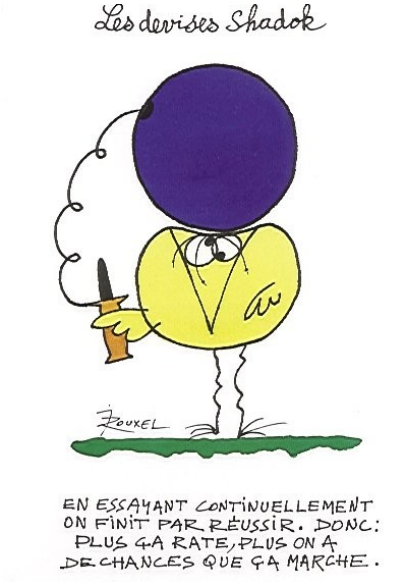
Table des matières



Les tests

Objectif

Le test est une recherche d'anomalie (défaut, appelé souvent *bug*) dans le comportement d'un logiciel.

Remarques:

- Si une batterie de tests ne montre pas de défaut cela n'implique pas que le logiciel est quand même exempt de défaut ...
- Le test n'a pas pour objectif : de diagnostiquer la cause des erreurs, de corriger les fautes ou de prouver la correction.
- On évalue à environ 40% la part des tests dans le coût d'un logiciel (et plus pour des logiciels critiques).

Principe d'une panne

A l'origine, il y a la **Faute** (*mistake*) : c'est la cause d'une erreur, puis :

Erreur (*Error*)		**Défaut, anomalie (*bug*)**		**Panne (*failure*)**
Un écart entre une valeur ou condition calculée (observée ou mesurée) et la valeur ou condition qui est vraie (spécifiée ou théoriquement correcte)	⇨	La manifestation d'une erreur dans un logiciel. Un défaut peut causer une panne	⇨	La fin de la capacité d'un système ou de l'un de ses composants d'effectuer la fonction requise, ou de l'effectuer à l'intérieur des limites spécifiés
IEEE 729				

Les tests dans le projet de BTS

Si les tests se déroulent parmi les activités finales du projet, ils se préparent dès le début. Il existe différentes formes de tests :

- Tests de fonctionnements (**unitaires** et **intégrations**) : pour vérifier que le **produit est bien fait**
- Tests de **validation** : permettent de vérifier que le **produit réalisé est le bon**
- Tests d'architecture : pour vérifier que le produit est utilisable dans son environnement d'exploitation.

Ces tests, quels qu'ils soient, sont **planifiés à l'avance**, dans l'étape correspondante à leur niveau de détails (voir cycle en V plus loin). Cela signifie qu'il faut, dans chaque étape du processus, spécifier quels tests seront effectués, à quel moment, par qui, dans quelles conditions, avec quels moyens et surtout de quelle manière.

En détaillant ces tests dans l'ordre dans lequel ils sont effectués:

- **Tests unitaires** : ils sont planifiés lors de la **conception détaillée**. Ils permettent de tester "les plus petites unités testables": méthodes, fonctions, etc. …
- **Tests d'intégrations** : planifiés en **analyse** et en **conception préliminaire**, ils évaluent les différentes unités intégrées, *packages*, groupement de classes, mais aussi les interactions matérielles – logiciel en cas de système embarqué. Ils utilisent les diagrammes d'interactions (séquence et/ou collaboration).
- **Tests de validations** : conçus dès les spécifications, ils permettent de vérifier l'adéquation du produit aux exigences fonctionnelles du client. Les points de départ sont les cas d'utilisation et les scénarios.
- Tests d'architecture: planifiés lors des spécifications techniques et en conception, ils doivent assurer que le système informatique est correctement dimensionné afin de garantir un fonctionnement confortable de l'application: temps de réponse convenables, bonne tenue à la montée en charge, sécurité optimale, …

La recette finale comprendra les mêmes type de tests que ceux effectués lors des tests de validation et d'architecture, mais réalisés par le client ou ses représentants. Elle permettra au client de déterminer si l'ensemble du système est bon pour le service.

Il est donc indispensable de tester unitairement les méthodes de chaque classe, puis l'intégration jusqu'à la validation qui permet de vérifier si les besoins captés à l'analyse sont bien couverts.

L'investissement dans les tests a un coût très important dans un projet logiciel.

En résumé

Tests unitaires
Chaque module du logiciel est testé séparément par rapport à ses spécifications.
En programmation C++, on fera des test unitaires au niveau des méthodes, puis au niveau de la classe.

Tests d'intégration
Les modules validés par les test unitaires sont rassemblés dans un composant logiciel. Le test d'intégration vérifie que l'intégration des modules n'a pas altéré leur comportement.

Tests de validation
Le test vérifie que le logiciel réalisé correspond bien aux besoins exprimés par le client.
La validation ou vérification d'un produit cherche donc à s'assurer qu'on a construit le bon produit.

Tests de recette (Tests d'intégration système)
L'application doit fonctionner dans son environnement de production, avec les autres applications présentes sur la plate-forme et avec le système d'exploitation.
Les utilisateurs vérifient sur site que le système répond de manière parfaitement correcte.

Exemple : **Test de validation**

Ce test s'élabore normalement au tout début du projet, une fois les besoins exprimés et les cas d'utilisations connus. Par exemple pour un logiciel de dessin :

Désignation	Démarche à suivre	Résultat attendu	oui/non	Remarques
Déplacement d'un objet	- Sélectionner dans la barre d'outil l'icône correspondante à un objet - Cliquer sur le champ de travail pour créer l'objet - Déplacer l'objet sur le champ de travail - Cliquer sur Annuler	L'objet revient à sa position initiale		
...				

Puis, en fin de développement, il est complété afin de valider le produit :

Désignation	Démarche à suivre	Résultat attendu	oui/non	Remarques
Déplacement d'un objet	Sélectionner dans la barre d'outil l'icône correspondante à un objet Cliquer sur le champ de travail pour créer l'objet Déplacer l'objet sur le champ de travail Cliquer sur Annuler	L'objet revient à sa position initiale	non	l'objet disparaît du champ de travail
...				

D'autres tests connus

Tests de boîte noire (test fonctionnel)
Le test porte sur le fonctionnement externe du système. La façon dont le système réalise les traitements n'entre pas dans le champ du test.

Tests de boîte blanche (test structurel)
Le test vérifie les détails de l'implémentation, c'est à dire le comportement interne du logiciel.

Tests de conformité
Le test vérifie la conformité du logiciel par rapport à ses spécifications et sa conception.

Tests de non conformité
Le test vérifie que les "cas non prévus" ne perturbent pas le fonctionnement du système.

Tests bêta
Réalisés par des développeurs ou des utilisateurs sélectionnés, ils vérifient que le logiciel se comporte pour l'utilisateur final comme prévu par le cahier des charges.

Tests alpha
Le logiciel n'est pas encore entièrement fonctionnel, les testeurs alpha vérifient la pré-version.

Tests fonctionnels
L'ensemble des fonctionnalités prévues est testé : fiabilité, performance, sécurité, affichages, etc

Tests de non régression (ou de régression)
Après chaque modification, correction ou adaptation du logiciel, il faut vérifier que le comportement des fonctionnalités n'a pas été perturbé, même lorsqu'elle ne sont pas concernées directement par la modification.

Test de Charge
Il s'agit d'un test au cours duquel on va simuler un nombre d'utilisateurs virtuels prédéfinis, afin de valider l'application pour une charge attendue d'utilisateurs. Ce type de test permet de mettre en évidence les points sensibles et critiques de l'architecture technique. Il permet en outre de mesurer le dimensionnement des serveurs, de la bande passante nécessaire sur le réseau, etc.

Test de Performance
Proche du Test de Charge, il s'agit d'un test au cours duquel on va mesurer les performances de l'application soumise à une charge d'utilisateurs. Les informations recueillies concernent les temps de réponse utilisateurs, les temps de réponse réseau et les temps de traitement d'une requête sur le(s) serveur(s). La nuance avec le type précédent réside dans le fait qu'on ne cherche pas ici à valider les performances pour la charge attendue en production, mais plutôt vérifier les performances intrinsèques à différents niveaux de charge d'utilisateurs.

Test de Dégradations des Transactions
Il s'agit d'un test technique primordial au cours duquel on ne va simuler que l'activité transactionnelle d'un seul scénario fonctionnel parmi tous les scénarios du périmètre des tests, de manière à déterminer

quelle charge limite simultanée le système est capable de supporter pour chaque scénario fonctionnel et d'isoler éventuellement les transactions qui dégradent le plus l'ensemble du système. Ce test peut tenir compte ou non de la cadence des itérations, la représentativité en termes d'utilisateurs simultanés vs. sessions simultanées n'étant pas ici un objectif obligatoire, s'agissant ici plutôt de déterminer les points de contention générés par chaque scénario fonctionnel. La démarche utilisée est d'effectuer une montée en charge linéaire jusqu'au premier point de blocage ou d'inflexion. Pour dépasser celui-ci, il faut paramétrer méthodiquement les composants systèmes ou applicatifs afin d'identifier les paramètres pertinents, ce jusqu'à obtention de résultats satisfaisants. Méthodologiquement, ce test est effectué avant les autres types de tests tels que performance, robustesse, etc. où tous les scénarios fonctionnels sont impliqués.

Test de stress
Il s'agit d'un test au cours duquel on va simuler l'activité maximale attendue tous scénarios fonctionnels confondus en heures de pointe de l'application, pour voir comment le système réagit au maximum de l'activité attendue des utilisateurs. La durée du palier en pleine charge, en général de 2 heures, doit tenir compte du remplissage des différents caches applicatifs et clients, ainsi que de la stabilisation de la plateforme de test suite à l'éventuel effet de pic-rebond consécutif à la montée en charge. Dans le cadre de ces tests, il est possible de pousser le stress jusqu'à simuler des défaillances systèmes ou applicatives afin d'effectuer des tests de récupération sur incident (Fail-over) ou pour vérifier le niveau de service en cas de défaillance.

Test de Robustesse, d'endurance, de fiabilité
Il s'agit de tests au cours duquel on va simuler une charge importante d'utilisateurs sur une durée relativement longue, pour voir si le système testé est capable de supporter une activité intense sur une longue période sans dégradations des performances et des ressources applicatives ou système. Le résultat est satisfaisant lorsque l'application a supporté une charge supérieure à la moitié de la capacité maximale du système, ou lorsque l'application a supporté l'activité d'une journée ou plusieurs jours/mois/années, pendant 8 à 10 heures, sans dégradation de performance (temps, erreurs), ni perte de ressources systèmes.

Test de capacité, Test de montée en charge
Il s'agit d'un test au cours duquel on va simuler un nombre d'utilisateurs sans cesse croissant de manière à déterminer quelle charge limite le système est capable de supporter. Éventuellement, des paramétrages peuvent être effectués, dans la même logique que lors des tests de dégradation, l'objectif du test étant néanmoins ici de déterminer la capacité maximale de l'ensemble système-applicatif dans une optique prévisionnelle (cf. Capacity Planning).

Test aux limites
Il s'agit d'un test au cours duquel on va simuler en général une activité bien supérieure à l'activité normale, pour voir comment le système réagit aux limites du modèle d'usage de l'application. Proche du test de capacité, il ne recouvre pas seulement l'augmentation d'un nombre d'utilisateurs simultanés qui se limite ici à un pourcentage en principe prédéfini, mais aussi les possibilités d'augmentation du nombre de processus métier réalisés dans une plage de temps ou en simultané, en jouant sur les cadences d'exécutions, les temps d'attente, mais aussi les configurations de la plateforme de test dans le cadre d'architectures redondées (Crash Tests).

Les méthodes de tests

Les tests statiques

Les méthodes de tests statiques consistent en l'analyse textuelle du code du logiciel afin d'y détecter des erreurs, sans exécution du programme.

Les méthodes utilisées sont : revue de code, analyse des types, analyse du domaine des variables, ...

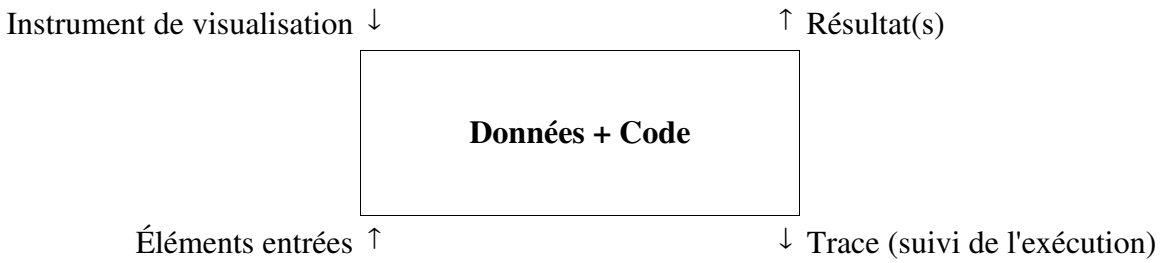
Les revues de code permettent l'examen détaillé d'une spécification, d'une conception ou d'une implémentation par une personne ou un groupe de personnes (lecture croisée), afin de déceler des fautes, des violations de normes de développement ou d'autres problèmes.

Avantages :
Méthodes efficaces et peu coûteuses (60 à 95% des erreurs sont détectées lors de contrôles statiques). Donc, les méthodes de tests statiques sont nécessaires.

Inconvénients :
Méthodes ne permettant pas de valider le comportement d'un programme au cours de son exécution. Donc, les méthodes de tests statiques ne sont pas suffisantes.

Les tests dynamiques

Les méthodes de tests dynamiques consistent en l'exécution du programme à valider à l'aide d'un jeu de tests. Elles visent à détecter des erreurs en confrontant les résultats obtenus à ceux attendus par la spécification.



Méthode aléatoire

Inconvénient : Ces méthodes ne garantissent pas une bonne couverture de l'ensemble des entrées du programme. En particulier, elles peuvent ne pas prendre en compte certains cas limites ou exceptionnels. Ces méthodes ont donc une efficacité très variable.

Méthode structurelle (Boîte blanche)

Le jeu de tests repose sur le code du programme. Le jeu de tests est choisi de manière à remplir certaines exigences:

- couverture de toutes les instructions.
- couverture de tous les chemins exécutables.
- couverture de toutes les conditions.

Méthode fonctionnelle (Boîte noire)

Le jeu de tests est dérivé de la spécification du programme. Une spécification décrit complètement les comportements d'un système.
La méthode de tests fonctionnelle vise à valider les fonctionnalités d'un programme.

Méthode expérimentale

Le jeu de tests est sélectionné sur la base de l'expérience.
Une base de données contenant toutes les erreurs découvertes dans un logiciel A peut servir de guide lors de la sélection du jeu de tests d'un logiciel B.

Les classes d'équivalence

On peut réduire le nombre de ces tests en utilisant la technique des classes d'équivalence. Cette technique consiste à identifier des classes d'équivalence dans le domaine des données d'entrées vis à vis d'une propriété d'une donnée de sortie. Tout test effectué avec une entrée quelconque appartenant à une classe d'équivalence déterminé entraîne un résultat soit correct (**classe valide**), soit incorrect (**classe invalide**).

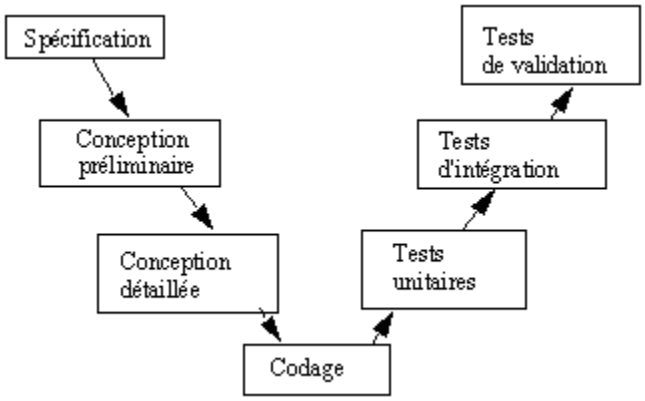
Une fois les classes déterminées, il suffit de prendre au moins un représentant pour chacune de ces classes (**jeu de test**).

Un plan de test se représente par un tableau contenant : une description du test, les valeurs en entrées du module et le résultat attendu. On numérotera chaque test (**phase**).

Exemple pour un test unitaire :

Testeur :		Date :	
Module testé :			Version : 1.0
Classe	Description	Valeurs en entrée	Résultats attendus
valide n°1	3 valeurs positives égales	(2, 2, 2)	3
valide n°2	3 valeurs positives dont deux égales telle que la somme des deux valeurs égales soit supérieure à la troisième	(3, 5, 5)	2
valide n°3	3 valeurs positives différentes telle que la somme de deux d'entre elles soit supérieure à la troisième	(3, 7, 9)	1
invalide n°4	Au moins une valeur négative	(-1, 6, 9)	-1
invalide n°5	3 valeurs positives telle que la somme de deux d'entre elles soit inférieure à la troisième	(3, 8, 4)	0

Le cycle en V



Remarques

- Les tests de validation sont planifiés en phase d'analyse (spécification)
- Les tests d'intégration sont planifiés en phase de conception préliminaire
- Les tests unitaires sont planifiés en phase de conception détaillée

Exercice

Soit une fonction **`rechercher`** permettant de retourner l'indice d'une valeur contenue dans un tableau d'entiers triés non vides (ils contiennent au moins un élément).

On définit 2 classes :

- classe n°1 : tableau de taille = 1
- classe n°2 : tableau de taille > 1

1 . Proposer un jeu de tests pour les deux classes définies.

Classe	Conditions du test			Résultats attendus
	Élément	Tableau	Valeur cherchée	Indice
1	Dans le tableau	{17}	17	0
	Pas dans le tableau	{17}	27	-1
2				

Séquence 1 – Tests unitaires avec CppUnit

Rappel

La meilleure façon d'exécuter des tests unitaires est d'utiliser une procédure automatique. En effet, les tests doivent pouvoir être exécutés sans intervention humaine et donner un résultat chiffré *(par exemple 80 tests réussis sur 100)*. Pour cela on va mettre en oeuvre le *framework* `CppUnit` (pour C++).
Un *framework* est une infrastructure logicielle qui facilite la conception des applications par l'utilisation de bibliothèques de classes ou de générateurs de programmes, soit dit en quelques mots : un cadre de développement.

Mise en situation

On désire tester unitairement un module que nous devons intégrer dans un projet en cours de développement. Ce module détermine et retourne le type (scalène, isocèle ou équilatéral) d'un triangle dont les valeurs des côtés sont passées en argument. Si les valeurs des côtés sont erronées, le module l'indiquera dans sa valeur de retour.

Le prototype de la méthode à tester est : `int isTriangle();`

Le code source à tester est fourni.

Mise en oeuvre de cppunit

Récupérer l'archive **`cppunit-x.x.x.tar.gz`** sur le serveur ou sur le site du projet (http://sourceforge.net/projects/cppunit) et effectuer la procédure suivante :

```
$ su root
# cd /usr/local
# tar zxvf cppunit-1.12.0.tar.gz
# cd /usr/local/cppunit-1.12.0/
# ./configure
# make
# make check
# make install
```

Les répertoires d'installation importants sont :

```
# ll /usr/local/lib/libcppunit*
# ll /usr/local/include/cppunit/
```

Vérification post-installation :

```
# cat /etc/ld.so.conf | grep "/usr/local/lib"
/usr/local/lib
```

Présence du répertoire **`/usr/local/lib`** dans **`/etc/ld.so.conf`** **sinon** l'ajouter et faire un **`ldconfig`** :

```
# echo "/usr/local/lib" >> /etc/ld.so.conf
# ldconfig
```

Maintenant, vous pouvez quitter la session **root** :

```
# exit
$
```

Travail demandé

On va mettre en place le jeu de tests ci-dessous à partir du *framework* cppunit.

Testeur :		Date :	
Module testé:			Version : 1.0
Classe	Description	Valeurs en entrée	Résultats attendus
valide n°1	**3 valeurs positives égales**	**(2, 2, 2)**	**3**
valide n°2	**3 valeurs positives dont deux égales telle que la somme des deux valeurs égales soit supérieure à la troisième**	**(3, 5, 5)**	**2**
valide n°3	**3 valeurs positives différentes telle que la somme de deux d'entre elles soit supérieure à la troisième**	**(3, 7, 9)**	**1**
invalide n°4	**Au moins une valeur négative**	**(-1, 6, 9)**	**-1**
invalide n°5	**3 valeurs positives telle que la somme de deux d'entre elles soit inférieure à la troisième**	**(3, 8, 4)** **(3, 4, 8)**	**0**

1 . Pour cela, on va écrire une classe de test nommée `TestTriangle`.
***Rappel :* il est très important de strictement séparer le code du test du code à tester.**

Fichier: TestTriangle.h

```
#ifndef CPP_UNIT_TESTTRIANGLE_H
#define CPP_UNIT_TESTTRIANGLE_H

#include <cppunit/extensions/HelperMacros.h>

class Triangle ; //la classe à tester
```

```
class TestTriangle : public CPPUNIT_NS::TestFixture
{
  CPPUNIT_TEST_SUITE( TestTriangle );
  CPPUNIT_TEST( testTriangleEquilateral );
  CPPUNIT_TEST( testTriangleIsocele );
  CPPUNIT_TEST( testTriangleScalene );
  CPPUNIT_TEST( testTriangleInvalide1 );
  CPPUNIT_TEST( testTriangleInvalide2 );
  CPPUNIT_TEST( testTriangleInvalide3 );
  CPPUNIT_TEST_SUITE_END();

public:
   TestTriangle();
   virtual ~TestTriangle();
  //void setUp();
  //void tearDown();

protected:
  void testTriangleEquilateral();
  void testTriangleIsocele();
  void testTriangleScalene();
  void testTriangleInvalide1();
  void testTriangleInvalide2();
  void testTriangleInvalide3();
};

#endif
```

Fichier: TestTriangle.cpp

```
#include <cppunit/config/SourcePrefix.h>
#include "TestTriangle.h"

#include "Triangle.h" // Classe à tester

CPPUNIT_TEST_SUITE_REGISTRATION( TestTriangle );

TestTriangle::TestTriangle()
{}

TestTriangle::~TestTriangle()
{}

//Classes valides :
//-C1 : 3 valeurs positives égales
//-C1 : (2, 2, 2)
void TestTriangle::testTriangleEquilateral()
{
   Triangle triangle(2, 2, 2);
   CPPUNIT_ASSERT_EQUAL( Triangle::EQUILATERAL, triangle.isTriangle() );
}
```

```
//-C2 : 3 valeurs positives dont deux égales telle que la somme des deux valeurs
égales soit supérieure à la troisième.
//-C2 : (3, 5, 5)
void TestTriangle::testTriangleIsocele()
{
   Triangle triangle(3, 5, 5);
   CPPUNIT_ASSERT_EQUAL( Triangle::ISOCELE, triangle.isTriangle() );
}

//-C3 : 3 valeurs positives différentes telle que la somme de deux d'entre elles
soit supérieure à la troisième.
//-C3 : (3, 7, 9)
void TestTriangle::testTriangleScalene()
{
   Triangle triangle(3, 7, 9);
   CPPUNIT_ASSERT_EQUAL( Triangle::SCALENE, triangle.isTriangle() );
}

//Classes invalides :
//-C4 : Au moins une valeur négative.
//-C4 : (-1, 6, 9)
void TestTriangle::testTriangleInvalide1()
{
   Triangle triangle(-1, 6, 9);
   CPPUNIT_ASSERT_EQUAL( Triangle::INVALIDE, triangle.isTriangle() );
}

//-C5 : 3 valeurs positives telle que la somme de deux d'entre elles soit
inférieure à la troisième.
//-C5 : (3, 8, 4)
// L'erreur n'a pas été detectée ! Elle aurait été avec ce jeu de test :
//-C5 : (3, 4, 8)
void TestTriangle::testTriangleInvalide2()
{
   Triangle triangle(3, 8, 4);
   CPPUNIT_ASSERT_EQUAL( Triangle::PAS_TRIANGLE, triangle.isTriangle() );
}

//-C5 : 3 valeurs positives telle que la somme de deux d'entre elles soit
inférieure à la troisième.
//-C5 : (3, 4, 8)
void TestTriangle::testTriangleInvalide3()
{
   Triangle triangle(3, 4, 8);
   CPPUNIT_ASSERT_EQUAL( Triangle::PAS_TRIANGLE, triangle.isTriangle() );
}
```

2 . Il ne reste plus qu'à écrire le programme de test.

Fichier : Main.cpp

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

int main( int argc, char* argv[] )
{
  // Create the event manager and test controller
  CPPUNIT_NS::TestResult controller;

  // Add a listener that colllects test result
  CPPUNIT_NS::TestResultCollector result;
  controller.addListener( &result );

  // Add a listener that print dots as test run.
  CPPUNIT_NS::BriefTestProgressListener progress;
  controller.addListener( &progress );

  // Add the top suite to the test runner
  CPPUNIT_NS::TestRunner runner;
  runner.addTest( CPPUNIT_NS::TestFactoryRegistry::getRegistry().makeTest() );
  runner.run( controller );

  // Print test in a compiler compatible format.
  CPPUNIT_NS::CompilerOutputter outputter( &result, CPPUNIT_NS::stdCOut() );
  outputter.write();

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

3 . On compile les différents fichiers sources et on exécute le programme de test :

```
$ g++ -g -O2 -c -I/usr/local/include/cppunit/ Triangle.cpp
$ g++ -g -O2 -c -I/usr/local/include/cppunit/ TestTriangle.cpp
$ g++ -g -O2 -c -I/usr/local/include/cppunit/ Main.cpp
$ g++ -g -O2 -o testTriangle -I/usr/local/include/cppunit/ -ldl -lcppunit
Triangle.o TestTriangle.o Main.o


$ ./testTriangle
TestTriangle::testTriangleEquilateral : OK
TestTriangle::testTriangleIsocele : OK
TestTriangle::testTriangleScalene : OK
TestTriangle::testTriangleInvalide1 : OK
TestTriangle::testTriangleInvalide2 : OK
TestTriangle::testTriangleInvalide3 : assertion
TestTriangle.cpp:65:Assertion
Test name: TestTriangle::testTriangleInvalide3
equality assertion failed
- Expected: 0
- Actual  : 1

Failures !!!
Run: 6   Failure total: 1   Failures: 1   Errors: 0
```

Remarque : le choix des classes de test est très important car on s'aperçoit ici que c'est le deuxième jeu de valeurs qui permet de détecter l'erreur.

4 . Compléter les tests existants (`TestTriangle.cpp`) pour différents jeux de valeur afin de couvrir plus précisément les différents cas. Exécuter alors le programme de test.

Testeur :		Date :	
Module testé:			Version : 1.0
Classe	Description	Valeurs en entrée	Résultats attendus
valide n°1	**3 valeurs positives égales**	**(2, 2, 2)**	**3**
valide n°2	**3 valeurs positives dont deux égales telle que la somme des deux valeurs égales soit supérieure à la troisième**	**(3, 5, 5)** **.....** **.....**	**2**
valide n°3	**3 valeurs positives différentes telle que la somme de deux d'entre elles soit supérieure à la troisième**	**(3, 7, 9)** **.....** **.....**	**1**
invalide n°4	**Au moins une valeur négative**	**(-1, 6, 9)** **.....** **.....**	**-1**
invalide n°5	**3 valeurs positives telle que la somme de deux d'entre elles soit inférieure à la troisième**	**(3, 8, 4)** **(3, 4, 8)** **.....**	**0**

5 . Corriger l'erreur dans le programme `Triangle.cpp` et ré-exécuter le programme de test.

Remarque : l'activité de correction ne fait partie des tests ! Elle est réalisée ici pour montrer l'utilité des tests automatiques.

Séquence 2 - Test de boîte noire

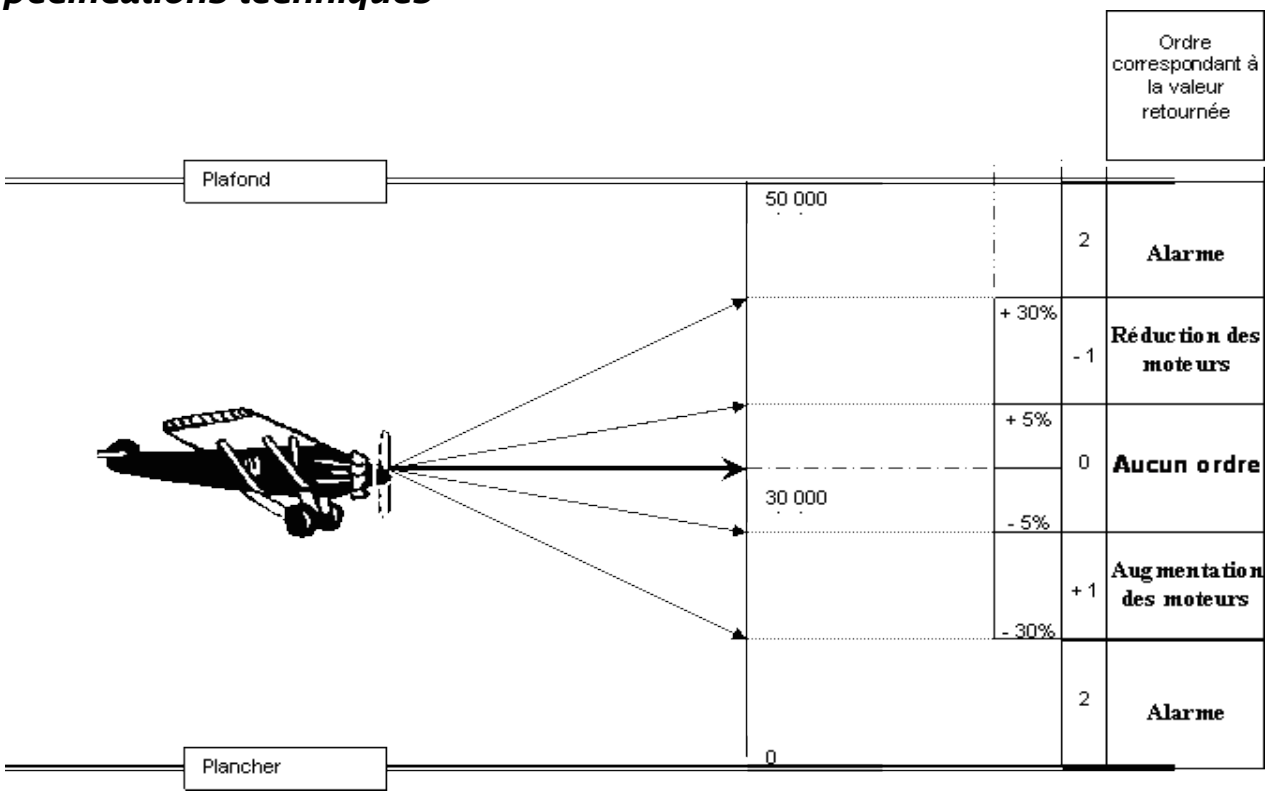
Rappel

Le test porte sur le fonctionnement externe du système. La façon dont le système réalise les traitements n'entre pas dans le champ du test. Le jeu de tests est dérivé de la spécification du programme. Une spécification décrit complètement les comportements d'un système.

Mise en situation

Employé(e) dans une entreprise aéronautique, il est confié à votre service, la lourde responsabilité de tester un nouveau système de pilotage automatique destiné à équiper les derniers appareils sortis des ateliers. Il vous incombe de tester la méthode **`NewAlti`** de la classe **`PiloteAuto`** qui, selon l'altitude captée, émet une alarme ou renvoie des ordres à un autre module chargé de la correction de la vitesse de vol.

Spécifications techniques



Description du module logiciel

Le module reçoit en entrée la nouvelle altitude captée. Il détermine et renvoie ensuite la valeur à retourner correspondante. Le prototype est fourni dans le fichier **`PiloteAuto.h`**
Valeur de l'altitude compensée préprogrammée : 30000
Module logiciel fourni : **`PiloteAuto.o`**
Remarque : **ici le code source du module n'est pas fourni (principe du test boîte noire)**

Travail demandé

La planification et les spécifications des tests unitaires à réaliser :

Testeur : *Shadok*		Date : 01/01/2005	
Module testé:			Version : 1.0
N° phase	Description	Valeurs en entrée	Résultats attendus
1	**Avion en danger (beaucoup trop bas)**	**Altitude < 21000pieds**	**2**
2	**Avion en danger (beaucoup trop haut)**	**Altitude ≥ 39000pieds**	**2**
3	**Avion bien positionné (dans la marge des +/- 5%)**	**Altitude ≥ 28500pieds** **Altitude < 31500pieds**	**0**
4	**Avion trop bas**	**Altitude ≥ 21000pieds** **Altitude < 28500pieds**	**1**
5	**Avion trop haut**	**Altitude ≥ 31500pieds** **Altitude < 39000pieds**	**-1**

1 . Développer la classe de test `TestPiloteAuto` et exécuter le programme de test unitaire.
Contraintes: Pour chaque test, utiliser trois jeux de valeurs en entrée (en terme d'altitude). Le programme de test utilise le *framework* **cppunit**.

2 . Établir le compte-rendu de test.

Testeur :		Date :		
Module testé :				Version : 1.0
N° phase	Description	Valeurs en entrée	Résultats attendus	Résultats obtenus
1	**Avion en danger (beaucoup trop bas)**	**Altitude < 21000pieds**	**2**	
2	**Avion en danger (beaucoup trop haut)**	**Altitude ≥ 39000pieds**	**2**	
3	**Avion bien positionné (dans la marge des +/- 5%)**	**Altitude ≥ 28500pieds** **Altitude < 31500pieds**	**0**	
4	**Avion trop bas**	**Altitude ≥ 21000pieds** **Altitude < 28500pieds**	**1**	
5	**Avion trop haut**	**Altitude ≥ 31500pieds** **Altitude < 39000pieds**	**-1**	

Séquence 3 – Pilotage par les tests

Rappel

Dans un développement piloté par les tests (comme dans la méthode *eXtremme Programming*), on écrit d'abord les tests et le code ensuite (les cas d'utilisation et les scénarios sont là pour aider à les écrire).
Remarque : Un programmeur écrit du code puis se demande : "quel tests pourraient valider ce code ?" (Les tests unitaires classiques sont toujours écrits après le code ou alors incomplets ou obsolètes). Un programmeur XP écrit un test puis se demande "quel code pourrait passer ce test ?" !

Mise en situation

Dans le cadre d'un travail de Technicien Supérieur en Informatique, on vous demande de réaliser la mise au point d'un module logiciel. Pour cela, vous devez :

- **suivre la planification et spécification de test fourni pour écrire la classe de test**
- **exécuter le programme de test**
- **coder la classe à tester en fonction des résultats des tests unitaires**

Description du module logiciel

Le module logiciel a validé est une classe **`Tableau`** qui gère une collection d'entiers. La méthode à tester unitairement est **`bool rechercher(int valRecherchee, unsigned int &position)`** dont la spécification est la suivante :

Définition : *recherche dichotomique d'un élément dans le tableau.*

Liste des paramètres :
Paramètre de sortie :
position :*position de la valeur recherchée.*

Paramètre d'entrée :
valRecherchee : *valeur à rechercher dans le tableau.*

Paramètre d'E/S : ***aucun***

Valeur retournée : true si la valeur a été trouvée dans le tableau, sinon ***false***.

Pré-condition : *tableau m_tab trié*

Post-condition :
résultat de la recherche=vrai et m_tab[position]=valRecherchee
OU
résultat de la recherche=faux

Invariant :
m_taille > 0

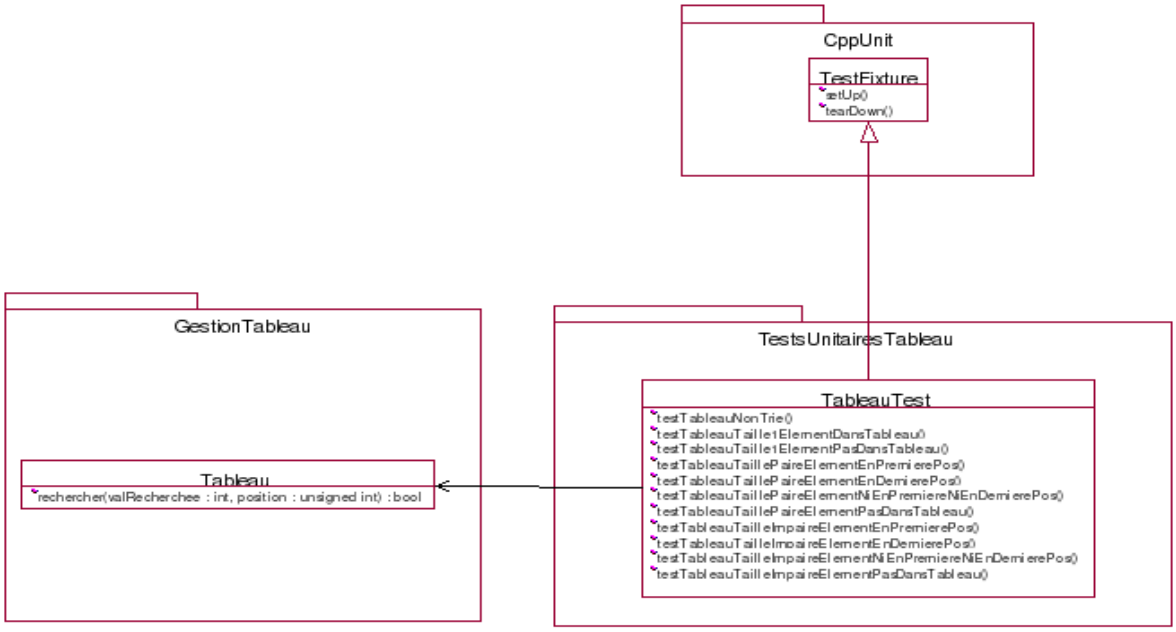
On a établi les 11 classes de test suivantes :

1. Tableau de taille quelconque, mais non trié.
2. Tableau trié de taille 1, élément dans le tableau,
3. Tableau trié de taille 1, élément pas dans le tableau,
4. Tableau trié de taille paire, élément en première position,
5. Tableau trié de taille paire, élément en dernière position,
6. Tableau trié de taille paire, élément ni en première, ni en dernière position,
7. Tableau trié de taille paire, élément pas dans le tableau,
8. Tableau trié de taille impaire, élément en première position,
9. Tableau trié de taille impaire, élément en dernière position,
10. Tableau trié de taille impaire, élément ni en première, ni en dernière position,
11. Tableau trié de taille impaire, élément pas dans le tableau.

Les résultats attendus sont :

Opérations	Résultats attendus
Test 1.1 *Tableau (T) de taille quelconque, mais non trié.* *T = 58, 4, 69, 158, 88, 27*	Valeur retournée par la méthode ***= false***
Test 1.2 *Tableau (T) trié de taille 1, élément dans le tableau.* *T = 4*	Valeur retournée par la méthode ***= true*** et ***oTableau[position] = 4***
Test 1.3 Tableau (T) trié de taille 1, élément pas dans le tableau.	Valeur retournée par la méthode ***= false***
Test 1.4 Tableau trié de taille paire, élément en première position	Valeur retournée par la méthode ***= true*** et ***oTableau[position] = 4***
Test 1.5 Tableau trié de taille paire, élément en dernière position	Valeur retournée par la méthode ***= true*** et ***oTableau[position] = 150***
Test 1.6 Tableau (T) trié de taille paire, élément ni en première, ni en dernière position. T = 4, 85, 99, 150	Valeur retournée par la méthode = ***true*** et ***oTableau[position] = 99***
Test 1.7 Tableau (T) trié de taille paire, élément pas dans le tableau. T = 4, 85, 99, 150	Valeur retournée par la méthode = ***false***
Test 1.8 Tableau (T) trié de taille impaire, élément en première position. T = 4, 85, 99	Valeur retournée par la méthode ***= true*** et ***oTableau[position] = 4***
Test 1.9 Tableau (T) trié de taille impaire, élément en première position. T = 4, 85, 99	Valeur retournée par la méthode ***= true*** et ***oTableau[position] = 99***
Test 1.10 Tableau (T) trié de taille impaire, élément ni en première, ni en dernière position. T = 4, 85, 99	Valeur retournée par la méthode ***= true*** et ***oTableau[position] = 85***
Test 1.11 Tableau (T) trié de taille paire, élément pas dans le tableau. T = 4, 85, 99	Valeur retournée par la méthode ***= false***

Le diagramme de classes sera le suivant :



Travail demandé

1 . A partir des documents fournis, finaliser la classe de test `TableauTest` en codant la méthode `TableauTest::testTableauTaille1ElementDansTableau()` soit le test 1.2.

2 . Compiler et exécuter le programme de test. Compléter les résultats obtenus :

```
Run: ...   Failure total: ...   Failures: ...   Errors: ...
```

3 . Coder la méthode `Tableau::estTrie()`

4 . Appliquer les itérations suivantes Tester-Coder/Mettre au point afin d'obtenir le résultat suivant :

```
TableauTest::testTableauNonTrie : OK
TableauTest::testTableauTaille1ElementDansTableau : OK
TableauTest::testTableauTaille1ElementPasDansTableau : OK
TableauTest::testTableauTaillePaireElementEnPremierePos : OK
TableauTest::testTableauTaillePaireElementEnDerniere Pos : OK
TableauTest::testTableauTaillePaireElementNiEnPremiereNiEnDerniere Pos : OK
TableauTest::testTableauTaillePaireElementPasDansTableau : OK
TableauTest::testTableauTailleImpaireElementEnPremierePos : OK
TableauTest::testTableauTailleImpaireElementEnDerniere Pos : OK
TableauTest::testTableauTailleImpaireElementNiEnPremiereNiEnDerniere Pos : OK
TableauTest::testTableauTailleImpaireElementPasDansTableau : OK
OK (11)
```

Séquence 4 – Test de non régression

Rappel

Après chaque modification, correction ou adaptation du logiciel, il faut vérifier que le comportement des fonctionnalités n'a pas été perturbé, même lorsqu'elle ne sont pas concernées directement par la modification.

Il faut donc « rejouer » les tests après toutes modifications du code.

Mise en situation

Reprendre le code de la séquence 3.

Travail demandé

1 . Planifier, Spécifier, et Ecrire les tests unitaires de la méthode `trier()` dans la classe de test `TableauTest`. Exécuter les tests.

2 . La classe `Tableau` utilise un tri à bulles comme technique de tri dans la méthode `Tableau::trier()`. On vous demande de re-coder cette méthode en implémentant une autre technique (tri par insertion, sélection, ...).

3 . Sans aucune modification de la classe de test, relancer l'exécution des tests unitaires. Réaliser la mise au point si nécessaire.

Remarque : Un test doit être **non intrusif**
Le code doit être intègre, c'est-à-dire non modifié pour réaliser le test. Par exemple pour un test fonctionnel on ne doit pas ajouter de **`printf`** ou de **`TRACE`** qui modifie entre autre les temps d'exécution et donc la validité du module testé.

Annexe 1

Exemple de procès-verbal d'un test unitaire (bilan du test)

Référence du module testé		
Date du test :	jj / mm / aaaa	
Type du test :	Fonctionnel ou structurel	
Objectif du test :	Cette rubrique rappelle le but du test.	
Conditions du test		
Etat initial	**Environnement du test**	
Description de l'état du module avant le lancement du test.	Description des moyens mis en œuvre pour la réalisation du test.	
Procédures du test		
Opérations	**Résultats attendus**	**Résultats obtenus**
Test n°1	Résultats attendus pour le test n°1.	Résultats obtenus pour le test n°1.
Test n°2	Résultats attendus pour le test n°2.	Résultats obtenus pour le test n°2.
Test n°N	Résultats attendus pour le test n°N.	Résultats obtenus pour le test n°N.
Nature des modifications apportées au fichier source du module testé		
Description des modifications apportées au module testé suite aux tests effectués.		