

INF3135

Construction et maintenance de logiciels

Cours 13: Tests

Alexandre Blondin Massé

Université du Québec à Montréal
Département d'informatique

Été 2020

Table des matières

1 Quiz

2 Travail pratique 3

3 Tests

4 Développement guidé par les tests

Quiz

Quiz

Quiz 3

- **Moyenne:** 68.7%

Quiz 4

Matière:

- Chapitre 8: tests
- Labo 10: intégration
- Labo 11: tests

Contenu:

- Une question courte sur la gestion de la **mémoire**
- Une question courte sur l'**intégration**
- Une question courte sur les **tests**
- Une question longue sur les **tests**

Travail pratique 3

Travail pratique 3

- **Dépôt:** <https://gitlab.info.uqam.ca/inf3135-ete2020/inf3135-ete2020-tp3>
- **Sujet:** <https://gitlab.info.uqam.ca/inf3135-ete2020/inf3135-ete2020-tp3/-/tree/sujet/sujet>
- Développer un petit **jeu vidéo**
- Inspiré du jeu **Super Hexagon** de Terry Cavanagh
- Avec bibliothèque **SDL2**
- **Coquille** de base disponible
- Peut être fait en **équipe** d'au plus 3
- **Attention!** je vérifie qui fait quoi

Tests

Pourquoi tester?

- Détecter des **bogues**
- Et éventuellement les **corriger**
- Avoir une plus grande **confiance** en notre programme
- S'assurer de ne pas introduire de **régression**

Idéal

- Prouver que notre programme est **sans bogue**
- **Impossible** dans la majorité des cas
- Sauf pour des **petits programmes**
- Ou en utilisant un outil de **vérification formelle**
- **Champ d'étude**: analyse de programme

Problème de l'arrêt (*halting problem*)

Problème

- Peut-on écrire un programme G qui
- Étant donné un **programme** P
- Décide si P **termine toujours**
- Peu importe les valeurs en **entrée**?

Réponse

- **1936**: Turing a prouvé qu'un tel programme G ne peut pas exister
- Plus formellement, le problème est **indécidable** (Gödel)

Conséquence

Impossible de **prouver** qu'un programme arbitraire est **sans bogue**

Exemple: collatz

```
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>

void print_collatz_sequence(unsigned int n) {
    while (true) {
        printf("%d ", n);
        if (n == 1)
            break;
        else if (n % 2 == 0)
            n = n / 2;
        else
            n = 3 * n + 1;
    }
}

int main(int argc, char *argv[]) {
    print_collatz_sequence(atoi(argv[1]));
    return 0;
}
```

Résultat:

```
$ gcc collatz.c -o collatz
$ ./collatz 85
85 256 128 64 32 16 8 4 2 1
```

Différents types de tests (1/3)

Compilation (*build*)

- **Compilation** correcte
- Édition des **liens**
- Avec **bibliothèques** tierces
- Ou autres **dépendances**

Intégration

- Interaction correcte **entre** les modules
- Autant **compilation**
- Que tests **unitaires**

Différents types de tests (2/3)

Unitaires

- Teste un aspect **spécifique**
- De façon **individuelle**

Fonctionnels

- **Comportement** du programme
- Est en adéquation avec ce qui était **demandé**
- Respecte le **cahier des charges**

Non-régression

- Pas de perte de **fonctionnement**
- Ou de perte importante de **performance**

Différents types de tests (3/3)

Configuration

- Fonctionne dans des **environnements** variés
- **Appareils** (ordinateur, mobile, console, ...)
- **Distribution** (Linux, MacOS, Windows, ...)
- **Architecture** (32 bits, 64 bits, autre processeur, ...)

Performance

- **Rapidité** du programme
- Utilisation de **mémoire**

Installation

L'application s'installe correctement

Propriétés d'un bon test

- **Juste**: il teste bien ce qu'il faut
- **Robuste**: il ne plante pas
- **Pur**: il est sans effet de bord
- **Reproductible**: il a le même comportement peu importe l'environnement
- **Pertinent**: il augmente notre confiance
- **Non redondant**: il teste quelque chose de distinct d'un autre test
- **Efficace**: il prend un temps raisonnable
- **Automatisable**: il ne nécessite pas d'intervention humaine

Automatisation

Tests automatisés

- Vérification **textuelle**, notamment à l'aide de *regex*
- Utilisation des **canaux standards** (`stdin`, `stdout`, `stderr`)
- Ou accès direct au **contenu** (tests internes)
- Construction de **scénarios**

Tests manuels

- Quand automatisation **pas possible**
- Vérification **humaine**
- Évaluation **visuelle**, **sonore**, ...
- Suite d'**événements**, parfois **asynchrones**
- Prennent du **temps** et souvent **coûteux**
- Typique des applications **graphiques**
- Tests d'**interface d'utilisation**

Tests shell

- De nombreuses **commandes shell** permettent de tester
- Avec la **sémantique** habituelle (0: succès, $\neq 0$: erreur)

Exemples:

```
# Vérifie si README.md contient un code permanent
#   -q: mode silencieux
#   -E: expression étendue
$ grep -qE "[A-Z]{4}[0-9]{8}" README.md

# Vérifie si bidon.c existe dans un sous-dossier de /tmp
$ find /tmp -name bidon.c | grep -q .

# Vérifie si fichier1 et fichier 2 sont presque identiques
#   -i: ignorer la casse
#   -w: ignorer les espaces
$ diff -iw fichier1 fichier2

# Vérifie si une commande engendre une fuite mémoire
# Valgrind retourne 0 si aucune fuite, 1 sinon
$ valgrind --leak-check=yes --error-exitcode=1 ./prog; echo $?
```


La commande test

Vérifier le type des fichiers et compare des valeurs:

```
test EXPRESSION [OPTION]
```

- Si l'expression est **vraie** alors la commande retourne 0
- Sinon elle retourne 1

Exemples:

```
# Vérifie si 1 < 2 (comparaison numérique)
```

```
$ test 1 -lt 2; echo $?
```

```
0
```

```
# Vérifie si linux = linux (en tant que chaînes)
```

```
$ test $(echo "linux") = "linux"; echo $?
```

```
0
```

```
# Vérifie s'il y a un Makefile dans le répertoire courant
```

```
$ test -f Makefile; echo $?
```

```
0
```

```
# Vérifie s'il y a un répertoire code dans le répertoire courant
```

```
$ test -d code; echo $?
```

```
0
```

Tests sur chaînes de caractères

```
test CHAINE1 OPERATEUR CHAINE2
```

Exemples:

```
# Vérifie si deux chaînes sont égales
```

```
$ test "linux" = "Linux"; echo $?
```

```
1
```

```
# Vérifie si deux chaînes sont différentes
```

```
$ test "linux" != "Linux"; echo $?
```

```
0
```

```
# Vérifie si une chaîne est vide
```

```
$ test -z ""; echo $?
```

```
0
```

```
$ test -z "linux"; echo $?
```

```
1
```

```
# Vérifie si une chaîne est non vide
```

```
$ test -n ""; echo $?
```

```
1
```

```
$ test -n "linux"; echo $?
```

```
0
```

Tests sur les valeurs numériques

```
test VALEUR1 OPERATEUR VALEUR2
```

Exemples:

```
# Vérifie si deux valeurs sont égales
$ test 1 -eq 1
# Vérifie si deux valeurs sont différentes
$ test 1 -ne 2
# Vérifie une inégalité
$ test 1 -lt 2
$ test 2 -le 2
$ test 2 -gt 1
$ test 1 -ge 1
# Attention à différencier `=` de `-eq`
$ test "01" = 1; echo $?
1
$ test "01" -eq 1; echo $?
0
```

Tests sur les fichiers

test OPTION CHEMIN

Exemples:

Est-ce que le chemin existe?

\$ test -e /usr/local/bin; echo \$?

0

Est-ce que le chemin est un fichier?

\$ test -f /usr/local/bin/bats; echo \$?

0

Est-ce que le chemin est un répertoire?

\$ test -d /usr/local/bin; echo \$?

0

Est-ce que le chemin est un fichier non vide?

\$ touch nouveau

\$ test -s nouveau; echo \$?

1

Est-ce que le chemin est accessible en lecture?

\$ test -r chemin

Est-ce que le chemin est accessible en écriture?

\$ test -w chemin

Est-ce que le chemin est exécutable?

\$ test -x chemin

Opérateurs logiques

```
test EXPRESSION1 OPERATEUR EXPRESSION2
```

Exemples:

```
# ET logique
```

```
$ test expr1 -a expr2
```

```
# OU logique
```

```
$ test expr1 -o expr2
```

```
# NON logique
```

```
$ test ! expr
```

```
# Retourne vrai si chemin est un fichier vide
```

```
test -f chemin -a ! -s chemin
```

Syntaxe allégée

La syntaxe

```
test EXPRESSION
```

est équivalente à

```
[ EXPRESSION ]
```

Exemples:

```
$ [ -f Makefile ]; echo $?
```

```
1
```

```
$ [ -d bin ]; echo $?
```

```
0
```

- Les **espaces** après [et avant] sont importants
- Le caractère] est optionnel (mais plus joli)

Tests Bash

Syntaxe avec **doubles crochets**:

```
[[ EXPRESSION ]]
```

Opérateurs possibles:

- && et ||: connecteur logiques ET et OU
- (et): pour parenthéser
- < et >: comparaison lexicographique de chaînes
- ==: la 2e opérande est un motif de type *glob*
- =~: la 2e opérande est une expression régulière étendue

```
# La première expression correspond avec b = ? et * = njour
# La deuxième aussi avec onjou
# [un]          alternative entre u et n
# j?           j optionnel
# (...){2}     motif répété exactement deux fois
$ [[ bonjour == ?o* && bonjour =~ (o[un]j?){2} ]]
$ echo $?
0
```

Bats

- *Bats* = *Bash Automated Testing System*
- **Lien:** <https://github.com/bats-core/bats-core>
- **Image:** [DockerHub](#)

*« Bats is a TAP-compliant testing framework for **Bash**. It provides a simple way to verify that the UNIX programs you write behave as expected.*

*A Bats test file is a Bash script with special syntax for defining test cases. Under the hood, each test case is **just a function with a description**. »*

- Autrement dit, il suffit d'utiliser des **commandes** de test
- Et on peut profiter des **expressions régulières**

Exemples de tests Bats

```
valgrind_options="--error-exitcode=1 --leak-check=full"
```

```
# On vérifie s'il y a une fuite mémoire
```

```
@test "No leak with default program" {  
    run valgrind $valgrind_options ./program  
    [ "$status" -eq 0 ]  
}
```

```
# On vérifie que le fichier diagram.dot est valide
```

```
@test "File diagram.dot is valid" {  
    run neato diagram.dot  
    [ "$status" -eq 0 ]  
}
```

```
# On vérifie les premières lignes
```

```
@test "Show help with -h" {  
    run ./program -h  
    [ "$status" -eq 0 ]  
    [ "${lines[0]}" = "Usage: ./program [-h|--help]" ]  
    [[ "${lines[0]}" == "[U]sage" ]]  
    [[ "${lines[0]}" =~ "h(elp)?" ]]  
}
```

Exemple plus complexe (1/2)

- Programme `tournament.c` qui **génère** une grille de tournoi
- En lisant sur l'**entrée standard**
- Chaque ligne correspond à un **joueur** ou une **équipe**
- On veut **limiter à 20** pour des raisons d'affichage

```
$ cat examples/tennis.in
```

```
Djokovic
```

```
Nadal
```

```
Federer
```

```
Murray
```

```
$ ./tournament -s table < examples/tennis.in
```

ID	Player	Day 1	Day 2	Day 3
--	-----	-----	-----	-----
1	Djokovic	4	2	3
2	Nadal	3	1	4
3	Federer	2	4	1
4	Murray	1	3	2

Exemple plus complexe (2/2)

```
# Vérifier l'affichage à espace près
# -Z ignorer les espaces en fin de ligne
# -B ignorer les lignes vides
@test "Tennis example with default options" {
    run diff -ZB examples/tennis-default.out \
        <(/tournament < examples/tennis.in)
    [ "$status" -eq 0 ]
}

# Détecter le cas où on a plus de 20 équipes
@test "Too many players" {
    run ./tournament < examples/soccer-long.in
    [[ "$output" =~ "Error.*too many players" ]]
    [ "$status" -eq 1 ]
}

# Permettre le cas où on a exactement 20 équipes
# On ne peut pas utiliser | en combinaison avec run
@test "Twenty players is ok" {
    head -n 20 examples/soccer-long.in | ./tournament
    [ "$?" -eq 0 ]
}
```

Développement guidé par les tests

3 lois

Extraites de [Wikipedia](#):

1. Vous devez écrire un test qui échoue avant de pouvoir écrire le code de production correspondant.
2. Vous devez écrire une seule assertion à la fois, qui fait échouer le test ou qui échoue à la compilation.
3. Vous devez écrire le minimum de code de production pour que l'assertion du test actuellement en échec soit satisfaite.

Proposée par **Robert C. Martin** (2014)

Cycle de développement en 5 étapes

1. Ajouter un test ou plusieurs tests

Qui mettent en évidence le comportement souhaité

2. Lancer tous les tests

Les nouveaux tests devraient échouer

3. Écrire du code

Pas besoin d'être parfait

4. Lancer tous les tests

Les nouveaux tests devraient réussir, les anciens aussi

5. Factorisation et nettoyage

En s'assurant que les tests réussissent toujours

Exemple: le module set (1/4)

Interface (fichier set.h):

```
struct set {                // A set of integers
    int *elements;          // Its elements
    unsigned int size;      // Its cardinality
    unsigned int capacity;  // Its capacity
};

// Create an empty set
struct set *set_create(void);
// Delete a set
void set_delete(struct set *set);
// Check if a set is empty
bool set_is_empty(const struct set *set);
// Check if a set contains an element
bool set_contains(const struct set *set, int element);
// Add an element to a set
void set_add(struct set *set, int element);
// Print a set to stdout
void set_print(const struct set *set);
```

Exemple: le module set (2/4)

Implémentation (fichier set.c):

```
struct set *set_create(void) {
    struct set *set = malloc(sizeof(struct set));
    set->elements = malloc(sizeof(int));
    set->capacity = 1;
    set->size = 0;
    return set;
}

void set_delete(struct set *set) {
    free(set->elements);
    free(set);
}

bool set_is_empty(const struct set *set) {
    return set->size == 0;
}

void set_print(const struct set *set) {
    printf("{");
    for (unsigned int i = 0; i < set->size; ++i) {
        if (i > 0) printf(", ");
        printf("%d", set->elements[i]);
    }
    printf("}");
}
```


Exemple: le module set (3/4)

Implémentation (fichier set.c):

```
int compare_ints(const void *i1, const void *i2) {
    return *(int*)i1 - *(int*)i2;
}

bool set_contains(const struct set *set,
                  int element) {
    return bsearch(&element, set->elements, set->size,
                  sizeof(int), compare_ints) != NULL;
}

void set_add(struct set *set,
             int element) {
    unsigned int i = 0;
    while (i < set->size && set->elements[i] < element)
        ++i;
    if (set->elements[i] == element) return;
    if (set->size == set->capacity) {
        set->capacity *= 2;
        set->elements = realloc(set->elements, set->capacity * sizeof(int));
    }
    for (unsigned int j = set->size; j > i; --j)
        set->elements[j] = set->elements[j - 1];
    set->elements[i] = element;
    ++set->size;
}
```

Exemple: le module set (4/4)

Tests (fichier test_set.c):

```
#include "set.h"
#include <tap.h>

int main(void) {
    struct set *set = set_create();
    ok(set_is_empty(set), "created set is empty");
    ok(set->size == 0, "size of set is 0");
    set_add(set, 3);
    set_add(set, 5);
    set_add(set, 2);
    diag("Adding 3, 5, 2");
    printf("# set = "); set_print(set); printf("\n");
    ok(set->size == 3, "size of set is 3");
    ok(set_contains(set, 3), "set contains 3");
    ok(!set_contains(set, 1), "set does not contains 1");
    diag("Adding 5 again");
    set_add(set, 5);
    printf("# set = "); set_print(set); printf("\n");
    ok(set->size == 3, "size of set is still 3");
    set_delete(set);
    return 0;
}
```

Ajout de la fonction de suppression d'un élément

1. Ajouter un test ou plusieurs tests

Deux cas: élément **présent** ou **absent**

2. Lancer tous les tests

Les deux tests devraient échouer

3. Écrire du code

Pas besoin d'être parfait

4. Lancer tous les tests

Les deux nouveaux tests et les anciens devraient réussir

5. Factorisation et nettoyage

En s'assurant que les tests réussissent toujours