Formation C++ Ubisoft - Module 6

Romain Arcila[1,2]
Charles de Rousiers[1]

10 mai 2009

[1] INRIA Grenoble
[2] Liris - CNRS Lyon

Plan

Test unitaire

Les principaux points abordés :

- Principe de base
- Mise en place dans un projet

Objectif

Comprendre le fonctionnement des tests unitaires

- Savoir comment les rédiger
- Savoir comment les intégrer dans du code existant

Plan

Exercices

Exercices pratiques

1. Écriture d'une classe simple et mise en place de test
2. Mise en place de tests dans du code existant

Plan

Généralités

Qu'est ce qu'un test ?

- Procédure de vérification partielle d'un système
- Paraphrase : Bout de code permettant d'exécuter un autre bout de code et d'analyse le résultat

Pourquoi faire des tests ?

- Trouver le nombre maximum de comportements problématiques.
- Permet de vérifier/valider le bon comportement du système par rapport à ce qui était prévu
- Permet de détecter au plus tôt des problèmes de fonctionnement (cf. assert de compilation)

Généralités

Schéma de test

- Données d'entrées
- Objet à tester
- Résultats attendus

Donnée entrée → Objet de test → Observation attendue

Si résultats différents des résultats attendus : Erreur !

- Modification de l'objet
- Relancement de la procédure de test

Remarques

Les tests doivent posséder les caractéristiques suivantes :

- **Répétabilité** : test doivent être indépendants de l'environnement, nécessaire pour l'exécution automatique des tests
- **Non exhaustivité** : pas possible de tout tester dû à la combinatoire trop élevée

Généralités

Types de tests : Différentes possibilités de classification

Par niveau de développement

- Test unitaire
- Test d'intégration
- Test système
- Test d'acceptation

Par niveau d'accéssibilité

- Test boîte blanche
- Test boîte noire

Par caractéristique

- Test fonctionnel
- Test de performance
- Test de robustesse
- Test de vulnérabilité

Généralités

Test boite blance / boite noire

- **Boite blanche** : connaissance sur la structure interne. Permet de tester plus facilement certaine partie du code.
- **Boite noire** : seule la spécification est connue, aucune information sur la structure interne

Test de non régression

- Permet d'assurer que ce qui fonctionnait fonctionne toujours avec les nouvelles modifications apportées au code
- Contrairement aux niveaux de test présenté précédement, un test de non régression est un type de test

→ Les risques de regression sont extrêmement important dans le code réutilisé et lors des étapes de maintenance

Test de recette vs. unitaire

Attention

Tests de recette vs. tests unitaires ⇒ **objectifs différents** !

Tests unitaires

- Donne un retour plus rapide
- Scénario simple
- Coût d'écriture faible
- Coût d'exécution faible
- Exécution après chaque compilation
- Connaissance de la structure interne (boîte blanche)
- Tests effectués très tôt dans le développement

Test de recette vs. unitaire

Test de recette

- Aucune connaissance a priori de la structure (boîte moire)
- Exécution uniquement lorsque toute l'application est compilée
- Tests effectués uniquement lorsque toutes les fonctionnalités sont implémentées
- Niveau de complexité supérieur
- Forte interaction entre plusieurs classes
- Coût d'exécution important (scénario complexe)
- Retour plus intéressant (vérification du comportement haut niveau) mais plus long

Dans la pratique

Constat

Méthode de test des développeurs au quotidien :

1. Truffer le code de traces temporaires
2. Lancer l'application
3. Vérifier ce qui est plus ou moins attendu
4. Supprimer / commenter le code de tracer

Bilan : Stratégie peu efficace

- Contrairement à ce qu'on pourrait croire : rédaction couteuse
- Ne sont que temporaires car supprimés == perte de temps (par d'amortissement d'écriture)
- Ne sont que pour des tests immédiats (pas pour des tests de non régression)

Pourquoi tester ?

Objectifs de la mise en place de tests

Avoir du code de tests pour :

- Déceler les bugs immédiats
- Déceler les bugs liés à l'évolution du code
- Effectuer ces vérification de façon automatique (gain de temps)

$\rightarrow$ **Permet d'amortir le coût de rédaction des tests**

Intérêts des **tests unitaires**

- Présente une 'simplicité' de mise en oeuvre
- Permet de tester efficacement le code de façon automatique
- Présente une granularité fine
- Permet un 'feeback' rapide pour un cout moins important

Les approches

Test-Driven

- Rédaction des spécifications et des tests d'une classe avant de coder son implémentation
- Tests uniquement avec des petites portions de code (scénario simple)
- Utiliser dans des méthodes de développement type eXterme programming

Principaux avantages

- Détection des problèmes de design (utilisation, couplage)
- Facilité la rédaction des tests : notion de testabilité
- Détection des bugs au plus tôt
- Permet d'écrire des spécifications plus claires
- Automatisation des tests
- Évite de frustrer le codeur de faire des tests qu'il croit inutiles

Code correspondant uniquement aux tests

- Code simple et non superflu
- Produit une conception simple : facile à utiliser et à maintenir

Les approches

Processus

Bonne pratique : écriture des tests avant même d'écrire le code de la fonction

1. Écriture des spécifications
2. Écriture du prototype
3. Écriture des tests
4. Écriture du code de la fonction
5. Test de la fonction

Dans méthode XP : travail en binôme

- Mise au point en commun
- Rédaction des tests par une personne
- Rédaction du code par l'autre
- Mise en commum pour effectuer les tests

Les approches

Si test échoue : réactivité facile et immédiate

- Programmeur connaît son code donc va plus vite
- Coût moins cher à corriger car pris à l'origine

Principaux désavantages :

- Demande un investissement
- Perçu comme laborieux et contraignant

Permet d'avoir un sentiment d'avancement : Code testé plusieurs fois par jour et réussi les tests

- Sentiment de confiance
- Sentiment d'avancement

Nécessite un investissement

- Apprendre à utiliser un framework de test
- Organiser le code pour le rendre testable
- Temps pour la rédaction des tests

→ Processus coûteux sur le court termes mais gagnant sur le moyen/long terme

Test unitaire

Organisation des tests :

- Cas de test : Porte sur une fonctionnalité (ex : méthode, portion de code) pour valider son comportement
- Suite de tests : Contient un ensemble de tests se rapportant à une même entité (ex : classe, module)

Couverture de code

- Consite à tester l'ensemble des instructions du code : teste tous les chemins d'exécution possibles
- Utilisé uniquement pour du code critique, car coûteux en mise en place et en exécution

Test de non régression

- Consiste à rejouer tous les tests déjà mis au point
- Assurer que les nouvelles modifications n'introduisent pas de nouveaux bugs

Mise en place

Comment faire ses test ?

- 1 classe == 1 ensemble de tests
- 1 méthode == 1 ou plusieurs tests

Problème

Avec cette technique : beaucoup de tests dont une grande majorité est **triviale** !

Définition

Scénario : suite d'action permettant de simuler une action donnée dans un environnement donnée

Mise en place

Choix des tests

Les test doivent être pertinents

- Ne sert à rien de tester les méthodes triviales type ‘assesseurs’

Trouver les bons cas de tests

- Partir de cas nominaux
- Étendre à des scénarios catastrophe

→ Donne point de départ et permet d’ajouter des tests d’invariant sur la classe

Remarque

- Les tests ne doivent pas couvrir 100% du code !
- Intérêt porté uniquement sur les portions importantes/à ‘risque’

Exemple

Exemple : Test d'une classe

Code

```
class BoundingBox
{
        vec3 GetMin(){ return min;}
        vec3 GetMax(){ return max;}
        void Merge(const BoundingBox& _box ) {
                min = Math::min(min,_box.min);
                max = Math::max(max,_box.max);
        }

        bool Intersect(const Ray& _ray ) {
                // Complex code ...
                return intersect;
        }
}
```

Exemple

Exemple : Test d'une classe

Code

```
void testMerge1() {
        BoundingBox box1(0,1);
        BoundingBox box2(0,2);
        box1.Merge(box2);
        ASSERT_EQUAL_MACRO(box1.min,0);
        ASSERT_EQUAL_MACRO(box1.max,2);
}

void testMerge2(){
        BoundingBox box1(0,1);
        BoundingBox box2(4,5);
        box1.Merge(box2);
        ASSERT_EQUAL_MACRO(box1.min,0);
        ASSERT_EQUAL_MACRO(box1.max,1);
}

bool testIntersect1() {
        // ...
}

bool testIntersect2() {
        // ...
}
```

Les données de test

Les données en entrée : quelle plage de données testées ?

- Ne pas tester toutes les valeurs possibles
- Test des valeurs représentatatives

Exemple : paramètre de type entier

- Test avec quelques chiffre dans la plage principales
- Test avec les chiffres bords(-1, 0, valeur maximum, ...)

Effet de bords

Faire attention à laisser l'environnement comme il était avant de faire le test
$\longrightarrow$ **Risque d'effet de bords**

Organisation du code

Comment organiser code source et code de test ?

Éviter de mélanger les deux types de code au même endroit :

- Nuit à l'organisation des sources
- Pas même personne ayant accès aux tests et au code

Recommendations

- Arborescance de tests parallèle
- Oragnisation différente
- Trouver une convention de nommage uniformes pour les fichiers de tests
- ...

Multi-threading

Comment effectuer les tests dans une application multi-threaded ?

Problèmes liés au multi-tâches

- Problème d'interblocage
- Problème de concurrence

Les problèmes lié à la testabilité

- Comportements non nécessairement reproductibles
- Temps d'exécution aléatoire/long
- ...

$\Rightarrow$ Tests difficiles pour ce type d'application

Mutli-threading

Comportement non reproductible

Infinité de comportement dû au comportement non déterminé

- Heureusement les tests sont soumis aux mêmes contraintes : donc constitue une infinité de tests
- Avec une grande exécution des tests : espérance de courir une bonne partie des cas possibles

Temps d'exécution aléatoire/long

Notion de temps : test doit durer un laps de temps maximum fixe
→ Permet de détecter interblockage ou autre ...

Mutli-threading

title

```
void subTestMT1()
{
        CollidableObject obj1, obj2;
        obj.collide(obj2); // Launch collision computation in new thread
}

void testMT1()
{
        Watchdog wd(subTestMT1);

        // Wait 5 seconds
        if (!wd.start(5000))
        {
                ASSERT_FAILED("Time out");
        }
}
```

Base de donnée

Contexte : Test de code ayant besoin d'un accès à une base de données

Base de test

Utiliser une base de tests indépendante de la base de production pour éviter :

- Suppression d'élément
- Ajout d'élément erroné
- Placer la base dans un état incohérent
- ...

Autre possibilité : Utilisation de 'mock' pour simuler les accès à la base de données (renvoie de données fixes et fictives)

Interface graphique

Contexte

Comment tester du code présentant une interface graphique ?

Les outils

- Pour les interfaces web : existance d'outils de saisie automatique
- Pour les interfaces lourdes : ? ? ?

Effectuer les tests sur le code sous jacent à l'interface graphique

- Éviter de tester les composants graphiques
- Tester les comportements invoqués sur les actions des composants graphiques

But : avoir un code de test automatique même pour les interfaces grahiques

Plan

Généralités

Contexte

- Comment mettre en place des tests unitaires ?
- De quoi a-t-on besoin ?

Les principaux besoins :

- Fonction de test
- Mécanisme d'assertion : test ok/ko
- Appel automatique de toutes les fonctions de tests
- Sortir la liste des tests ayant réussi/échoué
- En cas d'échec, localisation des tests en défaut

Remarque

Préférable que le framework de test soit robuste

- Ne plante pas pendant les tests
- Prend en charge la gestion des exceptions

→ **But** : Évite le lancement de tous les tests pendant la nuit et plantage au test 3 / 5000

Notion de framework

Comment faire pour avoir toutes ces fonctionnalités ?

- Coder tout à la main : long, pas forcement optimales
- Utiliser un framework de test tout fait

Existance de nombreux environnement de tests :

- CppUnit
- Boost.Test
- CppUnitLite
- NanoCppUnit
- Unit++
- CxxTest
- ...

Remarque

Existence de plusieurs sites comparant les avantages et les inconvénients de chacun :

- http ://gamesfromwithin.com/ ?p=29
- http ://www.puupuu.org/blog/index.php/2007/12/14/96-frameworks-de-tests-unitaires-en-c

Fonctionnalités

La plupart des framework proposent deux types de structure :

- Cas de test (test case) : structure/fonction contenant le test à effectuer
- Suite de test (test suite) : structure/fonction contenant tous les tests d'un élément (classe, module, ...)

Fixture

- Quand on teste une classe besoin potentiel d'un environnement donnée $\longrightarrow$ Simulation de l'environnement avec des fonction de context (fixtures)

Fonctions de contexte :

- Avant chaque appel de test utilisation de 'setup'
- Après chaque appel de test utilisation de 'teardown'

Fixture

Exemple : Besoin de tester une classe nécessitant un accès à une base de donnée

Étapes

Pour chaque fonction de test, on a :

1. Construction : Création du connecteur à la base de données
2. Fonction de test : Utilisation du connecteur à la base de données
3. Destruction : Destruction du connecteur à la base de données

Framework utilisé

Utilisation de framework : CppUnit

Principaux avantages

- Le/Un des framework le plus utilisé
- Format de sortie flexible : text console, XML, GUI, ...
- Documentation bien fournie
- Cross plateforme

Principaux inconvénients

- Assez verbeux
- Nécessite le RTTI
- Nécessite le support des exceptions

Utilisation

Trois types de classes importantes :

- **TestCase** : Classe la plus simple. Définit un cas de test. Redéfinition de la méthode runTest()
- **TestSuite** : Classe permettant de regrouper les différents cas de tests (Test Case) rattaché à une classe ou un module
- **TestFixture** : Classe permettant de regrouper plusieurs cas de tests et supportant un environnement de test

Remarque

En pratique on construit la plupart du temps les classes de tests à l'aide de la classe **TestFixture**.

Utilisation

Exemple : Utilisation de la classe TestCase

Code

```
class PseudoTest : public CppUnit::TestCase
{
public:
	PseudoTest(std::string name):CppUnit::TestCase(name) {}

	void runTest()
	{
		CPPUNIT_ASSERT( 1 == 1 );
		CPPUNIT_ASSERT( !(1 == 2) );
	}
};
```

Utilisation

Exemple : Utilisation de la classe TestCase

Code

```
class PseudoTest : public CppUnit::TestFixture
{
public:
        CPPUNIT_TEST_SUITE( PseudoTest );
                CPPUNIT_TEST(test1);
                CPPUNIT_TEST(test2);
        CPPUNIT_TEST_SUITE_END();

        void test1() {
                CPPUNIT_ASSERT(2 == 2);
        }

        void test2() {
                CPPUNIT_ASSERT(1 != 3);
        }
};
```

Utilisation

Exemple : Utilisation de la classe TestCase

Code

```
class PseudoTest : public CppUnit::TestFixture
{
public:
        CPPUNIT_TEST_SUITE( PseudoTest );
                CPPUNIT_TEST(test1);
                CPPUNIT_TEST(test2);
        CPPUNIT_TEST_SUITE_END();

        void test1() {
                CPPUNIT_ASSERT(b/a == 2);
        }

        void test2() {
                CPPUNIT_ASSERT(a + b == 3);
        }

        void setUp() {
                a = 1;
                b = 2;
        }

        void tearDown() {
                // nothing to do !
        }
private:
  int a, b;

};
```

Utilisation

Phase de construction d'un test

1. Dérivée la classe de test depuis CppUnit : :TestFixture
2. Construire les différentes fonctions de tests
3. Enregistrer la suite de tests avec CPPUNIT_TEST_SUITE_REGISTRATION (maClass) dans le .cpp
4. Enregistrer les fonction de tests dans la suite de tests dans le dans le .hpp
   - Commencer la liste avec CPPUNIT_TEST_SUITE(maClass)
   - Ajouter chaque fonction de test avec CPPUNIT_TEST(maFonction)

Remarque

Toutes ces macros peuvent sembler obscures mais permettent de simplifier la construction des tests

Utilisation

Gestion des l'environnement pour un cas de test

- Création de l'envrionnement : redéfinir la fonction setUp()
- Suppression de l'envirionnement : redéfinir la fonction tearDown()

Remarques

Pour pouvoir avoir un envrionnement propre à chaque test les attributs doivent :

- Être alloués dynamiquement dans la méthode setUp()
- Être désalloués dynamiquement dans la méthode tearDown()

Vérification des résultats : utilisation de fonction d'assertion

Type d'assertion

Différents types d'assertion disponibles suivant :

- Le type de vérification à faire
- Le type d'affichage souhaité

Fixture

Exemple : Besoin de tester une classe nécessitant un accès à une base de donnée

Code

```
class CustomerTest : public CppUnit::TestFixture {
public:
    void setUp() {
        proxy = new DatabaseProxy("login","pwd");
        proxy->Connect();
    }
    void tearDown() {
        proxy->Disconnect();
        delete proxy;
    }

    void testUpdateCustomer {
        // ...
    }

    DatabaseProxy* proxy;
};
```

Utilisation

Assertion de test

- CPPUNIT_ASSERT(condition) : teste une simple condition
- CPPUNIT_ASSERT_EQUAL (expected,actual) : teste si la valeur réelle est égale à la valeur attendue
- CPPUNIT_ASSERT_MESSAGE(message, condition) : teste la condtion et affiche un message si la condition est fausse
- CPPUNIT_ASSERT_DOUBLES_EQUAL(expected, actual, delta) : teste si la valeur réelle est égale à la attendue à un delta près

Assertion d'exception

- CPPUNIT_ASSERT_THROW(epression, exceptionType) : teste si l'expression lance l'exception attendue
- CPPUNIT_ASSERT_NO_THROW(expression) : teste si l'expression ne lance pas d'exception

Utilisation

Assertion d'erreur

- CPPUNIT_FAIL(message) : rate et affiche le message
- CPPUNIT_ASSERT_ASSERTION_FAIL(assertion) : teste si l'assertion rate
- CPPUNIT_ASSERT_ASSERTION_PASS(assertion) : teste si l'assertion réussi

Exemple

Exemple : Test d'une classe de fraction

Classe à tester

```
unsigned int gcf (unsigned int, unsigned int); // greatest common factor
unsigned int lcd (unsigned int, unsigned int); // least common denominator

class DivisionByZeroException{};

class Fraction {
    public:
        Fraction (int = 0, int = 1) throw (DivisionByZeroException);
        Fraction (const Fraction&);
        Fraction& operator= (const Fraction&);

        bool operator== (const Fraction&) const;
        bool operator!= (const Fraction&) const;

        friend Fraction operator+ (const Fraction&, const Fraction&);
        friend Fraction operator- (const Fraction&, const Fraction&);
        friend ostream& operator<< (ostream&, const Fraction&);

    private:
        void reduce (void);
        int numerator, denominator;
};
```

Exemple

Classe de test (.hpp)

```
#include <cppunit/TestFixture.h>
#include <cppunit/extensions/HelperMacros.h>

class fractiontest : public CPPUNIT_NS::TestFixture {
   CPPUNIT_TEST_SUITE(fractiontest);
             CPPUNIT_TEST(addTest);
             CPPUNIT_TEST(subTest);
             CPPUNIT_TEST(exceptionTest);
             CPPUNIT_TEST(equalTest);
   CPPUNIT_TEST_SUITE_END ();
   public:
      void setUp (void);
      void tearDown (void);
   protected:
      void addTest (void);
      void subTest (void);
      void exceptionTest (void);
      void equalTest (void);
   private:
      Fraction *a, *b, *c, *d, *e, *f, *g, *h;
};
```

Exemple

Classe de test (.cpp)

```
CPPUNIT_TEST_SUITE_REGISTRATION (fractiontest);

void fractiontest::setUp (void) {
    // set up test environment ( initializing  objects)
    a = new Fraction (1, 2);
    b = new Fraction (2, 3);
    c = new Fraction (2, 6);
    d = new Fraction (-5, 2);
    e = new Fraction (5, -2);
    f = new Fraction (-5, -2);
    g = new Fraction (5, 2);
    h = new Fraction ();
}

void fractiontest::tearDown (void) {
    // finally  delete  objects
    delete a; delete b; delete c; delete d;
    delete e; delete f; delete g; delete h;
}
```

Exemple

Classe de test (.cpp) (suite)

```
void fractiontest::addTest(void) {
    // check subtraction results
    CPPUNIT_ASSERT_EQUAL (*a + *b, Fraction (7, 6));
    CPPUNIT_ASSERT_EQUAL (*b + *c, Fraction (1));
    CPPUNIT_ASSERT_EQUAL (*d + *e, Fraction (-5));
    CPPUNIT_ASSERT_EQUAL (*e + *f, Fraction (0));
    CPPUNIT_ASSERT_EQUAL (*h + *c, Fraction (2, 6));
    CPPUNIT_ASSERT_EQUAL (*a + *b + *c + *d + *e + *f + *g + *h, Fraction (3, 2));
}

void fractiontest::subTest(void) {
    // check addition results
    CPPUNIT_ASSERT_EQUAL (*a - *b, Fraction (-1, 6));
    CPPUNIT_ASSERT_EQUAL (*b - *c, Fraction (1, 3));
    CPPUNIT_ASSERT_EQUAL (*b - *c, Fraction (2, 6));
    CPPUNIT_ASSERT_EQUAL (*d - *e, Fraction (0));
    CPPUNIT_ASSERT_EQUAL (*d - *e - *f - *g - *h, Fraction (-5));
}
```

Exemple

Classe de test (.cpp) (suite)

```
void fractiontest :: exceptionTest (void) {
    // an exception has to be thrown here
    CPPUNIT_ASSERT_THROW (Fraction (1, 0), DivisionByZeroException);
}

void fractiontest :: equalTest (void) {
    // test successful , if true is returned
    CPPUNIT_ASSERT (*d == *e);
    CPPUNIT_ASSERT (Fraction (1) == Fraction (2, 2));
    CPPUNIT_ASSERT (Fraction (1) != Fraction (1, 2));
    // both must have equal valued
    CPPUNIT_ASSERT_EQUAL (*f, *g);
    CPPUNIT_ASSERT_EQUAL (*h, Fraction (0));
    CPPUNIT_ASSERT_EQUAL (*h, Fraction (0, 1));
}
```

Exemple

Main

```
#include <cppunit/CompilerOutputter.h>
#include <cppunit/extensions/TestFactoryRegistry.h>
#include <cppunit/TestResult.h>
#include <cppunit/TestResultCollector.h>
#include <cppunit/TestRunner.h>
#include <cppunit/BriefTestProgressListener.h>

int main () {
    CPPUNIT_NS::TestResult testresult;
    CPPUNIT_NS::TestResultCollector collectedresults;
    testresult.addListener(&collectedresults);

    CPPUNIT_NS::BriefTestProgressListener progress;
    testresult.addListener (&progress);

        CPPUNIT_NS :: TestRunner testrunner;
    testrunner.addTest (CPPUNIT_NS::TestFactoryRegistry::getRegistry().makeTest());
    testrunner.run (testresult);

    CPPUNIT_NS::CompilerOutputter compileroutputter(&collectedresults, std::cerr);
    compileroutputter.write ();

    return collectedresults.wasSuccessful () ? 0 : 1;
}
```

Remarque

Recommendation : Pas plus d'une assertion par cas de test

Avantages

- Plus facile pour le débuggage : meilleure isolation du problème
- Test mieux ciblé (évite les scénarios compliqués)
- Risque plus faible d'effets de bord

Inconvénients

- Nécessite l'écriture de plus de fonctions

Plan

Mise en place

Question : Que doit-on tester ?

Tester avant tout les services rendus par la classe

- Méthode complexe
- Méthode métier

Attention

Pas de tests systématiques de toutes les méthodes d'une classe

- Ex : Les ascesseurs ne présentent pas d'intérêt

Possibilité de faire des tests pour des ensembles de classes

- Si les interactions sont fortes
- Et elles sont de petites tailles
- Et si elles ne présentent pas d'intérêt à être tester unitairement

Attention

Éviter de faire des tests non-unitaires (sur des ensembles de classe)

- Prend plus de temps à concevoir
- Difficulté d'investigation en présence de bug
- Évolution moins facile du code de test

Stratégie de mise en place

Étapes de conception d'une classe

1. Idée d'une classe
2. Rédaction des spécifications
3. Rédaction de l'interface
4. Rédaction des tests
5. Mise à joure de l'interface et des spécification
6. Rédaction de l'implémentation

Conception

La conception d'un programme doit s'adapter aux contraintes de testabilité !
→ Une application n'est testable (facilement) que si elle a été concue pour cela

Problèmes récurrents :

- Interface mal pensée
- Nombres de services trop important
- Nécessité de redécouper les actions en sous actions

Résultat

Adapation du design pour rendre la classe plus testable !
→ Permet d'avoir un ou plusieurs tests simples par méthode pertinente

Principaux avantages

- Permet de prendre en compte la conception
- Donne une vue utilisation de la classe
- Permet de juger rapidement des dépendances / du couplage

→ Donne une meilleure conception d'un point de vue utilisation

Conception

Exemple : Interface non testable car mal pensée

Code

```
class EmailChecker(){
        bool Check(const std::string& _email) const
        {
                //Code to check syntaxe
                //Code to check syntaxe
                //Code to check server
        }
};
```

Conception

Exemple : Transformation de l'interface pour la rendre testable

Code

```
class EmailChecker(){
        bool Check(const std::string& _email) const {
                CheckSyntaxe(_email);

                std::string domain = Tools::ExtractDomain(_email);
                CheckDomain(domain);

                std::string server = Tools::ExtractDomain(_email);
                CheckServer(server);
        }

        bool CheckSyntaxe() {
                //Code to check syntaxe
        }

        bool CheckDomain() {
                //Code to check syntaxe
        }

        bool CheckServer() {
                //Code to check server
        }
};
```

Conception

Contexte

Couplage obligatoire pour certaines classes

Comment tester code de façon indépendante ?

- Pour éviter les effets de bord
- Pour éviter la mise en échec non responsable
- Pour une débuggage plus efficace
- ...

Plusieurs solutions

- Self-shunt
- Mock object
- Extraction de code

Self-shunt

Définition

Idée : Implémenter les services nécessaires par la classe de test

- Technique de substitution de code
- La classe de test fait office de 'bouchon' (stub)

Caratéristiques :

- Héritage de la classe dépendante
- Implémentation rudimentaire des services nécessaires
- Nécessite un passage par référence ou par pointeur

Cas problématiques :

- Si l'objet testé détruit la classe dont elle dépend
- Si la classe communique avec plusieurs instances de la classe dont elle dépend
- Si le passage de l'objet dont elle dépend se fait par valeur (slicing effect)

Mock object

Définition

Idée : implémenter les services nécessaires par une classe factice

- Technique de substitution de code
- La classe factice fait office de 'bouchon' (stub)

Caratéristiques :

- Héritage de la classe dépendante
- Implémentation rudimentaire des services nécessaires
- Nécessite un passage par référence ou par pointeur

Self-shunt et Mock object

Comment mettre cela en place en C++ ?

- Méthodes à substituer **virtuelles** : dérivation simple
- Méthodes à substituer **non virtuelles** : impossible !

Remarque

Possibilité d'utiliser des frameworks spéciaux pour gérer les cas compliqués

- Mockpp
- MockItNow
- Moxy
- ...

Méthodes non virtuelles

Utilisation des méthodes de la classe mère donc substitution non valide

Mock object

Exemple : Mise en place d'un mock

Code

```
class A {
public :
        void FooA(const B& _b) {
                // ...
                _b.FooB();
                // ...
        }
};

class B {
        public: virtual void FooB() { /* ,,, */ }
};

class MockB : public B {
        public: virtual void FooB() { /* do other stuff */}
};

class TestCaseA : public CppUnit::TestCase
{
public:
        void runTest {
                A a;
                Mock mb;
                a.FooA(mb);
        }
}
```

Extraction

Définition

Idée : Extraire la portion de code intéressante et la mettre dans une nouvelle méthode

Caractéristiques :

- Ne nécessite plus d'écrire de code 'bouchon'
- Supprime le couplage sur le service à tester
- Nécessite un redesign de la classe
- Test devient plus facile à mettre en oeuvre

Extraction

Exemple : mise en place d'une extraction

Code

Voir code de refactorisation

Classe abstraite

Rappel

Classe abstraite : classe non instanciable dû à la non implémentation d'une ou plusieurs méthodes

Comment tester une classe abstraite ?

- Création d'un mock pour la classe
- Héritage de la classe abstraite
- Implémentation rudimentaire voire vide

$\longrightarrow$ Permet la validation des autres services de la classe

Plan

Généralités

Retro-fitting

Définition : intégration a posteriori des tests unitaires dans du code existant ('retrofitting unit tests')

Situations propices :

- Changement de politique de développement
- Réutilisation de code
- ...

Problème

Comment intégrer des tests unitaires a posteriori ?

Mise en place

Contexte : Souhait de mettre en place une politique de tests unitaires

Problème

Pas possible de geler tout le projet le temps de mettre en place tous les tests (trop coûteux)

Solution

Ajout progressif des tests

Mise en place

Situations propices à l'ajout de tests unitaires

- Ajout d'une nouvelle classe/méthode → Ajouter de tests unitaires
- Modification d'une classe/méthode → Ajouter de tests unitaires
- Refactorisation → Ajouter de tests unitaires
- Découverte d'un bug → Ajouter de tests unitaires
- Découverte de mauvais code → Refactorisation → Ajouter de tests unitaires

Si application fonctionne et souhait de mettre tout de même en place des tests unitaires

- Commencer par les tests des classes de plus haut niveau (approche top-down)
- Code a priori plus robuste à la base car plus ancien donc plus testé
- Permet de mettre en place rapidement des tests utiles

Mise en place

Détection de bug

- Trouver les cas permettant de reproduire le bug
- Associer le test à une classe
- Ajouter le test dans la base de tests

Problème

Une application n'est testable que si elle a été concue pour cela !

- Mise en place des tests : détection de problèmes de design
- Nécessité de faire de la refactorisation

→ Pas toujours évident : besoin de tout casser du code parfois robuste/délicat

Refactorisation

Phase de refactorisation

Conseil : utilisation de l'approche **TDD**

1. Rédaction des spécifications
2. Rédaction des tests
3. Puis enfin rédaction de la classe

Permet d'assurer :

- La testabilité de la classe
- Un couplage a priori faible de la classe

Refactorisation

Exemple : Refactorisation d'un distributeur de billet !

Code original

```
cashMachine.withdraw(200);

public class CashMachine {
  private User user = User.getCurrent();

  public void withdraw(int amount) {
    accountManager.debit(user.getAccountId(), amount);
  }
}
```

Code refactorisé

```
cashMachine.withdraw(200, User.getCurrent());

public class CashMachine {
//private User user = User.getCurrent();

  public void withdraw(int amount, User user) {
    accountManager.debit(user.getAccountId(), amount);
  }
}
```

Bénéfice et coûts

Bénéfices non immédiats car concerne principalement :

- Refactorisation du code pour une utilisation plus aisée (pas de plus-value immédiate)
- Détection de non-régression
- Documentation du code

Sur-coût

Entraine une inertie plus importante sur le projet momentanément
$\longrightarrow$ Ajoute un sur-coût au développement

Coûts

Problème

Pour les développeurs : premiers pas pour ce type d'approche entrain différents problèmes

- Validation de code qui marche a priori déjà
- Peu sembler une perte de temps
- Évaluation des bénéfices à court terme peu convaincant

Coûts

Problème

D'une manière générale les tests ont un coût :

- Volume code source $<$ Volume code test
- Temps d'exécution des tests grossissent
- Aspect pénible dans la rédaction de code de test qui marchera a priori (coût humain)

Remarque : Tendance à penser que le code n'évoluera pas (et donc pas besoin de faire des tests)

Conseils

- Isoler les tests dont le temps d'exécution est long
- Lancer leur execution périodiquement