Projet de Programmation

Benoit Donnet

Année Académique 2019 - 2020

Agenda

Partie 2: Outils

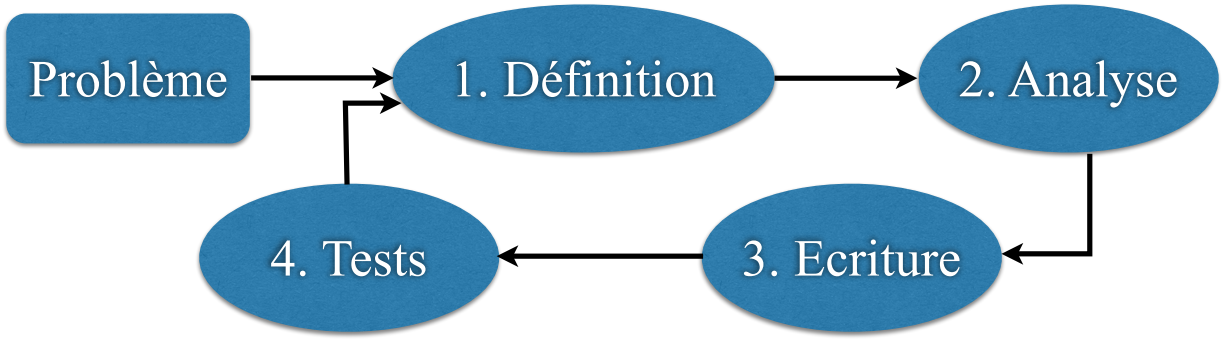
Agenda

- Chapitre 3: Tests
  - Introduction
  - Tests Unitaires
  - Tests d'Implémentation
  - Tests d'Intégration
  - Quand Arrêter les Tests?

Agenda

- Chapitre 3: Tests
  - Introduction
  - Tests Unitaires
  - Tests d'Implémentation
  - Tests d'Intégration
  - Quand Arrêter les Tests?

Introduction



Introduction (2)

- Les tests sont une activité essentielle du développement logiciel
  - ce n'est pas au client d'effectuer ce travail
  - ou alors, il doit en retirer un avantage
- Il existe différents types de tests, chacun avec des objectifs différents. Par exemple:
  - **tests de couverture**
    - ✓ vérification du comportement de la plus grande fraction du code possible
  - **tests de non-régression**
    - ✓ persistance de la validité du module vis-à-vis des spécifications de la version précédente
- cfr. Cours PROJ0010

Introduction (3)

- Pourquoi tester?
  - permet de s'assurer que le système rencontre sa spécification...
- ... mais pas seulement
  - *traçabilité*
    - ✓ les tests aident à remonter du composant vers l'exigence ayant causé le problème
  - *maintenabilité*
    - ✓ les tests de non-régression permettent de s'assurer que tout changement après livraison du produit ne remet pas en cause tout (ou partie) le logiciel
  - *compréhension*
    - ✓ un nouveau venu peut lire le code des tests et comprendre ce que fait le programme
    - ✓ écrire d'abord les tests encourage à rendre l'interface vraiment utilisable

Agenda

- Chapitre 3: Tests
  - Introduction
  - Tests Unitaires
    - ✓ Principe
    - ✓ 4 Phases
    - ✓ `Seatest`
    - ✓ Exemple
  - Tests d'Implémentation
  - Tests d'Intégration
  - Quand Arrêter les Tests?

Principe

- Objectif?
  - attester la validité de chacun des modules qui constituent le projet
    - ✓ module == fichier `.c`
- Moyens?
  - mettre en oeuvre l'ensemble des fonctionnalités décrites dans les spécifications
  - explorer le fonctionnement du module dans des conditions non-spécifiées
- En pratique?
  - tests de couverture et de non-régression de l'ensemble du code, y compris les blocs dédiés à la gestion des erreurs
  - définition de jeux de tests représentatifs
  - comparaison à des résultats attendus

Principe (2)

- Le code destiné aux tests unitaires doit faire partie du code du module
  - permet la réutilisabilité des tests en même temps que du code module proprement dit
  - facilite l'extensibilité des tests
    - ✓ et donc la mise en oeuvre des tests de non-régression

Principe (3)

- Tout module `module.c` doit disposer au moins d'un `module_main_test.c` contenant la procédure de test unitaire
  - contient une fonction `main()`
  - construit par la commande `make test_module`
  - renvoie un code de succès (`exit(0)`) ou d'échec
    - ✓ permet la conduite automatique de tests au moyen de scripts shell
- La procédure de test doit être documentée dans le manuel de maintenance

4 Phases

- On définit généralement 4 phases dans l'exécution d'un test unitaire
  1. *initialisation*
     - ✓ fonction `setup()`
     - ✓ définition d'un environnement de test complètement reproductible
       - **fixture**
  2. *exercice*
     - ✓ le module à tester est exécuté
  3. *vérification*
     - ✓ utilisation de la procédure `assert()`
     - ✓ comparaison des résultats obtenus avec un vecteur de résultats défini
  4. *désactivation*
     - ✓ désinstallation des fixtures pour retrouver l'état initial du système et ne pas polluer les tests suivants

Seatest

- Cadre de test unitaire pour les programmes C
  - architecture de style xUnit
  - portable
  - idéal pour les TDDs
  - http://code.google.com/p/seatest/
- Architecture de `seatest`
  - `seatest.h`
    - ✓ contient les interfaces
    - ✓ contient la définition des macros
  - `seatest.c`
    - ✓ contient les implémentations

seatest (2)

- Les tests unitaires sont implémentés via des macros
  - permet de vérifier le retour des fonctions
- Il y a une macro par type de retour
- Exemples

```
assert_true(test) ...

assert_int_equal(expected, actual) ...

assert_ulong_equal(expected, actual) ...

assert_n_array_equal(expected, actual, n) ...

assert_string_contains(expected, actual) ...

assert_string_starts_with(expected, actual) ...
```

seatest (3)

- On définit, dans un module séparé, les procédures de test
  - ce module contient aussi une fonction `main()` pour l'exécution des tests
  - on peut ajouter une cible particulière dans le Makefile pour les tests
    - ✓ cfr. Chapitre 1
- On écrit une procédure de test par fonction à tester
- Exemple

brol.c
`BrolStructure *foo(int x, char *y){ ... }`
brol-test.c
`void test_foo(){ ... }`

seatest (4)

- La plupart des cadres xUnit viennent avec un mécanisme permettant de découvrir et exécuter automatiquement les procédures de test
- Tout ce qu'on a à faire, c'est écrire la procédure de test
  - le cadre xUnit fait le reste
- Ce mécanisme est inexistant en C
  - on doit le forcer soi-même

seatest (5)

- Seatest force le programmeur à renseigner ses procédures de test pour une exécution automatique
- Implique l'écriture d'une procédure particulière
  - **fixture**

seatest (6)

- Canevas général

```
void test_fixture(){
  test_fixture_start();

  fixture_setup(my_own_setup);
  fixture_teardown(my_own_td);

  run_test(test_1);
  run_test(test_2);
  run_test(test_3);

  test_fixture_end();
}//end test_fixture()
```

démarre l'infrastructure de test

à exécuter avant chaque test

à exécuter après chaque test

enregistre le test_1

termine l'infrastructure de test

seatest (7)

- Une fois tous les fixtures définis, il suffit de tout mettre ensemble

```
void all_tests(){
  test_fixture1();
  test_fixture2();
  ...
  test_fixturen();
}//end all_tests()
```

- Construction du `main()` pour les tests

```
int main(){
  return run_tests(all_tests);
}//end main()
```


Exemple

- On veut créer une librairie qui manipule de l'argent
- L'objet argent est composé de deux informations
  - le montant
  - la devise
    - ✓ USD, EUR, ...
- On veut effectuer de l'arithmétique sur de l'argent
  - exemple: additionner deux objets argent
- Structuration du code:
  - `money.h`
    - ✓ contient les interfaces et une déclaration de structure
  - `money.c`
    - ✓ contient l'implémentation du type et des fonctions/procédures
  - `money-test.c`
    - ✓ contient tous les tests

Exemple (2)

- money.h

```
#ifndef __MONEY__
#define __MONEY__
typedef struct Money_t Money;

Money *create_money(int amount, char *currency);

int get_money_amount(Money *m);

char *get_money_currency(Money *m);

void free_money(Money *m);

void print_money(Money *m);

Money *add_money(Money *m1, Money *m2);

int equal_money(Money *m1, Money *m2);
#endif
```


Exemple (3)

- money.c

```
#include <stdlib.h>
#include "money.h"
struct Money_t{
  int amount;
  char *currency;
};

Money *create_money(int amount, char *currency){
  Money *m = malloc(sizeof(Money));
  if(m==NULL)
    return NULL;
  m->amount = amount;
  m->currency = currency;

  return m;
}//end create_money()

int get_money_amount(Money *m){ return m->amount; }
char *get_money_currency(Money *m){ return m->currency; }
```

Exemple (4)

- money-test.c

```
#include "seatest.h"
#include "money.h"

static void test_create_money(){
  Money *m15EUR = create_money(15, "EUR");

  assert_int_equal(15, get_money_amount(m15EUR));
  assert_string_equal("EUR", get_money_currency(m15EUR));

  free_money(m15EUR);
}//end test_create_money()
```


Exemple (5)

- money.c

```
void free_money(Money*m){free(m);}

void print_money(Money *m){
  printf("%d%s\n", m->amount, m->currency);
}//end money_print()

int equal_money(Money *m1, Money *m2){
  if(m1==NULL && m2==NULL)
    return 1;

  if(m1==NULL || m2==NULL)
    return 0;

  return m1->amount == m2->amount
    && !strcmp(m1->currency, m2->currency);
}//end equal_money()
```

Exemple (6)

- money-test.c

```
static void test_equal_money(){
  Money *m1 = create_money(12, "EUR");
  Money *m2 = create_money(12, "USD");
  Money *m3 = create_money(24, "EUR");

  assert_false(equal_money(m1, NULL));
  assert_true(equal_money(NULL, NULL));
  assert_false(equal_money(NULL, m1));
  assert_true(equal_money(m1, create_money(12, "EUR")));
  assert_false(equal_money(m1, m2));
  assert_false(equal_money(m1, m3));

  free_money(m1);
  free_money(m2);
  free_money(m3);
}//end test_create_money()
```


Exemple (7)

- money.c

```
Money *add_money(Money *m1, Money *m2){
  if(m1==NULL || m2==NULL)
    return NULL;

  if(strcmp(m1->currency, m2->currency))
    return NULL;

  return create_money(m1->amount+m2->amount, m1->currency);
}//end add_money()
```

Exemple (8)

- money-test.c

```
static void test_add_money(){
  Money *m12EUR = create_money(12, "EUR");
  Money *m14EUR = create_money(14, "EUR");
  Money *expected = create_money(26, "EUR");
  Money *result = add_money(m12EUR, m14EUR);

  assert_equal(expected, result);

  free_money(m12EUR);
  free_money(m14EUR);
  free_money(expected);
  free_money(result);
}//end test_create_money()
```


Exemple (9)

- money-test.c

```
static void test_fixture(){
  test_fixture_start();

  run_test(test_create_money);
  run_test(test_equal_money);
  run_test(test_add_money);

  test_fixture_end();
}//end test_fixture()

static void all_tests(){
  test_fixture();
}//end all_tests()

int main(){
  return run_tests(all_tests);
}//end main()
```

Exemple (10)

- Compilation

```
$>gcc -o main-test money.c seatest.c money-test.c
```

- Exécution

```
$>./main-test

------------------ money-test.c ------------------
                  3 run  0 failed

==================SEATEST v0.5====================

               ALL TESTS PASSED
                 3 tests run
                    in 0 ms
==================================================
```


Agenda

- Chapitre 3: Tests
  - Introduction
  - Tests Unitaires
  - Tests d'Implémentation
  - Tests d'Intégration
  - Quand Arrêter les Tests?

Tests d'Implémentation

- Tout module `module.c` gérant une structure de données `Module` doit contenir une fonction `verifie_module()`
- Objectif?
  - vérifier la cohérence de la structure de données passée en paramètre
    - ✓ test en mode "debug"
    - ✓ utilisation d'un flag `MODULE_DEBUG`
- Utilité?
  - seulement si il existe des conditions vérifiables

Tests d'Implémentation (2)

```
int fonction(Matrice *source, Matrice *destination, int param){
  //du code
  #ifdef MATRICE_DEBUG
    if(verifie_matrice(source)!=0){
      //du code
      return 1;
    }
  #endif

  //encore du code

  #ifdef MATRICE_DEBUG
    if(verifie_matrice(destination)==0)
      return -1;
  #endif

  return 0;
}//fin fonction()
```

Agenda

- Chapitre 3: Tests
  - Introduction
  - Tests Unitaires
  - Tests d'Implémentation
  - Tests d'Intégration
  - Quand Arrêter les Tests?

Tests d'Intégration

- Objectif?
  - attester la validité du projet dans son ensemble
- Moyen?
  - mise en oeuvre de jeux de tests de taille réelle
- Tout projet doit disposer d'un (ou plusieurs) fichiers contenant la procédure de test d'intégration
  - procédure documentée dans le manuel de maintenance
    - ✓ nécessaire au bon déroulement des tests de non-régression

Agenda

- Chapitre 3: Tests
  - Introduction
  - Tests Unitaires
  - Tests d'Implémentation
  - Tests d'Intégration
  - Quand Arrêter les Tests?

Quand Arrêter?

Testing can only show the presence of bugs, not their absence
(E. W. Dijkstra)

- Réponse cynique (*sad but true* © Metallica)
  - on ne finit jamais les tests
    - ✓ chaque exécution du programme est un nouveau test
    - ✓ chaque correction de bug doit s'accompagner de tests de non-régression
  - on arrête les tests quand on est à court de temps et/ou d'argent
    - ✓ inclure les tests dans les plans du projet
    - ✓ ne pas les faire en urgence
      - sur le long terme, une bonne planification des tests permet d'économiser du temps (et donc de l'argent)

Quand Arrêter (2)

- Tests statistiques
  - on teste jusqu'à ce qu'on ait amené le taux d'erreur en-dessous d'un seuil acceptable