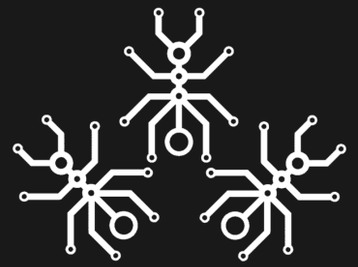


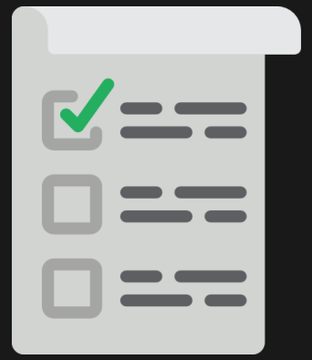
TESTS UNITAIRES POUR LES PROJETS MICROCONTRÔLEUR

par Maxime HEREDIA-HIDALGO

PRÉREQUIS

- Bonnes connaissances en langage C
- Comprendre les principes de l'architechture modulaire

OBJECTIFS DE LA FORMATION

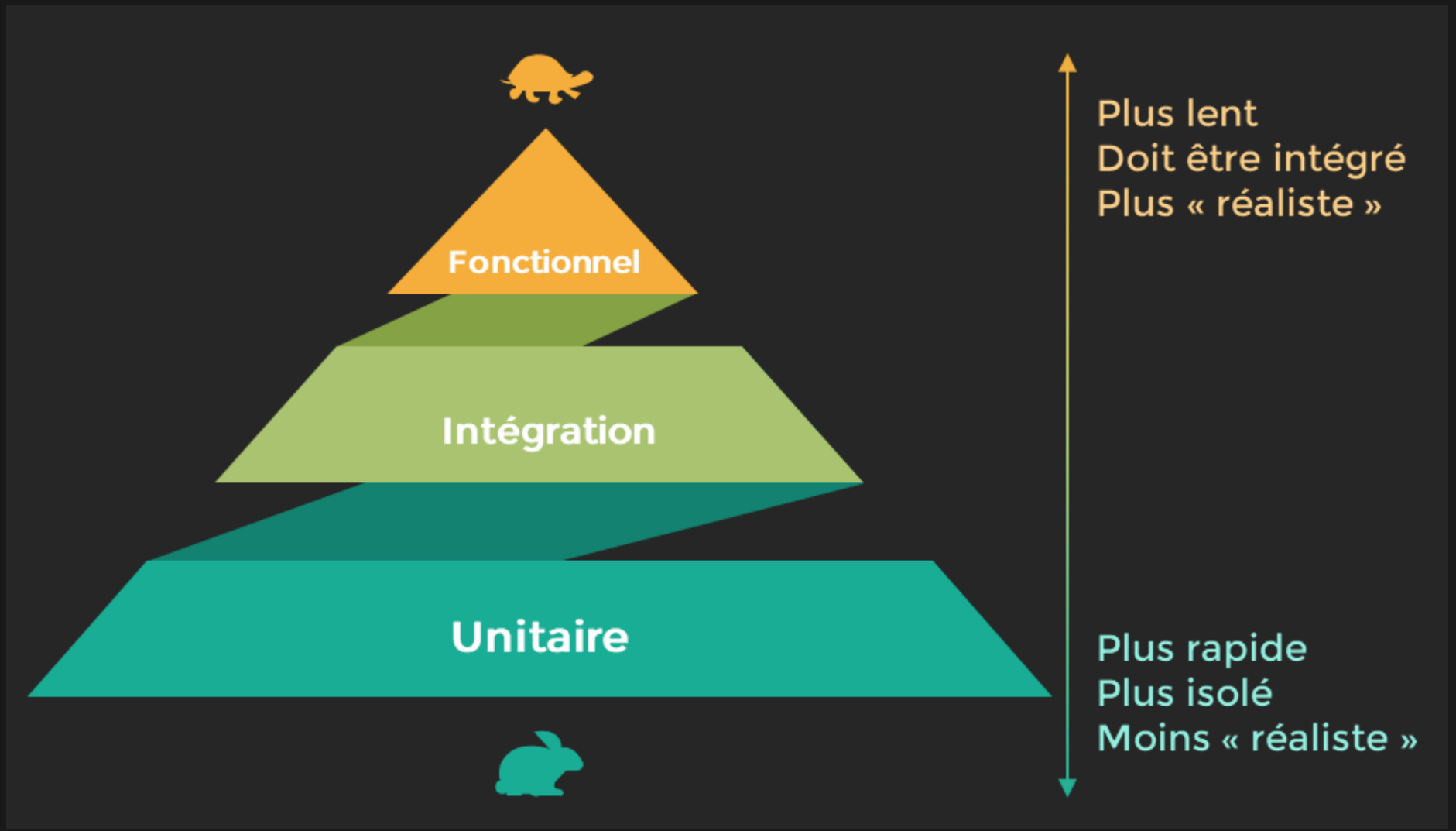


- Découvrir ce qu'est un test unitaire
- Comprendre les intérets de mettre en place des tests unitaires
- Découvrir un outils permettant la mise en place de tests unitaires pour code microcontrôleur

PLAN

1. Les types de tests
2. Pourquoi écrire des tests unitaires n'est pas une tâche secondaire
3. Comment écrire un test unitaire : assert , stub , mock
4. Un framework efficace pour le C embarqué : CMocka
5. Aller plus loin : Intégration continue , TDD

LES TYPES DE TESTS



LES TESTS UNITAIRES

- Valider des morceaux de code isolées
- Plus rapide, peux etre débarqué
- Couvrir l'integralité du code
- Détécter les bugs au plus tôt

LES TESTS D'INTEGRATIONS

- Valider un assemblage de modules
- Plus lent
- Détécter les erreurs fonctionel ou d'architecture

LES TESTS FONCTIONELS

- Encore plus lent, doivent etre executés sur la cible
- Garantir que le produit correspond aux spécifications/exigences
- Scénarii proche de cas réel vue par l'utilisateur
- peut etre automatisé ou manuel

POURQUOI ÉCRIRE DES TESTS UNITAIRES N'EST PAS UNE TÂCHE SECONDAIRE

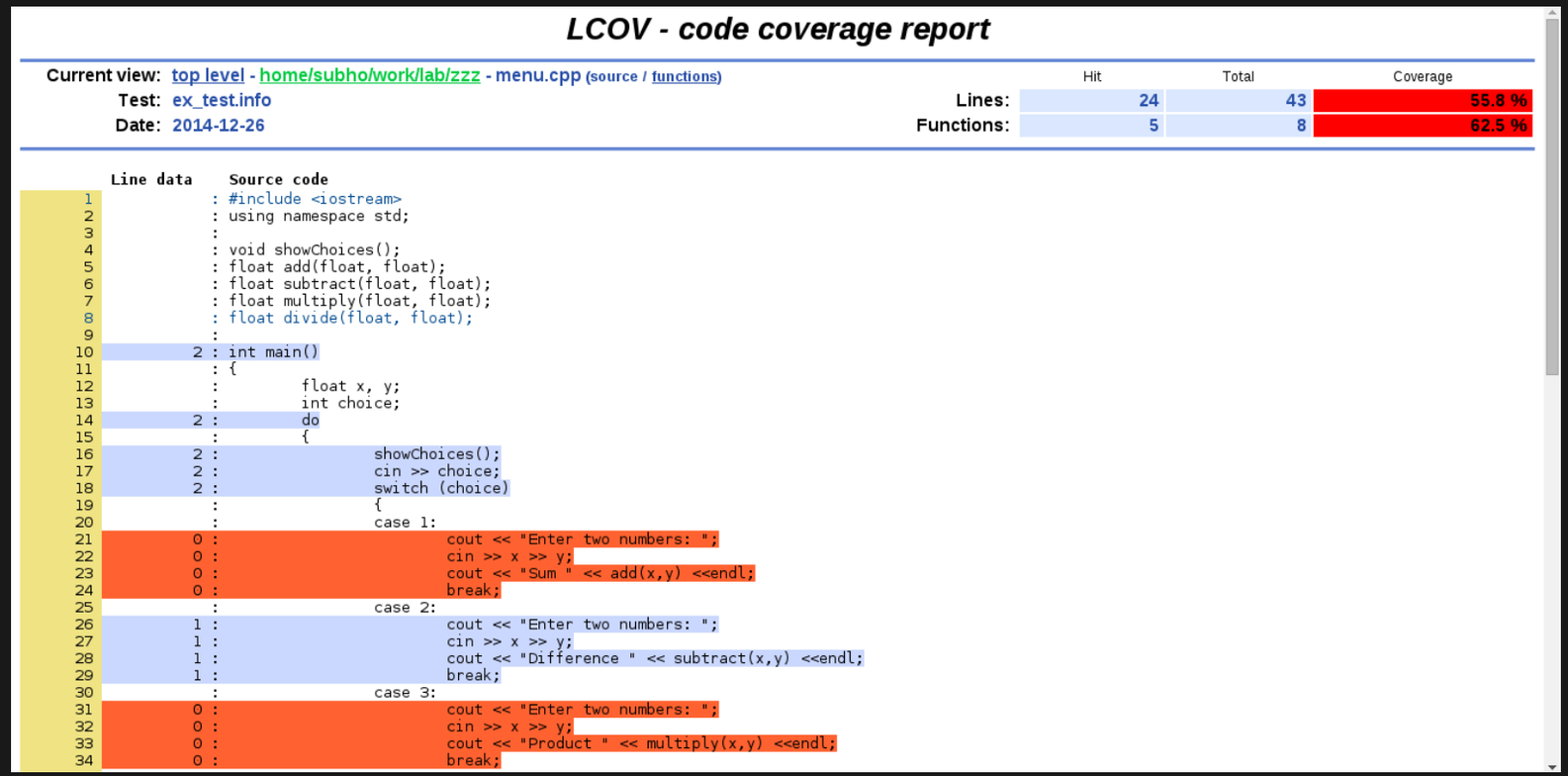
- Permet de valider un module
- Trouver rapidement les erreurs
- Permet de coder sans la cible (en débarqué)
- Permet d'eviter les régressions

LA COUVERTURE DE CODE (CODE COVERAGE)

- Généré par le framwork de test unitaire
- C'est n'est qu'un indicateur
- Permet de savoir si le code est couvert par des tests
- ne dit PAS si les tests sont pertinents

Une code couvert à 100% ne garantis pas qu'il est correctement testé

Un test verifie un comportement, il ne garantie pas l'absence de bug



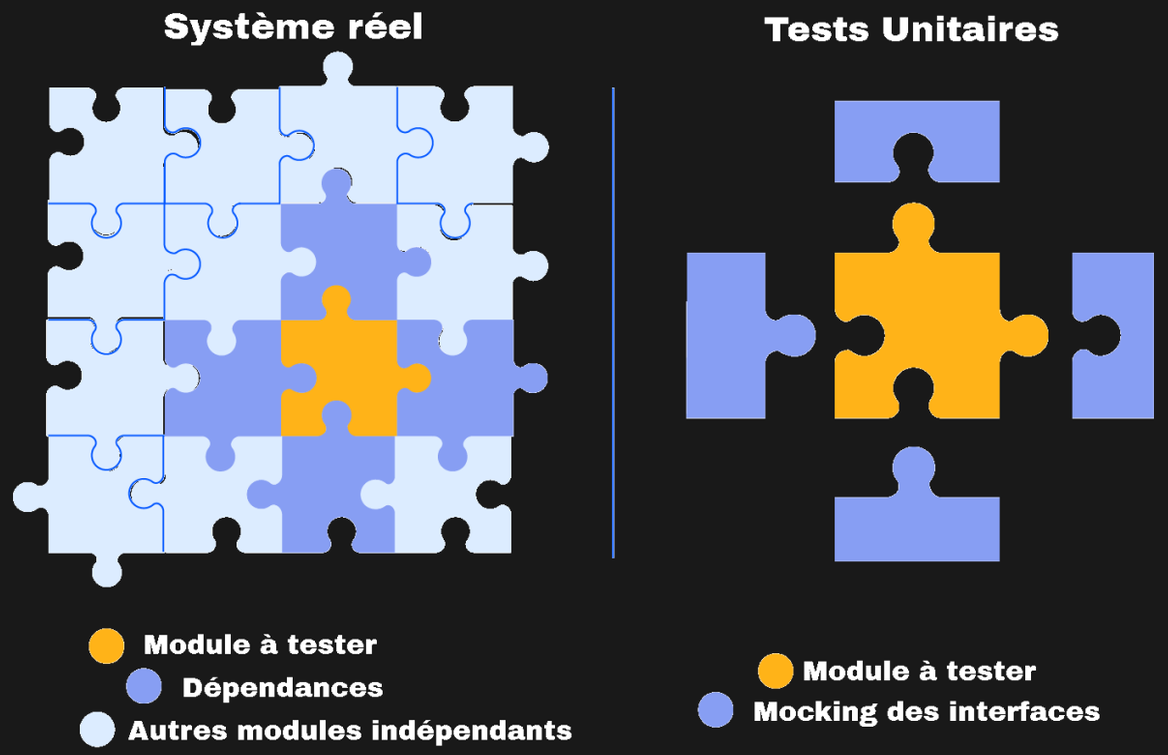
LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level - home/subho/work/lab/zzz - menu.cpp (source / functions)	Lines:	24	43	55.8 %
Test:	ex_test.info	Functions:	5	8	62.5 %
Date:	2014-12-26				

Line	Line data	Source code
1	:	#include <iostream>
2	:	using namespace std;
3	:	
4	:	void showChoices();
5	:	float add(float, float);
6	:	float subtract(float, float);
7	:	float multiply(float, float);
8	:	float divide(float, float);
9	:	
10	2	int main()
11	:	{
12	:	float x, y;
13	:	int choice;
14	2	do
15	:	{
16	2	showChoices();
17	2	cin >> choice;
18	2	switch (choice)
19	:	{
20	:	case 1:
21	0	cout << "Enter two numbers: ";
22	0	cin >> x >> y;
23	0	cout << "Sum " << add(x,y) <<endl;
24	0	break;
25	:	case 2:
26	1	cout << "Enter two numbers: ";
27	1	cin >> x >> y;
28	1	cout << "Difference " << subtract(x,y) <<endl;
29	1	break;
30	:	case 3:
31	0	cout << "Enter two numbers: ";
32	0	cin >> x >> y;
33	0	cout << "Product " << multiply(x,y) <<endl;
34	0	break;

ARCHITECTURE MODULAIRE ET TESTS UNITAIRES

- Architecture modulaire permet de définir des interfaces
- Les tests unitaires vont injecter et observer via ces interfaces



- Mocking : simuler un comportement spécifique des modules externes

COMMENT ÉCRIRE UN TEST UNITAIRE : ASSERT , STUB , MOCK

- Assert: une assertion permet de vérifier un resultat attendu
- Si une assertion n'est pas validé, le test echoue
- Il se trouve à l'extérieur du code à tester

```
1 int a = 0;
2 a = fonctionATester();
3 assert (a > 0);
```

ÉCRIRE UN TEST UNITAIRE : ASSERT , STUB , MOCK

Problèmatique : Le code à tester fait des appels à d'autres modules

Solution :

- Stub: Une "fausse" implémentation de la fonction appelé qui répond toujours la même chose
- Mock: Une "fausse" implémentation de la fonction appelé qui répond en fonction de ce qui a été défini dans le test

RUNTIME ASSERT

- Se trouve à l'intérieur du code à tester
- Librairie standard existante : <assert.h>
- Sert à détécter des valeurs interdites/impossible pendant le dévelopement
- Désactivable quand compilé en mode "Release"

- Exemple, detection d'un pointer null passé en parametre d'une fonction

```
1 void maFonction(int* monPointer, int maValeur)
2 {
3   assert( monPointer != NULL );
4   *monPointer = maValeur;
5 }
```

- Les tests unitaires vont redéfinir le comportement de la fonction assert() pour pouvoir tester leur présence

UN FRAMEWORK EFFICACE POUR LE C EMBARQUÉ : CMOCKA

- Mécanismes de mocking intégré
- Utilise uniquement les librairies standards
- Open source
- Documentation claire

```c
#include <stdarg.h>
#include <stddef.h>
#include <setjmp.h>
#include <cmocka.h>

#include "module_a_tester.h"

static void simple_test(void **state) {
  assert_int_equal( addition(10,23) , 33 );
}

int main(void) {
  const struct CMUnitTest tests[] = {
    cmocka_unit_test(simple_test),
  };

  return cmocka_run_group_tests(tests, NULL, NULL);
}
```

ALLER PLUS LOIN

- Intégration continue
- TDD : Test driven developpement