# 1 Tests unitaires

## 1.1 Tests

- Avant de livrer un logiciel, on doit vérifier que
  - on peut construire le logiciel
  - on peut l'exécuter
  - il se comporte comme prévu
  - les résultats produits sont corrects
- Pour cela il y a différents types de tests :
  - tests unitaires : est-ce que les objets font ce qu'il faut, est-ce qu'ils sont faciles à utiliser

- tests d'intégration : est-ce que des modules fonctionnent bien ensembles
- test d'acceptation : est-ce que le système en entier fonctionne avec la nouvelle fonctionnalité
- tests de régression : est-ce qu'après des changements dans le système des fonctionnalités ne fonctionnent plus
- tests de validation et de vérification
- tests en cas d'épuisement des ressources, d'erreurs : comment le système se comporte dans les conditions réelles d'utilisation
- tests de performance : est-ce que le système est suffisamment rapide, ne consomme pas trop de ressources
- tests d'utilisabilité : est-ce que le système est facilement utilisable par l'utilisateur final (du point de vue du facteur humain)

## 1.2 Tests unitaires

- Un test unitaire est un morceau de code
  - automatisé
  - qui appelle la fonction/méthode/classe en train d'être testée
  - qui vérifie ensuite la véracité d'hypothèses faites sur le comportement des cette fonction/méthode/classe
    - état interne
    - valeurs calculées
    - interactions avec d'autres objets
  - si ces hypothèses ne sont pas vérifiées, le test échoue

- Un test unitaire est en général écrit avec une infrastructure de test unitaire qui doit permettre de :
  - d'écrire les tests simplement
  - spécifier les mises en place des bancs d'essai et leur nettoyage après le test
  - sélectionner seulement certains tests à exécuter
  - analyser les résultats en fonction de résultats attendus (ou inattendus)
  - produire un rapport des échecs

## 1.3 Tests unitaires traditionnels avec xxxUnit

- Écrire une classe de test dérivée d'une classe de l'infrastructure
- Écrire les tests dans des méthodes de test
- Écrire dans les tests des assertions permettant de vérifier des conditions
- Les méthodes seront exécutées et le rapport de test indiquera les assertions qui ont été vérifiées et celles qui ont échoué
- Quand plusieurs tests partagent la même initialisation de données :
  - des méthodes spéciales d'initialisation et de destruction
  - seront appelées systématiquement au début et à la fin de chaque test

```
class testPoint : public ??? {
  point* p;
public:
  //mise en place des tests
  void setup()    { p = new point{10.0,-5.2}; }
  //nettoyage des tests
  void teardown() { delete p; }
  //un test
  void testLeConstructeurInitialiseLePoint() {
    ASSERT_EQUAL???(p.x(),10.0);  //assertion
    ASSERT_EQUAL???(p.y(),-5.2);
  }
  //un autre test
  void testMoveDeplaceLePoint() {
    double dx = 1.2, dy=-3.2;
    p.move(dx,dy);
    ASSERT_EQUAL???(p.x(),10.0+dx);
    ASSERT_EQUAL???(p.y(),-5.2+dy);
  }
};
```

# 2 Tests unitaires avec doctest

- doctest (github.com/onqtam/doctest) : infrastructure de tests
  - plus adaptée à C++
  - permettant plus facilement le développement dirigé par les tests

## 2.1 Mise en place

- Rendre le fichier `doctest.h` accessible
- Écrire un fichier principal de test (`testmain.cpp`) contenant

```
#define DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN
#include "doctest.h"
```

- Écrire les suites de tests dans des fichiers à part contenant

```
#include "doctest.h"
```

## 2.2 Cas de tests

– Pour chaque classe `toto` : fichier `totoTest.cpp` contenant les tests

– On écrit des cas de tests :

```
TEST_CASE("Description du test"... ) {
//tests
}
```

– Les descriptions de test devraient être uniques à chaque cas de test

– Les cas de test peuvent éventuellement être regroupés en suites de test

```
TEST_SUITE("math") {
  TEST_CASE("sqrt") {} // part of the math test suite
  TEST_CASE("sin")  {} // part of the math test suite
}
```

– À l'exécution de l'exécutable, on peut sélectionner les tests à exécuter en fonction de mots clés sur le nom des cas de test et/ou des suites de test

- Les tests sont des lignes de codes qui contiennent des assertions
- Les résultats des évaluations des expressions des assertions seront enregistrés (et imprimés)
- Trois niveaux d'assertion ; si l'assertion échoue
  - `REQUIRE` : marque le cas de test comme ayant échoué et quitte le cas de tests
  - `CHECK` : marque le cas de test comme ayant échoué mais continue le cas de tests
  - `WARN` : affiche seulement un message sans marquer le cas de test comme ayant échoué
- Assertions générales
  - `<LEVEL>(expression);` : l'expression doit être vraie
  - `<LEVEL>_FALSE(expression);` : l'expression doit être fausse
  - où `<LEVEL>` = `REQUIRE`, `CHECK` ou `WARN`

- Comparaison des réels :
  - `doctest::Approx(`*`réel`*`)` ou `doctest::Approx(`*`réel`*`).epsilon(`*`valeur`*`)` : réel à une imprécision près pour faire une comparaison `==` à epsilon près
- Vérifier que des exceptions sont lancées ou pas lors du calcul de l'expression :
  - `<`*`LEVEL`*`>_THROWS(expression)` : l'expression doit lancer une exception
  - `<`*`LEVEL`*`>_THROWS_AS(expression,exception type)` : l'expression doit lancer une exception du type indiqué
  - `<`*`LEVEL`*`>_NOTHROW(expression)` : l'expression ne doit lancer aucune exception
- Si l'évaluation de l'expression fait lever une exception non attrapée, c'est considéré comme un échec

```
#include "doctest.h"
#include "point.h"

TEST_CASE("[point] Les points sont bien construits") {
//constructeur avec x et y
  double x = 31.21, y = -32.3;
  point p1 {x,y};
  REQUIRE( p1.x() == x );
  REQUIRE( p1.y() == y );
//constructeur par défaut
  point p2 {};
  REQUIRE( p2.x() == 0.0 );
  REQUIRE( p2.y() == 0.0 );
}
```

- Les assertions peuvent être utilisées dans n'importe quel code
- Les tests doivent eux aussi être proprement programmés

```
void lesCoordonnesDuPointSontExactement(const point& p,double x,double y) {
  REQUIRE( p.x() == x );
  REQUIRE( p.y() == y );
}

void lesCoordonnesDuPointSont(const point& p, double x, double y) {
  REQUIRE( p.x() == Approx(x) );
  REQUIRE( p.y() == Approx(y) );
}

TEST_CASE( "[point] Les points sont bien construits") {
//constructeur avec x et y
  double x = 31.21, y = -32.3;
  lesCoordonnesDuPointSontExactement(point{x,y}, x, y);
//constructeur par défaut
  lesCoordonnesDuPointSontExactement(point{}, 0.0, 0.0);
}
```

## 2.3 Assertions binaires et unaires

- doctest fournit aussi les assertions suivantes qui compilent plus rapidement
- `<LEVEL>_EQ(left, right)` : équivalent à `<LEVEL>(left == right)`
- `<LEVEL>_NE(left, right)` : équivalent à `<LEVEL>(left != right)`
- `<LEVEL>_GT(left, right)` : équivalent à `<LEVEL>(left > right)`
- `<LEVEL>_LT(left, right)` : équivalent à `<LEVEL>(left < right)`
- `<LEVEL>_GE(left, right)` : équivalent à `<LEVEL>(left >= right)`
- `<LEVEL>_LE(left, right)` : équivalent à `<LEVEL>(left <= right)`
- `<LEVEL>_UNARY(expr)` : équivalent à `<LEVEL>(expr)`
- `<LEVEL>_UNARY_FALSE(expr)` : équivalent à `<LEVEL>_FALSE(expr)`

## 2.4 Assertions rapides

- doctest fournit aussi des assertions qui compilent encore plus rapidement
- Mais si une expression lance une exception, le cas de test en entier est terminé
- Pour les utiliser :
  - préfixer les assertions binaires et unaires par `FAST_`
  - rajouter `#define DOCTEST_CONFIG_SUPER_FAST_ASSERTS` au début

```
#define DOCTEST_CONFIG_SUPER_FAST_ASSERTS

void lesCoordonnesDuPointSontExactement(const point& p,double x,double y) {
  FAST_REQUIRE_EQ( p.x() , x );
  FAST_REQUIRE_EQ( p.y() , y );
}

void lesCoordonnesDuPointSont(const point& p, double x, double y) {
  FAST_REQUIRE_EQ( p.x() , doctest::Approx(x) );
  FAST_REQUIRE_EQ( p.y() , doctest::Approx(y) );
}

TEST_CASE( "Les points sont bien construits", "[point]" ) {
//constructeur avec x et y
  double x = 31.21, y = -32.3;
  lesCoordonnesDuPointSontExactement(point{x,y}, x, y);
//constructeur par défaut
  lesCoordonnesDuPointSontExactement(point{}, 0.0, 0.0);
}
```

## 2.5 Sous-cas

- Il est préférable de pouvoir isoler les différents tests de cas de test
- Découpage en sous-cas :

```
SUBCASE( "Description des tests" ) {
  //tests de la section
}
```

- Les sous-cas peuvent être imbriquées : un `SUBCASE` peut contenir des `SUBCASE`
- Chaque sous-cas feuille (sans sous-cas) sera exécuté exactement une fois à partir du `TEST_CASE`, dans un chemin d'exécution séparé de ceux des autres sous-cas feuilles
- Cela permet de factoriser dans le code des initialisations communes à différents sous-cas en les mettant avant ces sous-cas
- Un échec dans un sous-cas empêche les sous-cas imbriqués de s'exécuter

```
TEST_CASE( "Changements de coordonnées" ) {
  double x = 31.21, y=-32.3;
  point p{x,y};

  SUBCASE( "La copie est correcte" ) {
    double x2 = x+1.0, y2 = y-1.0;
    p = point{x2,y2};
    lesCoordonnesDuPointSontExactement(p, x2, y2);
  }

  SUBCASE( "La translation est correcte" ) {
    double dx = 1.0, dy = -1.0;
    p.move(dx,dy);
    lesCoordonnesDuPointSont(p, x+dx, y+dy);
  }
}
```

# 3 Tests propres

## 3.1 Organisation concrète au sein du projet

- Mettre le code de production et le code de test au même endroit
- Utiliser des scripts pour n'exporter en production que le code de production
- Pour chaque fichier `classe.cpp`, fichier de tests `Testclasse.cpp`
- Noms des classes de
  - suites de tests : préfixe `test` (xxxUnit)
  - tests : préfixe `testeQue` (xxxUnit)
  - doublures : préfixe `faux`, `mock`
- Les test doivent être organisés de façon à pouvoir trouver facilement les tests concernant la tache en cours

## 3.2 Tests propres

- Le code de test est aussi important que le code de production : il doit être de même qualité
- Les test doivent être propres : lisibles, c.-à-d. clairs et simples et compréhensibles ;
- Un test « sale » ne sert à rien
- Les tests doivent pouvoir être maintenus, c.-à-d. ils doivent pouvoir changer et évoluer avec le système
- Le nom d'un test doit être une phrase décrivant le comportement attendu avec le vocabulaire du domaine du problème
- Utiliser le vocabulaire du domaine du système créé

## Pas de duplication

```
TEST_CASE( "[point] Les points sont bien construits" ) {
//constructeur avec x et y
  point p1 {31.21,-32.3};
  REQUIRE( p1.x() == 31.21 );
  REQUIRE( p1.y() == -32.3 );
//constructeur par défaut
  point p2 {};
  REQUIRE( p2.x() == 0.0 );
  REQUIRE( p2.y() == 0.0 );
}
```

- Duplication :
  - de valeurs
  - de code

```
void lesCoordonnesDuPointSontExactement(const point& p,double x,double y) {
  REQUIRE( p.x() == x );
  REQUIRE( p.y() == y );
}

TEST_CASE( "[point] Les points sont bien construits" ) {
//constructeur avec x et y
  double x = 31.21, y=-32.3;
  point p1 {x,y};
  lesCoordonnesDuPointSontExactement(p1, x, y);
//constructeur par défaut
  point p2 {};
  lesCoordonnesDuPointSontExactement(p2, 0.0, 0.0);
}
```

– mieux séparer les tests

```
TEST_CASE("[point] Les points sont bien construits") {
  SUBCASE("Le constructeur avec des coordonnées est correct" ) {
    double x = 31.21, y=-32.3;
    point p1 {x,y};
    lesCoordonnesDuPointSontExactement(p1, x, y);
  }
  SUBCASE("Le constructeur par défaut est correct" ) {
    point p2 {};
    lesCoordonnesDuPointSontExactement(p2, 0.0, 0.0);
  }
}
```

## 3.3 Assertions

- Les assertions doivent exprimer une intention ou une supposition
- Ce sont des déclarations quand au comportement du code
- Elles doivent être écrites avec le bon niveau d'abstraction (p.ex. pas de nombres magiques $\rightarrow$ constantes)
- Séparer l'action des assertions :
  - on agit d'abord
  - on écrit ensuite les assertions à partir des résultats de l'action
  - éviter d'appeler des méthodes dans les assertions

## 3.4 Tests fiables

- Un test doit être digne de confiance :
- Un test doit pouvoir échouer :
  - quand on écrit un test pour la première fois
  - écrire (ou modifier temporairement) la fonction testée pour qu'elle renvoie n'importe quoi
  - et vérifier que le test échoue bien
- Il doit tester vraiment ce qu'il annonce tester, ni plus, ni moins
- Il doit échouer quand il le faut (c.-à-d. ne pas laisser passer d'erreurs)
- Il doit passer quand il doit passer
- Il ne doit pas avoir de bug

```
TEST_CASE("[vector] Vecteurs : ajout de valeur") {
  SUBCASE("Mauvais Ajouter une valeur augmente la taille de un") {
    vector<int> v{};
    v.push_back(10);
    REQUIRE(!v.empty());      //pas ce qui est annoncé
    REQUIRE(v.back() == 10);
  } //test réussit quand même si bug et taille augmente de 2
  SUBCASE("Bon Ajouter une valeur augmente la taille de un") {
    vector<int> v{};
    auto taille = v.size();
    v.push_back(10);
    REQUIRE(v.size() == taille + 1);
  }
  SUBCASE("Mieux Ajouter une valeur augmente la taille de un") {
    vector<int> v{};
    auto ancienneTaille = v.size();
    v.push_back(10);
    auto nouvelleTaille = v.size();
    REQUIRE(nouvelleTaille == ancienneTaille + 1);
  }
}
```

## 3.5 Tests simples

- Minimiser le nombre d'assertions dans un test (si possible une assertion par test)
- Ne tester qu'un concept dans un test
- Un test ne doit vérifier qu'une seule chose et doit bien la vérifier
- Un test ne doit avoir qu'une seule raison d'échouer (sinon il est difficile de connaître la raison qui a fait échouer le test)
- Éviter la logique et les structures de contrôle d'exécution dans les tests (éviter les tests, les boucles) :
  - s'il y a de la logique, c'est que le test teste plusieurs choses à la fois

  $\rightarrow$ séparer le test en plusieurs tests
- S'il y a des tests, toutes les branches des tests doivent pouvoir échouer

```
SUBCASE("Mauvais Ajouter une valeur augmente
         la capacité si besoin") {
  vector<int> v = vectorAuHasard();
  if (!v.empty()) {
    if (v.size() == v.capacity()) {
      auto capacite = v.capacity();
      v.push_back(10);
      REQUIRE(v.capacity() > capacite);
    }
    else {
      auto capacite = v.capacity();
      v.push_back(10);
      REQUIRE(v.capacity() == capacite);
    }
  }
}

//Bon
int valeur = 10;
SUBCASE("Un vecteur vide a une capacité nulle") {
  vector<int> v {};
  REQUIRE(v.capacity() == 0);
}
SUBCASE("Ajouter une valeur à un vecteur vide
         rend sa capacité non nulle") {
  vector<int> v {};
  v.push_back(valeur);
  REQUIRE(v.capacity() > 0);
}
```

```
SUBCASE("Ajouter une valeur à capacité pleine
         augmente la capacité") {
  vector<int> v = vectorAuHasardAPleineCapacite();
  auto ancienneCapacite = v.capacity();
  v.push_back(valeur);
  auto nouvelleCapacite = v.capacity();
  REQUIRE(nouvelleCapacite > ancienneCapacite);
}

SUBCASE("Ajouter une valeur à capacité non pleine
         ne change pas la capacité") {
  vector<int> v = vectorAuHasardNonAPleineCapacite();
  auto ancienneCapacite = v.capacity();
  v.push_back(valeur);
  auto nouvelleCapacite = v.capacity();
  REQUIRE(nouvelleCapacite == ancienneCapacite);
}
```

## 3.6 Tests à problèmes

### Assertions primitives

- Garder un seul niveau d'abstraction dans le test
- Assertions qui utilisent des éléments de plus bas niveau que le comportement testé

$\rightarrow$ reformuler avec un vocabulaire de même niveau

### Hyper-assertion

- Assertions qui vérifient plein de détails en bloc à la fois

$\rightarrow$ découper en tests qui vérifient différentes parties

## 3.7 Quoi et comment tester

- Un test doit tester un comportement et non pas un point technique ou l'implémentation
- Le test doit vérifier la validité du résultat et non pas comment le résultat a été obtenu
- Une suite de test doit tester tout ce qui pourrait casser, c.-à-d. doit tester toutes les conditions possibles et valider tous les calculs :
  - tester les usages réalistes et les conditions d'utilisation courante de l'utilisateur final
  - tester les conditions limites
  - tester les cas particuliers qui peuvent se produire
- Ne pas ignorer les tests triviaux : ils sont faciles à écrire et ont une valeur documentaire élevée

- Données de test :
  - données réelles
  - données synthétiques générées artificiellement
- Si on trouve un bug :
  - écrire un test qui expose le bug pour qu'il ne se répète plus
  - tester exhaustivement la fonction où se trouvait le bug pour vérifier qu'il n'y en a pas d'autre
- Les test devraient couvrir une bonne proportion du code
  - utiliser des outils qui testent la couverture d'un code
  - pour tester si du code est couvert par les tests : commenter cette partie et exécuter les tests. Si les tests passent c'est que la partie n'est pas ou mal couverte

## 3.8 Préparation du banc d'essai

- Un test peut être exprimé en quatre étapes
  - mettre en place les objets utilisés par le test
  - déclencher l'action
  - faire à la fin des assertions sur le résultat
  - nettoyer les ressources utilisées
- *Banc d'essai* (fixture) d'une suite de tests :
  - code de mise en place des objets (setup) exécuté avant chaque test
  - code de nettoyage des ressources (teardown) exécuté après chaque test
- Banc d'essai avec xxxUnit : mise en place et nettoyage dans deux méthodes spéciales
- Banc d'essai avec Catch : se fait naturellement par l'exécution du code qui va du début jusqu'à la fin de chaque section

- On ne peut comprendre le test sans comprendre ce code $\rightarrow$ ce code doit être écrit aussi bien que le reste
  - pas de duplication : factoriser les initialisations communes dans des fonctions à part
  - elles doivent être lisibles : découper si nécessaire dans des fonctions à part
- Les tests d'une même suite de test doivent avoir une cohésion : ils doivent notamment utiliser le même banc d'essai
  - si l'initialisation n'est pas exactement la même pour les tests de la suite
  - écrire ces initialisations dans des méthodes à part et les appeler dans les tests

## 3.9 Tests de qualité

– Les test unitaires doivent être de première (F.I.R.S.T) qualité : Fast, Independent, Repeatable, Self-Validating, Timely

### Rapide

– Un test doit s'exécuter rapidement

– Attention aux opérations qui prennent du temps

– Attention aux ressources dont l'accès prend du temps

  – accès à des fichiers

  – accès à une base de données

  – accès au réseau

– Utiliser des doublures pour ces ressources

## Indépendant

- Un test ne doit pas dépendre d'un autre
- Chaque test doit pouvoir être exécuté seul et indépendamment des autres
- Les tests doivent pouvoir être exécutés dans n'importe quel ordre

```
TEST_CASE( "[point] Entrées sorties" ) {
  SUBCASE( "L'affichage est correct" ) {
    point p {31.21,-32.3};
    string formatAttendu = "(31.21,-32.3)", formatLu;
    ofstream ost{"pointtest.txt"};
    ost<<p;
    ost.close();
    ifstream ist{"pointtest.txt"};
    getline(ist,formatLu);
    REQUIRE(formatLu == formatAttendu);
  }
  SUBCASE( "La lecture est correcte" ) {
    point p {};
    ifstream ist{"pointtest.txt"}; //problème
    ist>>p;
    lesCoordonnesDuPointSont(p,31.21,-32.3);
  }
}
```

```
//tests indépendants
//utiliser des stringstream à la place de fichiers

TEST_CASE( "[point] Entrées sorties" ) {
  SUBCASE( "L'affichage est correct" ) {
    point p {31.21,-32.3};
    string formatAttendu = "(31.21,-32.3)";
    ostringstream ost{};
    ost<<p;
    string formatLu = ost.str();
    REQUIRE(formatLu == formatAttendu);
  }
  SUBCASE( "La lecture est correcte" ) {
    point p {};
    string formatLu = "(31.21,-32.3)";
    istringstream ist{formatLu};
    ist>>p;
    lesCoordonnesDuPointSont(p,31.21,-32.3);
  }
}
```

**Répétable**

- Un test doit toujours donner le même résultat à chaque exécution
- Un test doit pouvoir être exécuté tel quel
  - dans n'importe quel environnement (de développement, de test, de production . . . )
  - sur n'importe quelle machine, plateforme

  $\rightarrow$ encapsuler dans des objets les différences entre systèmes

**Se valider lui-même**

- Un test doit avoir un résultat binaire : il passe ou il échoue

## À Temps

- Les tests doivent être écrits avant le code de production
- Il faut tester
  - le plus tôt possible
  - souvent
  - automatiquement
- Coder un petit peut, tester un petit peu (ou l'inverse)
- Un code n'est pas terminé tant que les tests n'ont pas été exécutés avec succès

# 4 Isoler les tests de l'extérieur : doublures

## 4.1 Doublures

- Si le test dépend de conditions extérieures qui gênent le test

→ employer des objets *doublures* à la place des objets représentants les vraies conditions extérieures

- Doublures : objets utilisés à la place de ceux de production
  - avec une implémentation plus simple : permettent d'accélérer l'exécution du test
  - isolés de l'extérieur : permettent d'isoler le code du test de l'extérieur
  - sans effet de bord : rendent l'exécution déterministe
  - permettent de simuler des conditions particulières
  - permettent d'accéder à de l'information cachée à l'extérieur

- deux grands types de doublures :
  - pour tester les résultats : *manchon de test* (test stub) ou *contrefaçon* (fake object)
  - pour tester le comportement (c.à-d. l'interaction avec l'extérieur) : *objet factice* (mock objects)

**Manchon de test**

- Remplace l'implémentation réelle par l'implémentation la plus simple possible :
  - des méthodes qui ne font rien si le résultat de leur exécution n'est pas important pour le test
  - des méthodes qui renvoient des valeurs prédéfinies pour simuler des scénarios précis

## Contrefaçon

- Version allégée et optimisée qui réplique le comportement de l'objet réel mais :
  - sans ses effets de bord
  - sans ses conséquences sur le reste du système
- Permet à l'objet testé d'accéder plus facilement aux données extérieures qu'il utilise
- Et seulement aux données nécessaires au test
- exemples :
  - utiliser des flots chaîne `stringstream` à la place de fichiers `fstream`
  - objet représentant une base de données stockant uniquement en mémoire les quelques informations nécessaires au test sans utiliser de base de données

## Objet Factice

- Objet utilisé à la place de celui de production
  - qui garde en mémoire les méthodes qui ont été appelées dessus
- À la fin du test, on peut vérifier dessus si l'objet testé a bien appelé dessus les bonnes méthodes dans le bon ordre avec les bons paramètres

## Faciliter l'utilisation de doublures

- Pour pouvoir utiliser des doubles plus facilement : injection des dépendances
  - ne pas la mettre la classe précise des collaborateurs utilisés
  - mais une classe de base abstraite
  - permet d'utiliser des doubles dans les classes dérivées
  - et aussi l'objet original encapsulé dans une classe dérivée

# 5 Utilisation des tests

## 5.1 Bénéfices directs des tests

- Les tests unitaires testent mécaniquement le fonctionnement correct des méthodes fonctions . . . ,
- Ils aident à réduire les bugs en permettant de :
  - signaler les bugs le plus tôt possible
  - trouver plus facilement la source de bugs
  - éviter les régressions
- Ils aident à mieux développer et faire évoluer le code en permettant de :
  - ajouter des nouvelles fonctionnalité en s'appuyant sur des bases solides
  - remanier et nettoyer le code sans introduire de régression

## 5.2 Tests comme documentation

- Les tests unitaires servent aussi de documentation d'un système
- Ils montrent comment utiliser les objets et les méthodes et ce que l'on peut en attendre
  - à priori : développement dirigé par les tests
  - à postériori : utilisation d'un module inconnu
- Pour se familiariser avec ou s'approprier un module peu documenté : écrire des tests unitaires
- Si on utilise une méthode qui n'a pas de test : en écrire
  - le test devient un médium de communication
  - permet de voir ce qu'on peut attendre ou non de la méthode
  - permet ensuite d'améliorer le code

## 5.3 Tests pour apprendre

### Test d'apprentissage

- Pour apprendre à utiliser une bibliothèque tierce non connue :
  - au lieu d'expérimenter la bibliothèque dans le code de production
  - écrire des test unitaires pour la comprendre
- Écrire des tests où on appelle la bibliothèque tel qu'on voudrait l'appeler dans le code de production :
  - permet de faire des expérimentations précises qui permettent de vérifier notre compréhension de la bibliothèque
  - permet aussi de vérifier s'il y a des changements lors d'une nouvelle version

## Test de caractérisation

- Pour comprendre le comportement réel d'un morceau de code
- Test de caractérisation : test qui caractérise et documente le comportement réel d'un morceau de code
  - utiliser le morceau de code dans l'infrastructure de test
  - écrire une assertion dont on sait qu'elle va échouer
  - laisser l'échec indiquer quel est le comportement réel
  - changer le test pour qu'il s'attende au comportement produit par le code

# 6 Développement Dirigé par le Comportement

## 6.1 Développement Dirigé par les Tests

– Développement itératif où on écrit les tests en premier

1. Écrire un test d'acceptation client (qui décrit le comportement du système du point de vue du client) qui échoue :
   - écrire un test qui échoue parce que la fonctionnalité testée n'existe pas encore
   - écrire la fonctionnalité avec un code minimum qui ne marche pas pour faire échouer le test
2. Développement itératif pour faire passer ce test :
   - écrire le code qui fait passer le test
   - remanier le code pour améliorer sa conception

## 6.2 Pourquoi écrire les tests en premier

- En écrivant les test en premier on se place du côté utilisateur
- Cela permet de mieux réfléchir sur le code en
  - clarifiant ses intentions
  - donnant une description non-ambigüe de ce qu'il devrait faire
- Cela aide alors à écrire un meilleur code
- Le remaniement des tests permet aussi d'entrevoir de nouveaux remaniements nécessaires du code testé

- Les 3 lois du TDD

  1. Tu ne dois pas écrire du code de production avant d'avoir écrit un test unitaire qui échoue
  2. Tu ne dois pas écrire plus d'un test unitaire que nécessaire pour échouer (ne pas compiler est un échec)
  3. Tu ne dois pas écrire plus de code de production que nécessaire pour passer les tests

- Le système doit toujours tourner :

  - on ne doit pas faire de changement qui casse le système
  - après chaque changement, le système doit continuer à fonctionner comme avant

## 6.3 Développement Dirigé par le Comportement

- Un test peut être exprimé en trois étapes :
  - mettre en place les objets utilisée par le test
  - déclencher l'action
  - faire à la fin des assertions sur le résultat
- Ce raisonnement peut être exprimé sous la forme *étant donné*, *quand*, *alors* (given, when, then) :
  - *étant donné* un contexte,
  - *quand* quelque chose/un événement arrive/se produit
  - *alors* on s'attend à un certain résultats
- Catch permet aussi d'écrire des tests avec ce vocabulaire

- Le *Développement Dirigé par le Comportement* formule les tests sous cet aspect :
  - qui met l'accent plus sur le comportement attendu du programme
  - que sur les tests en tant que tels qui ne sont pas une fin en soi
- doctest permet aussi d'écrire des tests avec ce vocabulaire :
  - `SCENARIO` à la place de `TEST_CASE`
  - `GIVEN`, `WHEN` et `THEN` à la place de `SUBCASE`
  - `AND_WHEN` et `AND_THEN` : version de `WHEN` et `THEN` pour mieux montrer un enchaînement

```
SCENARIO( "vectors can be sized and resized" ) {
  GIVEN( "A vector with some items" ) {
    std::vector<int> v( 5 );
    REQUIRE( v.size() == 5 );
    REQUIRE( v.capacity() >= 5 );
    WHEN( "the size is increased" ) {
      v.resize( 10 );
      THEN( "the size and capacity change" ) {
        REQUIRE( v.size() == 10 );
        REQUIRE( v.capacity() >= 10 );
      }
    }
    WHEN( "the size is reduced" ) {
      v.resize( 0 );
      THEN( "the size changes but not capacity" ) {
        REQUIRE( v.size() == 0 );
        REQUIRE( v.capacity() >= 5 );
      }
    }
  }
}
```

## 6.4 Conception testable

- Même si on n'applique pas de TDD, on doit faire une conception testable
- Étant donné un morceau de code, on doit pourvoir écrire facilement et rapidement un test unitaire pour le tester
- Testabilité : logiciel qu'il est *facile* de tester
- Une conception simple passe tous les tests :
  - une conception doit produire un système qui agit comme attendu
  - des systèmes non testables ne sont pas vérifiables
  - écrire des tests conduit aussi à une meilleure conception

## 6.5 Déverminage

- Avant de travailler sur le bug, vérifier que le code compile proprement sans avertissements (les mettre au plus haut niveau) : laisser le compilateur trouver certains problèmes
- Rendre le bug reproductible
- Visualiser les données
- Utiliser un débogueur