
# Table des matières



## Pré-requis

On dispose du *framework* CppUnit correctement installé dans `/usr/local`. La version utilisée ici est la `1.12.0`.

```
$ cppunit-config --version
1.12.0

$ cppunit-config --cflags
-I/usr/local/include

$ cppunit-config --libs
-L/usr/local/lib -lcppunit -ldl
```

## Objectif

Ce document décrit la **mise en œuvre d'une procédure de tests unitaires pour la classe `ParLed56` en utilisant le *framework* CppUnit**.

## L'appareil PAR Led 56

L'appareil**PAR Led 56** est un projecteur DMX à base de LED qui sont divisées en trois canaux : Le rouge, vert et bleu. Le nombre de canaux change en fonction du projecteur utilisé. Des canaux supplémentaires comme le DIMMER (intensité lumineuse) et le STROBE (clignotement) peuvent être disponibles. Le modèle décrit ici utilise 4 canaux DMX suivants :

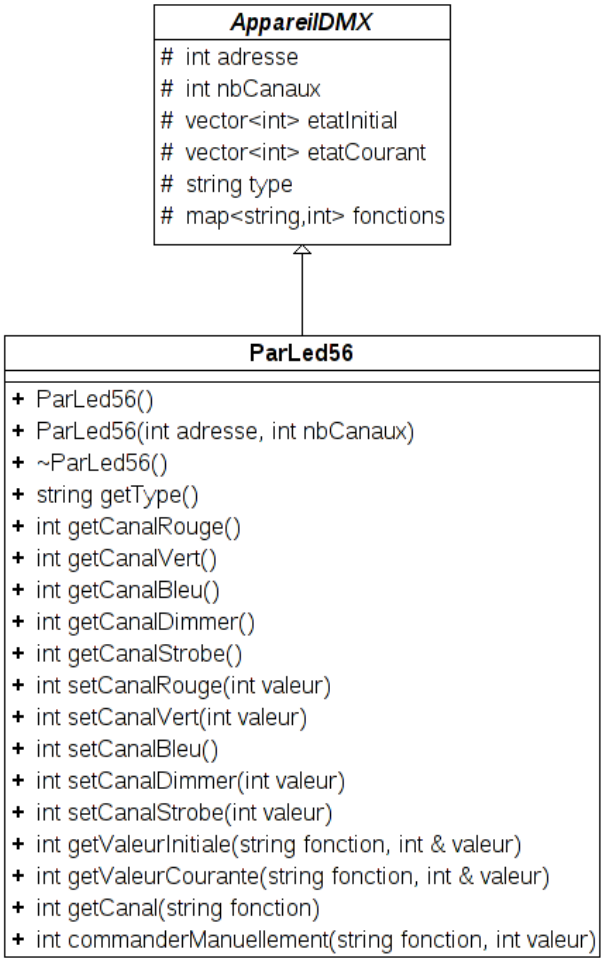
| Canaux | DMX | |
|---|---|---|
| canal 1 | 000 - 255 | Rouge |
| canal 2 | 000 - 255 | Vert |
| canal 3 | 000 - 255 | Bleu |
| canal 4 | 000 - 189 | DIMMER |
| | 190 - 250 | Clignotement |
| | 251 - 255 | Aucun changement |

*Remarque :* on considère ici l'adresse de l'appareil réglé sur le canal 1.

# La classe ParLed56

La classe `ParLed56` hérite de la classe abstraite `AppareilDMX`. Cette modélise un appareil « universel » raccordé sur le bus DMX512 et capable d'envoyer les valeurs de ses canaux vers une interface DMX.




*Ce qu'il faut retenir : En programmation orientée objet (**POO**), une **classe abstraite** est une classe dont l'implémentation n'est pas complète et qui n'est pas instanciable (on ne peut pas créer d'objet à partir d'une classe abstraite). Son nom apparaît en italique en UML.*

*Elle sert de base à d'autres classes dérivées (héritées). Le mécanisme des classes abstraites permet de définir des comportements (méthodes) qui devront être implémentés dans les classes filles, mais sans implémenter ces comportements (c'est-à-dire sans écrire de code pour cette méthode). Ainsi, on a l'assurance que les classes filles respecteront le contrat défini par la classe mère abstraite. Ce contrat est une interface de programmation.*

*Ce qu'il faut retenir : En **C++**, une classe est **abstraite** si elle contient au moins une méthode déclarée **virtuelle pure**, c'est-à-dire commençant par virtual et terminée par = 0. Ce type de classe n'est pas instanciable. Une fonction virtuelle pure doit être définie (ou redéclarée explicitement virtuelle pure) dans les classes dérivées.*

Les propriétés de la classe `AppareilDMX` dont héritent la classe `ParLed56` sont les suivants :

- `adresse` : un entier représentant le numéro du premier canal occupé par l'appareil sur le bus DMX (voir la document de l'appareil pour le réglage)
- nbCanaux : un entier représentant le nombre de canaux utilisé par cet appareil sur le bus DMX (exemple : 4 pour un PAR Led 56)
- `etatInitial` : un vecteur d'entiers exprimant les valeurs initiales à appliquer à chaque canal de l'appareil
- `etatCourant` : un vecteur d'entiers exprimant les valeurs courantes à appliquer à chaque canal de l'appareil
- `type` : une chaîne de caractères exprimant le type d'appareil (exemple : "PAR LED 56")
- `fonctions` : un vecteur de chaînes de caractères représentant l'ensemble des fonctions disponibles pour cet appareil (exemple : "Rouge" pour l'accès au canal rouge du projecteur, "Vert" ...)

La classe `ParLed56` fournit les services suivants :

- un ensemble d'accesseurs `get/set` permettant la manipulation des canaux Rouge, Vert, Bleu et Dimmer/Strobe.
- `getValeurInitiale()` qui permet d'obtenir la valeur initiale (`int`) d'un des canaux à partir de son nom de fonction (`string`)
- `getValeurCourante()` qui permet d'obtenir la valeur courante (`int`) d'un des canaux à partir de son nom de fonction (`string`)
- getCanal() qui permet de récupérer le numéro de canal (int) associé à une fonction (string). Exemple :
- **`commanderManuellement()` qui permet de piloter manuellement ce projecteur en passant en argument la fonction demandée (string) et la valeur à affecter (int).**

***Remarque :*** **tous les services d'une classe utilisent la convention suivante pour les retours d'appel**

- **0 : la méthode s'est exécutée avec succès**
- **<0 : la méthode a rencontré une erreur et elle retourne un code d'erreur**

# Mise en œuvre d'un test unitaire

## Principe

Le principe d'un test unitaire est simple : on va tester chaque fonction ou méthode individuellement (unitairement) et séparément par rapport à ses spécifications (c'est-à-dire ce qu'elle doit faire ou plus précisément ce que l'on attend d'elle).

Pour réaliser un test unitaire d'une fonction ou méthode, on procède de la manière suivante :

- on choisit un jeu de tests : c'est-à-dire les données d'entrée de la fonction (ses arguments)
- on définit le résultat attendu : c'est-à-dire le résultat qu'elle doit fournir
- puis on vérifie le résultat obtenu avec le résultat attendu

On validera une fonction lorsque tous les jeux de tests choisis ne détectent aucune anomalie (un résultat obtenu différent du résultat attendu). Toute la difficulté est de bien choisir ses jeux de tests pour couvrir le maximum de cas afin de détecter une possible anomalie.

## Le canal Rouge

On choisi de montrer la mise en œuvre du test unitaire de la gestion du canal Rouge de la classe ParLed56, c'est-à-dire la validation des méthodes `getCanalRouge()` et `setCanalRouge()`.

## Préparation

Il faut tout d'abord penser à séparer le code source de l'application de la suite de tests.

Exemple :

- `src` : répertoire contenant l'ensemble du code source de l'application
- `tests` : répertoire contenant l'ensemble du code source des tests unitaires de l'application

*Remarque :* Un test doit être non intrusif
Le code source 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.

## Spécification

La méthode `setCanalRouge()` reçoit en argument la valeur, sous la forme d'un entier, à affecter au canal Rouge et doit réaliser les traitements suivants :

- vérifier que la valeur du canal rouge est comprise entre 0 et 255 (c'est sa plage valide)
- retourner 0 si la plage est valide sinon retourner -1
- conserver la nouvelle valeur comme valeur courante pour le canal Rouge
- assurer la commande du bus DMX afin de mettre à jour l'état courant de cet appareil
- retourner l'état de la commande

## Les classes d'équivalence

Pour réduire le nombre de ces tests, on utilise 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 déterminée 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 (ce sera un jeu de test).

Un plan de test se représente par un tableau contenant : une description du test, les valeurs en entrées de la fonction et le résultat attendu.

| Module testé : | | | | Date : |
|---|---|---|---|---|
| Testeur : | | | | Version : |
| Classe | Description | Valeurs en entrée | Résultats attendus | Résultats observés |
| Valide n°1 | première valeur de la plage | 0 | valeur de retour : 0<br>valeur du canal : 0 | |
| Valide n°2 | Une valeur quelconque dans la plage (au milieu) | 128 | valeur de retour : 0<br>valeur du canal : 128 | |
| Valide n°3 | Dernière valeur de la plage | 255 | valeur de retour : 0<br>valeur du canal : 255 | |
| Invalide n°1 | Valeur inférieure au minimum de la plage | -1 | valeur de retour : -1<br>valeur du canal : inchangée | |
| Invalide n°2 | Valeur supérieure au maximum de la palge | 256 | valeur de retour : -1<br>valeur du canal : inchangée | |

Un rapport de test correspondra au tableau ci-dessus complété avec la possibilité d'ajouter des remarques et hypothèses en cas d'échec.

## Code source

Le code source des méthodes `getCanalRouge()` et `setCanalRouge()` :

```
int ParLed56::getCanalRouge() {
   // retourne la valeur courante affectée au canal Rouge
   return etatCourant[0];
}

int ParLed56::setCanalRouge(int valeur) {
   // Conserve la nouvelle valeur comme valeur courante pour le canal Rouge
   etatCourant[0] = valeur;
   // Envoie les valeurs des 512 canaux aux appareils DMX reliés sur le bus
   int etat = commander();
   return etat;
}
```

## La classe TestUnitaireParLed56

- La déclaration de la classe `TestUnitaireParLed56` :

```
#ifndef _TESTUNITAIREPARLED56_H
#define _TESTUNITAIREPARLED56_H

#include <cppunit/extensions/HelperMacros.h>

class ParLed56; // La classe à tester

class TestUnitaireParLed56 : public CPPUNIT_NS::TestFixture
{
  CPPUNIT_TEST_SUITE( TestUnitaireParLed56 );
  CPPUNIT_TEST( testCanalRouge ) ;
  CPPUNIT_TEST_SUITE_END();

private:
   ParLed56 *parLed56; // un pointeur sur une instance de la classe à tester
   int valeur;

public:
   TestUnitaireParLed56();
   virtual ~TestUnitaireParLed56();

   // Call before tests
   void setUp();
   // Call after tests
   void tearDown();

   // Liste des tests
   void testCanalRouge(); // le test unitaire du canal Rouge
};

#endif
```

- La définition de la classe TestUnitaireParLed56 :

```
#include <cppunit/config/SourcePrefix.h>
#include "TestUnitaireParLed56.h"
#include "ParLed56.h" // Classe à tester

// Enregistrement des différents cas de tests
CPPUNIT_TEST_SUITE_REGISTRATION( TestUnitaireParLed56 );

////////////////////////////////////////////////////////////////////
// Construction/Destruction
////////////////////////////////////////////////////////////////////
TestUnitaireParLed56::TestUnitaireParLed56() {}

TestUnitaireParLed56::~TestUnitaireParLed56() {}

void TestUnitaireParLed56::setUp()
{
   // Initialisation pour les tests
   parLed56 = new ParLed56(1); // instancie un objet ParLed56 sur le canal 1
   valeur = 0;
}

void TestUnitaireParLed56::tearDown()
{
   delete parLed56; // libère la mémoire
}

// Test unitaire du canal rouge : méthodes setCanalRouge() et getCanalRouge()
void TestUnitaireParLed56::testCanalRouge()
{
   int retour = 0;
   int valeurAttendue;

   // Appel méthode testée (lancement)
   valeurAttendue = valeur = 0; //classe valide (première valeur)
   // Vérification des résultats attendus
   retour = parLed56->setCanalRouge(valeur);
   CPPUNIT_ASSERT_MESSAGE( "classe valide (premiere valeur)", retour == 0 );
   // ou : CPPUNIT_ASSERT_EQUAL_MESSAGE(message, expected, actual);
   CPPUNIT_ASSERT_EQUAL( valeurAttendue, parLed56->getCanalRouge() ); // ou :
   //CPPUNIT_ASSERT( parLed56->getCanalRouge() == valeurAttendue );

   valeurAttendue = valeur = 128; //classe valide (valeur au milieu)
   retour = parLed56->setCanalRouge(valeur);
   CPPUNIT_ASSERT_MESSAGE( "classe valide (valeur au milieu)", retour == 0 );
   //CPPUNIT_ASSERT_EQUAL( valeurAttendue, parLed56->getCanalRouge() ); // <- ou
   CPPUNIT_ASSERT( parLed56->getCanalRouge() == valeurAttendue );

   valeurAttendue = valeur = 255; //classe valide (dernière valeur)
   retour = parLed56->setCanalRouge(valeur);
   CPPUNIT_ASSERT_MESSAGE( "classe valide (derniere valeur)", retour == 0 );
   CPPUNIT_ASSERT_EQUAL( valeurAttendue, parLed56->getCanalRouge() ); // ou :
   //CPPUNIT_ASSERT( parLed56->getCanalRouge() == valeurAttendue );

   valeur = -1; //classe invalide (inférieur à min)
   retour = parLed56->setCanalRouge(valeur);
```

```
   // Vérification des résultats attendus
   CPPUNIT_ASSERT_MESSAGE( "classe invalide (inferieur a min)", retour == -1 );
   CPPUNIT_ASSERT_EQUAL( valeurAttendue, parLed56->getCanalRouge() ); // ou :
   //CPPUNIT_ASSERT( parLed56->getCanalRouge() == valeurAttendue );

   valeur = 256; //classe invalide (supérieur à max)
   retour = parLed56->setCanalRouge(valeur);
   CPPUNIT_ASSERT_MESSAGE( "classe invalide (superieur a max)", retour == -1 );
   CPPUNIT_ASSERT_EQUAL( valeurAttendue, parLed56->getCanalRouge() ); // ou :
   //CPPUNIT_ASSERT( parLed56->getCanalRouge() == valeurAttendue );
}
```

## Exécution du test unitaire

```
$ make

$ ./testdmx
TestUnitaireParLed56::testCanalRouge : assertion
TestUnitaireParLed56.cpp:68:Assertion
Test name: TestUnitaireParLed56::testCanalRouge
assertion failed
- Expression: retour == -1
- classe invalide (inferieur a min)

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

Le test unitaire du canal Rouge pour la classe invalide n°1 à échouer ! Un rapport de test peut être rédigé par le testeur.

**Hypothèse : la méthode ne réalise pas de vérification de la plage valide pour la valeur du canal et accepte donc même des valeurs invalides.**

## Action corrective

À partir du rapport de test, le développeur en charge de cette fonction propose une action corrective sur le code source de la méthode `setCanalRouge()` :

```
int ParLed56::setCanalRouge(int valeur) {
   if(valeur >= 0 && valeur < 256)
   {
      // Conserve la nouvelle valeur comme valeur courante pour le canal Rouge
      etatCourant[0] = valeur;
      // Envoie les valeurs des 512 canaux aux appareils DMX reliés sur le bus
      int etat = commander();
      return etat;
   }
   return -1;
}
```

## Vérification et validation

***Attention :*** **il ne faut pas oublier de refabriquer (`make`) les modules logiciels (fichiers objets `.o`) de votre application avant de tester.**

On refabrique maintenant le programme de test et on exécute :

```
$ make
$ ./testdmx
TestUnitaireParLed56::testCanalRouge : OK
OK (1)
```

Le test unitaire est passé ! Aucun défaut (*bug*) n'a été détecté donc les méthodes sont validées.

*Remarque :* Test de non régression
Après chaque modification, correction ou adaptation du logiciel, il faudra vérifier que le comportement des fonctionnalités n'a pas été perturbé, même lorsqu'elle ne sont pas concernées directement par la modification. Il faudra donc « rejouer » les tests après toutes modifications du code.