Secteur Tertiaire Informatique
Filière étude - développement

Développer des composants d'interface

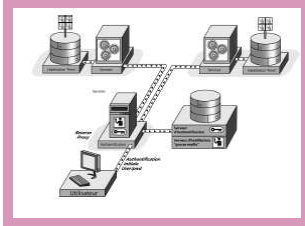
Les tests unitaires

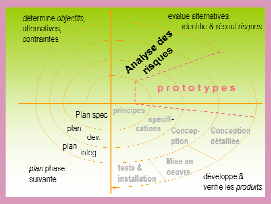
Accueil

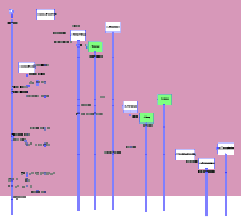
Apprentissage

Période en entreprise

Evaluation







Code barre

SOMMAIRE

I APPROCHE DU TEST 4

I.1 DEFINITION 4
I.2 TEST ET CYCLE DE VIE DU LOGICIEL 5
I.3 TERMINOLOGIE 7

II LA DEMARCHE DE TESTS UNITAIRES 8

II.1 UNE APPROCHE STATIQUE 8
II.2 UNE APPROCHE DYNAMIQUE 9
II.3 QUAND ARRETER LES TESTS ? 10

III COMMENT CHOISIR LE JEU DE TEST ? 11

III.1 METHODES « BOITE NOIRE » 11
III.1.1 La recherche intuitive d'erreurs 11
III.1.2 Les classes d'équivalence 11
III.1.3 Les tests aux limites 13
III.1.4 Tester plusieurs variables 14
III.1.5 Le graphe de causes à effets 14
III.2 METHODES « BOITE BLANCHE » 22
III.2.1 Analyse du flot de contrôle 23
III.2.2 Analyse du flot de données 25

I APPROCHE DU TEST

Le vol inaugural d'Ariane 5 qui eut lieu le 4 juin 1996 s'est soldé par un échec. Environ 40 secondes seulement après le démarrage de la séquence de vol, le lanceur, qui se trouvait alors à une altitude de quelques 3700 mètres, a dévié de sa trajectoire, s'est brisé et a explosé.
Analyse de l'échec :
Première erreur technique : Le code contient tout bêtement une affectation d'une donnée 64 bits vers une donnée 16 bits.
Deuxième erreur technique : Aucun mécanisme de gestion d'erreur n'est prévu.
L'exception est transmise au cœur du contrôle de vol, qui la traite comme n'importe quelle autre valeur fournie par le calculateur.
Il s'en est résulté un comportement aberrant, qui a entrainé une déviation de trajectoire, et son autodestruction.

Une erreur dans n'importe quelle étape de la fabrication d'un produit (spécification, conception, programmation) va avoir pour conséquence un défaut dans le logiciel qui entrainera une anomalie de fonctionnement.
La non qualité a un coût :

- Pendant le développement
  - Cout des reprises
  - Délais de livraison
  - Fatigue
  - Moral de l'équipe
- Après la livraison
  - Retour de produit
  - Coût du support client
  - Perte d'affaires, de part de marché
  - Dégradation de l'image de marque

L'assurance qualité permet de mettre en œuvre un ensemble de dispositions qui vont être prises tout au long des différentes phases de fabrication d'un logiciel pour accroître les chances d'obtenir un logiciel qui corresponde à ses objectifs (son cahier des charges).
La définition et la mise en place des activités de test ne sont qu'un sous-ensemble des activités de l'assurance qualité, et le test aura pour but de minimiser les chances d'apparition d'une anomalie lors de l'utilisation du logiciel.

I.1 DEFINITION

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

Définition donnée par
IEE STD 729: standard glossary of software engineering terminology
La pratique du test correspond donc à 2 objectifs :

- **La vérification** : consiste à vérifier que l'on a bien fait le logiciel (est-ce que le logiciel fonctionne correctement ?)
- **La validation** : consiste à vérifier qu'on a fait le bon logiciel (est-ce que le logiciel réalise les fonctions attendues ?)

Mais n'a pas pour objectif :

- De corriger le défaut détecté
- De prouver la bonne exécution d'un programme

Le test peut être **manuel** : le testeur entre les données de test au moyen d'une interface par exemple, lance les tests, observe les résultats et les compare avec les résultats attendus.
L'opération est fastidieuse, avec possibilité d'erreur humaine, et difficilement gérable pour les grosses applications.

Le test peut être **automatisé** : de nombreux outils été créés afin de minimiser les risques d'erreurs de programmation et de vérifier la conformité du produit intermédiaire ou final avec le cahier des charges. La capacité du produit à résister à différentes montées en charge fait également partie de la batterie de tests qu'une solution se doit de passer avant d'être mise sur le marché ;
3 catégories d'outils existent :

- Outils de tests reposant sur l'exécution du code source
- Outils de tests fonctionnels reposant sur l'analyse des spécifications de tout ou partie du logiciel, sans tenir compte de sa programmation intrinsèque
- Outils relatifs à la performance des logiciels développés, que ces derniers soient des applications Web, intranet ou des Web services.

Aujourd'hui, le test fait l'objet d'une pratique artisanale ; demain, le test devrait tendre vers la rigueur, basé sur des modèles et des théories et s'automatiser.

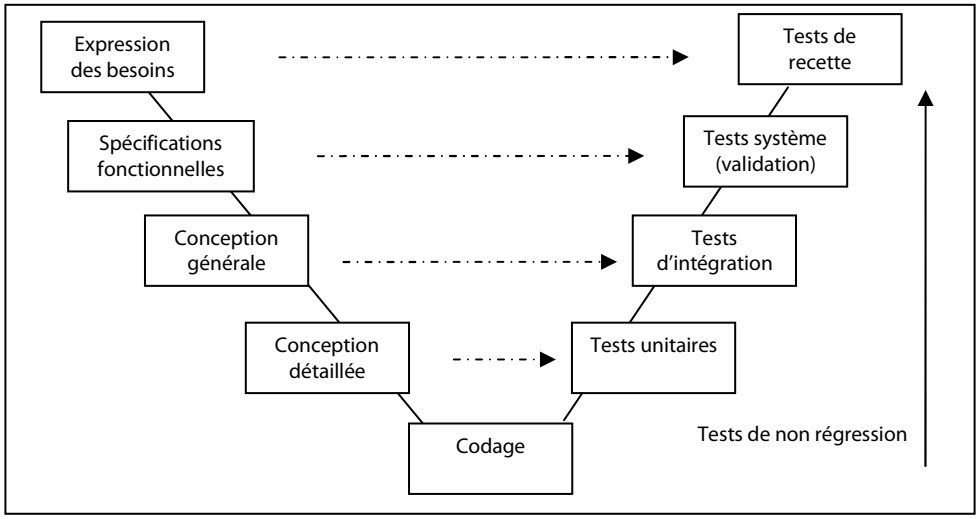
I.2 TEST ET CYCLE DE VIE DU LOGICIEL

Le Comité Français du Test Logiciel identifie 4 niveaux de tests :

- Les **tests unitaires** vérifient la conformité des unités

  On s'assurera que les composants logiciels, pris un à un sont conformes à leurs spécifications et prêts à être intégrés.
- Les **tests d'intégration** visent à démontrer que les unités communiquent et interagissent de manière correcte, stable et cohérente
- Les **tests système** (ou validation) qui permettent de tester les fonctionnalités et les exigences non fonctionnelles du produit (interface, performance, utilisabilité, sécurité, robustesse …).
- Les **tests de recette** qui permettent au client de reconnaitre que le produit livré par le fournisseur est conforme à la commande passée, qu'il est exploitable dans le système d'information de l'entreprise, et qu'il peut être mis à la disposition des utilisateurs.

Et tout au long de la réalisation du produit, les **tests de non régression** vérifieront que les nouvelles fonctionnalités apportées au produit n'altèrent pas les fonctionnalités déjà en place.
Les tests dépendent du cycle de développement : dans le cycle en V par exemple, on prépare et on anticipe les tests dans les phases descendantes.



Les phases de test dans le cycle en V

Dans le cas des applications orientées objet, toute intégration est représentée par un ou plusieurs objets : Il est souvent difficile d'établir une séparation nette entre les tests de composants et les tests d'intégration.

Dans un **cycle de développement incrémental**, les tests unitaires et d'intégration sont menés sur chaque incrément d'une itération, les tests système sont menés à la fin de l'itération.

La tendance à abandonner les modèles séquentiels au profit des modèles incrémentaux vise à éviter l'intégration tardive du logiciel, ce qui rejoint l'idée **d'eXtreme Programming** qui vise à maintenir le logiciel dans un état intégré permanent. L'intégration continue signifie que tous les changements même minimes sont intégrés au fur et à mesure, ce qui impose des tests de non régression automatisés, aucune équipe ne pouvant tester le logiciel en permanence.

Le développement piloté par les tests (Test Driven Development) se caractérise par les points suivants :

- Ecriture du code test avant toute écriture de code
- Ecriture du code qui va satisfaire aux exigences du test : si le test marche, le code est complet.
- Le développement se déroule alternant les tests et le codage, par pas maximum de 10 minutes.
- Lors de l'intégration du code dans l'application, les tests unitaires doivent être OK.

I.3 TERMINOLOGIE

Un jeu de test (série de données de tests comprenant les données d'entrée et les résultats attendus) est réussi si toutes les données de test qu'il contient produisent des résultats corrects.

Un cas de test spécifie une configuration des données d'entrée et de sortie permettant de vérifier un point particulier des spécifications du logiciel (par exemple, une règle de gestion)

Un scénario de test est un enchainement de cas de tests

Une procédure de test peut spécifier les conditions d'exécution d'un ou plusieurs cas de test (liste ordonnée des actions à exécuter par l'opérateur pour passer un cas de test).

Un cas de test produit un résultat, qui doit être évalué par un Oracle.

Oracle : ce qui permet de dire si un test est bon ou non : l'oracle est humain, logiciel ou résultat d'un processus qui, connaissant le fonctionnel sait valider les résultats obtenus.

Le **niveau de gravité** d'une anomalie peut-être :

- Bloquant : les tests ne peuvent être poursuivis
- Majeur : les tests peuvent se poursuivre en explorant d'autres aspects
- Mineur : sans conséquence sur les tests, la correction se fera ultérieurement

Le **critère d'arrêt** est déterminé par une règle prédéfinie précisant les conditions d'arrêt des campagnes de test en fonction des anomalies.

II LA DEMARCHE DE TESTS UNITAIRES

Les erreurs de codage sont inévitables et même les programmeurs les plus expérimentés se trompent de temps en temps. Vous devez donc vous attendre à ce que le code que vous venez de développer contienne des erreurs.

Le but de cette étape est de tester les composants implémentés en tant qu'unités individuelles, ce qui veut dire que chacun des tests ne porte pas sur l'ensemble de l'application, mais sur des composants de type fonction ou procédure dans un langage procédural, fenêtre dans un environnement graphique, classe ou son instance dans une application orientée objet.

Un bon test unitaire se compose de vérifications statiques, de tests dynamiques préparés, identifiés et fournissant une trace (exécution, résultats), archivés et/ou *rejouables*.

II.1 UNE APPROCHE STATIQUE

Elle consiste en l'analyse textuelle du code du logiciel afin d'y détecter des erreurs, sans exécution du programme.

- La revue de code

La revue de code est un examen systématique du code source d'un logiciel, il peut être comparé au processus ayant lieu dans un Comité de lecture, l'objectif est de trouver des Bogues ou des vulnérabilités potentielles ou de corriger des erreurs de conception afin d'améliorer la qualité et la sécurité du logiciel [source : Wikipédia]

La revue de code peut être mise en œuvre sous 2 formes :

- La lecture simple

  Elle met en relation deux intervenants, un auteur et un lecteur : l'auteur fournit son code au lecteur qui le lit, recherche les erreurs et écrit ses remarques sur le document ; l'auteur effectuera les corrections nécessaires à la lecture du document.

  L'auteur et le lecteur appartiennent à la même équipe, ce qui permet l'échange de rôles.

  La recherche d'erreurs se fait tous azimuts, sans en privilégier un type particulier.

- La lecture croisée

  Elle met en relation des intervenants, un auteur et des lecteurs : l'auteur fournit son code aux lecteurs qui étudient le code ; au cours d'une réunion, une liste des erreurs à corriger est synthétisée.

- L'inspection

  Elle met en relation plusieurs intervenants, un auteur et des inspecteurs: l'auteur fournit son code aux lecteurs qui recherchent les erreurs en y joignant le type d'erreur à détecter; au cours d'une réunion chaque inspecteur indique à l'auteur les erreurs à corriger.

Les inspecteurs (1 à 2 maximum) ne font généralement pas partie de l'équipe de développement.

La recherche d'erreurs ne se fait pas tous azimuts, mais est orientée sur des types d'erreur précis

Elle présente l'inconvénient d'un coût plus élevé, et impose la nécessité de planifier les interventions extérieures.

Le choix de l'une ou l'autre méthode va être fonction de la taille de l'équipe et de l'importance des exigences qualité.

- L'analyse du graphe de contrôle

Le graphe de contrôle modélise la structure interne du logiciel et est généralement dérivé de l'organigramme du programme (cf § Méthodes boite blanche).

Analyser le graphe de contrôle du programme peut révéler des erreurs (exemples: sauts anormaux, code inutilisé...).

Ces méthodes statiques sont efficaces et peu coûteuse : 60 à 95% des erreurs sont détectées lors de contrôles statiques [Laprie95], mais ne sont pas suffisantes : elles ne permettent pas de valider le comportement du programme durant son exécution.

II.2 UNE APPROCHE DYNAMIQUE

Elle consiste en l'exécution de l'unité à valider à l'aide d'un jeu de tests, sélectionné à partir des spécifications du composant. Elle vise à détecter des erreurs en confrontant les résultats obtenus par l'exécution du programme à ceux attendus par la spécification de l'application.

Le choix du jeu de test est essentiel à la réussite des tests ; Si le nombre de cas de tests est trop limité, les tests seront incomplets et des erreurs risquent d'être oubliées. Si le nombre de cas de tests est trop élevé, les résultats seront redondants, du temps (donc de l'argent !) sera perdu.

Le plan de test le plus approprié doit être choisi.

L'activité de test comporte :

- Une étape de définition des entrées (appelées 'données de test') qui seront fournies au logiciel pendant une exécution
  Exemple : DT = {x=3, y=2},
- Une étape d'exécution,
- Une étape d'observation des résultats, qui a pour objectif de répondre à la question : l'exécution a-t-elle retourné le bon résultat ? c'est le rôle de l'oracle qui décide du succès ou de l'échec du jeu de tests.
  Le jeu de tests sera un succès si chaque cas de test est un succès, il sera un échec si au moins un cas de test est un échec.

II.3 QUAND ARRETER LES TESTS ?

Un programme n'a plus besoin d'être testé, lorsque l'efficacité du jeu de tests sélectionné est conforme à certains critères de qualité, et lorsque ce jeu de tests est un succès.

Lors des tests unitaires classiques, on aborde la notion de couverture des tests : quelles proportions du code et des spécifications ont-elles été couvertes par les tests ?

La couverture par rapport aux spécifications indique dans quelle mesure les cas de test prennent en compte les spécifications du logiciel. La base de départ est par exemple représentée par des tableaux d'exigences testables, élaborés par le client ou l'analyste qui peuvent être traduites en tests unitaires.
La mesure de ce type de couverture est généralement obtenue par des inspections manuelles.

La couverture par rapport au code se rapporte au contrôle de flux du programme.

Certains outils du marché peuvent déterminer le pourcentage de couverture et identifier les parties de code non exécutées. Il peut être légitime d'utiliser ces informations, mais on peut douter de l'efficacité d'une recherche à tout prix d'une certaine valeur dont on pourrait se satisfaire.

Les résultats d'une mesure de couverture peuvent attirer l'attention sur des points faibles des tests effectués ; on peut par exemple distinguer différentes catégories de code non couvert :

- Du code non testé, mais qui devrait l'être
- Du code mort, qui devrait être éliminé
- Du code généré automatiquement qui n'est pas appelé dans l'application
- Du code qui ne pourrait être atteint par les tests qu'au prix de dépenses trop considérables, par exemple le code de traitement d'erreurs.

En conclusion, tester complètement un composant est impossible. Il faut avant tout prendre en compte les enjeux du projet (criticité, coût d'une défaillance), choisir le processus de test adapté aux enjeux et prévoir les moyens humains pour atteindre les objectifs, sachant que pour un logiciel critique, le coût du test peut représenter plus de entre 30 et 60% du coût de développement.

III COMMENT CHOISIR LE JEU DE TEST ?

Une application, pour être fiable, doit être testée avec différentes sortes de données. L'application doit se comporter normalement et donner les résultats escomptés quand les données transmises sont correctes, mais elle doit aussi composer avec des données qui sortiraient du cadre spécifié.

Vous devez donc la tester avec différentes entrées de données au format approprié mais aussi avec des données dans des formats erronés.

On distingue deux grandes classes de méthodes de sélection de jeu de tests :

- Les méthodes dites « **boite noire** », ou fonctionnelles qui vérifient le comportement du composant par son exécution.
- Les méthodes dites « **boite blanche »,** ou structurelles qui vérifient l'implémentation interne du composant par l'inspection du code.

III.1 METHODES « BOITE NOIRE »

Différentes méthodes permettent de vérifier le comportement du composant :

III.1.1 La recherche intuitive d'erreurs

Les cas de tests sont déterminés par :

- Des éléments de spécification complexes ou risquant d'être mal compris
- Des valeurs spéciales en entrée ou en sortie
- La connaissance du contexte
- L'expérience

On sait par exemple d'expérience, que l'exécution d'une division par 0 est catastrophique pour un programme, et systématiquement cette valeur sera testée pour toute division prévue.

III.1.2 Les classes d'équivalence

Pour chacune des variables d'entrée du composant, on va établir les différents ensembles de valeurs d'entrée pour lesquelles **l'objet est censé se comporter de manière identique**.

En testant le composant pour chaque combinaison de classes d'équivalence d'entrée, on réduit l'effort de test, tout en conservant une couverture de tests efficace.

Exemple 1 : Valeurs dans un intervalle
Pour une fonction qui attend en entrée un code département métropole compris entre 1 et 95, on définit 3 classes d'équivalence, dont on choisit un représentant

	Classe d'équivalence	Validité	Représentant
C1	[1,95]	Valide	40
C2	< 1	Invalide	-15
C3	> 95	Invalide	200

On en déduit 3 cas de test, pour lesquels le résultat attendu est :

C1	40	OK
C2	-15	N OK
C3	200	N OK

Exemple 2 : Valeurs dans un jeu de données
Pour une fonction qui attend en entrée une couleur spécifique parmi les 3 couleurs ROUGE et BLEU, on définira

	Classe d'équivalence	Validité	Représentant
C1	[ROUGE]	Valide	rouge
C2	[BLEU]	Valide	bleu
C3	Autre	Invalide	fleur

On en déduit 4 cas de test, pour lesquels le résultat attendu est :

C1	rouge	OK
C2	bleu	OK
C3	fleur	N OK

En résumé :

Si la donnée appartient à un intervalle, on définira :
- Une classe d'équivalence valide pour les valeurs de l'intervalle
- Une classe d'équivalence invalide pour les valeurs supérieures
- Une classe d'équivalence invalide pour les valeurs inférieures

Si la donnée appartient à une liste de valeurs, on définira :
- Une classe d'équivalence valide pour **chaque** élément de la liste
- Une classe d'équivalence invalide pour toutes les valeurs n'appartenant pas à la liste (les autres)

Si la donnée respecte une condition, on définira :
- Une classe d'équivalence valide avec la condition vraie
- Une classe d'équivalence invalide avec la condition fausse

Ex : le 1er caractère doit être une majuscule…

III.1.3 Les tests aux limites

L'expérience prouve que les erreurs se situent aux frontières de comportements différents :
- Indices de tableau
- Boucles (+ ou – une itération)
- Comparaison inférieur ou égal, ou strictement inférieur
- …

L'idée : Ajouter à la valeur représentative de la classe, les valeurs aux bornes pour tester des valeurs complètement en dehors de la portée normale de fonctionnement comme zéro pour des valeurs non nulles, des nombres négatifs pour des valeurs positives et ainsi de suite….

Pour la fonction précédemment définie dans l'exemple 1

Validité	Classe d'équivalence	Représentant
Valide	[1,95]	1,40, 95
Invalide	< 1	0, -15
Invalide	> 95	200, 96

III.1.4 Tester plusieurs variables

Les cas de test d'une unité à plusieurs variables seront créés en combinant les données représentatives de chaque classe.

Les combinaisons de valeurs des domaines d'entrée par **produit cartésien** font exploser le nombre de cas de tests : 2 variables avec 3 valeurs possibles donnent 9 (3^2) combinaisons, 4 variables avec 3 valeurs possibles donnent 81 (3^4) combinaisons!

Un jeu d'essai basé sur les classes d'équivalence permet de réduire le nombre de cas de test tout en conservant la fiabilité.
Pour construire ce jeu d'essai,

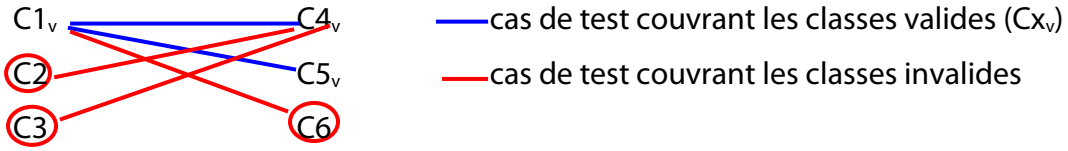
- on écrira un nouveau cas de test couvrant **le plus de classes valides non couvertes**, jusqu'à ce que **toutes les classes valides** soient couvertes.
- on écrira un nouveau cas de test couvrant **une et une seule classe invalide** jusqu'à ce que toutes les **classes invalides** soient couvertes.

Prenons le cas de la fenêtre comportant 2 champs à saisir,

- Un champ Département compris entre 1 et 95
- Un champ Couleur, pouvant contenir rouge ou bleu

Le jeu d'essai complet (produit cartésien) permettrait d'afficher 9 cas de tests différents :

Basé sur les classes d'équivalence :



Cas de test	Jeu d'essai	OK /NOK
C1, C4	1, rouge	OK
C1, C5	95, bleu	OK
C2, C4	0, rouge	NOK
C3, C4	96, rouge	NOK
C1, C6	1, fleur	NOK

III.1.5 Le graphe de causes à effets

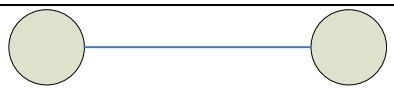
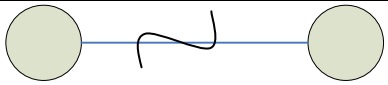
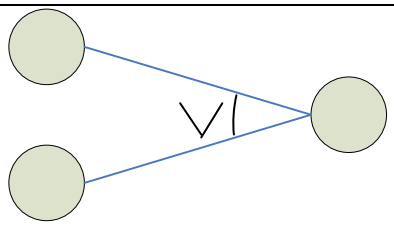
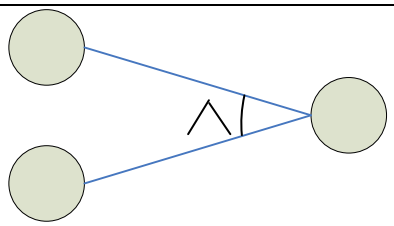
La méthode des graphes cause effets peut être vue comme une méthode à part ou comme une méthode pour créer des partitions relationnelles dans le cas multi variables.

A partir des spécifications découpées en entités simples, on isole la liste des actions (les causes) ayant un effet sur le système, ainsi que la liste des réponses que le système doit produire (les effets).

On construit alors un graphe dont les nœuds sont les causes et les effets précédemment mentionnés, ainsi que des combinaisons de ces derniers par conjonction, disjonction ou

négation. Les arcs du graphe relient chaque cause ou combinaison de cause à ses effets ou combinaison d'effets

Les principales notations :

	**Identité**
	**Négation** :
	**Ou**
	**Et**

A partir de ce graphe cause-effet, on construit une table contenant en entrée les causes et combinaisons de causes, en sortie les effets possibles. Les cases se trouvant à l'intersection d'une cause et d'un effet représentent les cas de test correspondants, une case vide indiquant un effet impossible à obtenir à partir de la cause correspondante.

Exemple : Une fenêtre comportant 2 champs à saisir

- Un champ Sexe, pouvant contenir M ou F
- Un champ Nom devant être non nul

Sera valide si les champs sont remplis correctement.

Dans le cas ou le champ Sexe n'est pas correctement rempli, on affichera le message « Sexe invalide ».

Dans le cas ou le champ Nom n'est pas saisi, on affichera le message « La saisie du nom est obligatoire ».

Les causes :

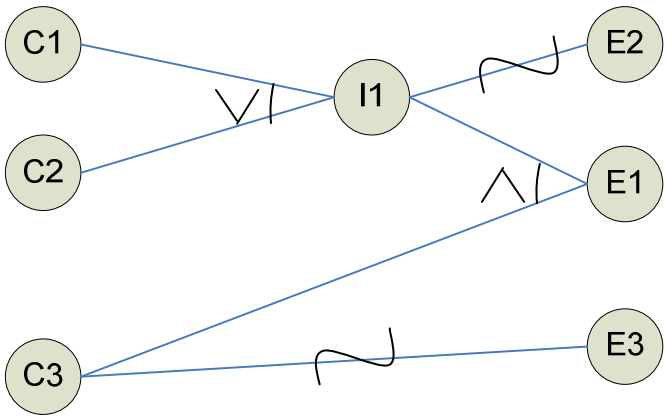
C1: Sexe est égal à M
C2: Sexe est égal à F
C3: Nom est saisi

Les effets :

E1 : La fenêtre est valide
E2 : affichage du message « Sexe invalide »
E3 : affichage du message « La saisie du nom est obligatoire »

Le graphe de causes à effets correspondant

On introduit un nœud intermédiaire



La table de décision correspondante contient l'ensemble des combinaisons des causes possibles (2^3 possibilités)

Cas	1	2	3	4	5	6	7	8
C1	0	0	0	0	1	1	1	1
C2	0	0	1	1	0	0	1	1
C3	0	1	0	1	0	1	0	1
I1	0	0	1	1	1	1		
E1	0	0	0	**1**	0	**1**		
E2	**1**	**1**	0	0	0	0		

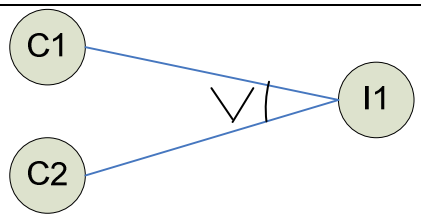
⇦ Cas impossibles au vu des spécifications

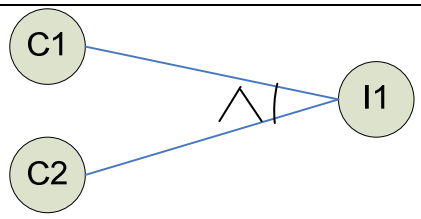
E3	**1**	0	**1**	0	**1**	0		

Il faudra donc choisir :

- 2 cas de test pour tester la fenêtre valide, par exemple :
  - {F, Dupont}
  - {M, Dupont}
- 2 cas de test pour tester l'affichage du message « Sexe invalide », par exemple :
  - {'A',''}
  - {'A', Dupont}
- 3 cas de test pour tester l'affichage du message « La saisie du nom est obligatoire », par exemple :
  - {'A',''}
  - {'F', ''}
  - {'M',''}

Des règles de simplification existent pour restreindre le nombre de cas de test.

<table>
<tr><td colspan="2">Forme OU</td></tr>
<tr><td> 
</td><td>• R1 : Pour I1 = 1
ne pas considérer C1 = C2 = 1.

• R2 : Pour I1=0
considérer tous les cas ou C1 = C2 = 0.</td></tr>
<tr><td colspan="2">Forme ET</td></tr>
<tr><td> 
</td><td>• R3 : Pour I1 = 1, considérer tous les cas ou C1= C2 =1.

• R4 : Pour I1 =0,

o 1 - considérer un seul cas ou C1=C2= 0.

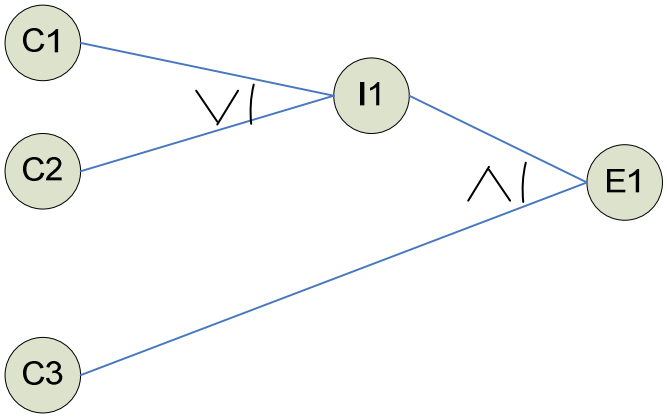
o 2 - Pour les autres cas 001, 010, 011, 100,101 ne considérer pour chaque variable qu'un cas ou la variable est à 1</td></tr>
</table>

Pour démontrer l'effet de ces simplifications, reprenons l'exemple du contrôle fenêtre en se focalisant sur l'effet « Fenêtre valide ».

Les causes sont les mêmes, l'effet analysé est E1 : La fenêtre est valide

Méthode 1 :

Le graphe de causes à effets correspondant



La table de décision correspondante :

Cas	1	2	3	4	5	6	7	8
C1	0	0	0	0	1	1	1	1
C2	0	0	1	1	0	0	1	1
C3	0	1	0	1	0	1	0	1
I1	0	0	1	1	1	1		
E1	0	0	0	**1**	0	**1**		

⇦ Cas impossibles au vu des spécifications

Les cas de test se déduisent de la table de décision : 2 cas de test pour tester la fenêtre valide, 4 cas de test pour tester la fenêtre invalide.

Méthode 2 :

En décomposant le graphe de contrôle en sous graphe, on obtient 2 tables de décision correspondant à chaque sous graphe :

Sous GrapheC1/C2/I1

Cas	1	2	3	4
C1	0	0	1	1
C2	0	1	0	1
I1	0	1	1	1

⇦ R1 : ne pas considérer C1=C2=1

Sous Graphe I1/C3/E1

Cas	1	2	3	4
I1	0	0	1	1
C3	0	1	0	1
E1	0	0	0	1

En appliquant les **règles de simplification** :

Jeu de tests pour E1 = 1

D'après R3, il faut considérer toutes les situations ou I1 = C3 =1
I1 = 1 pour {0,1} et {1,0} C3=1 pour {1}
Il faut donc retenir les cas de test {0, 1,1} et {1, 0,1}

Jeu de tests pour E1 = 0

D'après R4 1
il faut considérer une seule situation ou I1 = C3 =0
I1 = 0 pour {0,0} ou {0,1}
Nous retiendrons par exemple le cas de test {0, 0, 0}

D'après R4 2
E1 = 0 si I1 =0 et C3= 1 ou si I1=1 et C3= 0
Il ne faut considérer pour chaque variable qu'une seule situation où elle est à 1

C3 = 1 (une seule situation) I1 =0 pour {0,0}
Nous retiendrons donc le cas de test {0, 0, 1}

I1 = 1 pour {0,1} ou {1,0} C3=0
Nous retiendrons par exemple le cas de test {0, 1, 0}

L'ensemble des cas de test est ramené à :

{'M', Dupont}	fenêtre valide
{'F', Dupont}	fenêtre valide
{'', '' }	fenêtre invalide
{'A', 'Dupont' }	fenêtre invalide
{'F', ''}	fenêtre invalide

Au final, nous obtenons 2 cas de test pour tester la fenêtre valide, 3 cas de test pour tester la fenêtre invalide.

Nous avions peut-être mis en évidence intuitivement ces cas de test ; Mais dans une structure plus complexe, le graphe de causes à effets est d'une grande utilité pour élaborer des cas de test permettant de valider les différentes combinaisons de données.

Grande utilité, mais difficile à obtenir … !!!

III.2 METHODES « BOITE BLANCHE »

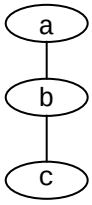
Le test structurel est caractérisé par une sélection des jeux de tests reposant sur la **description de la structure du logiciel** à tester.

L'objectif est de vérifier que les tests exercent tout le code et/ou emploient toutes les données, pour les tests unitaires, et passent dans tous les liens entre modules pour les tests d'intégration.

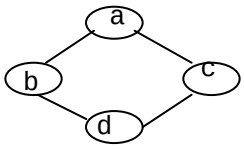
Pour ce faire, on utilisera un graphe de contrôle.

Le graphe de contrôle modélise la structure interne du logiciel : il se compose d'un ensemble de nœuds reliés par des arcs
Les principales notations utilisées:

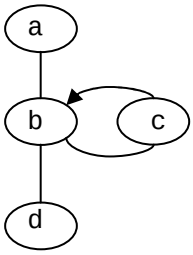
La sequence



If- Else



While



Expression du chemin

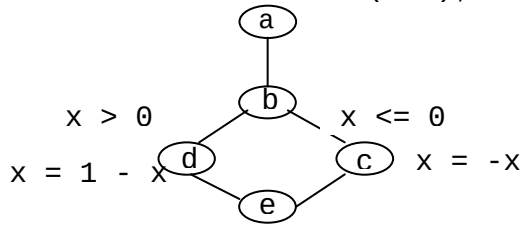
abc | a(b+c)d | ab $(cb)^n$d

Exemple :

```
Console.ReadLine( x );
if (x <= 0)
{
  x = -x ;
}
else
{
  x = 1-x ;
}
Console.WriteLine( x );
```

Ce programme admet le graphe de contrôle suivant :

```
Console.ReadLine( x );
```




```
Console.WriteLine( x );
```

- a, b, c, d, e sont des sommets qui contiennent des blocs d'instruction
- un chemin de contrôle définit une exécution possible
  [a,b,c,e] est un chemin de contrôle
- L'ensemble des chemins du graphe peut s'exprimer sous la forme :
  G = abce + abde = ab(c+d)e

A partir du graphe de contrôle, deux approches sont possibles : l'analyse du flot de contrôle et l'analyse du flot de données

III.2.1 Analyse du flot de contrôle

Le critère de sélection du jeu de test repose sur le choix de données d'entrée déclenchant l'exécution de certains chemins du graphe de contrôle.
Le jeu de test est choisi de manière à remplir certaines exigences :

- Couvrir toutes les instructions

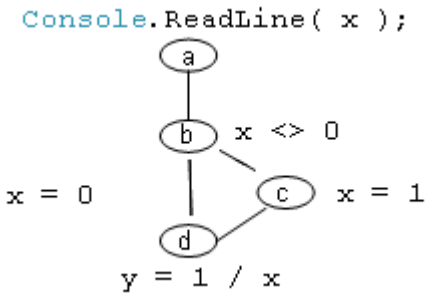
  Chaque bloc d'instructions doit être atteint par au moins un chemin.

  On définit le taux de couverture par :
  tx = nb d'instructions couvertes / nb total d'instructions

  Exemple 1:

```
Console. ReadLine( x );
if (x != 0)
{
  x = 1 ;
}
y = 1 /x  ;
```

  Ce programme admet le graphe de contrôle suivant :



  Le critère « toutes les instructions » est satisfait par le chemin [abcd].

  En choisissant comme donnée de test, la valeur x=2, chaque bloc d'instructions est atteint ; le taux de couverture est égal à 1, la couverture est complète, mais si x=0, une erreur se produira.

  En conclusion, ce critère est rarement suffisant.

- Couvrir tous les arcs

  Chaque arc du graphe doit être parcouru par au moins un chemin.

  On définit le taux de couverture par :
  tx = nb d'arcs couverts / nb total d'arcs

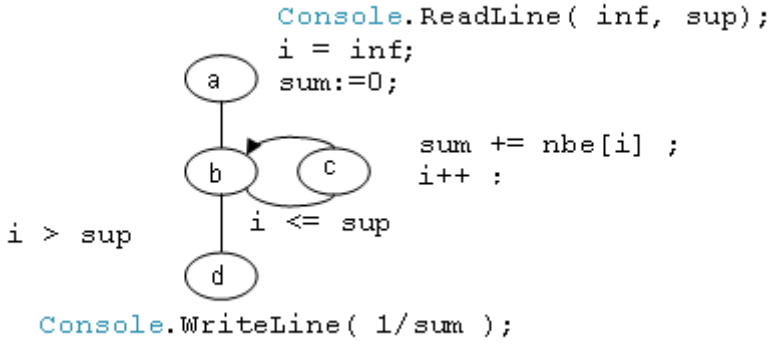
  Lorsque le critère « tous les arcs » est satisfait, le critère « toutes les instructions » est forcément satisfait.

  En reprenant l'exemple précédent, l'arc bd devra être couvert.
  L'ensemble des cas de test sera : DT1= {x=2}, DT2= {x=0}.
  Cet objectif a ses limites : il ne permet pas de détecter la non exécution d'une boucle.

Exemple 2:

```
Console.ReadLine( inf, sup );
i = inf;
sum:=0;
while (i <= sup)
{
  sum += nbe[i] ;
  i++ ;
}
Console.WriteLine( 1/sum );
```

Qui admet le graphe de contrôle suivant :



Le cas de tests :

DT1 ={nbe[1]=10, nbe[2]=20, nbe[3]=30, inf = 1, sup =3}

couvre tous les arcs en parcourant le chemin [abcbcbcbd].
La couverture de tests est égale à 1, et pourtant le programme se terminera en erreur si inf supérieur à sup, car sum = 0.
Il faut rajouter un cas de test pour couvrir le chemin [abd].

- Couvrir tous les chemins indépendants

Le critère « tous les chemins indépendants » vise à parcourir tous les arcs dans chaque configuration possible (et non pas au moins une fois comme dans le critère tous-les-arcs)

Le nombre de chemins indépendants est donné par le nombre de Mc Cabe, ou nombre cyclomatique :

V(G) = nb arcs – nb nœuds + 2

On définit le taux de couverture par :

tx = nb de chemins couverts / V(G)

Lorsque le critère « tous les chemins indépendants » est satisfait, le critère « tous les arcs » est forcément satisfait.
Sur l'exemple 2,
V(G) = 4 – 4 + 2 = 2
Pour que le taux de couverture soit égal à 1, 2 chemins doivent être parcourus : [abcbd].et [abd].

Dans la pratique, la limite supérieure du nombre cyclomatique serait de 30…Au-delà le test est difficile à réaliser…

On peut aller encore plus loin dans les exigences, avec le critère « Couvrir les PLCSS (Portions linéaires de code suivies d'un saut) », soit les séquences d'instructions entre deux branchements.

III.2.2 Analyse du flot de données

La sélection des cas de test repose sur l'analyse fine des relations entre instructions, en tenant compte des variables qu'elles définissent ou utilisent.

On définit deux types d'utilisation d'une donnée, après sa définition :

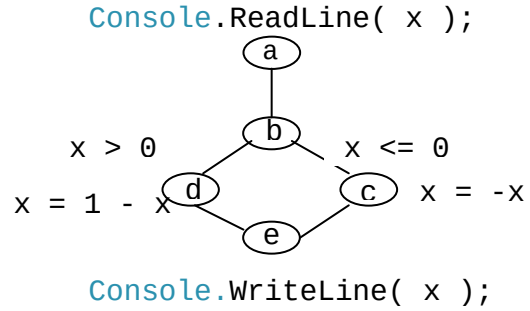
- Dans le prédicat d'une instruction de décision, la **p-utilisation**
- Dans un calcul ou un indice, la **c-utilisation**

On dit qu'une instruction I2 est utilisatrice par rapport à une instruction I1 d'une variable x si x est définie en I1 et est directement utilisée en I2, sans définition intermédiaire.

Un chemin d'utilisation est un chemin reliant l'instruction de définition d'une variable à une instruction utilisatrice (p-utilisation ou c-utilisation).

Exemple :

Pour le graphe de contrôle suivant :



- b est p-utilisatrice de x par rapport à a
- c est c-utilisatrice de x par rapport à a, pas e

Le jeu de test est choisi de manière à respecter certains critères :

- Critère « Toutes les définitions »

  Ce critère impose de couvrir au moins un chemin d'utilisation pour chaque définition.

- Critère « tous les p-utilisateurs »

  Ce critère impose de couvrir tous les arcs p-utilisateurs correspondant à toutes les définitions du graphe, par des chemins de p-utilisation.

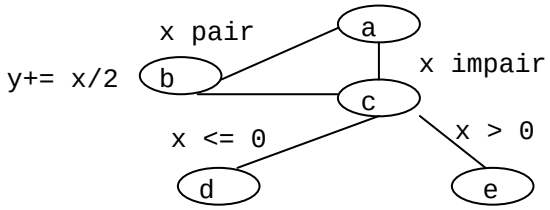
  Il entraine la couverture des arcs du graphe de contrôle.

  Il est limité dans le cas des variables qui n'admettent que des c-utilisateurs : une mauvaise affectation ne sera pas forcément détectée.

On peut également envisager les critères « Tous les p-utilisateurs, quelques c-utilisateurs », « Tous les c-utilisateurs, quelques p-utilisateurs », « Tous les p-utilisateurs, tous les c-utilisateurs »,

Exemple :

```
                    Console.ReadLine( x,y );
```



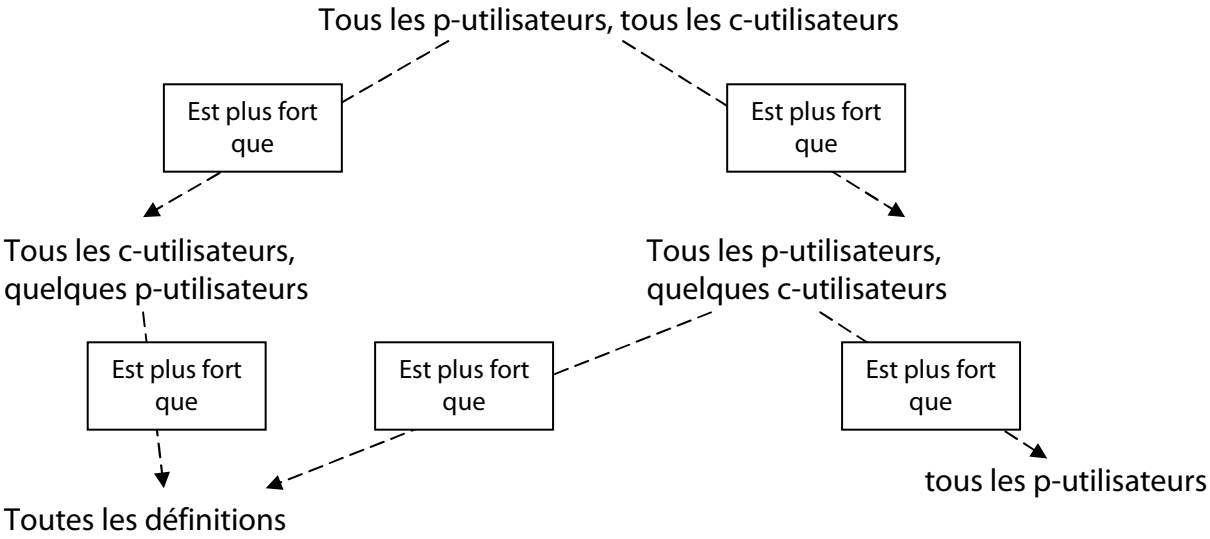

```
Console.WriteLine( y );          Console.WriteLine(("erreur ");
```

Le test exerçant les chemins [abce] et [acd] couvre tous les p-utilisateurs de x.

Si une erreur d'affectation sur y existe en b, elle n'est pas détectée par le test, il n'y a que des c-utilisateurs.

Le test exerçant en plus le chemin [abcd] couvre la définition de y en b et son utilisation en d et permet de détecter l'erreur.

Dans la hiérarchie,



Etablissement référent
Marseille Saint Jérôme

Equipe de conception
Elisabeth Cattanéo

Remerciements :

Date de mise à jour 12/11/2008
afpa © Date de dépôt légal nov.-08

gagnez en compétences

afpa / Direction de l'Ingénierie13 place du Générale de Gaulle / 93108 Montreuil Cedex
association nationale pour la formation professionnelle des adultes
Ministère des Affaires sociales du Travail et de la Solidarité