Vérification & Validation

Philippe Collet
2007-2008

- Principes
- Techniques de tests
- Tests en boîte noire
- L'outil check

Principes de V&V

- Deux aspects de la notion de qualité :
  - Conformité avec la définition : VALIDATION
    - Réponse à la question : faisons-nous le bon produit ?
    - Contrôle en cours de réalisation, le plus souvent avec le client
    - **Défauts** par rapport aux besoins que le produit doit satisfaire
  - Correction d'une phase ou de l'ensemble : VERIFICATION
    - Réponse à la question : faisons-nous le produit correctement ?
    - Tests
    - **Erreurs** par rapport aux définitions précises établies lors des phases antérieures de développement

V&V et cycle de vie

- Les spécifications fonctionnelles définissent **les intentions**
  - Elles sont créées lors de la phase d'analyse des besoins
- La vérification du produit consiste à vérifier la conformité vis-à-vis de ces spécifications fonctionnelles
  - Revues, inspections, analyses, tests fonctionnels et structurels en boîte blanche
- La validation du produit consiste à vérifier par le donneur d'ordre la conformité vis-à-vis des besoins
  - Le plus souvent, tests fonctionnels en boîte noire
  - Théoriquement, la validation devrait être plutôt faite par les utilisateurs, sans tenir compte du cahier des charges
  - En pratique, la validation s'appuie sur le cahier des charges pour créer des tests d'acceptation...

Techniques statiques

- Portent sur des documents (plutôt des programmes), sans **exécuter** le logiciel
- Avantages
  - contrôle systématique valable pour toute exécution, applicables à tout document
- Inconvénients
  - Ne portent pas forcément sur le code réel
  - Ne sont pas en situation réelle (interaction, environnement)
  - Vérifications sommaires, sauf pour les preuves
  - Ces preuves nécessient des spécifications formelles et **complètes**, donc difficiles

Techniques dynamiques

- Nécessitent une exécution du logiciel, une parmi des multitudes d'autres possibles
- Avantages
  - Vérification avec des conditions proches de la réalité
  - Plus à la portée du commun des programmeurs
- Inconvénients
  - Il faut provoquer des expériences, donc écrire du code et construire des données d'essais
  - Un test qui réussit ne démontre pas qu'il n'y a pas d'erreurs

☞ *Les techniques statiques et dynamiques sont donc complémentaires*

Tests

" Testing is the process of executing a program with the intent of finding errors."
Glen Myers

Tests : définition…

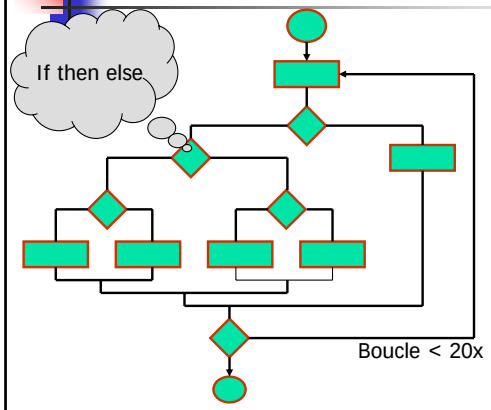
- Une expérience d'exécution, pour mettre en évidence un défaut ou une erreur
  - Diagnostic : quel est le problème
  - Besoin d'un **oracle**, qui indique si le résultat de l'expérience est conforme aux intentions
  - Localisation (si possible) : où est la cause du problème ?

☞ *Les tests doivent mettre en évidence des erreurs !*

☞ *On ne doit pas vouloir démontrer qu'un programme marche à l'aide de tests !*

- Souvent négligé car :
  - les chefs de projet n'investissent pas pour un résultat négatif
  - les développeurs ne considèrent pas les tests comme un processus destructeur

Tests exhaustifs ?



Il y a 5^{20} chemins possibles
En exécutant 1 test par milliseconde, cela prendrait **3024** ans pour tester ce programme !

Constituants d'un test

- Nom, objectif, commentaires, auteur
- Données : jeu de test
- Du code qui appelle des routines : cas de test
- Des oracles (vérifications de propriétés)
- Des traces, des résultats observables
- Un stockage de résultats : étalon
- Un compte-rendu, une synthèse...

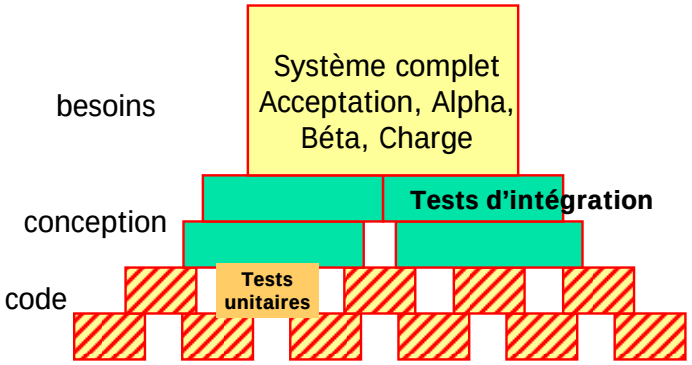
- Coût moyen : autant que le programme

Test vs. Essai vs. Débogage

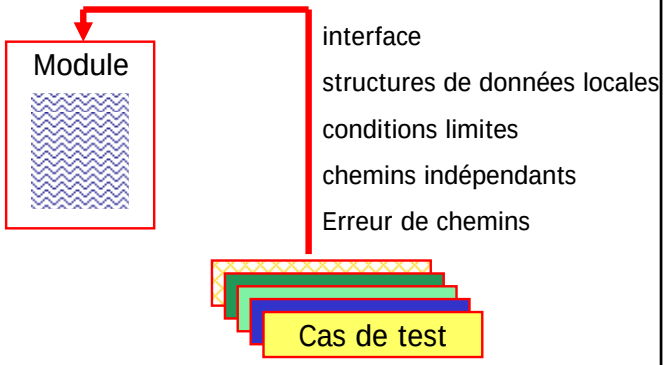
- On converse les données de test
  - Le coût du test est amorti
  - Car un test doit être **reproductible**
- Le test est différent d'un essai de mise au point

- Le débogage est une enquête
  - Difficilement reproductible
  - Qui cherche à expliquer un problème

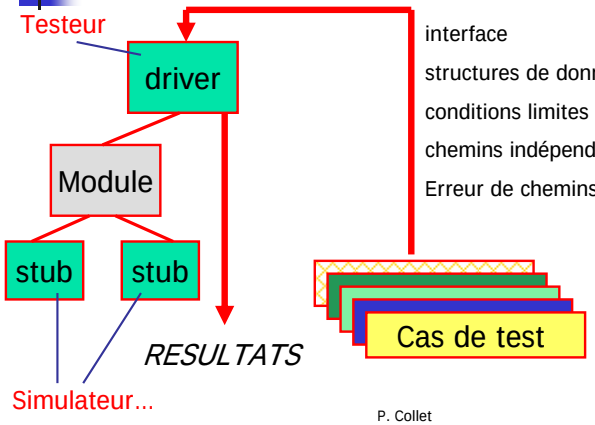
Les stratégies de test



Test unitaire



Environnement du test unitaire



interface
structures de données locales
conditions limites
chemins indépendants
Erreur de chemins

Tests d'intégration

Si tous les modules marchent bien séparément, pourquoi douter qu'ils ne marcheraient pas ensemble ?

Réunir les modules : **Interfacer**

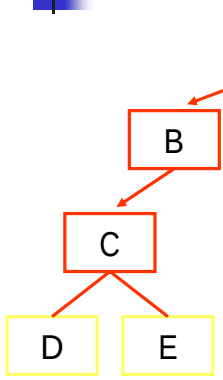
Big Bang

Stratégie de construction incrémentale

Intégration partielle

Test de non-régression

Intégration descendante



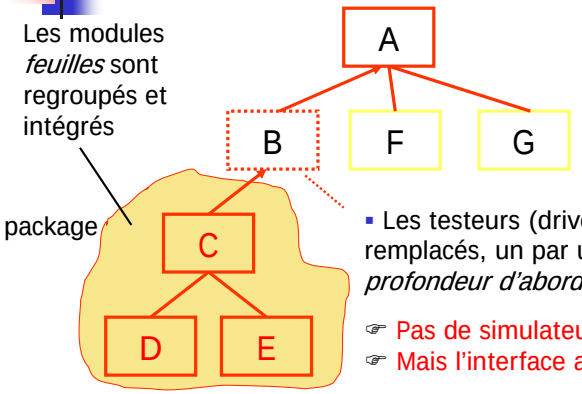
Les modules racines sont testés avec des simulateurs (stubs)

- Les simulateurs sont remplacés, un par un, *en profondeur d'abord*
- Lorsque de nouveaux modules sont intégrés, des sous-ensembles des tests sont ré-effectués

☞ Les modules complexes sont testés tardivement

Intégration ascendante

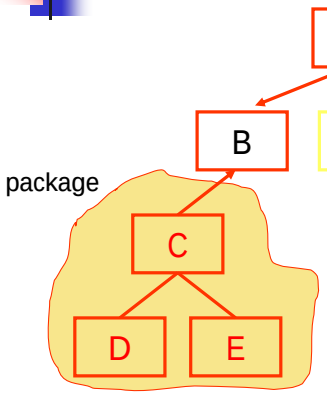
Les modules *feuilles* sont regroupés et intégrés



- Les testeurs (drivers) sont remplacés, un par un, *en profondeur d'abord*

☞ Pas de simulateur
☞ Mais l'interface apparaît tard

Intégration en *Sandwich*



Les modules racines sont testés avec des simulateurs (stubs)

Les modules *feuilles* sont regroupés et intégrés

Tests fonctionnels en boîte noire

- Principes
  - S'appuient sur des spécifications externes
  - Partitionnent les données à tester par **classes d'équivalence**
    - Une valeur attendue dans 1..10 donne [1..10], < 1 et > 10
  - Ajoutent des valeurs « pertinentes », liées à l'expérience du testeur
    - Tests aux bornes : sur les bornes pour l'acceptation, juste au delà des bornes pour des refus

Exemple : la recherche binaire

```
Table_binaire <elem -> comparable>
…
lower() : integer
…
chercher(key : elem) : integer
// si l'élement de clé key est dans la table, rend son
indice, sinon rend lower-1
…
```

- Classes d'équivalence ?
  - Données conformes aux prérequis (?)
  - Données non-conformes aux prérequis
  - Cas où l'élément recherché existe dans la table
  - Cas où l'élément recherché n'existe pas dans la table

Test : recherche binaire

- Deuxième regroupement plus pertinent
  - Table ordonnée, avec élément recherché présent
  - Table ordonnée, avec élément recherché absent
  - Table non ordonnée, avec élément recherché présent
  - Table non ordonnée, avec élément recherché absent
- Ajout de cas limites et *intuitifs*
  - Table à un seul élément
  - Table à un nombre impair d'éléments *(boite grise)*
  - Table à un nombre pair d'éléments *(boite grise)*
  - Table où l'élément recherché est le premier de la table
  - Table où l'élément recherché est le dernier de la table

Test : recherche binaire

- Classes d'équivalence
  - 1 seul élément, clé présente
  - 1 seul élément, clé absente
  - Nb pair d'éléments, clé absente
  - Nb pair d'éléments, clé en première position
  - ...
- Tests en boîte noire résultants
  - Nb pair d'éléments, clé ni en première, ni en dernière position
  - Nb impair d'éléments, clé absente
  - Nb impair d'éléments, clé en première position
  - ...

L'Outil Check

Check (check.sourceforge.net)

- Environnement de tests unitaires pour le langage C
  - Inspiré de JUnit (pour Java)
  - Plus ou moins intégré à l'approche *Extreme Programming*
- Trois avantages à l'approche incrémentale code+test
  - Comme les tests unitaires utilisent l'interface de l'unité à tester, ils amènent le développeur à réfléchir à l'utilisation de cette interface tôt dans l'implémentation
  - Ils permettent aux développeurs de détecter tôt des cas aberrants
  - **En fournissant un degré de correction documenté, ils permettent au développeur de modifier l'architecture du code en confiance**
- D'autres outils similaires : GNU AutoUnit, cUnit...

Principe de fonctionnement

- En Java, facile...
  - Un échec lève une exception et hop !
- En C... on risque plutôt un crash dans l'espace d'adressage !
- Check
  - utilise fork pour créer un nouvel espace d'adressage pour chaque test unitaire
  - utilise des files de messages pour retourner des comptes-rendus à l'environnement de tests
- Pour piloter le lancement des tests, check
  - Peut être appelée directement, mais surtout
  - Depuis un makefile, et encore mieux
  - Depuis autoconf/automake pour générer le tout...

Écrire un test

- Utiliser l'include **`#include <check.h>`**
- Ecrire le test entre les deux macros

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

- Exemple :

```
START_TEST (test_create)
{
 Money *m;
 m = money_create(5, "USD");
 fail_unless(money_amount(m) == 5, "Amount not correct on creation");
 fail_unless(strcmp(money_currency(m), "USD") == "Currency not set
   correctly on creation");
 money_free(m);
}
END_TEST
```

ORACLE

Alternatives pour les oracles

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

Aussi équivalent à

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

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

Est équivalent à

```
fail_unless(money_amount(m) == 5, "Assertion
            'money_amount (m) == 5' failed");
```

Liste variable d'arguments, style *printf* :

```
fail_unless(money_amount(m) == 5, "Amount was %d,
            instead of 5", money_amount(m));
```


Pilotage depuis automake/autoconf

Makefile.am

```
TESTS=check_money
noinst_PROGRAMS=check_money
check_money_SOURCES= money.h money.c check_money.c
check_money_INCLUDES= @CHECK_CFLAGS@
check_money_LIBS= @CHECK_LIBS@
```

configure.in (vue partielle)

```
AC_INIT()
AM_INIT_AUTOMAKE()
AC_PROG_CC
AM_PATH_CHECK()
AC_OUTPUT(Makefile)
```

autoconf
automake
configure

gentest

```
aclocal
autoconf
autoheader
automake --add-missing
./configure
```

Makefile

```
matrix> make -k check
```


Construction de l'exemple

- Money.h

```
typedef struct Money Money;
Money *money_create(int amount, char *currency);
int money_amount(Money *m);
char *money_currency(Money *m);
void money_free(Money *m);
```

- Money.c

```
#include <stdlib.h>
#include "money.h"
Money *money_create(int amount, char *currency) { return NULL; }
int money_amount(Money *m) { return 0; }
char *money_currency(Money *m) { return NULL; }
void money_free(Money *m) { return; }
```

Jeu de tests

- Création des cas de test
- Regroupement dans une *suite*
  - *Des test cases*
  - Pour chacun, des tests sont ajoutés
- Exécution depuis un *suite runner*
  - À partir d'une *suite*
  - Exécution en mode normal
  - Récupération du retour (qui sera interprété par make)

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

START_TEST (test_create)
…
END_TEST

Suite *money_suite(void) {
  Suite *s = suite_create("Money");
  TCase *tc_core = tcase_create("Core");
  suite_add_tcase (s, tc_core);
  tcase_add_test(tc_core, test_create);
  return s;
}

int main(void) {
  int nf;
  Suite *s = money_suite();
  SRunner *sr = srunner_create(s);
  srunner_run_all(sr, CK_NORMAL);
  nf = srunner_ntests_failed(sr);
  srunner_free(sr);
  return (nf == 0) ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

Paramétrage du *SRunner*

`void srunner_run_all(SRunner *sr, enum print_output print_mode);`

- Exécute tous les tests unitaires de tous les cas de test définis pour toutes les suites dans `sr`
- Collecte les résultats dans `sr`
- Affiche les résultats selon le `print_mode`
  - CK_SILENT : aucune sortie
  - CK_MINIMAL : résumé (number run, passed, failed, errors).
  - CK_NORMAL : résumé + un message par test échoué
  - CK_VERBOSE : résumé + un message par test (passé, échoué)
  - CK_ENV : récupère le mode depuis la variable d'environnement CK_VERBOSITY ("silent", "minimal", "normal, "verbose«)
- Pour les SRunners déjà exécutés, la fonction d'affichage séparé suivante peut être utilisée :

  `void srunner_print(SRunner *sr, enum print_output print_mode);`

Passons les tests

- En mode CK_NORMAL

```
matrix> make –k check
...
...
Running suite(s): Money
0%: Checks: 1, Failures: 1, Errors: 0
check_money.c:20:F:Core:test_create: Amount not set correctly
on creation
FAIL: check_money
==================
1 of 1 tests failed
==================
make[1]: *** [check-TESTS] Error 1
make[1]: Leaving directory `/home/collet/checkTestMoney'
make: *** [check-am] Error 2
make: Target `check' not remade because of errors.
```


Développons l'exemple

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

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

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

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

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

Plusieurs cas de test (1)

```
START_TEST (test_neg_create)
{
  Money *m = money_create(-1, "USD");
  fail_unless(m == NULL,
         "NULL should be returned on creation with a negative amount");
}
END_TEST

START_TEST (test_zero_create)
{
  Money *m = money_create(0, "USD");
  fail_unless(money_amount(m) == 0, "Zero is a valid amount of money");
}
END_TEST
```


Plusieurs cas de test (2)

```
Suite *money_suite (void) {
  Suite *s = suite_create("Money");
  TCase *tc_core = tcase_create("Core");
  TCase *tc_limits = tcase_create("Limits");
  suite_add_tcase(s, tc_core);
  suite_add_tcase(s, tc_limits);
  tcase_add_test(tc_core, test_create);
  tcase_add_test(tc_limits, test_neg_create);
  tcase_add_test(tc_limits, test_zero_create);
  return s;
}
```

Un nouveau TCase

Ajout des 2 tests

- Même *SRunner*, même *main*

Plusieurs cas de test

```
matrix> make -k check
...
Running suite(s): Money
66%: Checks: 3, Failures: 1, Errors: 0
check_money.c:29:F:Limits:test_neg_create:
NULL should be returned on attempt to
 create with a negative amount
FAIL: check_money
==================
1 of 1 tests failed
==================
```

- Résultat :
  - 1 test échoué
  - Il faut modifier le code de `money_create` pour implémenter le traitement d'un montant négatif

Test fixtures

- Plusieurs tests peuvent utiliser les mêmes données
  - Un code d'initialisation constant pour tous les tests concernés
  - **Test fixture** = code *setup* et/ou *teardown*
- *Checked fixtures*
  - Exécutés dans l'espace d'adressage du test unitaire
  - Avant chaque test unitaire dans un Tcase, la fonction *setup* est exécutée
  - Après chaque test unitaire dans un Tcase, la fonction *teardown* est exécutée
  - Si le code renvoie des signaux ou échoue, ce sera rattrapé et reporté par le *SRunner*
- *Unchecked fixtures*
  - Exécutés dans l'espace d'adressage du programme de test
  - Pas de signaux, pas de sortie, mais utilisation possible de fonctions *fail*
  - *Setup* et *teardown* sont resp. exécutées avant/après le cas de test

Exemples de *Test Fixture*

1. Définir les variables globales et les fonctions de signature void (void)

```
Money *five_dollars;
void setup(void) {
  five_dollars = money_create(5, "USD");
}
void teardown(void) {
  money_free(five_dollars);
}
```

2. Ajouter les fonctions au TCase avec ***tcase_add_checked_fixture***

```
tcase_add_checked_fixture(tc_core, setup, teardown);
```

3. *Écriture du test en conséquence (réécriture de notre 1er test)*

```
START_TEST (test_create) {
   fail_unless(money_amount(five_dollars) == 5,
               "Amount not set correctly on creation");
   fail_unless (strcmp(money_currency(five_dollars), "USD") == 0,
                "Currency not set correctly on creation");
} END_TEST
```


Option pour ne pas utiliser *fork*

- Comme check utilise *fork,* il n'est pas facile d'utiliser des outils de débogage
- L'utilisation de *fork* est désactivable par :
  - La variable d'environnement CK_FORK définie à « no »
  - La modification d'un SRunner grâce à la fonction
    - `void srunner_set_fork_status(SRunner *sr, enum fork_status fstat);`
    - L'enum fork_status définit CK_FORK et CK_NOFORK
  - Cette dernière définition prime sur la variable d'environnement
- Attention, en mode CK_NOFORK, des effets de bord (dans des *unchecked fixtures*, par exemple) peuvent impacter les tests suivants

Un *SRunner* et plusieurs *Suites*

- Une *suite* est censée tester un module du programme
- On peut gérer un programme de test par suite/module ou avoir un seul SRunner :

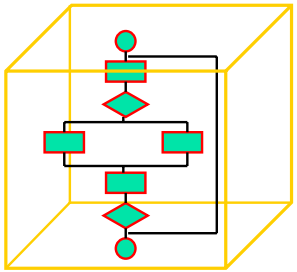
```
Suite *make_sub_suite(void);
Suite *make_sub2_suite(void);
Suite *make_master_suite(void);
Suite *make_list_suite(void);
Suite *make_msg_suite(void);
Suite *make_log_suite(void);
```

```
SRunner *sr;
sr = srunner_create(make_master_suite());
srunner_add_suite(sr, make_list_suite());
srunner_add_suite(sr, make_msg_suite());
srunner_add_suite(sr, make_log_suite());
```


Tests

Tests en boîte blanche

Tests (structurels) en boîte blanche



- Accès au code pour déduire des tests à faire, de manière plus complète
  - Test de couverture (passage par tous les blocs d'instructions solidaires)
  - Techniques d'analyse dynamique, en effectuant l'instrumentation du code
  - Outils de mesure de couverture (tcov)

Retour sur la recherche dichotomique

```
chercher(key : ELEM) : integer
// si l'élement de clé key est dans la table, rend son indice,
sinon rend lower()-1
// prérequis : estOrdonné
local bas, haut, milieu: integer; trouve : boolean;
begin
 bas:= lower(); haut:= upper();
 milieu:= (bas+haut) mod 2;
 trouve:= (itemAt(milieu)=key);
 while (not trouve and bas <= haut) begin
  if (itemAt(milieu)=key) then trouve:=true;
  else if (itemAt(milieu) < key) then bas:=milieu+1;
  else haut:=milieu-1;
  end if
 end while
 if trouve then result:=milieu;
 else result:=lower()-1;
 end if
end;
```

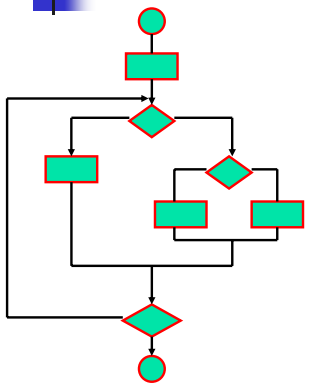
Affinage des jeux de tests

- L'examen du code montre qu'à chaque itération, le tableau est partitionné en 3 parties
  - Partie dont les indices sont inférieurs au milieu
  - Élément du milieu
  - Partie dont les indices sont supérieurs au milieu
- On peut ainsi ajouter les tests suivants :
  - Clé présente au centre du tableau
  - Clé présente juste avant le centre du tableau
  - Clé présente juste après le centre du tableau
- Assez empirique...

Tests des chemins d'exécution et de couverture

- A partir du graphe de flux de contrôle
  - On vérifie que l'on passe par tous les blocs d'instructions solidaires
- Plusieurs niveaux de contrôle sont envisageables
  - Tests des chemins d'exécution
    - Toutes les combinaisons possibles de passage de l'entrée à la sortie de la routine
    - Trop coûteux
  - Tests des chemins indépendants
    - Contiennent au moins un arc n'appartenant pas aux autres chemins
    - Moins coûteux
  - Tests de couverture
    - S'assure qu'on passe au moins une fois dans chaque bloc

Test de couverture de chemins



1) Dériver une mesure de la **complexité logique**

2) L'utiliser pour définir un ensemble de base des chemins d'exécution

D'où calcul de la **complexité cyclomatique**

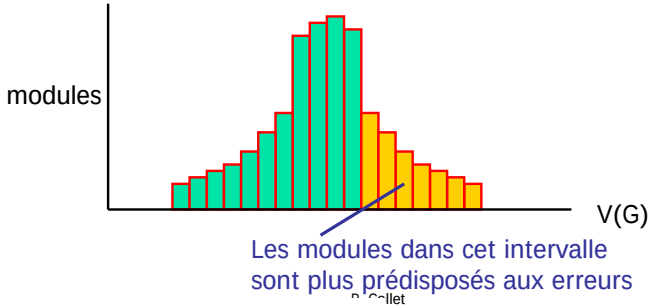
- V(G) = nb_d'arcs - nb_de_nœuds + 2 * nb_de_composants_connectés

Dans notre cas, V (G) = 11 - 9 + 2*1 = 4

V(G) fournit une borne supérieure des tests qui doivent être exécutés pour garantir la couverture de toutes les instructions du programme

Complexité cyclomatique [McCabe76]

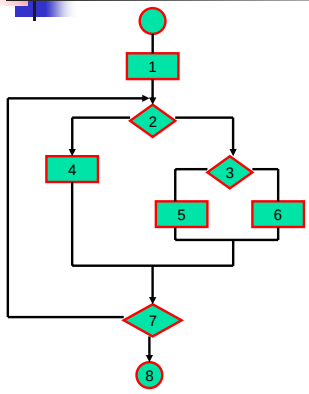
De nombreuses études industrielles ont montré que plus V(G) est grand, plus la probabilité d'erreurs est importante.



Ensemble de couverture

- La complexité cyclomatique V(G) définit le nombre de chemins indépendants dans l'ensemble de base
  - De nombreuses études industrielles ont montré que plus V(G) est grand, plus la probabilité d'erreurs est importante.
- L'ensemble de couverture des chemins est alors l'ensemble des chemins qui exécuteront toutes les instructions et toutes les conditions au moins une fois dans un programme
  - But : définir des cas de test pour l'ensemble de base
  - L'ensemble de couverture des chemins n'est pas unique
- Les tests de couverture doivent être appliqués aux modules critiques

Application



On peut dériver les chemins indépendants :

Comme V(G) = 4, il y a 4 chemins

Chemin 1: 1,2,3,6,7,8

Chemin 2: 1,2,3,5,7,8

Chemin 3: 1,2,4,7,8

Chemin 4: 1,2,4,7,2,4...7,8

Puis, on dériver des cas de tests pour confronter ces chemins.