Conduite de Projet
Cours 8 — Testing

Stefano Zacchiroli
zack@irif.fr

Laboratoire IRIF, Université Paris Diderot

2017–2018

Outline

1. Software testing — an introduction
2. The “Check” unit testing framework

Outline

1. Software testing — an introduction

2. The “Check” unit testing framework

Méthodes de V&V complémentaires

- Model-checking :
  - ✔ Exhaustif, automatique
  - ✗ Mise en œuvre moyennement difficile (modèles formelles, logique temporelle)
- Preuve :
  - ✔ Exhaustif
  - ✗ Mise en œuvre difficile, limitation de taille
- Test :
  - ✔ Nécessaire : exécution du système réel, découverte d'erreurs à tous les niveaux (spécification, conception, implantation)
  - ✗ Pas suffisant : exhaustivité impossible

Test de logiciel

Selon IEEE (Standard Glossary of Software Engineering Terminology)

« Le test est l'exécution ou l'évaluation d'un système ou d'un composant, par des moyens automatiques ou manuels, pour vérifier qu'il répond à ses spécifications ou identifier les différences entre les résultats attendus et les résultats obtenus. »

- Validation dynamique (exécution du système
- Comparaison entre système et spécification

Qu'est ce qu'un « bug » ?

(vocabulaire IEEE)

- Anomalie (fonctionnement) : différence entre comportement attendu et comportement observé
- Défaut (interne) : élément ou absence d'élément dans le logiciel entraînant une anomalie
- Erreur (programmation, conception) : comportement du programmeur ou du concepteur conduisant à un défaut

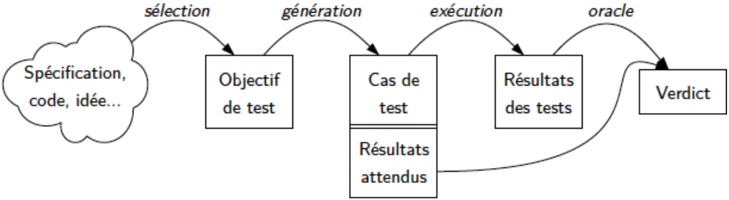
erreur ➔ défaut ➔ anomalie

Qu'est ce qu'un test ?

(vocabulaire IEEE)

- SUT (System Under Test) : le système testé.
- Objectif de test : comportement SUT à tester
- Données de test : données à fournir en entrée au système de manière à déclencher un objectif de test
- Résultats d'un test : conséquences ou sorties de l'exécution d'un test
  - (affichage à l'écran, modification des variables, envoi de messages...)
- Cas de test : données d'entrée **et** résultats attendus associés à un objectif de test

Qu’est ce qu’un test ?



Exemple

- Spécification : Le programme prend en entrée trois entiers, interprétés comme étant les longueurs des côtés d'un triangle. Le programme retourne la propriété du triangle correspondant : scalène, isocèle ou équilatéral.

- Exercice : Écrire un ensemble de tests pour ce programme

Exemple

Cas valides	Données	Résultat attendu
triangle scalène valide	(10,5,7)	scalène
triangle isocèle valide + permutations	(3,5,5)	isocèle
triangle équilatéral valide	(3,3,3)	équilatéral
triangle plat (a+b=c) + permutations	(2,2,4)	scalène

Cas invalides

pas un triangle (a+b<c) + permutations	(2,1,5)	triangle invalide
une valeur à 0	(3,0,4)	triangle invalide
toutes les valeurs à 0	(0,0,0)	triangle invalide
une valeur négative	(2,-1,6)	triangle invalide/entrée invalide
une valeur non entière	('a',4,2)	entrée invalide
mauvais nombre d'arguments	(3,5)	entrée invalide

Un autre exemple : tri d'une liste

Objectif de test	Donnée d'entrée	Résultat attendu	Résultat du test
liste vide	[ ]	[ ]	[. . .]
liste à 1 élément	[3]	[3]	[. . .]
liste ≥ 2 éléments, déjà triée	[2;6;9;13]	[2;6;9;13]	[. . .]
liste ≥ 2 éléments, non triée	[7;10;3;8;5]	[3;5;7;8;10]	[. . .]

égalité ?

Problème de l'oracle

- Oracle : décision de la réussite de l'exécution d'un test, comparaison entre le résultat attendu et le résultat obtenu
- Problème : décision pouvant être complexe
  - types de données sans prédicat d'égalité
  - système non déterminisme : sortie possible mais pas celle attendue
  - heuristique : approximation du résultat optimal attendu
    - Exemple : problème du sac à dos
- Risques : Échec d'un programme conforme si définition trop stricte du résultat attendu
  - => faux positifs (false fails)

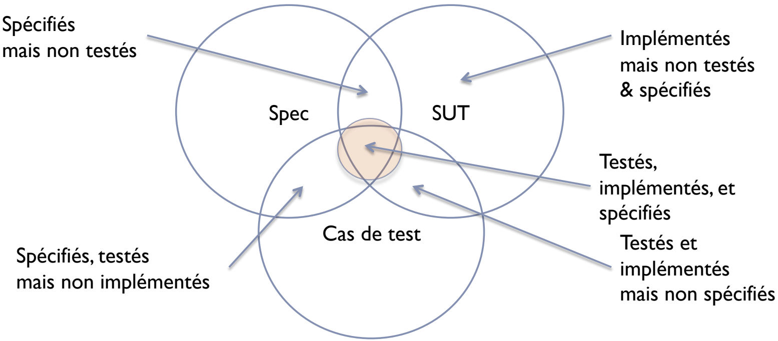
Faux positifs et faux négatifs

- Validité des tests : Les tests n'échouent que sur des programmes incorrects
- Faux positif (*false-fail*) *: fait échouer un programme correct*

- Complétude des tests : Les tests ne réussissent que sur des programmes corrects
- Faux négatif (*false-pass*) *: fait passer un programme incorrect*

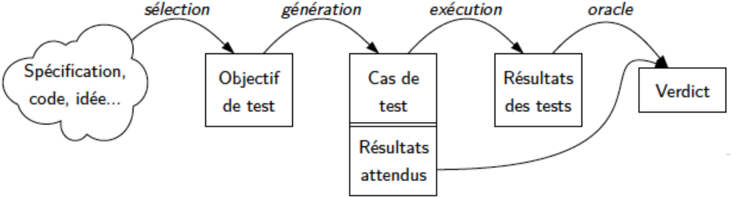
- Validité indispensable, complétude impossible en pratique
  - Toujours s'assurer que les tests sont valides

Validité et complétude des tests



Processus de test

1. Choisir les comportements à tester (objectifs de test)
2. Choisir des données de test permettant de déclencher ces comportements + décrire le résultat attendu pour ces données
3. Exécuter les cas de test sur le système + collecter les résultats
4. Comparer les résultats obtenus aux résultats attendus pour établir un verdict



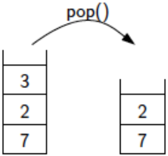
Exécution d’un test

- Scénario de test :
  - Préambule : Suite d'actions amenant le programme dans l'état nécessaire pour exécuter le cas de test
  - Corps : Exécution des fonctions du cas de test
  - Identification (facultatif) : Opérations d'observation rendant l'oracle possible
  - Postambule : Suite d'actions permettant de revenir à un état initial

Exécution de test

Ex : Pop (supprimer le sommet d'une pile)

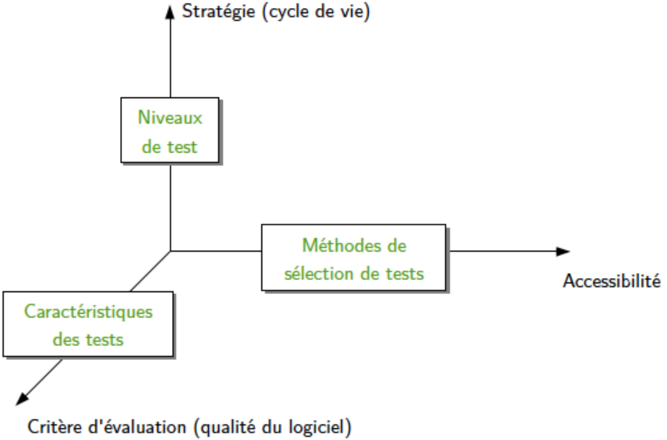
Cas de test :



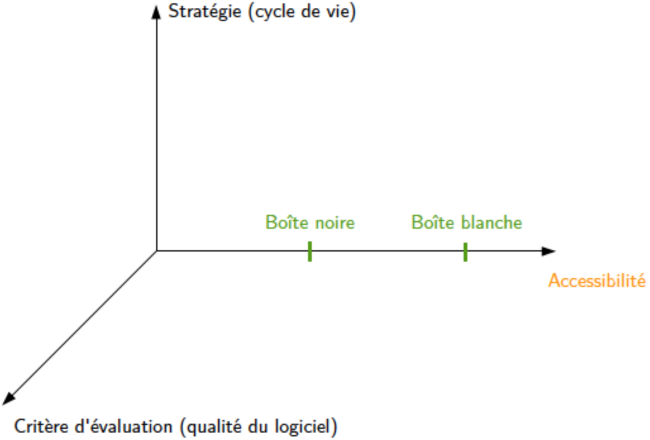
Exécution du test :

Préambule	push(7)
	push(2)
	push(3)
Corps	pop()
Identification	top() = 2
	pop()
	top() = 7
	pop()
	top() = *empty*

Types de test

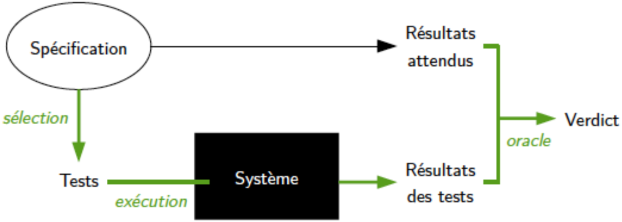


Types de test



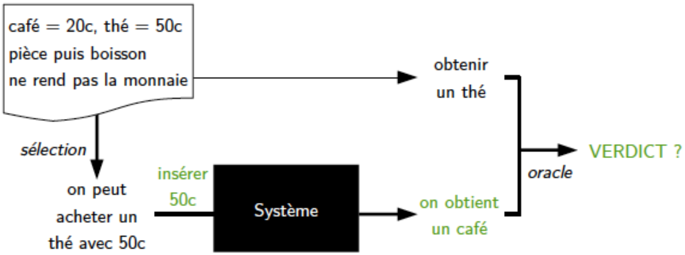
Test en boîte noire

- Sélection des tests à partir d'une spécification du système (formelle ou informelle), sans connaissance de l'implantation.
  - Test « fonctionnel »
- Possibilité de construire les tests pendant la conception, avant le codage



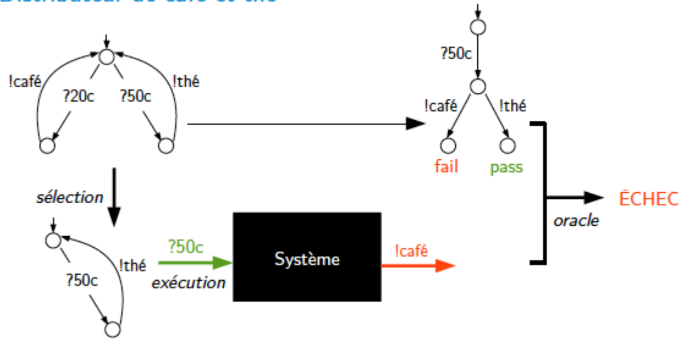
Test en boîte noire

Ex : Distributeur de café et thé



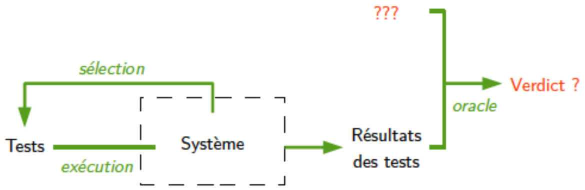
Test en boîte noire

Ex : Distributeur de café et thé



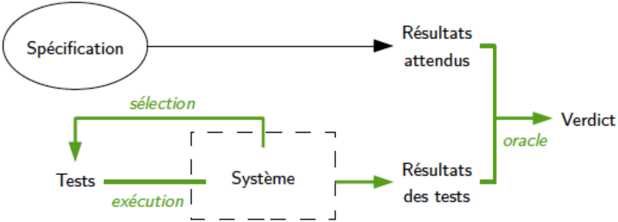
Test en boîte blanche

- Sélection des tests à partir de l'analyse du code source du système
  - Test « structurel »
- Construction des tests uniquement pour du code déjà écrit !

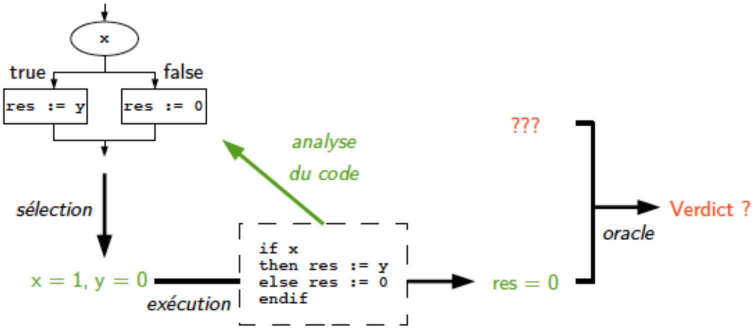


Test en boîte blanche

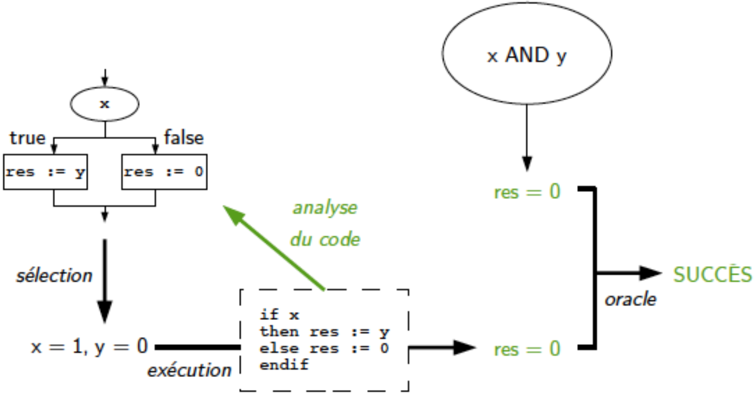
- Sélection des tests à partir de l'analyse du code source du système
  - Test « structurel »
- Construction des tests uniquement pour du code déjà écrit !



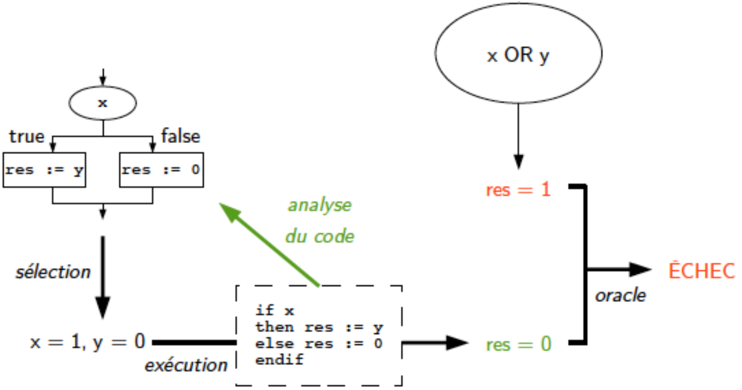
Test en boîte blanche



Test en boîte blanche



Test en boîte blanche



Test en boîte blanche

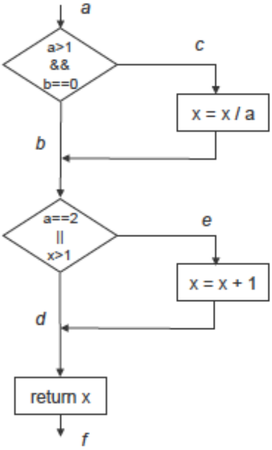
- Comment sélectionner des données de test qui détectent le plus d'erreurs ?
- Réponse : Couvrir les
  - Instructions du programme — simple
  - Changement de contrôle du programme
  - Décisions du programme
  - Comportements du programme — impossible

  - Flot de données (définition -> utilisation de variable)

Test en boîte blanche

```
int foo (int a, int b, int x) {
    if ((a>1) && (b==0))
        x = x div a;
    if ((a==2) || (x>1))
            x = x + 1;
    return x;
}
```

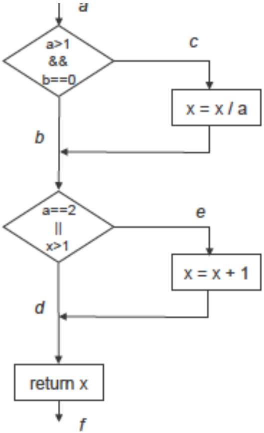
C code



Flow control graph

Test en boîte blanche

- Couverture des instructions
  - Exemple : acef
  - Données de test : a=2, b=0, x=3
  - Question : Et si le && était un || ?
  - Question : Et le chemin abdf ?
  - Très peu exhaustive, peu utilisé !

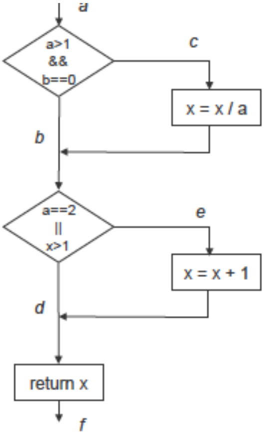


Test en boîte blanche

- Couverture des décisions

= chaque décision est vraie ou fausse au moins une fois

  - Exemple : {acef,abdf} ou {acdf,abef}
  - Données de test :
    - a=2, b=0, x=3
    - a=2, b=1, x=1
  - Question : Et si x > 1 était incorrecte ?

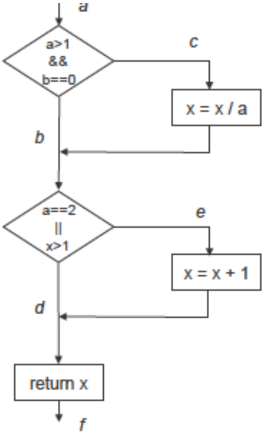


Test en boîte blanche

- Couverture de plusieurs décisions

= toutes les combinaison de valeurs pour les décisions sont testées au moins une fois

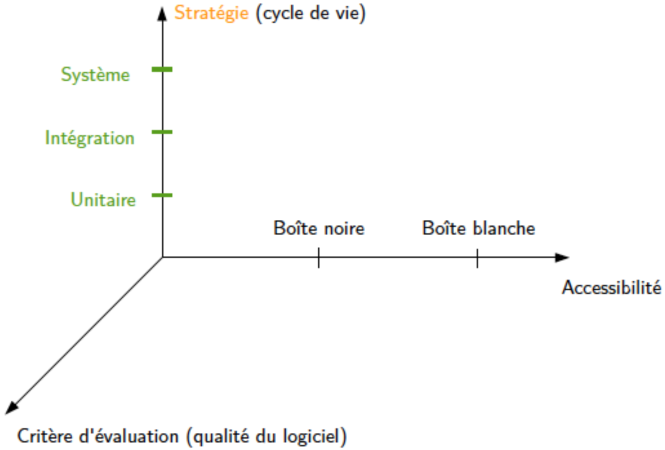
  - Exemple :
    1. a>1,b=0
    2. a=2,x>1
    3. a>1,b<>0
    4. a=2,x<=1
    5. a<=1,b=0
    6. a<>2,x>1
    7. a<=1,b<>0
    8. a<>2,x<=1
  - Données de test :
    - a=2, b=0, x=4 couvre 1 & 2
    - a=2, b=1, x=1 couvre 3 & 4
    - a=1, b=0, x=2 couvre 5 & 6
    - a=1, b=1, x=1 couvre 7 & 8
  - Toutes les exécutions sont couvertes ?



Boîte noire vs. boîte blanche

- Complémentarité : détection de fautes différentes
  - Boîte noire : détecte les oublis ou les erreurs par rapport à la spécification
  - Boîte blanche : détecte les erreurs de programmation

Types de test



A hierarchy of tests

Disclaimers:

- there are other hierarchies/taxonomies, on different angles
- terminology is not clear cut (as it often happens in SWE)
- the granularity trend—from small to big—however matters and is agreed upon

Test hierarchy

- acceptance

  Does the whole system work?
  (AKA: system tests)

- integration

  Does our code work against (other) code (we can't change)?

- unit

  Do our code units (i.e., classes, objects, modules, etc.) do the right thing and are convenient to work with?

Acceptance test

Does the whole system work?

Acceptance tests represent features that the system should have. Both their lack and their misbehaviour imply that the system is not working as it should. Intuition:

- 1 feature → 1+ acceptance test(s)
- 1 user story → 1+ acceptance test(s) (when using user stories)

Exercise (name 2+ acceptance tests for this "user login" story)

After creating a user, the system will know that you are that user when you login with that user's id and password; if you are not authenticated, or if you supply a bad id/password pair, or other error cases, the login page is displayed. If a CMS folder is marked as requiring authentication, access to any page under that folder will result in an authentication check. http://c2.com/cgi/wiki?AcceptanceTestExamples

Preview: we will use acceptance tests to guide feature development

Unit test

> *Do our code units do the right thing and are convenient to work with?*

Before implementing any unit of our software, we have (to have) an idea of what the code should do. Unit tests show convincing evidence that—in a limited number of cases—it is actually the case. [1]

Example (some unit tests for a `List` module)

1. remember: tests reveal bugs, but don't prove their absence!

Unit test

> *Do our code units do the right thing and are convenient to work with?*

Before implementing any unit of our software, we have (to have) an idea of what the code should do. Unit tests show convincing evidence that—in a limited number of cases—it is actually the case. [1]

Example (some unit tests for a `List` module)

- calling `List.length` on an empty list returns 0
- calling `List.length` on a singleton list returns 1
- calling `List.last` after `List.append` returns the added element
- calling `List.head` on an empty list throws an exception
- calling `List.length` on the concatenation of two lists returns the sum of the respective `List.length`s
- ...

1. remember: tests reveal bugs, but don't prove their absence!

Integration test

Does our code work against (other) code (we can't change)?

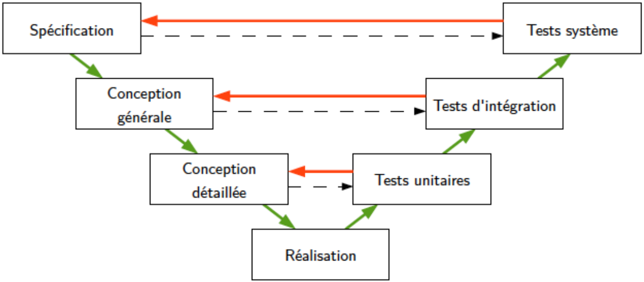
"Code we can't change" =

- 3rd party libraries/framework
  - be them proprietary or Free/Open Source Software
- code developed by other teams that we don't "own"
  - (strict code ownership is bad, though)
- code that we do not want/cannot modify in the current phase of development, for whatever reason

Example

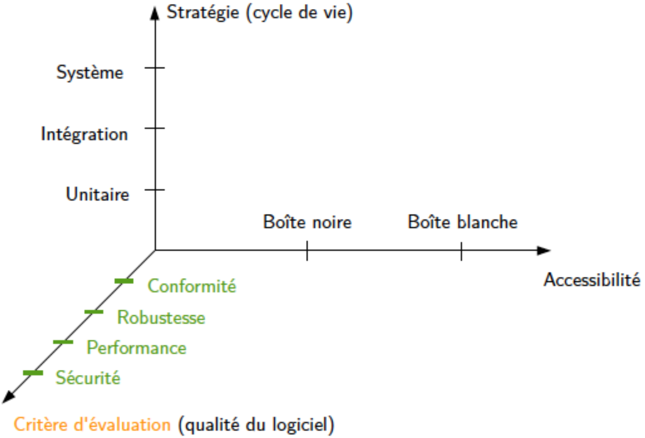
- our `BankClient` should not call the `getBalance` method on `BankingService` before calling `login` and having verified that it didn't throw an exception
- `xmlInitParser` should be called before any other parsing function of `libxml2`
- the DocBook markup returned by `CMSEditor.save` should be parsable by `PDFPublisher`'s constructor

Phases de production d'un logiciel



- Test unitaire = test des (petites) parties du code, séparément.
- Test d'intégration = test de la composition de modules.
- Test du système = de la conformité du produit fini par rapport au cahier des charges, effectué en boîte noire.

Types de test



Test de conformité

- But : Assurer que le système présente les fonctionnalités attendues par l'utilisateur
- Méthode : Sélection des tests à partir de la spécification, de façon à contrôler que toutes les fonctionnalités spécifiées sont implantées selon leurs spécifications
- *Ex : Service de paiement en ligne*
  - Scénarios avec transaction acceptée/refusée, couverture des différents cas et cas d'erreur prévus

Test de robustesse

- But : Assurer que le système supporte les utilisations imprévues
- Méthode : Sélection des tests en dehors des comportements spécifiés (entrées hors domaine, utilisation incorrecte de l'interface, environnement dégradé...)
- *Ex : Service de paiement en ligne*
  - Login dépassant la taille du buffer
  - Coupure réseau pendant la transaction

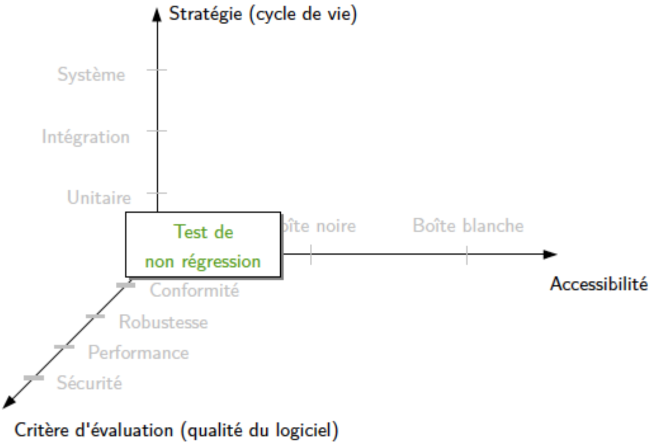
Test de sécurité

- But : Assurer que le système ne possède pas de vulnérabilités permettant une attaque de l'extérieur
- Méthode : Simulation d'attaques pour découvrir les faiblesses du système qui permettraient de porter atteinte à son intégrité
- *Ex : Service de paiement en ligne*
  - Essayer d'utiliser les données d'un autre utilisateur
  - Faire passer la transaction pour terminée sans avoir payé

Test de performance

- But : Assurer que le système garde des temps de réponse satisfaisants à différents niveaux de charge
- Méthode : Simulation à différents niveaux de charge d'utilisateurs pour mesurer les temps de réponse du système, l'utilisation des ressources...
- *Ex : Service de paiement en ligne*
  - Lancer plusieurs centaines puis milliers de transactions en même temps

Types de test



Test de non régression

- But : Assurer que les corrections et les évolutions du code n'ont pas introduit de nouveaux défauts
- Méthode : À chaque ajout ou modification de fonctionnalité, rejouer les tests pour cette fonctionnalité, puis pour celles qui en dépendent, puis les tests des niveaux supérieurs
  - Lourd mais indispensable
  - Automatisable en grande partie

Automatisation des tests

- Outils :
  - Générateur de test : aléatoire ou guidé par propriétés (BN) ou par des critères de couverture (BB).
    - Microsoft : DART, SAGE
    - INRIA : TGV
  - Analyseur de couverture : calcule le pourcentage de code couvert durant le test.
    - Coverlipse, gcov
  - "Record & playback" (Exécutif de test) : enregistre les actions de l'utilisateur pour pouvoir les rejouer à la demande ; utile pour le test des IHM et le test de régression.

  - Gestionnaire de test : maintient des suites de test, leurs résultat et produit des rapports.
    - Xunit (avec X=C, Java, Python)

Outline

Check

Check[2] is one of the most well-known unit testing framework for the C programming language.

Features overview

- C library
- xUnit style
- fixtures
- address space isolation
- autotools integration
- ~~mocking~~

References

- API reference: https://libcheck.github.io/check/doc/doxygen/html/check_8h.html
- source code (LGPL): https://github.com/libcheck/check/

2. https://libcheck.github.io/check/

Code organization for testing

Logical organization

- structure the code to be tested as a library...
- ...with a well-defined API
- corollary: almost empty `main()`, that just calls the main API entry point

Unit testing encourages to think at your API early in the project life-cycle, as your unit tests become your first client code.

Physical organization Up to you, but typically:

- `src/` (top-level dir): library code + `main()`
- `tests/` (top-level dir): Check tests
  - **#include** "../src/mylib.h" or equivalent

SUT — money.h

```c
#ifndef MONEY_H
#define MONEY_H

typedef struct Money Money;

Money *money_create(int amount, char *currency);
void money_free(Money * m);

int money_amount(Money * m);
char *money_currency(Money * m);

#endif /* MONEY_H */
```

SUT — money.c I

```
#include <stdlib.h>
#include "money.h"


struct Money {
        int amount;
        char *currency;
};

void money_free(Money *m) {
        free(m);
        return;
}
```

SUT — money.c II

```
Money *money_create(int amount, char *currency) {
        Money *m;

        if (amount < 0)
                return NULL;

        m = malloc(sizeof(Money));
        if (m == NULL)
                return NULL;

        m->amount = amount;
        m->currency = currency;

        return m;
}
```

SUT — money.c III

```
int money_amount(Money *m) {
        return m->amount;
}

char *money_currency(Money *m) {
        return m->currency;
}
```

Unit test skeleton

- the smallest units of executable tests in Check are unit tests
- unit tests are defined in regular .c files, using suitable preprocessor macros

```
#include <check.h>

START_TEST (test_name)
{
    /* unit test code */
}
END_TEST
```

Exercise

find the macro definitions of START_TEST and END_TEST and describe what they do

Hello, world

```
#include <check.h>  // testing framework
#include "../src/money.h"  // SUT

START_TEST(test_money_create) {
    Money *m;  // setup

    m = money_create(5, "USD");  // exercise SUT

    // test oracle
    ck_assert_int_eq(money_amount(m), 5);
    ck_assert_str_eq(money_currency(m), "USD");

    money_free(m);  // clean up
}
END_TEST
```

Assertion API

For basic data types, pre-defined "typed" assertions are available and can be used as simple and readable test oracles:

ck_assert_int_eq
: asserts that two signed integers values are equal; display a suitable error message upon failure

ck_assert_int_{ne,lt,le,gt,ge}
: like ck_assert_int_eq, but using different comparison operators

ck_assert_uint_*
: like ck_assert_int_*, but for unsigned integers

Assertion API

For basic data types, pre-defined “typed” assertions are available and can be used as simple and readable test oracles:

ck_assert_int_eq
: asserts that two signed integers values are equal; display a suitable error message upon failure

ck_assert_int_{ne,lt,le,gt,ge}
: like ck_assert_int_eq, but using different comparison operators

ck_assert_uint_*
: like ck_assert_int_*, but for unsigned integers

ck_assert_str_*
: like ck_assert_int_*, but for `char *` string values, using `strcmp()` for comparisons

Assertion API

For basic data types, pre-defined “typed” assertions are available and can be used as simple and readable test oracles:

ck_assert_int_eq
: asserts that two signed integers values are equal; display a suitable error message upon failure

ck_assert_int_{ne,lt,le,gt,ge}
: like ck_assert_int_eq, but using different comparison operators

ck_assert_uint_*
: like ck_assert_int_*, but for unsigned integers

ck_assert_str_*
: like ck_assert_int_*, but for `char *` string values, using `strcmp()` for comparisons

ck_assert_ptr_{eq,ne}
: like ck_assert_int_*, but for `void *` pointers

Assertion API (cont.)

For other data types, you can cook your own test oracles on top of more basic assertion primitives:

- **ck_assert** make test fail if supplied condition evaluates to false
- **ck_assert_msg** `ck_assert` + displays user provided message

Assertion API (cont.)

For other data types, you can cook your own test oracles on top of more basic assertion primitives:

ck_assert make test fail if supplied condition evaluates to false

ck_assert_msg `ck_assert` + displays user provided message

ck_abort make test fail unconditionally

ck_abort_msg ditto, with user supplied message

Assertion API — examples

```
ck_assert_str_eq(money_currency(m), "USD");
```

is equivalent to the following alternative formulations

```
ck_assert(strcmp(money_currency(m), "USD") == 0);

ck_assert_msg(strcmp(money_currency(m), "USD") == 0,
    "Was expecting a currency of USD, but found %s",
    money_currency(m));

if (strcmp(money_currency(m), "USD") != 0)
  ck_abort_msg("Currency not set correctly on creation");
```

Assertion API — examples (cont.)

```
ck_assert(money_amount(m) == 5);
```

is equivalent to:

```
ck_assert_msg(money_amount(m) == 5, NULL);
```

If money_amount(m) != 5 it will automatically synthesize the message:

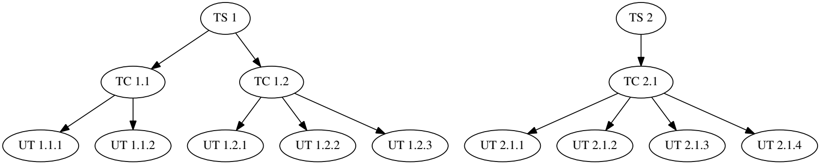
```
"Assertion 'money_amount(m) == 5' failed"
```

Test suites

In Check terminology:

- unit tests are grouped into test cases
- test cases are grouped into test suites

Test suites are what you can ask a test runner to run for you, recursively, down to individual unit tests.



Example:

```
START_TEST(test_money_create) {
    // as before
}
END_TEST
// other unit tests here
```

Test suites (cont.)

```
Suite *money_suite(void) {
    TCase *tc_core;
    Suite *s;

    tc_core = tcase_create("Core");
    tcase_add_test(tc_core, test_money_create);
    // tcase_add_test(tc_core, test_foo);
    // ...

    s = suite_create("Money");
    suite_add_tcase(s, tc_core);
    // suite_add_tcase(s, tc_bar);
    // ...

    return s;
}
```

Suite runner

In Check terminology, a suite runner is responsible for recursively running all unit tests reachable from a (set of) suite(s).

```
// create a suite runner, including a single suite
SRunner *srunner_create(Suite *);

// add a suite to a suite runner
void srunner_add_suite(SRunner *sr, Suite *s)

// destroy a suite runner
void srunner_free(SRunner *);
```

Suite runner (cont.)

```
// run all unit tests reachable from all (added) suites
void srunner_run_all(SRunner *sr,
                     enum print_output print_mode);

// run the unit tests corresponding to suite sname and
// test case tcname. Either can be NULL to mean "all"
void srunner_run(SRunner *sr,
                 const char *sname,
                 const char *tcname,
                 enum print_output print_mode);
```

`print_output` controls the on-screen output of the test runner:

- **CK_SILENT** no output
- **CK_MINIMAL** summary output
- **CK_NORMAL** summary + list of failed tests
- **CK_VERBOSE** summary + list of all tests
- **CK_ENV** infer from env CK_VERBOSITY (default: CK_NORMAL)

Suite runner (cont.)

After tests have been run, test result information can be extracted from the suite runner.

srunner_ntests_failed number of failed tests

srunner_ntests_run number of tests ran

srunner_failures

srunner_results

access to detailed, per test result information (see API reference)

Suite runner (cont.)

```c
int main(void) {
    int failures;
    Suite *s;
    SRunner *sr;

    s = money_suite();
    sr = srunner_create(s);

    srunner_run_all(sr, CK_VERBOSE);
    failures = srunner_ntests_failed(sr);
    srunner_free(sr);

    return (failures == 0) ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

Suite runner (cont.)

```
./check_money
Running suite(s): Money
100%: Checks: 1, Failures: 0, Errors: 0
check_money.c:13:P:Core:test_money_create:0: Passed
```

Demo

More tests

```
START_TEST(test_money_create_neg) {
    Money *m = money_create(-1, "USD");

    ck_assert_msg(m == NULL,
                  "NULL should be returned on attempt "
                  "to create with a negative amount");
}
END_TEST

START_TEST(test_money_create_zero) {
    Money *m = money_create(0, "USD");

    if (money_amount(m) != 0)
        ck_abort_msg("0 is a valid amount of money");
}
END_TEST
```

More tests (cont.)

```c
Suite *money_suite(void) {
    TCase *tc_core, *tc_limits;
    Suite *s;

    tc_core = tcase_create("Core");
    tcase_add_test(tc_core, test_money_create);

    tc_limits = tcase_create("Limits");
    tcase_add_test(tc_limits, test_money_create_neg);
    tcase_add_test(tc_limits, test_money_create_zero);

    s = suite_create("Money");
    suite_add_tcase(s, tc_core);
    suite_add_tcase(s, tc_limits);

    return s;
}
```

More tests (cont.)

```
./check_money
Running suite(s): Money
100%: Checks: 1, Failures: 0, Errors: 0
check_money.c:13:P:Core:test_money_create:0: Passed
check_money.c:24:P:Limits:test_money_create_neg:0: Passed
check_money.c:28:P:Limits:test_money_create_zero:0: Passed
```

Compiling with Check

src/Makefile

```
CC = gcc
CFLAGS = -Wall
LDFLAGS = $(CFLAGS)

all: main
main: main.o money.o money.h
money.o: money.h

clean:
	rm -f main *.o
```

Exercise

Where are the actual compilation commands defined?

Compiling with Check (cont.)

tests/Makefile

```
CC = gcc
CFLAGS = -Wall
LDFLAGS = $(CFLAGS)
LDLIBS = -Wall $(shell pkg-config --libs check)

all: check_money
check_money: check_money.o ../src/money.o

test: check_money
	./$<

clean:
	rm -f check_money *.o
```

Compiling with Check (cont.)

```
$ cd src/
src$ make
gcc -Wall -c -o main.o main.c
gcc -Wall -c -o money.o money.c
gcc -Wall main.o money.o money.h -o main

src$ cd ../tests
tests$ make
gcc -Wall -c -o check_money.o check_money.c
gcc -Wall check_money.o ../src/money.o -lcheck_pic -pthread \
    -lrt -lm -lsubunit -o check_money

tests$ make test
./check_money
Running suite(s): Money
100%: Checks: 1, Failures: 0, Errors: 0
check_money.c:13:P:Core:test_money_create:0: Passed

tests$
```

Unit testing C — memory safety

```
#include <check.h>

START_TEST(null_deref) {
    int *p = NULL;
    ck_assert_int_eq(p[1], 42);
}
END_TEST

int main(void) {
        TCase *tc;  Suite *s;  SRunner *sr;

        tc = tcase_create("segfault");
        tcase_add_test(tc, null_deref);
        s = suite_create("memsafety");
        suite_add_tcase(s, tc);
        sr = srunner_create(s);
        srunner_run_all(sr, CK_VERBOSE);
        return (srunner_ntests_failed(sr) == 0 ? 0 : 1);
}       // what will this program do?
```

Unit testing C — memory safety (cont.)

```
$ ./test-segfault
Running suite(s): memsafety
0%: Checks: 1, Failures: 0, Errors: 1
test-segfault.c:3:E:segfault:null_deref:0:
  (after this point) Received signal 11 (Segmentation fault)

$ echo $?
$ 1
```

Unit testing C — memory safety (cont.)

```
$ ./test-segfault
Running suite(s): memsafety
0%: Checks: 1, Failures: 0, Errors: 1
test-segfault.c:3:E:segfault:null_deref:0:
  (after this point) Received signal 11 (Segmentation fault)

$ echo $?
$ 1
```

- the program did *not* crash
- Check reported a test failure and “detected” the segfault
- how come?

Address space separation

- unit testing C might be difficult in general because all tests run in the same address space
- if a test induces memory corruption, *other* tests will suffer the consequences too (including crashes)

Address space separation

- unit testing C might be difficult in general because all tests run in the same address space
- if a test induces memory corruption, *other* tests will suffer the consequences too (including crashes)
- as a way around, several C test frameworks run tests in separate processes, `fork()`-ing before each test

Address space separation

- unit testing C might be difficult in general because all tests run in the same address space
- if a test induces memory corruption, *other* tests will suffer the consequences too (including crashes)
- as a way around, several C test frameworks run tests in separate processes, `fork()`-ing before each test
- by default Check runs each unit test in a separate process ("fork mode")
- "no fork mode" can be requested explicitly
  - define the environment variable CK_FORK=no
  - ```
    void srunner_set_fork_status(SRunner *,
                                 enum fork_status);
    ```

    with `fork_status` = CK_FORK / CK_NOFORK
    ```

Check (no) fork mode — example

```
$ ./test-segfault
Running suite(s): memsafety
0%: Checks: 1, Failures: 0, Errors: 1
test-segfault.c:3:E:segfault:null_deref:0:
  (after this point) Received signal 11 (Segmentation fault)

$ echo $?
$ 1

$ CK_FORK=no ./test-segfault
Running suite(s): memsafety
[1]    5750 segmentation fault  CK_FORK=no ./test-segfault

$ echo $?
$ 139
```

- after disabling fork mode the program did crash :-(
- the suite runner would have been unable to run further tests in the suite

Test suite best practices

- often a group of tests should be run on the same initial state...
- ...but tests execution might alter that state

1. we want test isolation: each test should behave the same no matter the test execution order *(dynamic requirement)*
   - each test should initialize all of its required state (setup)
   - each test should clean up after itself (tear down)
2. we also wish to not duplicate test initialization across tests, as it violates the DRY principle *(static requirement)*

Note: Check's fork mode helps with (1), but not with (2).

We want a mechanism to factor out setup and tear down code across multiple tests.

Test fixtures

A test fixture, or test context, is a pair $\langle \mathit{setup}, \mathit{teardown} \rangle$ of functions to be executed before and after the test body.

- setup should initialize the entire state needed to evaluate the test (i.e., exercising SUT + test oracle)
- teardown should clean up the entire the state affected by test execution (i.e., setup + exercising SUT)

The code that implements text fixtures is independent from the actual test code.
Therefore, it can be shared across multiple tests.

Test fixtures in Check

In Check, test fixtures are associated with test cases.
They are hence shared among all unit tests of the same test case.

In terms of isolation, Check distinguishes two kinds of fixtures:

checked fixtures
: are run within the address space of unit tests (if fork mode is on), once for each unit test

unchecked fixtures
: are run in the address space of the test program, once for each test case

Warning: memory corruption introduced by unchecked fixtures might crash the entire test suites.

Test fixtures in Check (cont.)

For a Check test case with 2 unit tests—`unit_test_1` and `*_2`—the execution order of test and fixture functions will be:

1. unchecked_setup();
2. fork();
3. checked_setup();
4. unit_test_1();
5. checked_teardown();
6. wait();
7. fork();
8. checked_setup();
9. unit_test_2();
10. checked_teardown();
11. wait();
12. unchecked_teardown();

Test fixtures in Check — example

```
Money *five_dollars;

void setup(void) {
    five_dollars = money_create(5, "USD");
}

void teardown(void) {
    money_free(five_dollars);
}

START_TEST(test_money_create_amount) {
    ck_assert_int_eq(money_amount(five_dollars), 5);
}
END_TEST

START_TEST(test_money_create_currency) {
    ck_assert_str_eq(money_currency(five_dollars), "USD");
}
END_TEST
```

Test fixtures in Check — example (cont.)

```
Suite * money_suite(void) {
    Suite *s;
    TCase *tc_core;
    TCase *tc_limits;

    s = suite_create("Money");

    tc_core = tcase_create("Core");
    tcase_add_checked_fixture(tc_core, setup, teardown);
    tcase_add_test(tc_core, test_money_create_amount);
    tcase_add_test(tc_core, test_money_create_currency);
    suite_add_tcase(s, tc_core);

    tc_limits = tcase_create("Limits");
    tcase_add_test(tc_limits, test_money_create_neg);
    tcase_add_test(tc_limits, test_money_create_zero);
    suite_add_tcase(s, tc_limits);

    return s;
}
```

Test fixtures in Check — example (cont.)

```
$ ./check_money
Running suite(s): Money
100%: Checks: 4, Failures: 0, Errors: 0
check_money.c:17:P:Core:test_money_create_amount:0: Passed
check_money.c:22:P:Core:test_money_create_currency:0: Passed
check_money.c:31:P:Limits:test_money_create_neg:0: Passed
check_money.c:35:P:Limits:test_money_create_zero:0: Passed

$
```

Selectively running tests

It might be important to run only a few tests

- e.g., when debugging a specific test failure
- e.g., to run fast vs slow tests in different phases of your development process

```
$ ./check_money
Running suite(s): Money
100%: Checks: 4, Failures: 0, Errors: 0
check_money.c:17:P:Core:test_money_create_amount:0: Passed
check_money.c:22:P:Core:test_money_create_currency:0: Passed
check_money.c:31:P:Limits:test_money_create_neg:0: Passed
check_money.c:35:P:Limits:test_money_create_zero:0: Passed

$ CK_RUN_CASE=Limits ./check_money
Running suite(s): Money
100%: Checks: 2, Failures: 0, Errors: 0
check_money.c:31:P:Limits:test_money_create_neg:0: Passed
check_money.c:35:P:Limits:test_money_create_zero:0: Passed
```

A similar CK_RUN_SUITE environment variable also exists.

Check boilerplate

```c
#include <stdlib.h>
#include <stdint.h>
#include <check.h>  // testing framework
#include "../src/money.h"  // SUT

START_TEST(test_money_create) {
    Money *m;  // setup

    m = money_create(5, "USD");  // exercise SUT

    // test oracle
    ck_assert_int_eq(money_amount(m), 5);
    ck_assert_str_eq(money_currency(m), "USD");

    money_free(m);  // clean up
}
END_TEST

Suite *money_suite(void) {
    TCase *tc_core;
    Suite *s;

    tc_core = tcase_create("Core");
    tcase_add_test(tc_core, test_money_create);
    // tcase_add_test(tc_core, test_foo);
    // ...

    s = suite_create("Money");
    suite_add_tcase(s, tc_core);
    // suite_add_tcase(s, tc_bar);
    // ...

    return s;
}
```

How do you like it?

Check boilerplate

```
#include <stdlib.h>
#include <stdint.h>
#include <check.h>  // testing framework
#include "../src/money.h"  // SUT

START_TEST(test_money_create) {
    Money *m;  // setup

    m = money_create(5, "USD");  // exercise SUT

    // test oracle
    ck_assert_int_eq(money_amount(m), 5);
    ck_assert_str_eq(money_currency(m), "USD");

    money_free(m);  // clean up
}
END_TEST

Suite *money_suite(void) {
    TCase *tc_core;
    Suite *s;

    tc_core = tcase_create("Core");
    tcase_add_test(tc_core, test_money_create);
    // tcase_add_test(tc_core, test_foo);
    // ...

    s = suite_create("Money");
    suite_add_tcase(s, tc_core);
    // suite_add_tcase(s, tc_bar);
    // ...

    return s;
}
```

How do you like it?

- quite a bit of boilerplate
- for relatively few lines of actual test code

checkmk

checkmk[3] can be used to reduce the amount of Check boilerplate to write and focus on the actual test code.

checkmk is used as a custom C preprocessor that expand specific #-directives to suitable calls to the Check API.

Some checkmk directives:

#suite	define a suite
#tcase	define a test case
#test	define a unit test
#main-pre	`main` preamble (e.g., to declare fixtures)

See the `checkmk(1)` manpage for a full list.

3. `http://micah.cowan.name/projects/checkmk/`

checkmk – example

```
#include <stdlib.h>
#include "../src/money.h"

#suite Money

#tcase Core

#test test_money_create_amount
    ck_assert_int_eq(money_amount(five_dollars), 5);

#test test_money_create_currency
    ck_assert_str_eq(money_currency(five_dollars), "USD")
```

checkmk – example (cont.)

```
#tcase Limits

#test test_money_create_neg
    Money *m = money_create(-1, "USD");
    ck_assert_msg(m == NULL,
                  "NULL should be returned on attempt "
                  "to create with a negative amount");

#test test_money_create_zero
    Money *m = money_create(0, "USD");
    if (money_amount(m) != 0)
        ck_abort_msg("0 is a valid amount of money");
```

checkmk – example (cont.)

```
Money *five_dollars;

void setup(void) {
    five_dollars = money_create(5, "USD");
}

void teardown(void) {
    money_free(five_dollars);
}

#main-pre
    tcase_add_checked_fixture(tc1_1, setup, teardown);
```

checkmk – example (cont.)

```
$ checkmk check_money.check > check_money.c

$ head check_money.c
/*
 * DO NOT EDIT THIS FILE. Generated by checkmk.
 * Edit the original source file "check_money.check" instead.
 */

#include <check.h>

#line 1 "check_money.check"
#include <stdlib.h>
#include "../src/money.h"
```

checkmk – example (cont.)

```
CC = gcc
CFLAGS = -Wall
LDFLAGS = $(CFLAGS)
LDLIBS = -Wall $(shell pkg-config --libs check)

all: check_money
check_money: check_money.o ../src/money.o

check_money.c: check_money.check
	checkmk $< > $@

test: check_money
	./$<

clean:
	rm -f check_money *.o
```

checkmk – example (cont.)

```
$ make
gcc -Wall   -c -o check_money.o check_money.c
gcc -Wall  check_money.o ../src/money.o  -Wall -lcheck_pic \
  -pthread -lrt -lm -lsubunit -o check_money

$ CK_VERBOSITY=verbose ./check_money
Running suite(s): Money
100%: Checks: 4, Failures: 0, Errors: 0
check_money.check:19:P:Core:test_money_create_amount:0: Passed
check_money.check:22:P:Core:test_money_create_currency:0: Passed
check_money.check:30:P:Limits:test_money_create_neg:0: Passed
check_money.check:35:P:Limits:test_money_create_zero:0: Passed
```