de générer du code. Grâce aux commentaires qui vont être ajoutés aux méthodes des classes métiers, cette macro va générer automatiquement les fonctions de test du même type que celles qui viennent d’être supprimées.

Le code du fichier Main.hx devient :

import haxe.unit.TestCase;

import haxe.unit.TestRunner;

@:build(hx.doctest.DocTestGenerator.generateDocTests())

class **Main extends TestCase** {

static function main() {

**var runner = new TestRunner();**

**runner.add(new Main());**

**runner.run();**

}

function new() { super(); }

}

### La classe Etudiant

Ajouter les commentaires suivants au-dessus du constructeur :

/\*\*

\* <pre><code><i>Tests:</i>

\* **>>> new Etudiant(null, null) throws "'nom' should not be null or empty !"**

**\* >>> new Etudiant("Asimov", "Isaac") throws nothing**

**\* >>> new Etudiant("Asimov", "Isaac").getNom() == "Asimov"**

\* </code></pre>

\* @param nom the student lastname (a non null nor empty String)

\* @param prenom the student firstname (a non null nor empty String)

\* @return the new object instance

\* @throws String if nom or prenom are empty or null

\*/

Chaque ligne commençant par >>> est un test. Le format d’une telle ligne est le suivant :

instruction **throw** valeur l’instruction est censée lever une exception

instruction **throw** nothing l’instruction ne devrait pas lever d’exception

instruction **==** valeur l’instruction devrait retourner la valeur

Les directives @param, @return et @throws permettent respectivement d’expliciter les paramètres, la valeur retournée par la fonction et les exceptions qui pourraient être levées.

Ajouter deux tests :

* le premier pour vérifier qu’une exception est levée lorsque le nom est valide mais pas le prénom ;
* le second pour vérifier que le prénom est bien affecté.

Ajouter les commentaires suivants à la méthode setNom :

/\*\*

\* <pre><code><i>Tests:</i>

\* >>> new Etudiant("Asimo", "Isaac").setNom("") == "Asimo"

\* >>> new Etudiant("Asimo", "Isaac").setNom(null) == "Asimo"

\* >>> new Etudiant("Asimo", "Isaac").setNom("Asimov") == "Asimov"

\* </code></pre>

\* Change the student name.

\* @param nom the new lastname

\* @return the student current lastname that should have been changed if valid

\*/

De même, commenter la méthode setPrenom.

### La classe Section

Ajouter le commentaire suivant à la méthode ajoutEtudiant :

/\*\*

\* <pre><code><i>Tests:</i>

\* >>> new Section().ajoutEtudiant(new Etudiant("...", "...")).getNbEtudiants() == 1

\* >>> new Section().ajoutEtudiant(new Etudiant("Herbert", "Frank")).getEtudiants()[0].   
 getNom() == "Herbert"

\* </code></pre>

\* Add a student to the section.

\* @param e the student to add

\* @return the section, that should have been added a new student (unless null)

\*/

## Documentation

La documentation a déjà été intégrée dans le code et devrait permettre à d’autres développeurs d’utiliser les classes métier avec facilité. Pour améliorer l’ergonomie, l’outil (la bibliothèque) « dox » permet de générer une documentation au format HTML, du même type que celle d’OpenFL ou de la bibliothèque standard.

Pour automatiser cette tâche, créer le fichier gendoc.hxml avec le contenu suivant :

-cp src

-D doc-gen

-xml doc/doc.xml #inspecte les classes et génère doc.xml

-main Main

-lib haxe-doctest #nécessaire à cause de la macro

--next #pour indiquer une deuxième opération

-cmd haxelib run dox -i doc -o doc #génère les fichiers html à partir du XML

Générer la documentation avec la commande ; haxe gendoc.hxml, puis ouvrir le fichier index.html du dossier doc (qui vient d’être créé) pour vérifier le résultat.

*Remarque : la documentation a été écrite en respectant le standard « JavaDoc » (également utilisé par les autres langages de programmation).*