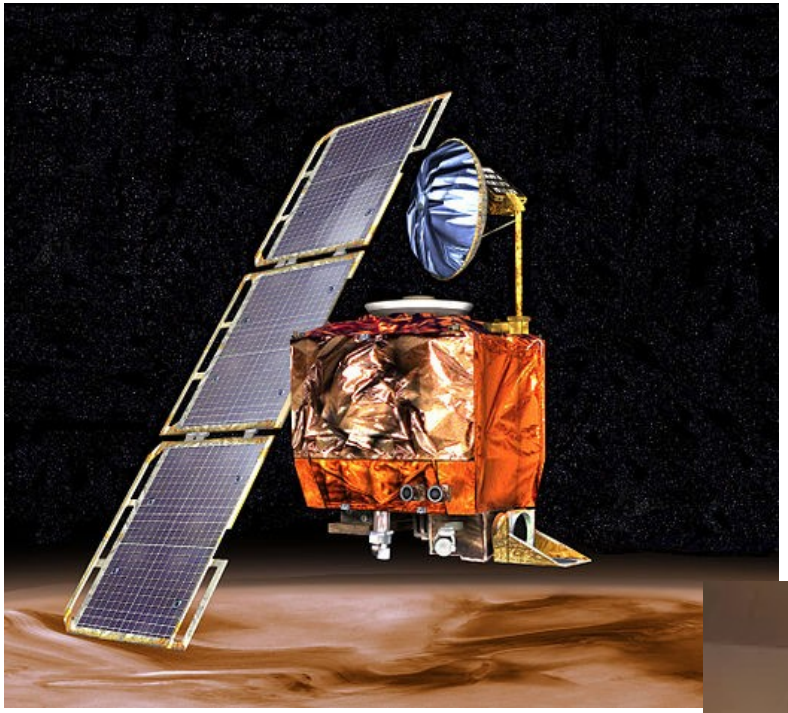
Projet d’immersion

Test


TESTER
C'EST DOUTER
COMMITSTRIP.COM

- Feu d’artifice à ~500M$
  - Erreur logicielle



...

Du code ? On sait faire !… ?

- Étude du Standish Group, 2014
- Projets aux USA
  - Type 1 : 16,2 % opérationnels avec le budget et le timing prévu
  - Type 2 : 52,7 % opérationnels mais
    - Budget plus élevé que prévu
    - Et/ou temps plus long que prévu
    - Et/ou moins de fonctionnalités que prévu
  - Type 3 : 31,1 % abandonnés

TESTER !

- Exécuter son code pour voir s'il marche
  - Fait naturellement en phase de développement
  - Refaire les validations à chaque changement
- Oui mais pas tout
  - Plein d'aspects différents
  - Plein de bouts de code différents
  - Plein de cas différents

    → Trop à gérer !

TESTER !

- Exécuter son code pour voir s'il marche
- Trop à gérer !
- Solution :

  Faire exécuter son code pour voir s'il marche…
  … par du code !

  - Exécution automatique
    - Plein de cas, aspects, bouts, ...
  - Dit ça marche
    - Ou alors ça marche pas, pourquoi et où
  - Un deuxième programme à développer

Exemple

- Un code fonctionnel :

```
/* dummy.c */ int add(int a, int b) { return a+b; }
```

- Un test :

```
// dummy.test.c

#include "dummy.h"
#include <stdio.h>

int test_add_1_2() { // automatique ; retourne marche / marche pas
    const int lhs = 1, rhs = 2; // les données du test
    const int expected = 3; // le résultat attendu
    const int actual = add(lhs, rhs); // on lance le code (sous test)
    if (actual != expected) { // et on voit si ça marche !
        // Message : qui a un souci ? Dans quel cas ? Quel test ?
        fprintf(stderr, "test_add_1_2: add: expecting %d while got %d", expected, actual);
        return 0; // On signale qu'il y a eu un problème
    }
    return 1;
}
```

Un test suffit-il ?

- Un autre code fonctionnel :

```
// dummy.c
#include <stdio.h>

int nat_add(int a, int b) {
    if (a < 0 || b < 0) {
        fprintf(stderr, "add: illegal values (%d, %d) ; should be all positive", a, b);
        return -1;
    }

    int ret = a+b;

    if (ret < a || ret < b) {
        fprintf(stderr, "add: adding %d and %d overflows", a, b);
        return -1;
    }

    return a+b;

}
```

Plein de cas

- a et b positifs ou nuls
  - Tester quelques cas significatifs
- a ou b négatifs
  - Tester les deux (cas au moins)
- a ou b très très grand
  - Tester quand c'est trop grand
  - Tester quand ça marche quand même

Un test par cas !

(au moins)

Exemple : grand mais marche

```c
// dummy.test.c

#include "dummy.h"
#include <stdio.h>
#include <limits.h>
…
// Un cas bien précis ; on peut en avoir vraiment beaucoup
int test_nat_add_1_to_max_minus_1_works() { // nom représente bien le cas testé
    // Noms de variables finement choisis
    // Valeurs dures à lire en constante
    const int lhs = INT_MAX-1, rhs = 1, expected = INT_MAX, actual = nat_add(lhs, rhs);
    if (actual != expected) { // Un contrôle lisible
        // Un message clair : qu'est-ce qui échoue, où ça échoue, pourquoi ça échoue
        printf("test_nat_add_1_to_max_minus_1_works: ");
        printf("nat_add(%d, %d): ", lhs, rhs);
        printf("expecting %d while got %d\n", expected, actual);
        return 0;
    }
    return 1;
}
```

Comment lancer tout ça ?

```
int main(int argc, char *argv[]) {
    int test_count = 0, failures = 0;

    if (! test_nat_add_ok()) failures++;
    test_count++;
    if (! test_nat_add_a_neg()) failures++;
    test_count++;
    if (! test_nat_add_a_and_b_neg()) failures++;
    test_count++;
    if (! test_nat_add_1_to_max_overflows()) failures++;
    test_count++;
    if (! test_nat_add_1_to_max_minus_1_works()) failures++;
    test_count++;

    printf("%d failures out of %d tests\n", failures, test_count);
    return failures;  // Pour bash, 0 c'est OK...
}
```

Comment lancer tout ça ? Soyons futés !

```
// Lancer un test dont la fonction est passée en paramètre et les stats par pointeur
void run_test(int (*f)(), int *failures, int *runs) {
    if (!f()) (*failures)++;
    (*runs)++;
}


// Lancer les tests les uns à la suite des autres
int main(int argc, char *argv[]) {
    int test_count = 0, failures = 0;
    run_test(test_nat_add_ok, &failures, &test_count);
    run_test(test_nat_add_a_neg, &failures, &test_count);
    run_test(test_nat_add_a_and_b_neg, &failures, &test_count);
    run_test(test_nat_add_1_to_max_overflows, &failures, &test_count);
    run_test(test_nat_add_1_to_max_minus_1_works, &failures, &test_count);
    printf("%d failures out of %d tests\n", failures, test_count);
    return failures;
}
```

Les cas compliqués

- Appel d'un truc hors du module sous test
  - Appel de `printf` à contrôler
    - Attraper stdio et le mettre dans un fichier

      ```
      freopen("/tmp/load_test_capture","w+",stdout);
      …
      freopen ("/dev/tty", "w", stdout);
      ```

      Le contenu du fichier est à vérifier
  - Appel de méthode GTK
    - On oublie ici
      - Par contre, code intelligent (à tester) ailleurs !

        Pas de calculs dans le code GTK, seulement des appels

Oui mais quand j’ai fini de tester ?

- Jamais !
  - 1+1
  - 1+2
  - 1+3
  - 1+4
  - …
  - 2+1
  - …

Une métrique

- La couverture de code !
  - Montre par où sont passés les tests
    - Et donc ce qu'on a (pas) oublié

LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level - src - dummy.c (source / functions)				
Test:	dummy-cov.info	**Lines:**	8	10	80.0 %
Date:	2019-09-10 13:07:08	**Functions:**	2	2	100.0 %

```
      Line data    Source code
    1             : // dummy.c
    2             : #include <stdio.h>
    3             :
    4           1 : int add(int a, int b) { return a+b; }
    5             :
    6             : int nat_add(int a, int b) {
    7           4 :     if (a < 0 || b < 0) {
    8           2 :         fprintf(stderr, "add: illegal values (%d, %d) ; should be all positive\n", a, b);
    9           2 :         return -1;
   10             :     }
   11           1 :     int ret = a+b;
   12           2 :     if (ret < a || ret < b) {
   13           0 :         fprintf(stderr, "add: adding %d and %d overflows\n", a, b);
   14           0 :         return -1;
   15             :     }
   16           1 :     return a+b;
   17             :
   18           3 : }
```

Generated by: LCOV version 1.13

Une métrique

- La couverture de code !
  - Montre par où sont passés les tests
  - Sur le code fonctionnel (pas les tests)

LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level - src - dummy.c (source / functions)				
Test:	dummy-cov.info	**Lines:**	8	10	80.0 %
Date:	2019-09-10 13:07:08	**Functions:**	2	2	100.0 %

Line	Line data	Source code
1	:	// dummy.c
2	:	#include <stdio.h>
3	:	
4	1 :	int add(int a, int b) { return a+b; }
5	:	
6	:	int nat_add(int a, int b) {
7	4 :	if (a < 0 || b < 0) {
8	2 :	fprintf(stderr, "add: illegal values (%d, %d) ; should be all positive\n", a, b);
9	2 :	return -1;
10	:	}
11	1 :	int ret = a+b;
12	2 :	if (ret < a || ret < b) {
13	0 :	fprintf(stderr, "add: adding %d and %d overflows\n", a, b);
14	0 :	return -1;
15	:	}
16	1 :	return a+b;
17	:	
18	3 :	}

Generated by: LCOV version 1.13

De jolis rapports

- Compiler en mode couverture
  - Options *-fprofile-arcs* et *-ftest-coverage* de `gcc`
    - Seulement au moment des tests !
- Exécution → fichiers (binaires) de couverture
- Fichiers de couverture lisibles
  - Gcov pour des rapports textuels
  - Lcov + genhtml pour des rapports HTML

    ⚠ options b et d

Torchons et serviettes

- Ne pas mélanger le code fonctionnel avec le code de test !
  - On ne distribue pas les tests dans le binaire
  - Les rapports ne sont intéressants que sur le code fonctionnel
  - Options de compilation différentes
  - Le code fonctionnel dans `src`
    Le code de test dans `tests`
- `gcc` va râler
  - `-I` pour lui expliquer où sont les `.h` (de `src`)
    - Check the man, man !
  - Compiler tous les `.c` (dans `src` – *sauf main* – et `test`) quand on veut tester
  - Compiler tous les `.c` de `src` pour le binaire final

Travail attendu

- Des tests
  - Plein de tests pour chaque module (`.h`)
    - Une méthode (par module) pour les appeler tous !
  - Tous appelé à un seul endroit
    - Un main pour le test
    - Ne pas inclure le main fonctionnel
- Un rapport de couverture
  - En texte, ou mieux html
  - Qui montre que vous avez bien testé

    Le code fonctionnel
- Et…

Travail attendu

- Des tests
- Un rapport de couverture
- Quelque chose pour lancer/générer tout ça
  - Une seule commande doit
    - Compiler pour les tests
      - Tests inclus et options ésotériques positionnées
    - Tester
    - Faire les rapports
    - Compiler pour distribuer
      - Options ésotériques, mais pour le binaire à distribuer