Introduction aux tests du logiciel

F.X. Fornari
xavier.fornari@esterel-technologies.com
P. Manoury
pascal.manoury@pps.jussieu.fr

2011

Contents

1 Présentation du cours

Contenu du cours

- Le test logiciel: Nous en faisons, mais qu'est-ce précisément ?
- Quand et comment fait-on du test ?
- But du cours: sensibiliser au test en général:
  - Le sujet est vaste
  - les applications variées

2 Introduction aux tests logiciels

2.1 Qu'est-ce qu'un logiciel

Un logiciel, ce n'est pas seulement du code, mais aussi:

- des besoins;
- Une gestion de projet;
- Une spécification (fonctionalités, contraintes, facteur de qualité, interface);
- Une conception: fonctionnelle, architecturale, algorithmes, conception détaillée,..
- Un code source;
- Un exécutable;
- ... et des tests !

L'importance des tests se voit en cas de problèmes: système de gestion de licences de la FFV s'effondrant dès la mise en opération, Ariane 5, portes bloquées sur une automobile, plantage serveur lors de la coupe du monde de foot...

Le Logiciel

- Q0: Qu'est-ce que du logiciel ?
- des besoins, exprimés par un client
- une gestion de projet
  - cycle de vie (classique, agile, ...) avec ses délivrables
  - les outils et l'environnement de développement
- une spécification: la définition du logiciel en réponse aux besoins

- une conception: globale, architecturale, détaillée. Les composants et leurs échanges sont définis
- du code
- un *produit*
- *... des tests !*

2.2 Qu'est-ce que tester un logiciel

Q1: que veut dire "tester un logiciel" ? Etant donné un logiciel, comment le tester ? Le tester, c'est vérifier qu'il est conforme, mais à quoi, sur quel critères ?

Définition possible "Processus d'analyse d'un programme avec l'intention de détecter des anomalies dans le but de le valider"

Tester un logiciel

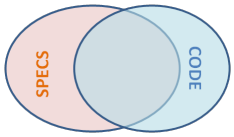
- Q1: Que veut dire "tester un logiciel" ?
- C'est valider sa conformité,
- Par rapport à des exigences

  C'est-à-dire par rapport à l'ensemble de la spécification, et de la conception du logiciel
- En fonction de critères.

  Par exemple, domaines d'entrées, taux d'erreurs acceptable, ...
- Q2: Y-a-t-il plusieurs types de tests ?
- Il y a différentes méthodes:
  - Test boîte noire-blanche,
  - Prédiction d'erreur (on sait ce qu'on cherche)
  - Tests automatiques, ...
- il y a différents types de tests: fonctionnels, non-fonctionels, structurels

Types de tests

- Fonctionnels:
  - *est-ce que le logiciel est conforme à la spécification ?*
  - liés à la spécification, à la qualité, à la performance, à l'interfaçage, ...
- Non-Fonctionnels:
  - *est-ce que son usage est conforme ?*
  - liés à la configuration, la compatibilité, à la documentation, au *stress*, ...
- Structurels:
  - *est-ce que le codage est correct ?*
  - fautes d'implémentation, ou fonctions non prévues.



- Q3: Y-a-t-il des alternatives ?
  - méthodes formelles:
    - *model-checking*, SAT, ...,
    - méthode par raffinements (B)
    - Interprétation abstraite (estimation d'erreur, runtime-error, ...)

    Ces méthodes sont de plus en plus employées. Problème: adéquation modèle réalité, passage à l'échelle
  - relecture de code: détection d'erreurs statiques, mais pas dynamiques.
    - C'est lourd, mais ça favorise la connaissance du code.
    - Les règles de codage sont importantes, et peuvent être vérifiées par des outils
  - analyse de sécurité: validation d'une architecture. C'est l'identification en amont de tous les problèmes potentiels.
- Q4: Coût du test ?
  - 30 à 40% du coût de développement, voire plus. Mais un bug peut coûter encore plus cher...

Un métier à part entière

- Seule activité dans le cycle de développement où l'on peut voir toutes les fonctionnalités d'un produit.
- Différent des développeurs spécialisés,
- C'est une activité créatrice:
  - il faut imaginer les scenarii pouvant mettre le logiciel en défaut,
  - il faut imaginer les bancs de tests, les environnements de simulations (logiciel de conduite de train, de controle des commandes de vol ⇒ comment faire ?)
  - demande rigueur et compétence
- Mais c'est une activité mal perçue:
  - le testeur est en fin de chaîne ⇒ retards
  - certains tests sont répétitifs
  - mal considéres par les développeurs

C'est pourtant une activité essentielle de R&D

2.3 Que teste-t-on ?

Typologie des logiciels

Tentative de classification:

- Transformationnels: logiciel de paie, compilateur, ...
- Interactif: Windows manager, WEB, DAB, ...
- Réactifs: comportement cyclique (chaîne de production)
- Réactifs sûrs: en milieu contraints: nucléaire, avionique, défense, ...

Pour chacune de ces classes, des contraintes différentes sont à gérer.

Exemples de contraintes

- Base de données
  - volume des données, intégrité
- Web
  - disponibilité, multi-navigateur, liens,
- Compilateur

  - test du langage d'entrée, des optimisations, du code généré, ...

- Interface graphique
  - multi-threading, sequences d'actions, *undo*, vivacité
- Code embarqués
  - tests sur hôte, et tests sur cibles, traçabilité
  - avionique, nucléaire, ... : exhaustivité, proche du 0 défaut
  - téléphone mobile: time-to-market très court !
- OS
  - ...

2.4 Quelques principes de base

Quelques principes de base

P1: Indépendance
un programmeur ne doit pas tester ses propres programmes Mais il vaut mieux faire ses tests avant de délivrer, et aussi les conserver !

P2: Paranoïa
Ne pas faire de tests avec l'hypothèse qu'il n'y a pas d'erreur (code trivial, déjà vu, ...) $\Rightarrow$ bug assuré !

Conseil: un test doit retourner *erreur* par *défaut*. Le retour *ok* doit être *forcé*.

P3: Prédiction
La définition des sorties/résultats attendus doit être effectuée avant l'exécution des tests. C'est un produit de la spécification.

- c'est nécessaire pour des développements certifiés
- les données sont fournies parfois au niveau système (ex: Matlab), mais les résultats seront différents à l'implementation.
- parfois les données sont trop complexes à fournir directement (éléments de compilation, environement complexe...)

P4: Vérification

- Il faut inspecter minutieusement les résultats de chaque test.
- Mais aussi la pertinence des tests
- (DO-178B): le process assure un "tuyau" propre, mais il faut quand même des filtres $\equiv$ vérification

- C'est la séparation de l'exécution et de l'analyse.

P5: Robustesse
Les jeux de tests doivent être écrits avec des jeux valides, mais aussi invalides ou incohérentes: on ne sait jamais ce qui peut arriver

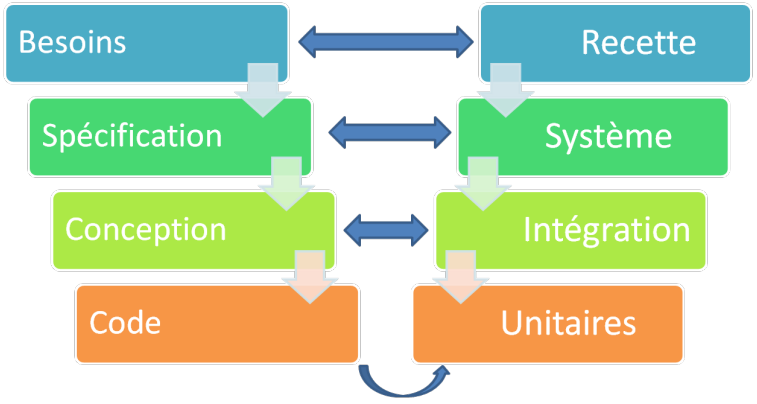
P6: Complétude
Vérifier un logiciel pour vérifier qu'il ne réalise pas ce qu'il est supposé faire n'est que la moitié du travail. Il faut aussi vérifier ce que fait le programme lorsqu'il n'est pas supposé le faire

En cas de bug ?

- Vérifier que le test est bien correct (P4)
- Vérifier que le problème n'est pas déjà répertorié (base de bugs par exemple)
- Etablir un rapport de bug
  - donner un synopsis succint et précis
  - donner une description claire, avec tous les détails de reproduction du bug
  - si possible, essayer de réduire l'exemple.
    Renvoyer un "tas" en disant "ça marche pas" ... ne marche pas.

3 Les différentes étapes du test des logiciels

Quand commencer ?
Le test commence de suite ! Ici, le cycle en V. mais aussi approches incrémentale,



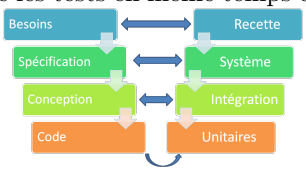
agiles, ...

Les différents niveaux

- Tests de recette: test de réception du logiciel chez le client final
- Tests intégration système: test de l'intégration du logiciel avec d'autres logiciels
- Tests systeme: test d'acception du logiciel avant livraison (nouvelle version par exemple)
- Tests Integration: test de l'intégration des différents composants (avec ou sans hardware)
- Tests Unitaires: tests élémentaires des composants logiciels (une fonction, un module, ...)
- Tests de non-régression
- "Smoke-tests": jeu réduit de tests pour valider un *build* avant de passer à la validation

La planification

- Ces différents niveaux de tests doivent être planifiés (c'est l'objet du plan projet).
- Théoriquement, on prépare les tests en même temps que le développement correspondant au niveau.



- On peut aussi le faire incrémentalement, ou en *pipeline*. Demande une gestion de projet très fine.
- En Extreme-Programmming: paire de développeurs/paires de testeurs.
- De toute façon, il y a rebouclage permanent (bugs, changements de specs): il faut s'adapter !

4 Les différents type de tests

4.1 Les tests fonctionnels

Tests fonctionnels

- Objectif: Analyse du comportement
  - conformité, comportement nominal, aux limites, robustesse

- Basés sur les spécifications, qui donnent le comportement attendus.
  - Attention: il ne suffit pas de donner des tests pour chaque exigence, mais il faut aussi comprendre les interactions possibles.
- Il faut mettre en place des stratégies.
  - De même qu'une exigence doit pouvoir être codée, elle doit pouvoir être testée ⇒ il faut une relecture de la spécification par les testeurs
- Différentes méthodes seront abordées dans la suite de ce cours

4.2 Les tests structurels

Tests structurels

- Objectif: Détecter les fautes d'implémentation
- Détecter si:
  - détection de cas de "plantage"
  - le logiciel n'en fait pas trop
- Niveau architectural & code:
  - Analyse des flots de contrôle, de données, condition logiques et des itérations.
  - Dépend de l'implémentation. Selon les cas, le testeur a accès ou non au code.
- De même qu'une exigence fonctionnelle doit pouvoir être codée, une exigence de codage doit pouvoir être testée
- Différentes méthodes seront abordées dans la suite de ce cours

5 Mise en œuvre

5.1 Les différents documents

Documents

Le test s'inscrit dans le cycle de vie du projet. Les documents suivant font partie des éléments nécessaires pour le test:

- Plan Projet: c'est le plan global du projet, servant de référence pour l'ensemble
- Spécification / Conception: les documents permettant l'élaboration du code et des tests associés
- Plan de tests: ce document doit décrire:

  - L'organisation du tests: équipes, environnement
  - Les stratégies: en fonction des éléments d'objectifs du projet, de spécifications, d'architectures, différentes stratégies de tests sont à envisager.
  - Ex WEB: test de l'interface, test de la logique "business" (authentification, transaction,...), test de la base, test de performances, de stress, ...
  - Les critères d'arrêt des tests: 100% de couverture de code n'est pas 100% des possibilités.

- Un standard de test. Assure une homogénéïté d'écriture et de validation.
- Les rapports de tests. Il faut être en mesure de donner un état précis de ce qui a été testé, et des résultats.
- En particulier, il faut pouvoir indiquer:
  - quelles sont les exigences couvertes ?
  - et comment les a-t-on couvertes (combien de tests/exigences, tests normaux, aux limites, de robustesse)
- La traçabilité entre les exigences et les tests doit être aussi assurée. En particulier, cela permet d'avoir une connaissance de l'impact d'une modification de spécification.

5.2 Organisation du test

Organisation du test

L'organisation du test recouvre différents aspects:

- Humain: il faut définir qui fait quoi (Plan de test)
  - Comment est constituée une équipe de validation
  - Comment elle interagit avec le développement
  - Comment le département Qualité (s'il existe) interagit
  - Quelle est l'interaction avec l'utilisateur final
- Technique: le comment
  - Comment met-on en place la base (ou les bases) de tests
  - L'environnement matériel éventuel (hôtes, mais cartes embarquées, autre matériel, réseau, ...). Il faut être au plus près de la mise en œuvre finale.
  - Quel outillage pour les tests automatiques ? Pour les mesures de tests ?
  - Utilisation d'outils maison ou COTS: s'est-on assuré de leur validité ?

5.3 Conclusion

Conclusion

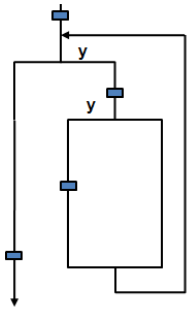
- Le test est une activité importante
- Demande compétence, rigueur, créativité
- Le test ne s'improvise pas: il faut le penser dès le début
- C'est une activité structurée, dans le cadre du projet
  - C'est un travail collectif entre le développent et la validation
  - Une bonne base de tests, avec une regression simple à mettre en œuvre est ausi très appréciée du dev !
- Ce n'est pas l'apannage de l'*industrie*, mais aussi des projets OpenSource (ex: Perl, Linux, ...) ... et aussi de vos développements !

6 Jeux de tests

Des jeux de tests : pour quoi faire ?

- Trace pour le contrôle (le vérificateur vérifie que le testeur a effectué les tests du jeu de test),
- Trace entre la conception et l'exécution du test (entre le moment de la spécification et celui o'u le code est écrit et peut être testé)
- Séparer l'exécution des tests de l'analyse des résultats (ce n'est pas au moment o'u on fait le test qu'il faut juger de la pertinence des sorties, ce n'est pas la même personne qui le fait la plupart du temps)

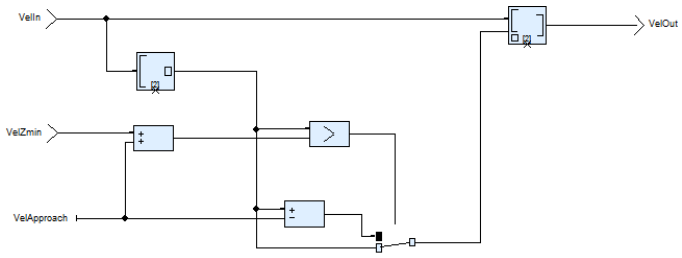
Flot de contrôle



L'analyse du flot de contrôle permet la construction des lignes du tableau de tests

```
Found  = False;
Indice = -1;
while (! Found && Indice < TabMax ) {
  Indice++;
  if ( Tab[Indice] == Value ) {
    Found = True;
  }
}
return Indice;
```

Flot de données



L'analyse du flot de données permet la construction des colonnes du tableau de tests

Composant: exemple

```
VG1, VG2 : Int;
procedure Proc (P1: in Int;
                P2: in out Int;
                P3: out Int)
is
  L1, L2 : Int;
begin
  -- corps de la procédure
end Proc;
```

Composant

Un composant (c'est-à-dire une procédure ou une fonction, *unit* en anglais) peut être caractérisé par :

- ses entrées :
  - paramètres d'entrée du composant
  - variables globales lues dans le composant
- ses sorties :
  - paramètres de sortie du composant
  - variables globales écrites dans le composant

Identifiables soit syntaxiquement, soit par une analyse du code

- les relations entre les entrées et les sorties de la procédure (corps de la procédure)
- les variables lues/écrites par les composantes appelées (causes d'indétermination si elles ne sont pas initialisées, cf bouchonnage)

Jeux de test

	Entrées du jeu			Sorties du jeu			
	P1	P2	VG1	P2	P3	VG1	VG2
jeu 1							
jeu 2							
jeu 3							

En supposant que les variables globales *VG1* et *VG2* soient:

- *VG1* lue et écrite
- *VG2* seulement écrite

L'Oracle

Procédure qui permet de prédire la valeur des sorties par rapport aux valeurs d'entrées.

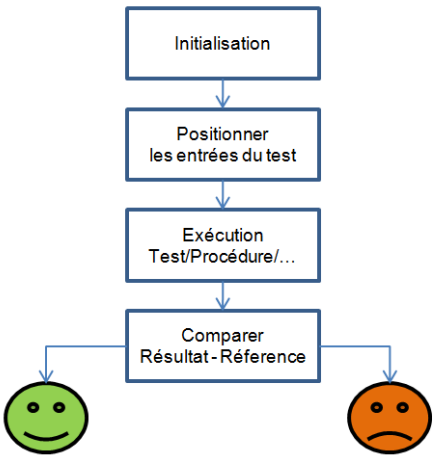
- Bas niveau : valeur précise
- Haut niveau : de la valeur précise à une simple propriété à vérifier (intervalle, propriété mathématique, propriété de sûreté, etc.)

Difficulté du test d'un logiciel

Toute la difficulté du test d'un logiciel provient de la détermination :

- d'un jeu d'entrées par rapport à une couverture de tests
- des valeurs de sortie d'une procédure par rapport à un jeu d'entrées déterminé (problème de l'oracle)
- traitement des composants appelés (cf le bouchonnage)

Exécution d'un test



Importance du jeu de tests

On voit ici l'importance du jeu de tests, car tout repose sur lui :

- Le jeu de tests est une représentation pour des valeurs particulières de la spécification d'une procédure.
- Si l'on veut tester complètement un logiciel, il faut élaborer tous les jeux de tests permettant de couvrir la spécification.
- La combinatoire de n'importe quel problème même de très petite taille est trop importante pour faire un test exhaustif. Par exemple, pour vérifier l'addition sur les entiers 32 bits, cela nécessiterait 2^{64} jeux de tests.

Equilibre d'arbitrage OK/KO

- trop de OK : le client n'est pas content
- trop de KO : le développeur est mécontent (30 à 50% des KO sont liés à des erreurs du testeur)

Il faut séparer (au moins temporellement) l'arbitrage et l'exécution des tests.

Couverture de tests, critère d'arrêt

Couverture de tests

- niveau de confiance dans le logiciel pour le client,
- le contrat entre le client et le testeur, jugé par le vérificateur,
- pour le testeur, critère de mesure

Critère d'arrêt : quand s'arrêter de tester ?

- négatif : bugs bloquants, on n'est pas en mesure de tester la suite
- positif (taux de couverture)

On ne cherche pas 100% de la couverture à chaque fois ($> 90\%$ bien fait, sauf normes de sûreté très spécifiques)

Choix de la couverture de tests

3 critères de choix :

- criticité du logiciel (normes de sûreté, imposé par le vérficateur)
- contraintes imposées au logiciel (facteurs qualité imposés par le client : temporelles, portabilité, etc.)
- type de logiciel (domaine : BD, réseaux, embarqué, etc.)

Taux de couverture
C'est une mesure de la couverture effective des tests.

Des justifications peuvent être nécessaires.

Chaque jeu de test doit augmenter la couverture de tests

7 Test Structurels

7.1 LCSAJ

Couverture de code: LCSAJ

"Linear Code Subpath And Jump": il s'agit d'une portion linéaire de code suivie d'un saut.

L'idée est de "couper" le code en tronçons linéaires, et de couvrir tous les chemins.

Pour un LCSAJ, nous avons:

- le début de la séquence;
- sa fin;
- la cible du saut.

Ces éléments sont exprimés en numéros de ligne.

LCSAJs ont un ratio d'efficacité de test de niveau 3.

Test Effectiveness Ratio

C'est une mesure de l'efficacité de couverture des tests.

$$TER1 = \frac{\sharp\ d'instructions\ executees}{\sharp\ total\ d'instructions\ executables}$$

$$TER2 = \frac{\sharp\ de\ branches\ executees}{\sharp\ total\ de\ branches}$$

$$TER3 = \frac{\sharp\ LCSAJs\ executes}{\sharp\ total\ de\ LCSAJs}$$

A noter:

$$TER3 = 100\% \Rightarrow TER2 = 100\% \text{ et } TER1 = 100\%$$

Un exemple

```
1  #include    <stdlib.h>
2  #include    <string.h>
3  #include    <math.h>
4
5  #define MAXCOLUMNS  26
```

```
6  #define MAXROW      20
7  #define MAXCOUNT    90
8  #define ITERATIONS  750
9
10 int main (void)
11 {
12     int count = 0, totals[MAXCOLUMNS], val = 0;
13
14     memset (totals, 0, MAXCOLUMNS * sizeof(int));
15
16     count = 0;
17     while ( count < ITERATIONS )
18     {
19         val = abs(rand()) % MAXCOLUMNS;
20         totals[val] += 1;
21         if ( totals[val] > MAXCOUNT )
22         {
23             totals[val] = MAXCOUNT;
24         }
25         count++;
26     }
27
28     return (0);
29
30 }
```

- Combinaison de boucle et de branchement;
- Nous avons donc des “bouts” de code séquentiel, ainsi que des sauts (branchements)
- Les sauts correspondent aux tests, et aux branches explicites ou implicites.

Un exemple

```
1  #include    <stdlib.h>
2  #include    <string.h>
3  #include    <math.h>
4
5  #define MAXCOLUMNS  26
6  #define MAXROW      20
7  #define MAXCOUNT    90
8  #define ITERATIONS  750
9
10 int main (void)
11 {
12     int count = 0, totals[MAXCOLUMNS], val = 0;
13
14     memset (totals, 0, MAXCOLUMNS * sizeof(int));
```

```
15
16      count = 0;
17      while ( count < ITERATIONS )
18      {
19          val = abs(rand()) % MAXCOLUMNS;
20          totals[val] += 1;
21          if ( totals[val] > MAXCOUNT )
22          {
23              totals[val] = MAXCOUNT;
24          }
25          count++;
26      }
27
28      return (0);
29
30  }
```

LCSAJ	Start	End	Jmp
1	10	17	28
2	10	21	25
3	10	26	17
4	17	17	28
5	17	21	25
6	17	26	17
7	25	26	17
8	28	28	-1

Un exemple

```
1  #include    <stdlib.h>
2  #include    <string.h>
3  #include    <math.h>
4
5  #define MAXCOLUMNS  26
6  #define MAXROW      20
7  #define MAXCOUNT    90
8  #define ITERATIONS  750
9
10 int main (void)
11 {
12     int count = 0, totals[MAXCOLUMNS], val = 0;
13
14     memset (totals, 0, MAXCOLUMNS * sizeof(int));
15
16     count = 0;
17     while ( count < ITERATIONS )
18     {
19         val = abs(rand()) % MAXCOLUMNS;
20         totals[val] += 1;
21         if ( totals[val] > MAXCOUNT )
22         {
23             totals[val] = MAXCOUNT;
24         }
25         count++;
26     }
27
28     return (0);
29
30 }
```

LCSAJ	Start	End	Jmp
1	10	17	28
2	10	21	25
3	10	26	17
4	17	17	28
5	17	21	25
6	17	26	17
7	25	26	17
8	28	28	-1

Un exemple

```
1  #include    <stdlib.h>
2  #include    <string.h>
3  #include    <math.h>
4
5  #define MAXCOLUMNS  26
6  #define MAXROW      20
7  #define MAXCOUNT    90
8  #define ITERATIONS  750
9
10 int main (void)
11 {
12     int count = 0, totals[MAXCOLUMNS], val = 0;
13
14     memset (totals, 0, MAXCOLUMNS * sizeof(int));
15
16     count = 0;
17     while ( count < ITERATIONS )
18     {
19         val = abs(rand()) % MAXCOLUMNS;
20         totals[val] += 1;
21         if ( totals[val] > MAXCOUNT )
22         {
23             totals[val] = MAXCOUNT;
24         }
25         count++;
26     }
27
28     return (0);
29
30 }
```

LCSAJ	Start	End	Jmp
1	10	17	28
2	10	21	25
3	10	26	17
4	17	17	28
5	17	21	25
6	17	26	17
7	25	26	17
8	28	28	-1

Un exemple

```
1  #include    <stdlib.h>
2  #include    <string.h>
3  #include    <math.h>
4
5  #define MAXCOLUMNS  26
6  #define MAXROW      20
7  #define MAXCOUNT    90
8  #define ITERATIONS  750
9
10 int main (void)
11 {
12     int count = 0, totals[MAXCOLUMNS], val = 0;
13
14     memset (totals, 0, MAXCOLUMNS * sizeof(int));
15
16     count = 0;
17     while ( count < ITERATIONS )
18     {
19         val = abs(rand()) % MAXCOLUMNS;
20         totals[val] += 1;
21         if ( totals[val] > MAXCOUNT )
22         {
23             totals[val] = MAXCOUNT;
24         }
25         count++;
26     }
27
28     return (0);
29
30 }
```

LCSAJ	Start	End	Jmp
1	10	17	28
2	10	21	25
3	10	26	17
4	17	17	28
5	17	21	25
6	17	26	17
7	25	26	17
8	28	28	-1

Un exemple

```
1  #include    <stdlib.h>
2  #include    <string.h>
3  #include    <math.h>
4
5  #define MAXCOLUMNS  26
6  #define MAXROW      20
7  #define MAXCOUNT    90
8  #define ITERATIONS  750
9
10 int main (void)
11 {
12     int count = 0, totals[MAXCOLUMNS], val = 0;
13
14     memset (totals, 0, MAXCOLUMNS * sizeof(int));
15
16     count = 0;
17     while ( count < ITERATIONS )
18     {
19         val = abs(rand()) % MAXCOLUMNS;
20         totals[val] += 1;
21         if ( totals[val] > MAXCOUNT )
22         {
23             totals[val] = MAXCOUNT;
24         }
25         count++;
26     }
27
28     return (0);
29
30 }
```

LCSAJ	Start	End	Jmp
1	10	17	28
2	10	21	25
3	10	26	17
4	17	17	28
5	17	21	25
6	17	26	17
7	25	26	17
8	28	28	-1

Un exemple

```
1  #include    <stdlib.h>
2  #include    <string.h>
3  #include    <math.h>
4
5  #define MAXCOLUMNS  26
6  #define MAXROW      20
7  #define MAXCOUNT    90
8  #define ITERATIONS  750
9
10 int main (void)
11 {
12     int count = 0, totals[MAXCOLUMNS], val = 0;
13
14     memset (totals, 0, MAXCOLUMNS * sizeof(int));
15
16     count = 0;
17     while ( count < ITERATIONS )
18     {
19         val = abs(rand()) % MAXCOLUMNS;
20         totals[val] += 1;
21         if ( totals[val] > MAXCOUNT )
22         {
23             totals[val] = MAXCOUNT;
24         }
25         count++;
26     }
27
28     return (0);
29
30 }
```

LCSAJ	Start	End	Jmp
1	10	17	28
2	10	21	25
3	10	26	17
4	17	17	28
5	17	21	25
6	17	26	17
7	25	26	17
8	28	28	-1

Un exemple

```
1  #include    <stdlib.h>
2  #include    <string.h>
3  #include    <math.h>
4
5  #define MAXCOLUMNS  26
6  #define MAXROW      20
7  #define MAXCOUNT    90
8  #define ITERATIONS  750
9
10 int main (void)
11 {
12     int count = 0, totals[MAXCOLUMNS], val = 0;
13
14     memset (totals, 0, MAXCOLUMNS * sizeof(int));
15
16     count = 0;
17     while ( count < ITERATIONS )
18     {
19         val = abs(rand()) % MAXCOLUMNS;
20         totals[val] += 1;
21         if ( totals[val] > MAXCOUNT )
22         {
23             totals[val] = MAXCOUNT;
24         }
25         count++;
26     }
27
28     return (0);
29
30 }
```

LCSAJ	Start	End	Jmp
1	10	17	28
2	10	21	25
3	10	26	17
4	17	17	28
5	17	21	25
6	17	26	17
7	25	26	17
8	28	28	-1

Un exemple

```
1  #include    <stdlib.h>
2  #include    <string.h>
3  #include    <math.h>
4
5  #define MAXCOLUMNS  26
6  #define MAXROW      20
7  #define MAXCOUNT    90
8  #define ITERATIONS  750
9
10 int main (void)
11 {
12     int count = 0, totals[MAXCOLUMNS], val = 0;
13
14     memset (totals, 0, MAXCOLUMNS * sizeof(int));
15
16     count = 0;
17     while ( count < ITERATIONS )
18     {
19         val = abs(rand()) % MAXCOLUMNS;
20         totals[val] += 1;
21         if ( totals[val] > MAXCOUNT )
22         {
23             totals[val] = MAXCOUNT;
24         }
25         count++;
26     }
27
28     return (0);
29
30 }
```

LCSAJ	Start	End	Jmp
1	10	17	28
2	10	21	25
3	10	26	17
4	17	17	28
5	17	21	25
6	17	26	17
7	25	26	17
8	28	28	-1

Calculé automatiquement d'ordinaire, inutile de s'inquiéter !

Avantage La couverture est sensiblement meilleure que la couverture des branches sans être trop explosive (bon ratio coût/niveau de confiance, entre couverture des branches et couverture des chemins).

Inconvénient Elle ne couvre pas la totalité de la spécification. Elle dépend du code du programme plus que de la structure réelle du graphe de contrôle.

7.2 Couverture MC/DC

Résumé des couvertures

Instruction couverture des instructions uniquement, mais les conditions ne sont pas couvertes complètement

Décision Couverture des branches, il faut un test à *vrai* et un test à *faux*

Condition Chaque condition doit être à *vrai* ou *faux*: Ex: "A or B" ⇒ `TF` et `FT`

Condition/Decision Combinaison des 2 précédantes, mais ne permet pas de distinguer `TT` et `FF` pour "A and B" et "A or B"

MC/DC *Modified Condition/Decision Coverage*. Demande l'indépendance de chaque condition sur la sortie. "A or B" ⇒ `TF`, `FT` et `FF`

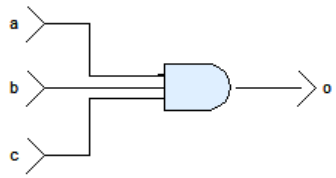
Multiple Condition Faire les 2^n cas. N'est pas praticable en général.

Critères satisfaits en fonction des couvertures.

Critère de couverture	**Instruction**	**Décision**	**Condition**	**Condition Décision**	**MC/DC**	**Multiple de Conditions**
Chaque point d'entrée/de sortie du programme exécuté au moins une fois		x	x	x	x	x
Chaque instruction du programme exécutée une fois	x					
Chaque décision a pris toutes les valeurs possible au moins une fois		x		x	x	x
Chaque condition d'une décision a pris toutes les valeurs possible au moins une fois			x	x	x	x
Chaque condition d'une décision a affecté la décision de manière indépendante					x	x
Chaque combinaison de conditions d'une décision a été faite au moins une fois						x

Couverture MC/DC

Un exemple avec l'opérateur "and":



Test	1	2	3	4
a	T	F	T	T
b	T	T	F	T
c	T	T	T	F
o	T	F	F	F

- La sortie "o" est modifiée par une seule entrée
- 4 tests au lieu de $2^3 = 8$
- Le problème du masquage: la cause "unique" n'est pas toujours possible.
- AND: FF inutile, OR: TT inutile, etc...

Test	1	2	3	4
a	T	*F*	*F*	T
b	T	*T*	*F*	T
c	T	T	T	F
o	T	F	F	F

MC/DC: méthodologie

1. Avoir une représentation schématique du code (portes logiques)
2. Identifier les tests en fonction des spécifications. Propager les tests sur la représentation
3. Eliminer les cas de masquages
4. Valider que les tests restants forment la couverture MC/DC
5. Examiner la sortie des tests pour confirmer la correction du logiciel

7.3 Couverture Itérations

Couverture des itérations

procedure p **is**	test 1	test 2	test 3
begin			
while c1	F	V,F	V,...,V,F
loop			
s1;		s1	s1;...;s1
end loop;			
s2;	s2	s2	s2
end p;			

Pour chaque boucle

- 0 itération
- 1 itération

- max - 1 itérations
- max itérations
- max + 1 itérations

7.4 Couverture des données

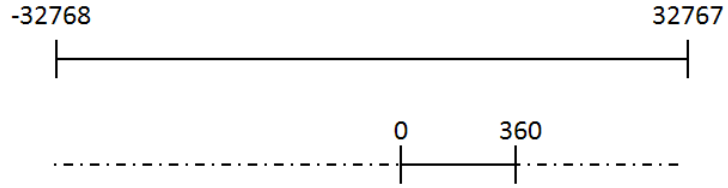
Couverture des donnees

Dans les couvertures précédentes : on ne s'est intéressé ni au format des données manipulées ni aux valeurs numériques permettant de rendre une condition vraie ou fausse. La couverture des données consiste à choisir :

- les "bonnes valeurs numériques" pour rendre une condition vraie ou fausse (valeurs remarquables)
- les valeurs limites des données d'entrée
- les valeurs numériques permettant d'obtenir les valeurs ou les erreurs prévisibles (overflow, division par zéro, ...)

Couverture des données: domaine

Il faut distinguer les domaines de définitions des données, et les domaines d'utilisation:



Couverture d'une donnée

Un exemple de valeurs compte tenu de l'interval fonctionnel, du test, et du type de donnée:

	test 1	test 2	test 3	test 4	test 5	test 6
procedure p **is**						
begin						
if e > 30	360	31	30	0	-32768	32767
then						
s1;	s1	s1				s1
endif;						
end p;						

Couverture d'une donnée

- Une procédure évalue les données pour des valeurs remarquables.
- Par exemple la conditionnelle : $e > 30$ montre que la valeur 30 est une valeur remarquable pour la donnée e.
- Il est possible de découper le domaine de chaque donnée en classes d'équivalence. Une classe définit un comportement propre à la donnée.
- Sur l'exemple précédent, la donnée peut être découpée en 2 classes d'équivalence [val_min; 30], [31; val_max].

Couverture d'une donnée

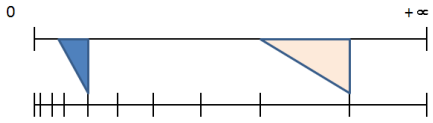
Du point de vue du testeur, cela nécessite deux types de test :

- les tests *aux limites* pour vérifier le comportement du logiciel pour chaque borne de chaque classe d'équivalence (les limites fonctionnelles des données et les valeurs remarquables)
- les tests *hors limites* pour vérifier le comportement du logiciel avec des valeurs en dehors de son domaine fonctionnel.

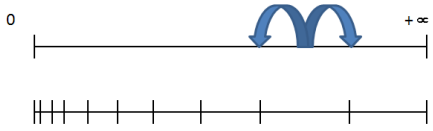
Cas des réels

Le cas des réels pose le problème de la représentation de flottants:

- Les valeurs sont discrétisées, et les intervalles ne sont pas les mêmes (nombre de bits fixe).



- Selon le calcul, le résultat peut changer d'intervalle.



- Un programme devrait faire: `if`$(val1 - val2) < \epsilon$)... et non `if`$(val1 = val2)$...
- Idem pour tester les résultats: il faut définir la précision attendue par rapport à la spécification.

Couverture de données: enumeration

Considérons le programme C suivant:

```
switch (size-bound) {
case 3:
  value += mBvGetbit(p_bv, byte_idx)<<2;
  byte_idx--;
case 2:
  value += mBvGetbit(p_bv, byte_idx)<<1;
  byte_idx--;
case 1:
  value += mBvGetbit(p_bv, byte_idx);
}
```

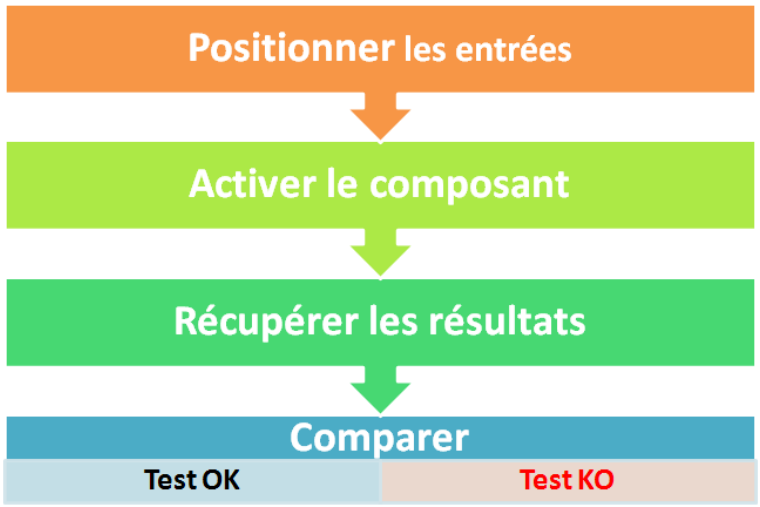
- Si $(size - bound) = 3$, tout le code est couvert.
- Certains outils l'acceptent, d'autres demandent la couverture de tous les cas
- Vérifier que **toutes** les valeurs possibles de l'énumération sont effectivement prises
- Faire le test avec des valeurs hors énumération (facile en C, plus dur en Ada)

7.5 Developpement de tests structurels

Developpement des tests structurels

- Pour les tests structurels, il faut écrire des programmes de tests pour exécuter les tests
- Ces programmes contiennent les tests, et sont *linkés* avec le composant à tester.
- Ils sont souvent écrits automatiquement par des logiciels d'éxecution de tests (Ex: RTRT d'IBM, LDRA TestBed, ...)

Execution d'un test



Programmes de tests

Les programmes de test sont architecturés comme suit :

- Initialisation de la valeur de retour du test à *KO* !
- Initialisation des variables **globales** utilisées par le composant à tester à partir des valeurs d'entrée des jeux de tests
- Appel du composant à tester avec les paramètres initialisées à partir des valeurs d'entrée des jeux de tests
  - En fonction des spécifications
  - Les données sont lues ou mise dans le source du test. *Attention:* Peut s'avérer complexe.
- Récupération des sorties produites par le composant (paramètres en sortie **et** variables globales)
- Comparaison des valeurs des sorties obtenues avec les valeurs des sorties décrites dans les jeux de tests
  - La comparaison peut-être plus ou moins simple (chaîne de caractères, réel vs flottant, structures, bitstream....)
- Enregistrement et affichage du résultat de test.
  - Certains tests sont manuels: il faut aussi enregistrer leurs résultats !

Bouchonnage

Le composant fait appel à une fonction externe `f`.

Que l'on dispose ou non de cette fonction, on souhaite dans un premier temps tester le comportement du composant indépendamment de celui de `f`. La technique du bouchonnage consiste donc à :

- ajouter une entrée *artificielle* dans les jeux de test. Cette entrée correspondra à la sortie de la fonction f
- écrire un *bouchon* pour remplacer le véritable code de la fonction `f`: ce code ne fait que retourner la valeur positionnée dans le jeu de test.

Ainsi le testeur peut maîtriser complétement les tests réalisés.

7.6 Points particuliers

Code défensif

- Certaines parties ne sont pas accessible durant l'exécution *normale* du code:

```
typedef
  enum {RED, ORANGE, GREEN}
  Light_t;
switch (light) {
case RED: ...
          break;
case ORANGE: ...
          break;
case GREEN: ...
          break;
default:
       /* handler */
}
```

```
void GetName(Person* pP) {
  assert(pP != (Person*)0);
  ...
}
```

- Normalement, ce type de situation ne doit pas se produire, seulement sur des versions en tests.
- Cela peut se compliquer avec l'utilisation d'exceptions.

7.7 Tests Structurels: Conclusion

En Résumé

- Il existe différents tests structurels, qui offrent plus ou moins de garanties
- Le test structurel est dépendant du code

  - Toute modification du code peut entrainer des modifications des tests
  - Le code est plus ou moins proche de la spécification (degré de liberté du développeur)
  - Il faut une bonne traçabilité, et une gestion de configuration
- Quelques idées générales ont été présentées, à adapter
  - En fonction des applications, des données, des algorithmes
  - En fonction des langages: C, C++/Java (héritage, dispatch, ...), O'Caml, assembleur.

8 Test fonctionnels

Test Fonctionnels
Aussi appelés test "boite-noire"

But

- vérification du comportement du logiciel par rapport aux spécifications (fonctions non conformes, manquantes, erreurs, ...)
- respect des contraintes (performances, memoire, delai, ...), et de qualité (portabilité, maintainabilité, documentation, ...)

Critère d'arrêt
Le nombre de tests nécessaires ne peut être connu a priori. Le **seul** élément certain est la *spécification*. Le testeur doit y trouver:

- les fonctions à implémenter
- L'interface du logiciel
- Les contraintes:
  - delai, taille
  - hardware
  - sûreté
  - ...

Les critères d'arrêt se font sur des mesures qualitatives.

Catégories de tests fonctionnels

Que test-t-on ? Comment ?

- Quoi ? couverture des tests
- Comment ? : analyse partitionnelle pour le test des fonctions de la spécification

Catégories de tests fonctionnels

- tests nominaux et aux limites: reliés aux fonctions du logiciel
  - Nominal: test la conformité par rapport à la spécification pour un comportement attendu
  - Limites: test du logiciel et de ses limites *définies*
- Robustesse: facteur de qualité
  - Liée à l'environnement: tests hors limites, charge/stress, défaut d'équipement externe, ...
- Tests de conformité: autres facteurs de qualités et/ou contraintes
  - perf, ergonomie, portabilité, ...

8.1 Analyse partionnelle

Analyse Partionnelle

Problème
Déterminer les jeux d'entrées

Première solution
Force brutale: produit cartésion des domaines d'entrées.
Défaut: potentiellement astronomique (addition $\rightarrow 2^{32}$).
Il faudrait des valeurs particulières *représentatives*.

Seconde solution
Trouver des classes d'équivalences d'entrées: ensemble des entrées aboutissant au même comportement fonctionnel.
Par exemple: 2 tests pour l'addition: avec et sans débordement.

Analyse Partionnelle
Pour chaque fonction de la spécification:

- déterminer les entrées et leurs domaines
- en fonction de la partie contrôle de la spécification, découper chaque domaine en classes d'équivalence
- Pour chaque classe:
  - sélectionner un représentant
  - déterminer le résultat attendu

Valeurs de sortie ou propriétés

Problème de l'oracle

- algorithme trop complexe (régulation en automatisme)
- toutes les entrées nécessaires par forcément accessibles au testeur (horloge système, positionnée par l'environnement, ...)

Partition

Partition
Soit un domain D, les ensembles C_1, ... C_n sont une partition en classes d'équivalence pour D si:

$$\begin{aligned} \forall i, j, C_i \cap C_j &= \emptyset \text{ règle 1: exclusion} \\ \bigcup_{i=1}^{n} C_i &= D \text{ règle 2: overlay} \end{aligned}$$

Règle 1 non satisfaite : La spécification n'est pas déterministe Règle 2 non satisfaite : La spécification incomplète

Exemple

Programme calculant : $\sqrt{\frac{1}{x}}$
3 classes d'équivalence:

- $x < 0$
- $x = 0$
- $x > 0$

Une seule classe est valide

Détermination des classes d'équivalence

Différentes méthodes sont possibles:

- Langages formels ⇒ détermination des chemins de la spécification
- Automates, réseaux de Petri ⇒ parcours des automates, ...
- Matrices cause/effet ⇒ parcours des matrices
- Langage naturel ⇒ spécification re-modélisée en langage formalisé ou automate.

Spécification informelle

Si la spécification ne peut pas être testée, elle peut être rejetée, ou alors:

- un modèle doit être développé
- ce modèle doit être validé par l'équipe de développement
- des classes d'équivalence peuvent être définies

Ce process de remodélisation permet souvent de détecter des problèmes très tôt:

- incohérences entre des parties de la spécification
- incomplétude des cas traités.

Test Nominaux

- sélection d'*une* valeur "intelligente" dans la classe d'équivalence
- varier les valeurs à l'intérieur d'un même intervalle (entropie)

Exemple de partition

Function:	AbsoluteValProd
Inputs:	E1, E2, integers
Outputs:	S
Role:	calcule la valeur absolue du produit des entrées

Classes d'équivalence pour les entrées

E1	E2
[Min_Int, -1]	[Min_Int, -1]
[0, Max_Int]	[0, Max_int]

Tests nominaux: choix "intelligent"

E1		E1	
[Min_Int,-1]	-734	[Min_Int,-1]	-525
[Min_Int,-1]	-7445	[0, Max_Int]	3765
[0, Max_Int]	7643	[Min_Int,-1]	-765
[0, Max_Int]	9864	[0, Max_Int]	3783

Test aux et hors limites fonctionnelles

- Tests aux limites fonctionnelles: sélection de valeurs *aux bornes* de chaque classe d'équivalence fonctionnelles
- Tests hors limites fonctionnelles: sélection de valeurs *hors bornes* de chaque classe d'équivalence fonctionnelles

Tests aux limites et hors limites

Si les entrées E1 et E2 sont dans le domaine [-100, 100].

Limites fonctionnelles

E1		E1	
[-100,-1]	-100	[-100,-1]	-57
[-100,-1]	-1	[0, +100]	64
[0, +100]	0	[-100,-1]	-5
[0, +100]	100	[0, +100]	98
[-100,-1]	-59	[-100,-1]	-1
[0, +100]	48	[-100,-1]	-100
[-100,-1]	-63	[0, +100]	0
[0, +100]	75	[0, +100]	100

Hors limites

E1		E1	
[-100,-1]	-234	[-100,-1]	-42
[0, +100]	174	[0, +100]	39
[-100,-1]	-84	[Min Int, -1]	-115
[0, +100]	48	[0, +100]	120

8.2 Les autres tests fonctionnels

Tests de robustesse

Vérifier le comportement du logiciel face à des événements non spécifiés ou dans des situations dégradées:

- Tests en charge
- Tests des pannes des équipements externes
- Données incorrectes (lié aux tests hors limites),
- Mémoire insuffisante,
- Options incorrectes, ...

Teste en charge

Vérifier le comportement du logiciel en cas de stress du logiciel tel que:

- avalanche d'alarmes
- saturation des réseaux
- saturation des requêtes
- saturation des resources
- ...

Tests de pannes des équipements externes

Simuler des pannes sur les équipement en interface avec le logiciel afin de vérifier son comportement:

- arrêt inopiné de l'équipement
- débranchement brutal de l'équipement (USB...)
- changement brusque de valeurs
- ...

Tests de pannes des équipements externes: connaissances requises

Ces tests nécessitent une bonne connaissance du hardware afin de spécifier les bons modèles de défaillance des équipements.

Par exemple pour un interrupteur:

- collage à 0 ou 1,
- bagottements (acquisition intermittente)
- parasitage à différentes fréquences

Le test des interfaces

Le but des tests des des interfaces est double:

- vérifier les interfaces logicielles entre les composants d'un sous-système logiciel
- vérifier les interfaces physiques entre le logiciel est la machine cible (carte par ex, mais aussi drivers)

Composants logiciels :Il y a deux types de tests à faire

- vérifier l'échange des données
- vérifier l'activation des composants

8.3 Tests fonctionnels: conclusion

Conclusion pour les tests fonctionnels

Ce ne sont que quelques exemples: tout dépend énormément du métier pour lequel le logiciel est développé.

Les tests fonctionnels font intervenir l'environnement, alors que les tests structurels se concentrent sur l'intérieur du logiciel.

L'exemple du triangle

Spécification
Un programme a trois entiers en entrée. Ces entiers sont interprétés comme les longueurs des côtés d'un triangle.

Le programme indique s'il s'agit d'un triangle ordinaire (scalène), isocèle ou équilatéral

Question
Produire une suite de tests pour ce programme. Combien de cas ?

L'exemple du triangle

cas de tests possibles – GM Myers "The Art of Software Testing"

1. cas normal ordinaire (1,2,3 et 2,5,10 ne le sont pas. Pourquoi ?)
2. cas normal équilatéral
3. cas isocèle normal (2,2,4 n'est pas valide)
4. cas isocèle avec 3 permutations (3,3,4; 3,4,3, 4,3,3)
5. avec la valeur 0
6. avec une valeur négative
7. la somme de 2 entrées égale à la 3ième
8. 3 cas pour le test 7 avec permutations
9. cas où la somme des 2 entrées est supérieure à la 3ième
10. 3 cas pour le test 9 avec permutations
11. cas où les 3 entrées sont nulles
12. 3 cas avec une valeur non entière
13. cas avec un nombre incorrect d'entrées
14. pour chaque test, avez vous prédit le résultat ?

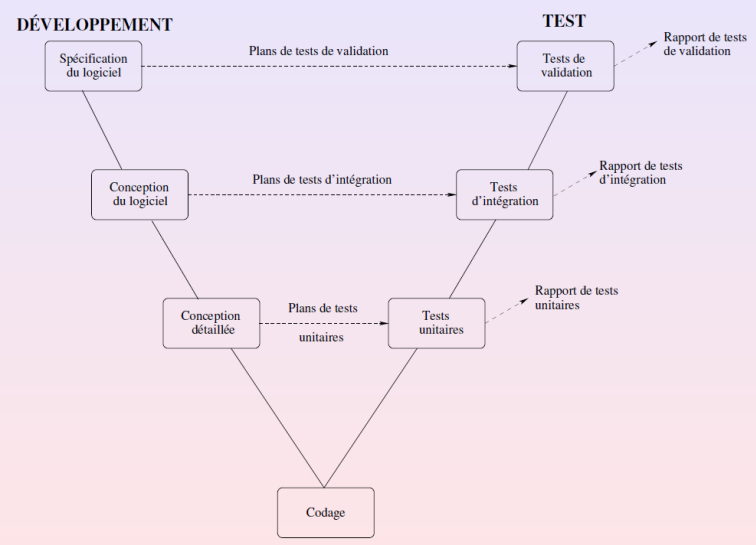
9 Cycle de vie

Introduction

- Les cours précédants ont abordés les techniques de tests
- Ces techniques s'appliquent lors de différentes phases
- L'objectif du cours est de montrer comment *formaliser* les approches.

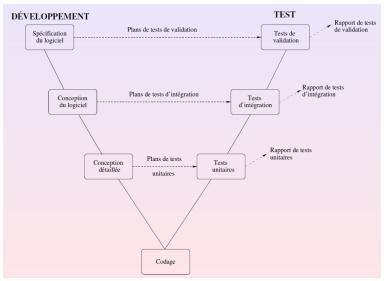
9.1 Les Phases de tests

Cycle de Vie



Les Phases de tests

- Le travail de testeur ne commence pas *après* la phase de codage, mais *en même temps*, et se poursuit en parallèle
- Afin d'éviter le "Big-Bang" à l'intégration, les phases de tests valident le logiciel depuis le module jusqu'au logiciel complet.



Les Phases de tests

- Chaque phase de tests a un but précis:
  - c'est la découvert d'anomalies
  - et la construction de la confiance attendue
  - c'est une brique pour la phase suivante

- Un *trou* dans la couverture d'une phase entraîne:
  - soit le masquage d'un problème lors de la mise en production
  - soit une difficulté de diagnostic lors d'anomalies durant les phases suivantes.

Plus un défaut est découvert tardivement plus il coûte cher à corriger

9.2 Les Tests par Phases

Les Tests Unitaires (TU]

But Validation de la conformité de chaque composant logiciel pris unitairement par rapport à sa spécification détaillée.

Quand ? Dès qu'une pièce de code a été codée et compilée correctement

Type de tests structurels

Comment ? sur machine hôte, généralement sans banc de tests

Qui ? Pour les logiciels de faible criticité, elle peut être réalisée par l'équipe de développement (mais pas par le développeur ayant codé la pièce de code).

Les Tests d'Intégration (TI)

But Validation des sous-systèmes logiciels entre eux

- Tests d'Intégration Logiciel/Logiciel (interface entre composants logiciels)
- Tests d'Intégration Logiciel/Matériel (interface entre le logiciel et le matériel)

Quand ? Dès qu'un sous-système fonctionnel (module, objet) est entièrement testé unitairement

Type de tests des interfaces

Comment ? logiciel/logiciel généralement sur machine hôte logiciel/matériel sur machine cible avec un banc de test minimal (simulation des entrées, acquisition des sorties).

Qui ? toujours par une équipe de tests indépendante de l'équipe de développement.

Les Tests de Validation (TV)

But Vérifier la conformité du logiciel à la spécification du logiciel

Quand ? Dès que l'ensemble des sous-systèmes fonctionnels ont été testé et intégré

Type de tests fonctionnels et de robustesse

Comment ? toujours réalisés sur machine cible et nécessitent généralement la fabrication d'un banc de tests élaboré

Qui ? Ces tests sont toujours réalisés par une équipe de tests indépendante de l'équipe de développement.

Autres Tests dans le cycle

On peut avoir les tests suivants:

- Test d'Intégration Système:
  - Ces tests permettent d'assurer la compatibilité du logiciel avec d'autres
  - Par exemple: un logiciel de gestion interopérant avec d'autres logiciels chez un client
- Test de recette:
  - Test servant à assurer la conformité et la confiance avant livraison finale.
  - Se décompose parfois entre *test de recette utilisateur*, impliquant les utilisateurs finaux, et *test d'exploitation*, impliquant le service informatique client.
- Test de non-régression
  - Assure la non-régression entre deux versions du logiciel, pour les mêmes fonctionnalités
  - Est utilisé au cours du cycle de développement, si celui-ci est fait par incréments

9.3 ... En Descente

Travail en phase de descente

Durant les phases de descente du cycle, le testeur élabore les *Plans de Tests*, et fabrique les bancs de tests. Les plans de tests décrivent essentiellement:

- La stratégie de tests mise en place
- Les moyens mis en œuvre (matériel, logiciel, et humain)
- L'ensemble des fiches de tests

Plan de Tests

C'est la description globale de la mise en œuvre. (fiche)

1. Introduction
   - Contexte, Organisation, Réferences
2. Objectifs et Obligations
   - Quels sont les objectifs du test
   - Quels sont les contraintes (délai, régression, ...)
3. Risques et dépendances
   - relevés des risques (code non livré, ...), liens avec d'autres outils
4. Hypothèses et exclusions
5. Critères de départ et d'arrêt
   - départ: définit les éléments nécessaires pour tester
   - arrêt : définition de critères (taux d'erreurs, revues, ...)
6. Contrôle du projet
   - Qui fait quoi
   - Besoins (formations, etc...)
   - Communication des problèmes, Rapports d'avancement

Spécification de test

La spécification de test détaille la réalisation de chaque test/catégories de tests (fiche)

1. Introduction
2. Exigences
   (a) Politique de test: vue d'ensemble, domaines, catégorisation des tests, ..
   (b) Environnement du test
   (c) Attributions des participants
   (d) Caractéristique du test
      - les scénarios, enregistrement des résultats, acceptance,
      - documentation des tests
3. Mode de test
   (a) Activités préliminaires
   (b) Réalisation du test
   (c) Scénario de test

9.4 ... En Montée

Travail en phase de remontée

Le testeur exécute les fiches de tests décrites dans les plans et produit les rapports de tests associés.

Ces rapports contiennent essentiellement:

- la synthèse des résultats de tests
- les résultats de tests détaillés
- la trace d'exécution des tests
- Exemple:
  - Modèle de résultats
  - Journal de tests

Rapport de test

C'est l'élément final après une "campagne" de tests (fiche)

1. Introduction
2. Vue d'ensemble
   - descriptions des activités
3. Ecarts
   - s'il y a des points particuliers, les mettre en exergue
4. Analyse
   - détail de la réalisation des tests, des problèmes rencontrés éventuellement
   - détail de la couverture de code si besoin
5. Résultats
   - résumé des résultats
6. Résumé des activités
   - résumé synthétique des résultats mais aussi des activités et de leurs déroulements

9.5 Gestion de processus

La gestion du processus de tests

- Tester un logiciel est un activité à part entière
- Cette activité doit être en constante amélioration
- Il faut donc pouvoir mesurer l'activité
- Il faut avoir des éléments de mesure ⇒ il faut des données **objectives**
- ces données doivent être collectées pour être utilisées lors du *Post-Mortem*

Exemple d'analyse

- Processus de développement et de
  - Est-ce que le logiciel et les tests ont été bien faits ? (fiche)
- Mise en œuvre du processus de test (fiche)

Environnement

- De même l'environnement de test doit être analysé
- En amont, pour sa mise en place
- En aval, pour déterminer s'il a répondu aux demandes, et s'il faut l'améliorer
- C'est le cas par exemple pour un outil (pour automatiser les tâches, ou faire de l'analyse). Exemple:
  - L'adéquation aux besoins (fiche)
  - Ses caractéristiques (fiche)
  - Son usage (fiche)
  - Son accessibilité (fiche)

Conclusion

- La mise en œuvre de tests demande de la rigueur
- Il faut définir un cadre de réalisation
  - méthodes, objectifs, technologies, resources, ...
  - Ce cadre doit être connu de tous
- Les tests font partie du produit, l'accompagne durant sa vie
  - ils évoluent avec lui

  - ils sont aussi un produit de la spécification

- Le *reporting* est important
  - il donne la mesure de la confiance
  - il doit être lui-même vérifiable !