Les tests logiciels : séance 1

M.Madani

ISIL - HEPL

Février 2016

Références

- Software Testing Foundations, Andreas Spillner, Tilo Linz, Hans Schaefer, Rocky Nook, 2007.
- Foundations of Software Testing, Aditya P. Mathur,Pearson Education India,2008.
- Software Testing and Quality Assurance,Kshirasagar Naik,Priyadarshi Tripathy,Wiley,2008.
- Software Error Detection Through Testing and Analysis,J.C.Huang,Wiley,2009.
- The Art of Software Test,Myers,Wiley,2012.

Première partie I

Introduction aux tests logiciels

Erreur, anomalie et faute

- Les termes sont souvent confondus :
  - Erreur(Error) :désigne l'action d'un intervenant qui entraîne un résultat incorrect.
  - Anomalie(Failure) : C'est la description de la situation lorsqu'un utilisateur fait face à un problème.
  - Faute(Fault) : Une anomalie ne peut survenir que s'il existe une faute dans le programme. (faute = defect, internal error).
- L'IEEE 610.12 - IEEE Standard Glossary of Software Engineering Terminology.

Pourquoi des fautes ?

- Dans chaque phase, les fautes peuvent trouver leur origine.
  - Une exigence non comprise du client.
  - Une exigence comprise lors de la conception mais non remplie.
  - Simplement dans l'implémentation.
- En termes d'implémentation, cela signifie
  - des chemins manquants dans le code ;
  - des branchements inadéquats (if, while,...) ;
  - des actions inappropriées ou manquantes.

Les fautes en cachent d'autres...

```
1  #include<stdio.h>
2  void f(int ,int);
3  void g(int ,int);
4  int main()
5  {
6   int a,b;
7   scanf("%d",&a);
8   scanf("%d",&b);
9   if(a>b)    // should be a>=b
10            f(a,b);
11    else    g(a,b);
12 }
13 void f(int x,int y)
14 {
15 if(x==y)printf("incorrect result");
16 }
17 void g(int x,int y)
18 {
19 if(x<y)printf("x lt y");
20 }
```

Les fautes en cachent d'autres...

```
1  #include<stdio.h>
2  void f(int ,int);
3  void g(int ,int);
4  int main()
5  {
6   int a,b;
7   scanf("%d",&a);
8   scanf("%d",&b);
9   if(a>b)   // should be a>=b
10           f(a,b);
11    else   g(a,b);
12 }
13 void f(int x,int y)
14 {
15 if(x==y)printf("incorrect result");
16 }
17 void g(int x,int y)
18 {
19 if(x<y)printf("x lt y");
20 }
```

Les fautes dans un programme peuvent être masquées par d'autres, ce qui rend leur localisation parfois difficile. Lorsque cela arrive, on parle de defect masking. Dans l'exemple, imaginez que la fonction f fournit un résultat incorrect si les deux entiers qu'elle prend en paramètre sont égaux. Il existe aussi une faute dans le programme, car normalement la ligne 9 aurait du être écrite avec une égalité. Dans ces conditions, même si les deux entiers du programme principal ont la même valeur, c'est g qui est exécuté au lieu de f et le problème relatif à f ne peut pas être détecté.

Cas de test

- Un cas de test est défini par un ensemble d'entrées, et un résultat de sortie attendu.Un cas de test peut être construit sur base
  - des exigences ;
  - du code ;
  - du domaine d'entrée de l'élément testé ;
  - de profils opérationnels ;
  - de modèles de fautes.

Cas de test

- Un cas de test correspond à un but/objectif : tester un chemin particulier dans le code, tester une exigence particulière(requirements), ...
- Deux grandes familles :
  - Test en boîte blanche(white box testing) $\rightarrow$ le code sert pour établir le(s) cas de test.
  - Test en boîte noire $\rightarrow$ on dérive les cas de test des exigences.
- Pour un système avec état, un cas de test peut être représenté par plusieurs tuples entrée(s)-sortie(s). Test d'un retrait sur un compte en banque : le solde a une valeur au préalable(pré condition). Le cas de test : TC = < <solde, 500.00>, <retirer, ‘‘montant ?’’>, <300.00, ‘‘300.00’’>, <solde, 200.00> >.

Un profil opérationnel

- Un profil opérationnel est par exemple une distribution de probabilité des entrées possibles lorsque le programme est en production. C'est donc une donnée numérique qui décrit comment le programme est utilisé.

Suite et plan de test

- Une suite de tests est un ensemble de cas de test.
- Un plan de test est un document décrivant
  - la portée ;
  - l'approche ;
  - les ressources et la plannification des activités à mettre en oeuvre pour tester.
- Il comprend notamment
  - les tâches à réaliser ;
  - qui les effectue ;
  - l'environnement de test ;
  - les techniques de test utilisées ;
  - le(s) critère(s) d'arrêt des tests.

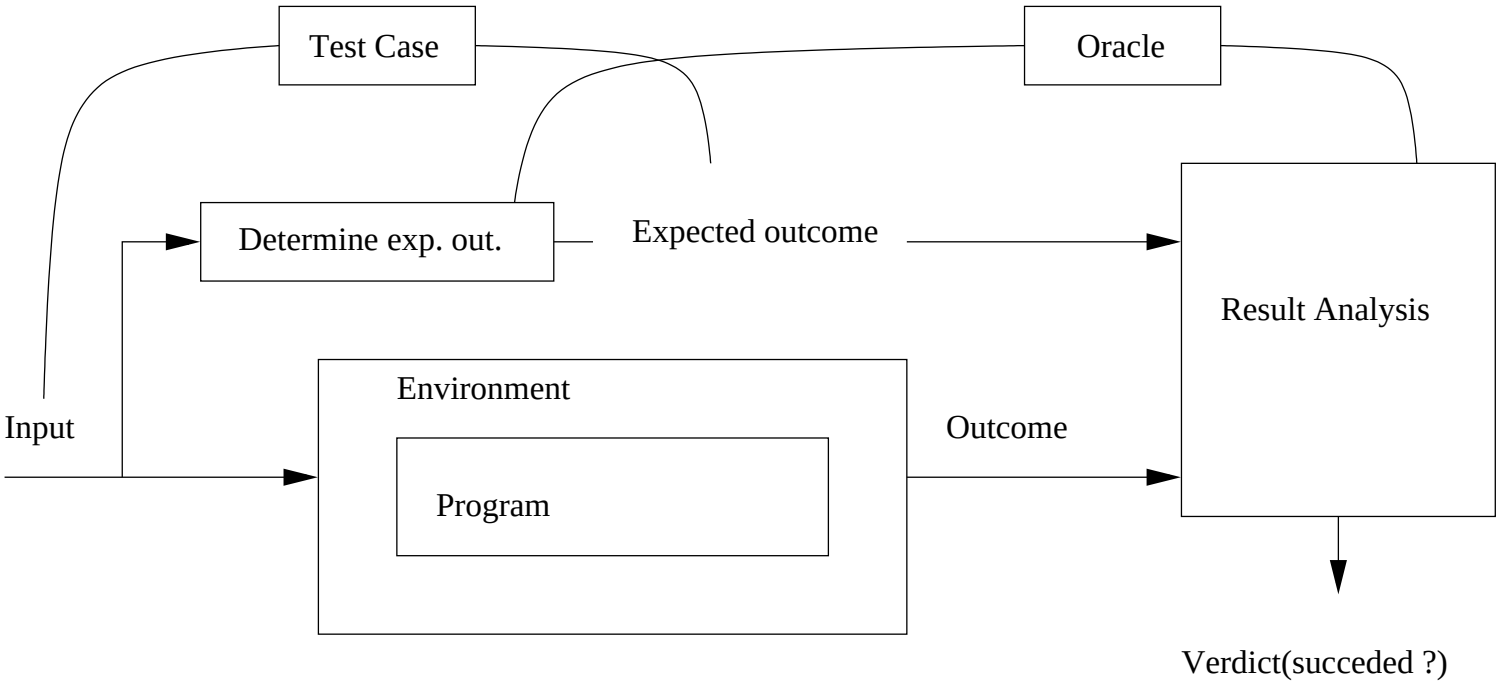
Validité et complétude des tests

- Complétude : les tests ne passent que sur des programmes corrects(Faux négatif - test négatif à tort - : la suite passe sur un programme incorrect).
- Validité : les tests n'échouent que sur des programmes incorrects(Faux positif - test positif à tort - est un test qui fait échouer un programme correct).

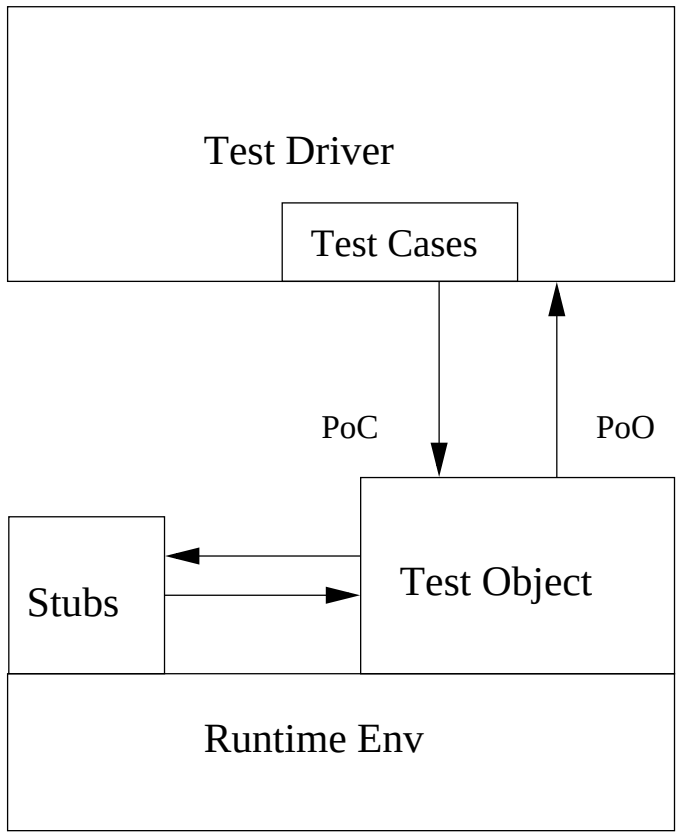
Un oracle

- Un oracle est l'élément, l'entité qui permet d'obtenir la sortie attendue d'un test, parfois au préalable à l'exécution du test.
  - L'oracle peut être une personne humaine. Les inconvénients sont la lenteur, le risque d'erreurs et la difficulté de déterminer la sortie pour des cas non triviaux.
  - L'oracle peut être un programme : par exemple, pour tester l'inversion d'une matrice, on utilisera un programme de multiplication de matrices. Si A est la matrice à inverser, A^{-1} la matrice inversée alors on vérifiera $A \times A^{-1} = I$

Les activités de base pour tester



Environnement de test



Environnement de test

- Les stubs ont pour but de remplacer une partie du système, du code pour faciliter la mise en place du scénario de test. Ils ne peuvent pas faire échouer le test.
- Le terme test harness englobe le driver et les stubs et correspond à l'environnement de test nécessaire pour les conduire.
- Le test driver est chargé de faire exécuter les cas de test de manière isolée.

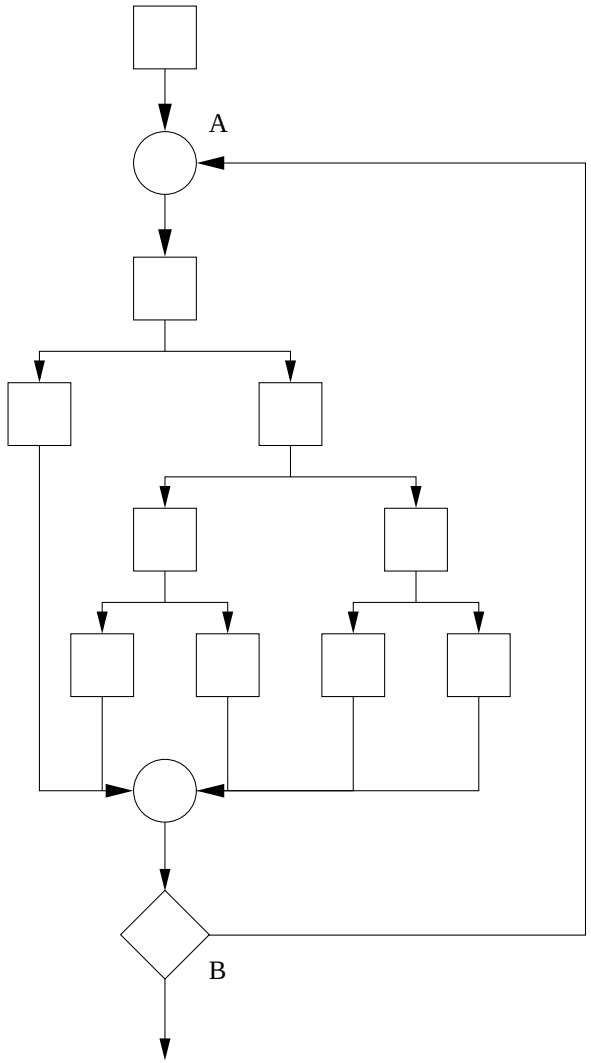
Difficulté de tout tester : exemple 1

Le domaine d'entrée(Input Domain) est l'ensemble de toutes les entrées possibles d'un programme. Pour la fonction

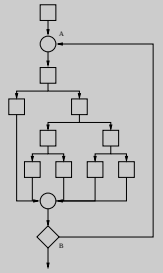
```
1 int sum(int a, int b){
2   return a+b;
3 }
```

tester l'ensemble des entrées possibles si un entier est codé sur 32 bits revient à tester $2^{32} \times 2^{32}$ cas. Si on considère qu'un cas de test met environ 10^{-9} secondes pour s'exécuter, il faudrait $1,8 \times 10^{19} \times 10^{-9}$ secondes pour tout tester, soit ≈ 570 ans.

Difficulté de tout tester : exemple 2



(source : The Art of Software Testing - Myers)



(source : The Art of Software Testing - Myers)

L'exemple porte sur l'application de tests en boîte blanche, c'est à dire en considérant le code pour dériver les cas de tests. Le graphe de contrôle correspond à une boucle do...while dans laquelle se trouve quatre if indépendants. On pourrait se dire que si on ne teste pas tous les chemins possibles entre le point A et le point B(le problème est alors de déterminer quelles sont les données en entrée pour remplir ce critère), le programme correspondant au CFG n'a pas été suffisement testé. Le nombre de chemins, si on ne fait qu'un tour de boucle, est de 5^1. Si on fait deux tours, il est de $5^1 \times 5^1 = 5^2$. Le nombre de chemins total $= 5^1 + 5^2 + ... + 5^{20} \approx 10^{14}$. Si 1 test prend 10^-9 seconde pour être généré, exécuté et vérifié, alors il faudra ≈ 27 heures pour tester tous les chemins. A raison d'un test par seconde, il faut $\approx 3 \times 10^6$ années pour tout tester !

Pourquoi tester ?

- Pour chercher des défauts.
- Pour mesurer la qualité du logiciel.

Un programme est correct s'il se comporte comme attendu pour chaque donnée de son domaine d'entrée. Il l'est ou pas (résultat binaire)au regard des exigences spécifiées dans un document.

- L'objectif de tester n'est presque jamais de vérifier complètement qu'un programme est correct !
- Pour cela, il faut établir une preuve mathématiquement sur base d'une spécification formelle des exigences.

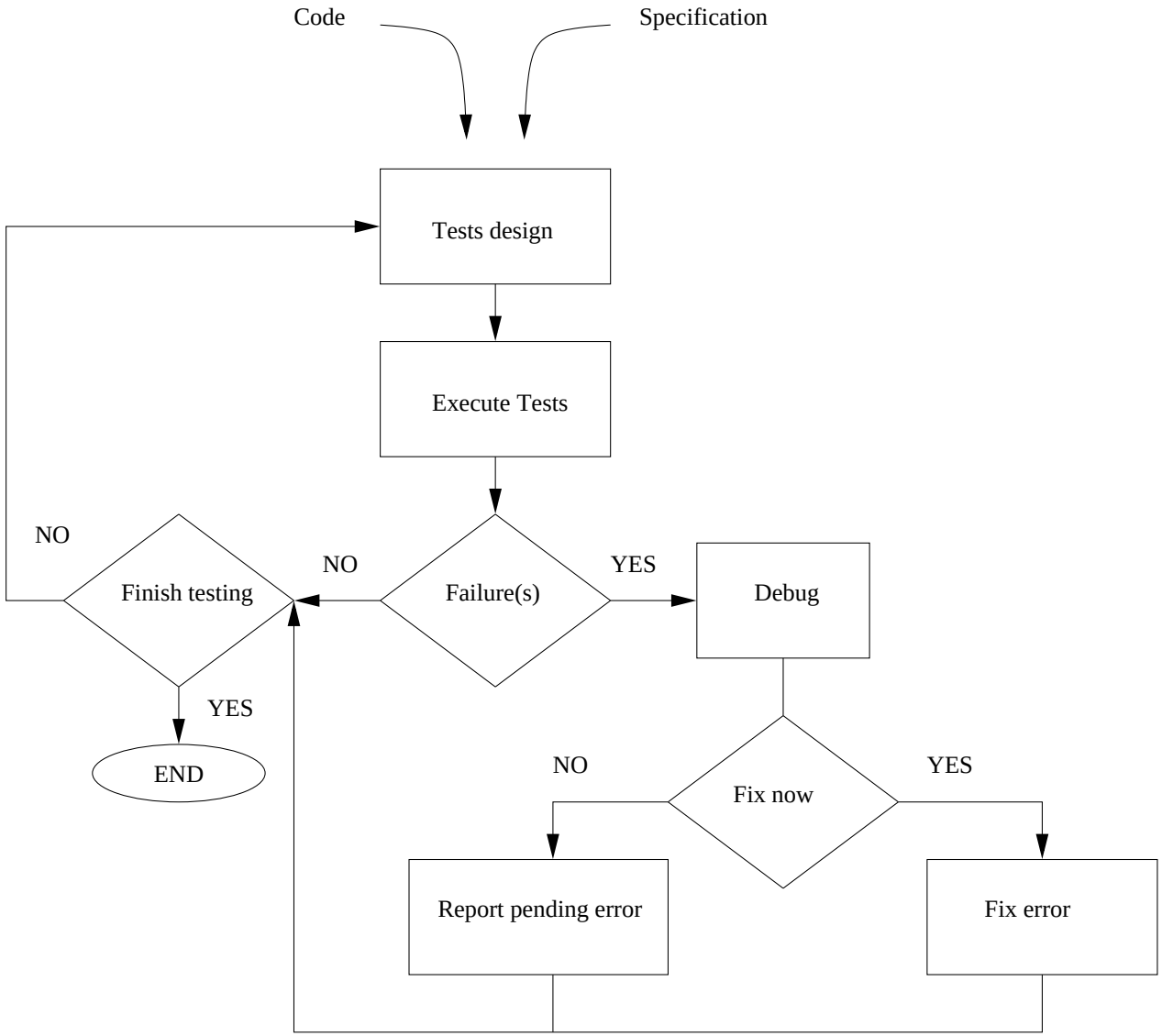
Les tests et la qualité

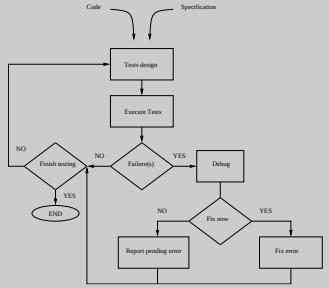
- Tester contribue à améliorer la qualité du logiciel que l'on construit. La qualité d'un logiciel se fait aussi par les aspects suivants :
  - fonctionnel(i.e. correct, adéquat,interropérable) ;
  - fiable ;
  - facilité d'utilisation(i.e. d'apprentissage, de compréhension) ;
  - facilité de maintenance(i.e. ajouts et analyse faciles, testable) ;
  - portabilité(i.e. facilité d'adaptation, d'installation sur un autre environnement) ;
  - efficience.
- Il existe plusieurs normes qui traitent de la qualité des logiciels. L'ISO 25000 (SQUARE) reprend l' ISO 9126 qui définit les exigences relatives à la qualité des logiciels et l'ISO 14598 qui traite de l'évaluation de ces exigences.

Tester n'est pas debugger

- Debugger est la tâche qui consiste à localiser et corriger des fautes dans le code
- Tester consiste à chercher à mettre en évidence des anomalies le plus souvent en exécutant le code.
- Tester ne permet pas
  - d'identifier la cause des erreurs ;
  - de les corriger ;
  - de prouver quoi que ce soit quand à la correction effectuée.

Tester et debugger





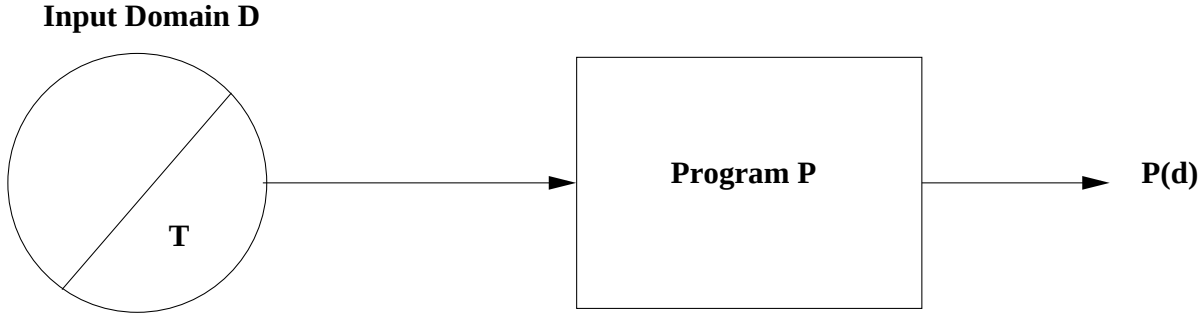
Le processus de debuggage peut être inductif ou déductif. Pour plus d'informations : The Art of Software Testing - Myers.

Tester n'est pas debugger

- Tester peut être effectué par quelqu'un d'extérieur au code, alors que débugger pas.
- Les tests doivent être plannifiés et "designés".
- La génération de cas de tests et l'exécution des tests peuvent être automatisés. Automatiser le processus de debuggage paraît compliqué.
- Les fondements et les limites des tests sont assez bien connus. La/une théorie relative au debugging est quasi inexistante dans la littérature.

Notion de test idéal :Théorie de Goodenough et Gerhart

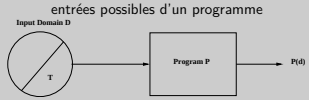
Le domaine d'entrée(input domain) est l'ensemble de toutes les entrées possibles d'un programme



- $OK(d)$ est un prédicat qui exprime si le résultat de l'exécution du programme est correct ou non. $OK(d) = true$ si et seulement si P(d) donne un résultat acceptable.
- Pour un ensemble $T \subseteq D$, $SUCCESSFUL(T) = true$ ssi $\forall t \in T$, $OK(t)$.
  Un test idéal : T constitue un test idéal si $OK(t) \forall t \in T \Rightarrow OK(d) \forall d \in D$.

Notion de test idéal :Théorie de Goodenough et Gerhart

Le domaine d'entrée(input domain) est l'ensemble de toutes les entrées possibles d'un programme



- $OK(d)$ est un prédicat qui exprime si le résultat de l'exécution du programme est correct ou non. $OK(d) = true$ si et seulement si P(d) donne un résultat acceptable.
- Pour un ensemble $T \subseteq D$, $SUCCESSFUL(T) = true$ ssi $\forall t \in T$, $OK(t)$.
  Un test idéal : T constitue un test idéal si $OK(t) \forall t \in T \Rightarrow OK(d) \forall d \in D$.

- Un test idéal signifie qu'après avoir exécuté le programme sur une partie du domaine d'entrée T on conclut que le programme ne contient pas d'erreurs. Dans ce cas, T est qualifié de test idéal. Tout dépend évidemment si T teste le programme de manière approfondie. Et là est le problème. En fait, lorsque $T = D$, le programme est testé de manière approfondie puisque complètement. Mais on sait que ce n'est presque jamais possible de procéder de la sorte...
- On notera que le domaine en entrée et le programme lui-même peuvent évoluer au cours du temps. Lorsque le programme évolue, on peut imaginer les cas de figure suivants : de nouvelles fonctionnalités sont ajoutées et/ou des modifications sont effectuées sur le code existant. Il en resulte que le domaine pour l'ensemble a plus que probablement changé.

Critère valide et fiable

On note 2^D Un ensemble de sous ensembles de D. Un ensemble de tests est spécifié en utilisant un critère C de sélection de cas de test. C correspond en fait aux propriétés du programme que l'on veut tester

- Critère fiable : soit tous les tests sélectionnés par C sont ok ou aucun ne l'est.
  $RELIABLE(C) \equiv (\forall T1)_{2^D}(\forall T2)_{2^D}(C(T1) \wedge C(T2) \supset (SUCCESSFUL(T1) \equiv SUCCESSFUL(T2)))$
- Critère valide : si P a des fautes, C sélectionne au moins un T pas ok pour P.
  $VALID(C) \equiv (\exists d)_D(\neg OK(d)) \supset (\exists T)_{2^D}(C(T) \wedge \neg SUCCESSFUL(T))$

Théorie de Goodenough et Gerhart

Théorème

$(\exists C)(VALID(C) \wedge RELIABLE(C) \wedge (\exists T)_{2^D}(C(T) \wedge SUCCESSFUL(T))) \supset SUCCESSFUL(D)$

En considérant le prédicat $COMPLETE(T, C)$

$(\exists T \subseteq D)(COMPLETE(T, C) \wedge RELIABLE(C) \wedge VALID(C) \wedge SUCCESSFUL(T)) \Rightarrow (\forall d \in D) OK(d)$

Théorème
$(\exists C)(VALID(C) \wedge RELIABLE(C) \wedge (\exists T)_{2^D}(C(T) \wedge SUCCESSFUL(T))) \supset SUCCESSFUL(D)$
En considérant le prédicat $COMPLETE(T, C)$
$(\exists T \subseteq D)(COMPLETE(T, C) \wedge RELIABLE(C) \wedge VALID(C) \wedge SUCCESSFUL(T)) \Rightarrow (\forall d \in D)OK(d)$

C un ensemble de prédicats de tests.
$COMPLETE(T, C) \equiv (\forall c \in C)(\exists t \in T)c(t) \wedge (\forall t \in T)(\exists c \in C)c(t)$

P	Q	$P \Rightarrow Q$
0	0	1
0	1	1
1	0	0
1	1	1

Pour démontrer le théorème, il suffit de montrer que si Q est faux, P ne peut être que toujours faux et donc que toute la proposition est toujours vrai. On suppose donc que $\neg OK(d)$. $VALID(C)$ implique qu'il existe un T tel que $\neg SUCCESSFUL(T)$. $RELIABLE(C)$ implique que si un T ne réussit pas alors tous les autres T que l'on aurait pu sélectionner ne réussissent pas. Et donc P vaut toujours faux.

Exemple

Soit un programme qui calcule $P(d) = d \mod 7$. Si, par erreur, $P(d) = d \mod 3$, on a $OK(21k+0)$ et $OK(21k+1)$

- $C_1(T) \equiv (T = \{1\}) \vee (T = \{2\})$: fiable mais non valide.
- $C_2(T) \equiv (T = \{t\}) \wedge (T \in \{1,2,3,4,5,6\})$: valide mais non fiable.
- $C_3(T) \equiv (T = \{t\}) \wedge (T \in \{3,4,5,6,7,8\})$: fiable et valide.
- $C_4(T) \equiv (T = \{t, t+1, t+2, t+3\}) \wedge (t \in D)$: fiable et valide.

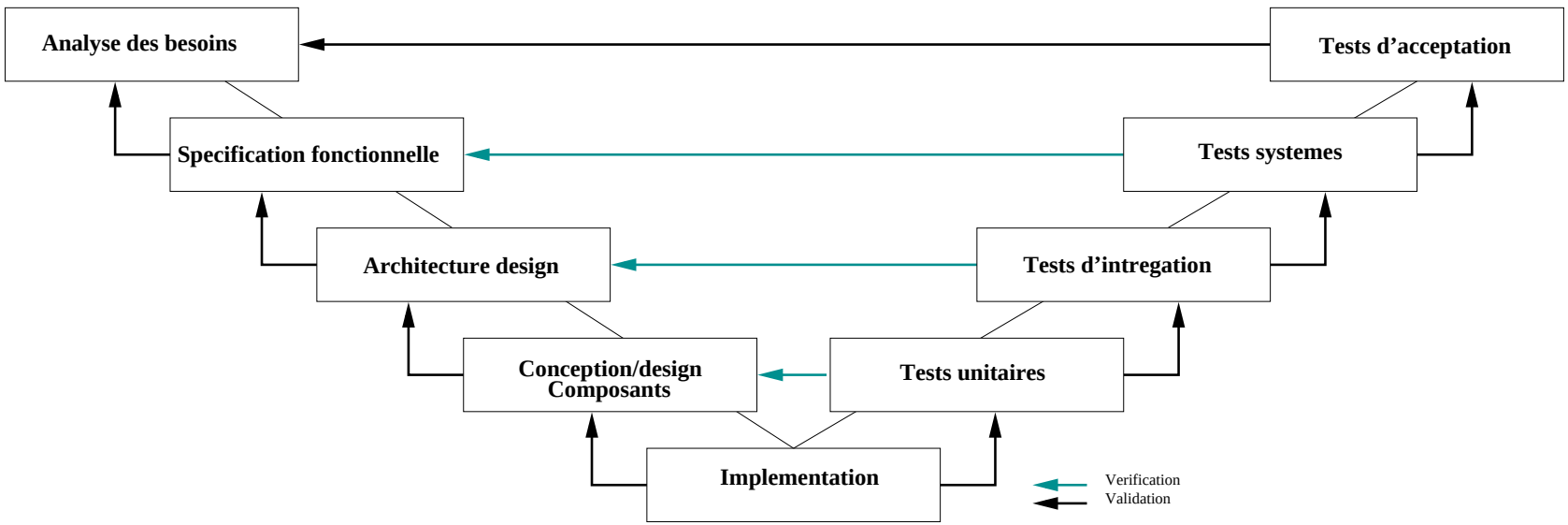
Praticable ?

- Les erreurs ne sont pas connues $\Rightarrow$ on ne sait pas prouver la validité et la fiabilité.
- Si C sélectionne $D \Rightarrow$ ok mais impraticable.
- La validité et la fiabilité sont suceptibles de changer suite au débuggage.

A retenir

- Il n'existe pas de méthode qui permette de conclure après un test réussit que le programme ne contienne aucune faute.
- Il n'est pas possible d'automatiser sur base d'un programme et de D la génération d'un $T \subset D$ tel que $(\forall t)\, T(OK(t)) \supset (\forall d) D(OK(d))$
- Il existe d'autres théories relatives aux tests dont notamment celle de Gourlay mais aussi de Weyuker et Ostrand.

Les tests dans le modèle de cycle de vie du projet



Validation ou vérification ?

- Validation : a pour but de vérifier si le logiciel créé rencontre les spécifications le concernant et rempli l'objectif pour lequel on l'a construit.
- Vérification : a pour but de répondre à la question : "a-t-on réalisé tout ce qui avait été prévu de réaliser ?". La vérification permet donc de s'assurer qu'une phase particulière du développement a été menée correctement et complètement selon sa spécification.
- Test = verification partielle.

Classification des tests

- Selon les techniques utilisées et l'entrée :
  - Tests en boîte blanche vs en boîte noire.
  - Exigences(requirements), code.

Classification des tests - suite

- Selon l'objectif recherché, les tests :
  - de charge ;
  - d'utilisabilité ;
  - de robustesse par rapport aux entrées invalides.

Classification des tests - suite II

- Selon la phase du projet, les tests
  - unitaires ;
  - d'intégration ;
  - fonctionnels ;
  - d'acceptation.
- Mais aussi
  - de régression (maintenance) :les tests de régression consistent à sélectionner et exécuter des cas de test existants pour s'assurer que de nouvelles fautes n'ont pas été introduites ;
  - manuels vs automatisés ;
  - statiques vs dynamiques.

Classification des tests - suite II

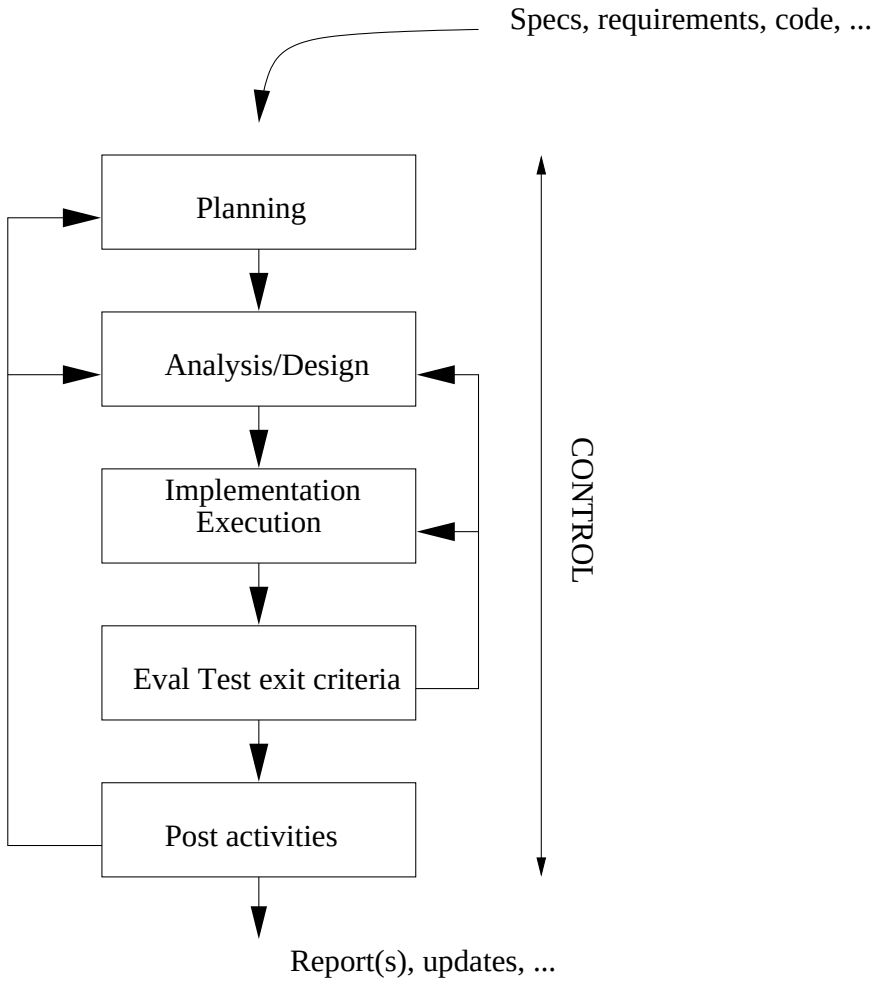
- Selon la phase du projet, les tests
  - unitaires ;
  - d'intégration ;
  - fonctionnels ;
  - d'acceptation.
- Mais aussi
  - de régression (maintenance) :les tests de régression consistent à sélectionner et exécuter des cas de test existants pour s'assurer que de nouvelles fautes n'ont pas été introduites ;
  - manuels vs automatisés ;
  - statiques vs dynamiques.

Pour les tests relatifs aux phases d'un projet, l'ordre dans lequel ils sont mis en oeuvre n'est absolument pas précisé. Ainsi, il est tout à fait possible d'élaborer des tests d'acceptation avant même d'avoir implémenté la moindre classe. Lors de l'implémentation, il est aussi tout à fait possible de créer une classe et la tester sans pour autant disposer des classes dont elle dépend. Nous reviendrons sur ce point surprenant par la suite.

Tests statiques/dynamiques

- Le test statique signifie tester un système(au niveau des exigences ou du code) sans en exécuter le code. Cela peut signifier
  - un examen du code par un ensemble de personnes sur base d'éléments déterminés à l'avance ;
  - un parcours systématique de portions de code dans le but de les comprendre et les analyser ;
  - utiliser des outils d'analyse du code source.
- Un test dynamique est un test qui nécessite l'exécution de l'élément soumis au test.

Le processus de test



La plannification

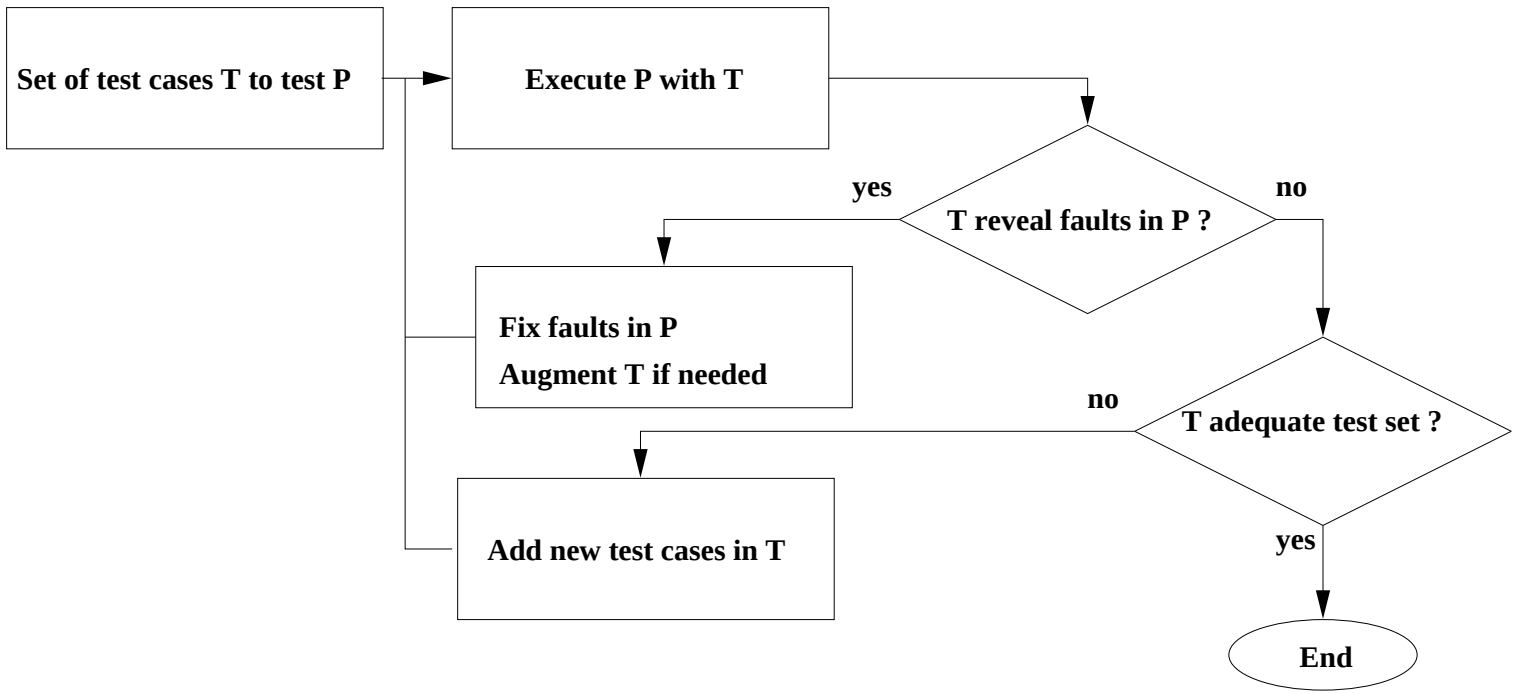
- Planning :
  - Détermination des ressources et plannification.
  - Elaboration d'une stratégie de test sur base des sous éléments du système.
  - Détermination de l'intensité et des critères d'arrêt des tests.
  - Prioritisation des tests.
  - ...

La clôture

- Clôture des activités :
  - Ce qui a été prévu a-t-il été achevé entièrement ?
  - Que s'est-il passé de non prévu ? Pourquoi ?
  - ...

Adéquation d'une suite de tests

- Le problème lorsqu'on teste est que l'on voudrait savoir si suffisement de tests ont été menés. Pour cela, on fixe un critère et la suite de tests doit satisfaire ce critère.



Quand arrêter ?

- Il arrive également que l'on arrête de tester pour les raisons suivantes :
  - Le délai éventuellement imparti au préalable pour les tests a expiré.
  - Le version du logiciel doit être publiée.
  - Les cas de test prévus et ajoutés ont tous été exécutés et plus aucune faute n'a été révélée.

Tests manuels et tests automatisés

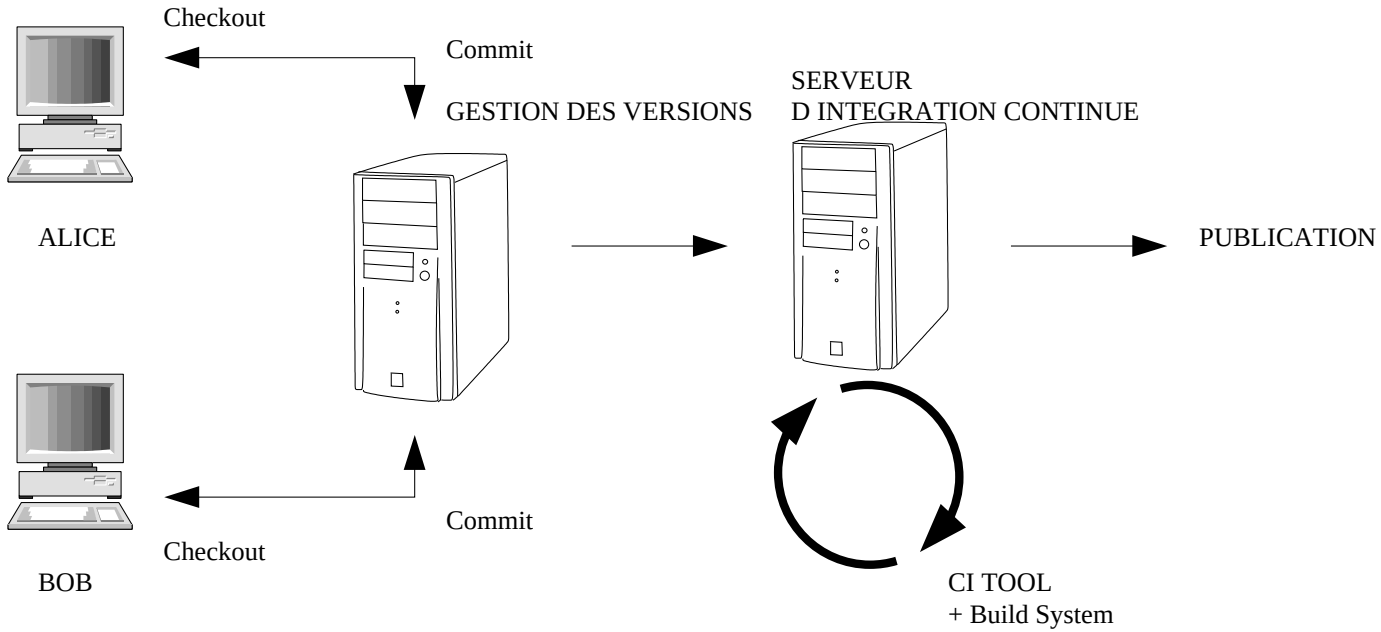
- Un test manuel(vs test automatisé)est un test pour lequel une intervention humaine est nécessaire pour lancer le test et décider si le test a réussit ou pas. Dans un test manuel, la vérité logique "test réussit" est calculée par le testeur.
- En fonction du degré d'automatisation, la batterie de tests peut être lancée automatiquement par un outil et les résultats vérifiés par un opérateur(peu courant) ou la batterie peut être lancée manuellement et le rapport de tests générés automatiquement.
- En général, les tests automatisés sont accumulés au fur et à mesure du temps.
- Les tests manuels sont parfois associés aux tests dans l'environnement réel.
- Les tests automatisés s'inscrivent dans un processus d'intégration continue.

Tests manuels et tests automatisés

- Quelques avantages des tests automatisés :
  - Améliore la couverture pour les tests de régression.
  - L'accumulation de cas de tests.
  - La localisation des fautes est facilitée lors des tests de régression.
  - Le coût en temps pour réexécuter le(s)test(s) faible par rapport aux tests manuels.
  - Réduction des coûts pour la maintenance.
  - Les cas de tests sont clairement définis.

L'intégration continue

- L'intégration continue est une pratique de développement qui consiste principalement à intégrer le travail d'une équipe le plus fréquemment possible. L'objectif, en procédant de la sorte, est de détecter les fautes le plus rapidement possible pour éviter une accumulation des problèmes.



Mettre en place l'intégration continue

- Partager le code source dans un dépôt.
- Enregistrer(commit) régulièrement(par jour) les ajouts/modifications.
- Ecrire des tests d'intégrations.

En général, les processus de compilation et de déploiement sont automatisés.

- Partager le code source dans un dépôt.
- Enregistrer(commit) régulièrement(par jour) les ajouts/modifications.
- Ecrire des tests d'intégrations.

En général, les processus de compilation et de déploiement sont automatisés.

Il existe plusieurs logiciels permettant de mettre en place des principes d'intégration continue. A titre d'exemples :

- CruiseControl ;
- Hudson ;
- Jenkins ;
- Apache Continuum.

Développement dirigé par les tests

- L'idée du TDD est de conduire le développement par les tests. On écrit d'abord un test puis le code de production correspondant.
- Il faut respecter trois principes lorsqu'on utilise le développement dirigé par les tests :
  - Il ne faut pas écrire de code de production sans avoir écrit au préalable un test qui échoue pour ce code.
  - Il suffit d'écrire uniquement un test qui échoue pour le code de production que l'on compte développer.
  - Il ne faut pas écrire du code de production qui nécéssiterait plus que le test qui a été élaboré au préalable.

Avantages du TDD

En respectant les trois principes énoncés,

- on minimise le temps entre le développement des tests et l'implémentation du code effectif $\Rightarrow$ les erreurs sont détectées et corrigées plus tôt, ce qui permet de réduire les coûts ;
- le programmeur réfléchit sur comment utiliser le code $\Rightarrow$ cela participe aux questions de design.

En respectant les trois principes énoncés,

- on minimise le temps entre le développement des tests et l'implémentation du code effectif ⇒ les erreurs sont détectées et corrigées plus tôt, ce qui permet de réduire les coûts ;
- le programmeur réfléchit sur comment utiliser le code ⇒ cela participe aux questions de design.

Ce n'est pas toujours possible de procéder de la sorte : par exemple si on hérite d'une classe abstraite et la classe fille ne peut être abstraite. Dans ce cas, il faut implémenter les méthodes pour que la compilation passe ou fournir une implémentation minimale.

En résumé

- Les tests montrent la présence de fautes, par leur absence.
- Tester de manière exhaustive n'est pas possible.
- Tester doit se faire de manière planifiée et le plus tôt possible.
- L'intensité des activités de test dépend du contexte et donc du type d'application.