public class Main {

public static void main(String[] args) {}

// Calcule le nb de carrés entre A et B

public static int nbCarre(int A, int B){

int cpt = 0;

for(int i=A; i<=B; ++i){

\* Test de la racine carré :

\* - convertion de la racine carré en int (perte de la virgule)

\* - re-convertion en float

\* - si ce float == la racine carré, alors la perte de la virgule n'a pas eu d'incidence

\* => nous sommes en présence d'un carré.

if(Math.sqrt(i) == ((float) ((int) Math.sqrt(i))) ){

cpt++; } }return cpt; }

\* Créer une matrice de nombre aléatoire entre A et B

public static int[][] matrice(int M, int N, int A, int B){

int[][] matrix = new int[M][N];

for(int i=0; i<M; ++i){

for(int j=0; j<N; ++j){

matrix[i][j] = rnd(A, B); }} return matrix; }

\* Créer un nombre aléatoire entre la partie haute (A) et la partie basse (B)

public static int rnd(int A, int B){

return (int)(Math.random() \* (B - A)) + A; }}

package com.td; package com.test;

public class Main { import static org.junit.Assert.\*;

public static void main(String[] args) { import org.junit.Test;

public static int multiplier(int a, int b){ import com.td.Main;

return a\*b; }} public class TestMultiplier {

@Test

public void testMultiplier(){

assertEquals(6, Main.multiplier(2, 3));

assertEquals(6, Main.multiplier(3, 2));

assertEquals(0, Main.multiplier(0, 3));

assertEquals(0, Main.multiplier(3, 0));

assertEquals(-4, Main.multiplier(-2, 2)); }}

package com.test; TEST NbCARRE

import static org.junit.Assert.\*;

import org.junit.Test;

import com.td.Main;

public class TestNbCarre {

\* Dans cette fonction nous testons l'encadrement de notre système

@Test

public void testNbCarre(){

//0x0 donne toujours 1 carré, 0...

assertEquals(1, Main.nbCarre(0, 0));

//Idem 1x1

assertEquals(1, Main.nbCarre(1, 1));

//Il n'y a pas de racine carré sur les nbs négatifs

assertEquals(0, Main.nbCarre(-3000, -1));

//Entre 2 et 25, nous avons exactement 4 carrés : 2x2, 3x3, 4x4 et 5x5

assertEquals(4, Main.nbCarre(2, 25));

}}

//Ici si nous voulions pousser au maximum, nous devrions avoir une récursivité sur un pas de 10

//Permettant de vérifier que le nbs de carrés sur des tranches de 10, ne peut que diminuer au fur et à mesure

MONEY.JAVA

package com.td;

public class Money {

private int fAmount;

private String fCurrency;

public Money(int fAmount, String fCurrency){

this.fAmount = fAmount;

this.fCurrency = fCurrency; }

public int amount(){

return this.fAmount; }

public String currency(){

return this.fCurrency; }

public Money add(Money m){

return this.add(m.amount(), m.currency()); }

public Money add(int namount, String ncurrency){

if(namount >= 0 && ncurrency.length() == 3){

return new Money(this.fAmount + namount, ncurrency); }

return this; }}

package com.test; TEST MATRICE

import static org.junit.Assert.\*;

import org.junit.Test;

import com.td.Main;

public class TestMatrice {

//Dans cette fonction, nous testons certaines paramètres déterministes

//Puis nous testons l'aléatoire

@Test

public void testMatrice(){

int[][] test = Main.matrice(5, 20, 4, 19);

//Test MxN

assertEquals(5, test.length);

assertEquals(20, test[0].length);

//Test tous les nombres sont entre 4 et 9

for(int i=0; i<test.length; ++i){

for(int j=0; j<test[0].length; ++j){

//Dans une assertion, nous pouvons envoyer un test "if"

assertFalse( (test[i][j] > 19 || test[i][j] < 4) ); }}

//Maintenant on test que deux matrices

//Pas plus de 10 cases identiques (5x20 : 100, donc 10% d'indetiques max)

int max = 10;

int[][] test2 = Main.matrice(5, 20, 4, 19);

for(int i=0; i<test.length; ++i){

for(int j=0; j<test[0].length; ++j){

if(test[i][j] == test2[i][j]){

max--; }}}

//Si max est négatif, nous avons rencontré plus de 5x le même nombre, sur la même case

assertTrue(max > 0); }}

package com.test; TEST MONEY

import static org.junit.Assert.\*;

import org.junit.Test;

import com.td.Money;

public class MoneyTest {

@Test

public void testMoney(){

Money m = new Money(5, "EUR");

assertEquals(5, m.amount());

assertEquals("EUR", m.currency());

//Valeur négative refusée, la valeur courante est donc reprise

m = m.add(new Money(-20, "EUR"));

assertEquals(5, m.amount());

assertEquals("EUR", m.currency());

//Mauvaise devise, refusé aussi...

m = m.add(new Money(20, "EUROS"));

assertEquals(5, m.amount());

assertEquals("EUR", m.currency());

m = m.add(new Money(30, "YEN"));

assertEquals(35, m.amount());

assertEquals("YEN", m.currency()); }}

package com.test; TEST MONEYBAG

import static org.junit.Assert.\*;

import java.util.ArrayList;

import org.junit.Test;

import com.td.Money;

import com.td.MoneyBag;

public class MoneyBagTest {

@Test

public void testMoneyEquals(){

Money m1 = new Money(5, "EUR");

Money m2 = new Money(5, "EUR");

Money m3 = new Money(10, "EUR");

Money m4 = new Money(5, "YEN");

MoneyBag test = new MoneyBag();

assertTrue(test.equalMoney(m1, m2));

assertFalse(test.equalMoney(m1, m3));

assertFalse(test.equalMoney(m1, m4));

assertFalse(test.equalMoney(m2, m3));

assertFalse(test.equalMoney(m2, m4)); }

@Test

public void testMoneyAdd(){

MoneyBag test = new MoneyBag();

//Test liste vide

assertEquals(0, test.getBag().size());

//Ajout d'un élément

test.add(new Money(5, "EUR"));

//On prend la liste et on vérifie quel est le seul élément dedans...

ArrayList<Money> list = test.getBag();

assertEquals(list.size(), 1);

Money item1 = list.get(0);

assertEquals(item1.currency(), "EUR");

assertEquals(item1.amount(), 5); }

@Test

public void testMoneySub(){

MoneyBag test = new MoneyBag();

//Test liste vide

assertEquals(0, test.getBag().size());

//Ajout d'un élément

test.add(new Money(5, "EUR"));

//On prend la liste et on vérifie quel est le seul élément dedans...

assertEquals(test.getBag().size(), 1);

//Ne dois pas enlever un élément

test.sub(new Money(10, "EUR"));

assertEquals(test.getBag().size(), 1);

//Ne dois pas enlever un élément (2)

test.sub(new Money(5, "YEN"));

assertEquals(test.getBag().size(), 1);

//Dois enlever l'élément

test.sub(new Money(5, "EUR"));

assertEquals(test.getBag().size(), 0); }

@Test

public void testMoneyNormalise(){

Money m1 = new Money(4, "EUR");

Money m2 = new Money(4, "EUR");

Money m3 = new Money(1000, "YEN");

Money m4 = new Money(10, "EUR");

MoneyBag test = new MoneyBag();

test.add(m1);

//m2 ne va pas être normalisé car identique

assertTrue(test.equalMoney(test.normalise(m2), m2));

//idem m1

assertTrue(test.equalMoney(test.normalise(m2), m1));

//Par contre m3 va être normalé en m4 (convertion YEN - EUR)

assertTrue(test.equalMoney(test.normalise(m3), m4));

//TODO : a vous de compléter les tests manquants (il manque 5 cas de tests) }}

package com.td; MONEYBAG.JAVA

import java.util.ArrayList;

public class MoneyBag {

private ArrayList<Money> Bag = new ArrayList<Money>();

public ArrayList<Money> getBag(){

return this.Bag; }

\* Normalise, transforme la monaie en monaie de la bonne devise

public Money normalise(Money m){

if(this.Bag != null && this.Bag.size() > 0){

Money first = this.Bag.get(0);

//Normalisation

If(!first.currency().equals(m.currency())){

if(first.currency().equals("EUR") && m.currency().equals("YEN")){

return new Money((int) (m.amount() / 100), "EUR");

}else if(first.currency().equals("EUR") && m.currency().equals("GBP")){

return new Money((int) (m.amount() / 1.2), "EUR");

}else if(first.currency().equals("YEN") && m.currency().equals("EUR")){

return new Money((int) (m.amount() \* 100), "YEN");

}else if(first.currency().equals("YEN") && m.currency().equals("GBP")){

return new Money((int) (m.amount() \* 120), "YEN");

}else if(first.currency().equals("GBP") && m.currency().equals("YEN")){

return new Money((int) (m.amount() / 120), "GBP");

}else if(first.currency().equals("GBP") && m.currency().equals("EUR")){

return new Money((int) (m.amount() / 1.2), "GBP"); }}}

//On valide la valeur par défaut s'il n'y a rien a faire

return m; }

\* Test l'égalité

public boolean equalMoney(Money m1, Money m2){

return (m1.amount() == m2.amount() && m1.currency()

.equals(m2.currency())); }

\* Ajout d'un Money dans la liste

public void add(Money m){

this.Bag.add(m); }

\* Suppression d'un Money dans la liste

public void sub(Money m){

for(int i=0; i<this.Bag.size(); ++i){

if(this.equalMoney(this.Bag.get(i), m)){

this.Bag.remove(i); }}}}

**Types de tests**

Il y a plusieurs type de tests qui interviennent tout au long du développement logicie :

· Tests unitaires : test de chaque module du logiciel indépendamment des autres. La granularité d'un module peut varier selon les cas (fonction, package, classe, …)

· Tests d'intégration : test des différents modules les uns avec les autres

· Tests systèmes ou de validation : valider le logiciel par rapport aux spécifications du client en évaluant également des composantes spécifiques telles que la sécurité ou les performances

· Tests de régression : vérifier que les mise à jours du logiciel n'ont pas introduits de bugs.

**Déroulement classique d'un test**

Quelque soit le type de test, on retrouve majoritairement la procédure suivant :

1. Choisir un cas de test (CT)

2. Instancier une donnée de test (DT) à partir du CT

3. Prévoir le résultat attendu à partir de la DT (Oracle)

4. Exécuter le programme sur la DT

5. Comparer le résultat obtenu avec l'oracle

On distingue néanmoins les tests faits à partir de spécifications: Boîte Noir (BN) et les test faits à partir du code source : Boîte Blanche (BB). Les tests BN ne nécessites pas de disposer du code source mais restent approprié au tests systèmes ainsi qu'à certains tests unitaires. Les tests BB, permettent de découvrir des défauts fins de programmation mais pas les fonctionnalités absentes. Une bonne suite de tests mélange donc habilement BB et BN. Il existe enfin une dernière manière d'effectuer des tests, dite aléatoire ou statistique mais nous ne la détaillerons pas dans ce cours.

**Sélection des tests**

Selon le type de tests (BB ou BN), la sélection des tests à réaliser va dépendre de critères différents.

**1 Sélection des tests en BB :**

Dans le cadre des BB, le code source est disponible et nous allons nous en servir afin d'explorer les différents comportements possible du programme. L'objectif est de déterminer les chemins minimaux entre l'entrée et la sortie du programme afin d'assurer que :

· toutes les conditions d'arrêts des instructions itératives (boucles) ont été testées.

· Toutes les branches des instructions conditionnelles (if) ont été testées.

· Toutes les structures de données internes (variables locales) ont été testées.

Pour ce faire, une méthode consiste à déterminer la mesure de complexité de Mac Cabr. Cette mesure se calcule à partir du graphe de flot du programme par la formule suivante :

Nombre cyclomatique = Nb Arcs – Nb Noeud +2

Cette valeur détermine le nombre de chemins minimums (égal au nombre de régions du graphe de flot) qu'il va falloir tester (un test par chemin pour tester toutes les possibilités du programme)

**2 Sélection des tests en BN**

Dans le cadre des BN, le programme est vu sous un aspect fonctionnel au lieu de structurel car on a pas accès au code source. Il faut alors partitionner le domaine de définition des entrées en plusieurs classes et générer des cas de test aux limites des classes

**1 Présentation :**

JUnit est un « framework » permettant l'automatisation des tests unitaires pour les classes Java,

crée par Kent Beck et Erich Gamma. La version actuelle est Junit 4, qui apporte des évolutions

majeurs par rapport à la précédente version Junit 3.8. Nous nous efforcerons de présenter les

différences essentielles entre ces deux versions, bien qu'il existe une compatibilité descendante

entre Junit 4 et Junit 3.8. En effet, la plupart des entreprises utilise encore la version 3, qui

n'autorise par exemple pas les annotations dont la version 4 fait largement usage. Ces annotations

permettent de tirer partie des ajouts faits dans Java à partir de Java 5.0.

Junit peut être utilisé directement en ligne de commande (Il s'agit d'un jar), ou via un plugin pour

la plupart des IDE tels eclipse, netbeans ou intellij.

**2 Test Unitaires :**

Les tests unitaires sont des programmes écrits par le développeur qui exécute des fonctionnalités

spécifiques d'un code à travers certains tests. L'objectif des test unitaires est de s'assurer qu'un

code fonctionne comme prévu par les différents tests choisis (cf cours précédent). Les tests

unitaires permettent également de valider qu'un code fonctionne toujours correctement après

certaines modifications qu'on lui aurait apporté. On parle dans ce cas de tests de non régression,

sur lesquelles nous reviendront au prochain cours.

Les différentes étapes pour l 'exécution de tests unitaires sont les suivantes :

1. Préparer (set up) l'environnement de test pour être exactement dans les conditions requises

par le test. Cela revient à définir et appliquer des pré-conditions (par exemple initialiser des

champs, activer les logs, …)

2. Exécuter les cas de test (CT) que vous avez prévus lors de la phase d'écriture du

programme (cf cours précédent).

3. Evaluer les résultats ou effets de bords engenfrés par l'exécution des CT pour vérifier qui'ils

correspondent à ce qui a été prévu par les oracles.

4. Nettoyer (tear down) l'environnement de test si besoin, afin que les tests suivants puissent

être exécutés sans problèmes ( post-conditions).

**3 Implémentation des tests unitaires avec Junit 4 :**

JUnit est un « framework » permettant d'implémenter et de d'exécuter des test unitaires selon les

étapes vues ci-dessus. L'ensemble de ces étapes est implémenté au sein d'une classe Java test.

Les méthodes de la classe test sont décorées par des annotations permettant à Junit d'effectuer

les actions suivantes :

1. Set up : les actions devant être exécutées pour préparer l'environnement sont

implémentées dans une ou plusieurs méthodes, annotées avec ***@Before***.

2. Test : les CT sont implémentés dans des méthodes annotées avec ***@Test***. L'évaluation des

résultats attendus (oracle), est également implémentée dans ces méthodes en utilisant des

assertions (par exemple ***assertEquals(ValeurAttendue,ValeurCalculee)***).

3. Tear down : le nettoyage de l'environnement peut être implémenté dans une ou plusieurs

méthodes annotées par ***@After***, qui sont exécutées après chaque test.

**1 Test unitaire de non régression:**

Un test unitaire de régression s'applique à chaque fois que le logiciel est modifié, afin de s'assurer

que ce qui fonctionnait avant les modifications fonctionne toujours

L'intérêt de modifier un code déjà testé, permet soit de corriger des défauts, soit d'ajouter des

fonctionnalités. Ces ajouts ont lieu en phase de maintenance (TMA) ou en évolution de versions.

De manière générale, les tests de non régressions s'appliquent à tous les types de tests (unitaires,

intégration, systèmes), mais nous ne nous interessons pour l'instant qu'aux tests unitaires.

On distingue deux problèmes spécifiques :

1. Evolution/maintenance : les tests (DT/Oracles) sont-ils toujours valides

2. Sélection/minimisation : faut-il tout rejouer, ce qui peut être coûteux, et sinon comment

sélectionner un sous ensemble pertinent.

Grâce à des plugins comme celui présent dans Eclipse, il est possible de facilement rejouer des

tests et ou détecter les couples (DT/Oracles) qui ne sont plus syntaxiquement corrects.

**2 Assertions :**

***asserEquals(a,b)*** : teste si a,b sont des valeurs de type primitif égales ou

des objets possédant une méthodes equals

***assertTrue(a)***:teste si a (expression booléenne) est vrai

***assertFalse(a)*** :teste si a (expression booléenne) est vrai

***assertSame(a,b)*** :teste si a et b réfèrent au même objet

***assertNotSame(a,b)*** :teste si a et b ne réfèrent pas au même objet

***assertNull(a)*** : teste si un objet a est null

***assertNotNull(a)***: teste si un objet a n'est pas null

***fail(message)*** : teste si le test doit échouer (levée d'exception)

**Tester en isolation** Le test des interactions est un moyen pour tester une classe en isolation, cad

indépendamment des classes dont elle dépend. Exemple : pour une classe MachineCrous qui utilise

une classe CarteCrous, on souhaite tester les fonctionnalités de MachineCrous sans du tout utiliser

la classe CarteCrous. Si on teste MachineCrous en utilisant directement CarteCrous, c'est plutôt du test

d'intégration.

**Pourquoi tester en isolation ?**

{ Si un test de MachineCrous échoue, doit-on chercher l'erreur dans MachineCrous ou dans CarteCrous ?

{ Les dépendances ne sont peut-être pas encore développées.

{ Les dépendances sont parfois difficilement contrôlables :

{ comment simuler une panne de réseau ?

{ que faire si la dépendance a un comportement aléatoire ?

{ Les dépendances sont parfois lourdes \_a mettre en place : base de données.

**Tester les interactions** Le principe est de :

{ remplacer les implantations des dépendances par des doublures, ou mocks. Les mocks sont des objets

du test qui simulent les objets du domaine pour la durée du test.

{ vérifier que la classe à tester appelle comme il faut les méthodes des mocks.

{ attention : on mocke les dépendances, on ne mocke pas la classe \_a tester !

**Dépendances explicites** Pour pouvoir remplacer les implantations par des doublures, il faut aussi

que les dépendances apparaissent explicitement (par exemple en argument du constructeur ou d'une

méthode). Ce pattern s'appelle injection des dépendances : il permet un couplage faible entre les classes

et favorise les architectures testables.

**Dépendances \_a des abstractions** Il faut que la classe \_a tester dépende d'interfaces (abstractions) et

non directement des implantations.

**Test des interactions / Test d'état** On n'a pas systématiquement besoin de l'un ou de l'autre, on

a parfois besoin des deux. Le test d'interaction demande obligatoirement un mock.

**Description des mocks** Les mocks peuvent ^etre codés \_a la main. Ils peuvent avantageusement ^être

génerés automatiquement \_a partir d'une description de leur comportement (stubbing). Les vérifications à

faire en interrogeant les mocks peuvent aussi être décrites. cf Mockito.

package com.memory; MEMORYSTORE TP NOTE

import java.util.HashMap;

import java.util.Timer;

import java.util.TimerTask;

class TimeoutMemoryStore extends TimerTask{

private String key;

\* Constructor

public TimeoutMemoryStore(String key){ this.key = key; }

@Override

public void run() {

if(this.key != null & this.key.length() > 0){

MemoryStore.delete(this.key); } } }

public class MemoryStore {

//This will store any kind of object, related to a specific key value in string

private static HashMap<String, Object> mem;

public static String add(Object obj){

return add(obj, 0); }

public static String add(Object obj, long timeout){

//If the system is not functionnal

if(mem == null){

erase(); }

//Generating a unique id

String gen = generate(16);

while(mem.containsKey(gen)){

gen = generate(16); }

mem.put(gen, obj);

//Create a delete operation after timeout

if(timeout > 0){

new Timer().schedule(new TimeoutMemoryStore(gen), timeout); } return gen; }

public static Object get(String key){

if(mem != null && mem.containsKey(key)){

return mem.get(key);

}else{ return null; } }

public static boolean delete(String key){

if(mem != null && mem.containsKey(key)){

mem.remove(key);

return true; }

return false; }

public static void erase(){

mem = new HashMap<String, Object>(); }

private static String generate(int length){

String chars = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890";

int charLength = chars.length();

String pass = "";

for(int x=0; x<length; ++x){

int i = (int) Math.floor(Math.random() \* charLength);

pass += chars.charAt(i); }

return pass; } }

import junit.framework.JUnit4TestAdapter;

import org.junit.Before;

import org.junit.BeforeClass;

import org.junit.Ignore;

import org.junit.Test;

\* A sample test case, testing {@link java.util.Vector}.

public class ListTest {

protected List<Integer> fEmpty;

protected List<Integer> fFull;

protected static List<Integer> fgHeavy;

public static void main(String... args) {

junit.textui.TestRunner.run(suite()); }

@BeforeClass

public static void setUpOnce() {

fgHeavy = new ArrayList<Integer>();

for (int i = 0; i < 1000; i++) {

fgHeavy.add(i); } }

@Before

public void setUp() {

fEmpty = new ArrayList<Integer>();

fFull = new ArrayList<Integer>();

fFull.add(1);

fFull.add(2);

fFull.add(3); }

public static junit.framework.Test suite() {

return new JUnit4TestAdapter(ListTest.class); }

@Ignore("not today")

@Test

public void capacity() {

int size = fFull.size();

for (int i = 0; i < 100; i++) {

fFull.add(i); }

assertTrue(fFull.size() == 100 + size); }

@Test

public void testCopy() {

List<Integer> copy = new ArrayList<Integer>(fFull.size());

copy.addAll(fFull);

assertTrue(copy.size() == fFull.size());

assertTrue(copy.contains(1)); }

@Test

public void contains() {

assertTrue(fFull.contains(1));

assertTrue(!fEmpty.contains(1));

}

@Test(expected = IndexOutOfBoundsException.class)

public void elementAt() {

int i = fFull.get(0);

assertTrue(i == 1);

fFull.get(fFull.size());

// Should throw IndexOutOfBoundsException

}

@Test

public void removeAll() {

fFull.removeAll(fFull);

fEmpty.removeAll(fEmpty);

assertTrue(fFull.isEmpty());

assertTrue(fEmpty.isEmpty());

}

@Test

public void removeElement() {

fFull.remove(new Integer(3));

assertTrue(!fFull.contains(3));

}

}

package com.memory; MAIN TP NOTE

public class Main {

public static void main(String[] args) {

//Setting it to 200 ms, so you can check by yourself it is working

long timeout = 200;

String key = MemoryStore.add(new String("yatta"), timeout);

for(int i=0; i<10000; ++i){

System.out.println((String) MemoryStore.get(key));

}

}

}

package test.memory;

import static org.junit.Assert.\*; TESTMEMORYSTORE TP NOTE

import org.junit.After;

import org.junit.Test;

import com.memory.MemoryStore;

public class StoreTest {

@After

public void after(){

//recreate an empty system

MemoryStore.erase(); }

@Test

public void testAdd(){

String test = "ok";

//Try a first time

String key = MemoryStore.add(test);

String result = (String) MemoryStore.get(key);

assertEquals("ok", result);

//Try a second time

String key2 = MemoryStore.add(test);

String result2 = (String) MemoryStore.get(key2);

assertEquals("ok", result2); }

@Test

public void testErase(){

String test = "ok", test2 = "ok2";

String key = MemoryStore.add(test);

String key2 = MemoryStore.add(test2);

assertEquals("ok", MemoryStore.get(key));

assertEquals("ok2", MemoryStore.get(key2));

MemoryStore.erase();

assertEquals(null, MemoryStore.get(key));

assertEquals(null, MemoryStore.get(key2));

}

@Test

public void testDelete(){

String test = "ok", test2 = "ok2";

String key = MemoryStore.add(test);

String key2 = MemoryStore.add(test2);

assertEquals("ok", MemoryStore.get(key));

assertEquals("ok2", MemoryStore.get(key2));

MemoryStore.delete(key);

assertEquals(null, MemoryStore.get(key));

assertEquals("ok2", MemoryStore.get(key2));

//Test deleting not existing item

assertEquals(false, MemoryStore.delete("ab09"));

}

@Test

public void testAutoDelete(){

//Test the temporary system is working great

String test = "ok";

String key = MemoryStore.add(test, 200);

assertEquals("ok", MemoryStore.get(key));

try {

Thread.sleep(400);

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} finally {

assertEquals(null, MemoryStore.get(key)); }}}