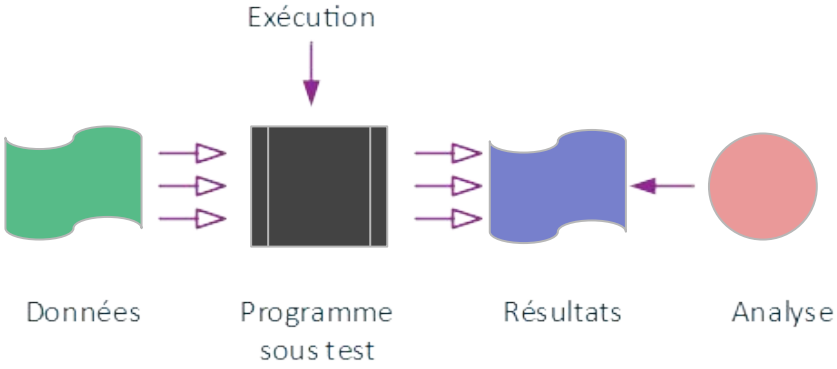
INF-1011
Génie Logiciel

Chapitre 8 :
Test de logiciel

Définition

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

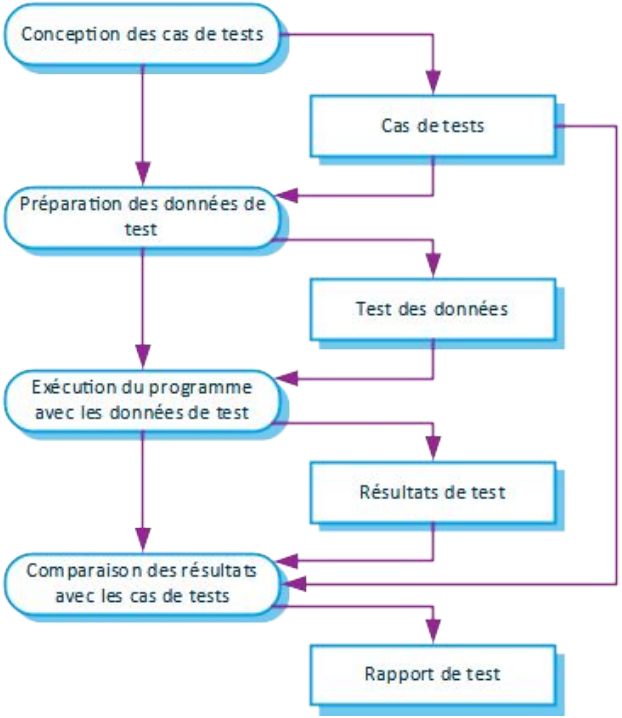
Définition



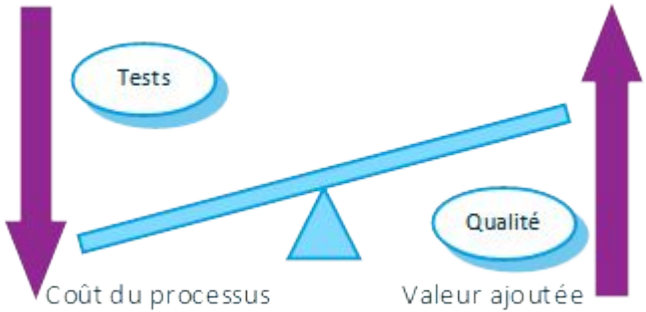
- Méthode de validation la plus mise en oeuvre en assurance qualité
- Le test constitue une tâche essentielle dans l'élaboration de la qualité des logiciels
  - 50% du coût total de développement
  - Largement accepté par l'industrie

Objectifs

- Mettre en oeuvre le programme (complet ou partiel) en utilisant des données similaires aux données réelles
- Observer les résultats (comportement du programme)
- Détecter les anomalies (défaillances)
  - Écarts entre les spécifications et le comportement réel observé
- Déduire la présence d'erreur



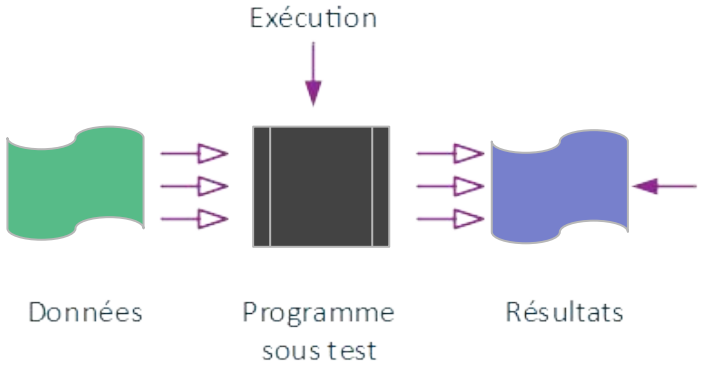
Objectifs



- Le coût de ce processus doit être compensé par l'amélioration de la valeur du produit logiciel
  - Fiabilité
  - Qualité

Construction des cas de tests

- Le test exhaustif est difficile à envisager
  - Impossible à réaliser dans de nombreux cas
  - On ne peut pas tester toutes les données d'entrées possibles
- On doit donc sélectionner certaines données représentatives de l'ensemble
  - Il s'agit d'un des problèmes fondamentaux du test



Construction des cas de test

- Il existe deux principales approches pour la construction des cas de test
  - Boîte fermée (boîte noire)
  - Boîte ouverte (boîte blanche)

Première approche : Boîte fermée

- Les cas de tests sont construits à partir des spécifications
  - Souvent appelés tests fonctionnels ou boîte noire
  - On s’intéresse aux fonctionnalités du logiciel
- L’élément testé (procédure, fonction, module) n’est considéré qu’au travers de ses interfaces
  - Données d’entrées et données de sorties
  - On ignore la structure de sa réalisation

Deuxième approche : Boîte ouverte

- Les cas de test sont construits à partir de la structure interne du programme
  - Souvent appelés tests structurels ou boîte blanche
  - Prend en compte la structure de contrôle du programme et de ses données
- Les données d'entrées sont choisies pour suivre des chemins particuliers au travers de la structure du code

Construction des cas de test

- Dans les deux cas, le cas de test d’un programme comporte les mêmes informations
  - Spécification des données d’entrées
  - Spécification des résultats attendus
  - Description des fonctions du système qui sont testées

Couverture de test

« La couverture de test est le rapport entre le nombre de tests effectués et le nombre de tests nécessaires pour que soit vérifiée une certaine propriété du programme testé. »

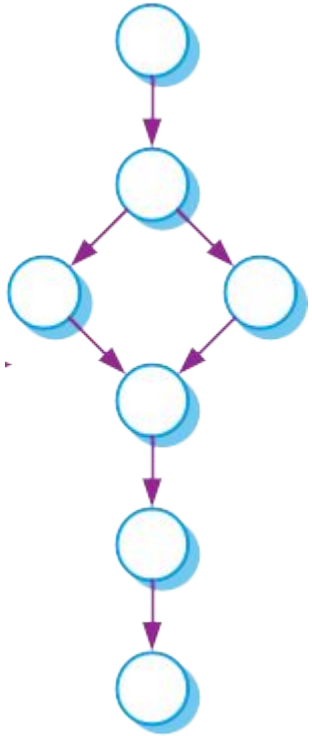
- Il existe deux types de couverture de test
  - Couverture fonctionnelle
  - Couverture structurelle

Couverture fonctionnelle

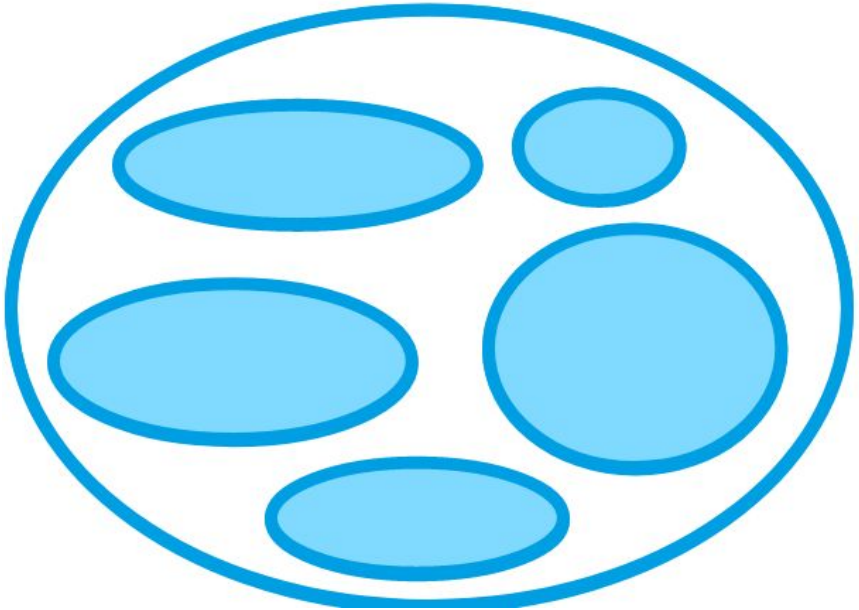
- On s’intéresse à la couverture des variables d’entrée et de sortie des divers niveaux du programme

Couverture structurelle

- On s’intéresse à la couverture d’éléments de la structure interne du programme
  - Instructions
  - Conditions simples et multiples
  - Branches élémentaires
  - Chemins



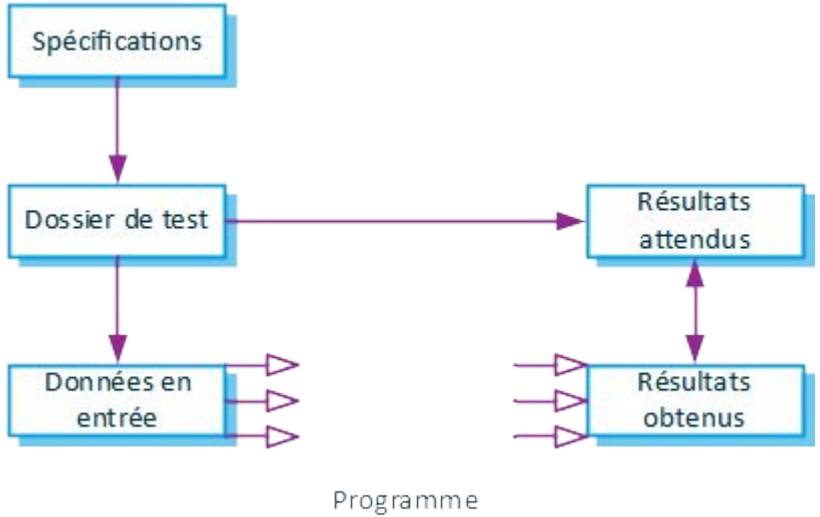
Techniques de test



- L’un des objectifs principaux du test est donc de déterminer des sous-ensembles de données pertinents sur lesquels le programme va être testé
- Pour se guider, on utilise certaines approches structurées
  - Construire les données de tests
  - Dépend également de l’aptitude et de l’expérience du testeur

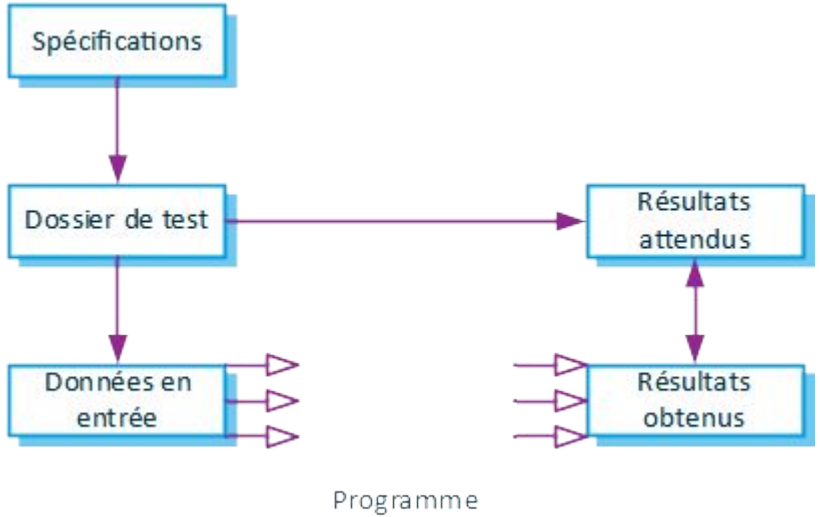
Tests fonctionnels

- La structure interne des programmes est ignorées
- Consiste à exécuter le programme avec des jeux d'essai
  - Établis à partir d'une analyse des spécifications
- On s'assure que les résultats sont conformes
  - Document de référence (oracle)

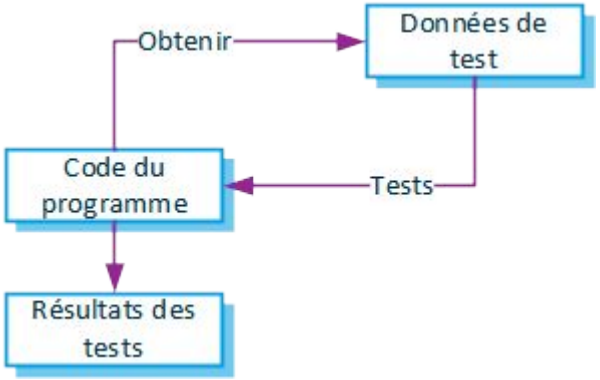


Tests fonctionnels

- Permet de s’assurer que le fonctionnement du programme est conforme à ses spécifications
- Plusieurs techniques basées sur cette approche
  - Test de partition
  - Test par classes d’équivalence
  - Test aux limites



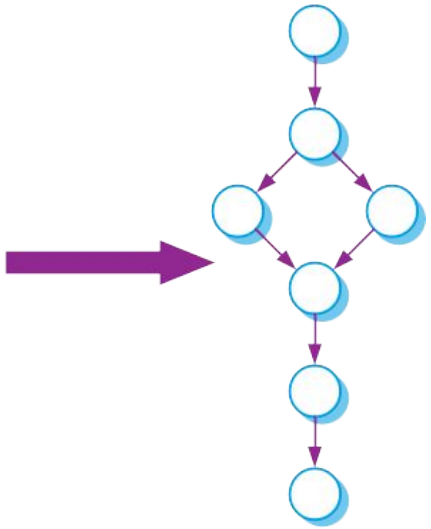
Test structurels



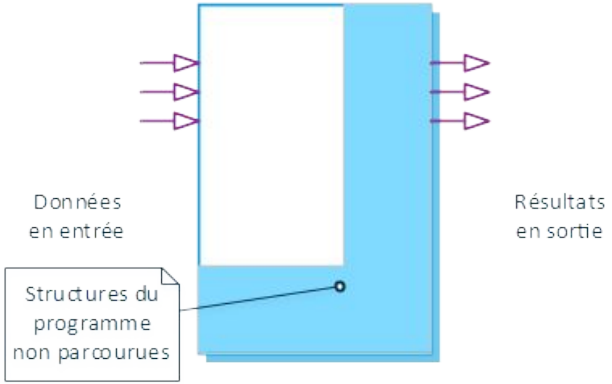
- Fondés sur les structures internes du programme testé
  - Principalement les graphes de contrôle
- Un test structurel est défini comme un parcours de graphe

Test structurels

- Exécuter le programme avec des données de tests préalablement établies
  - Analyse du code
  - S’assurer que la structure interne du programme est bien parcourue



Test structurels



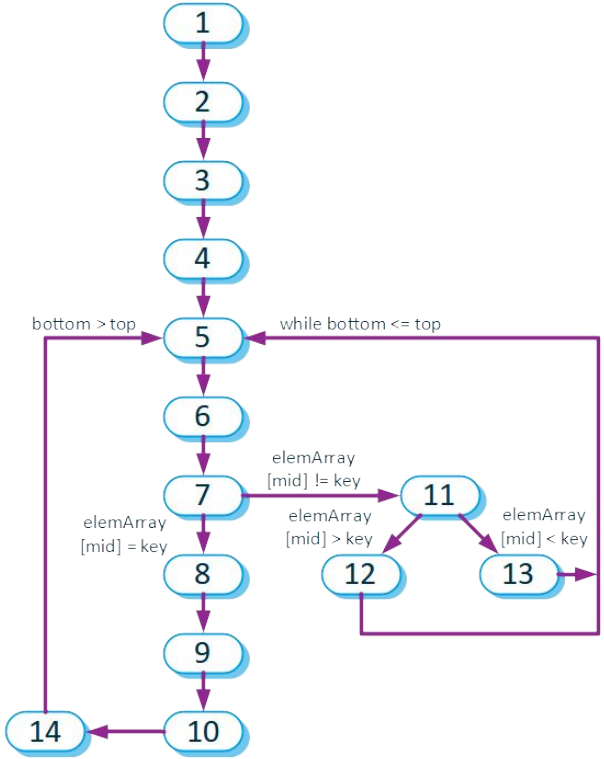
- Dans cette approche, on considère le programme comme une boîte ouverte
  - Instructions
  - Conditions
  - Branches
  - Chemins

Test d’instructions

- Consiste à exécuter au moins une fois chaque instruction du programme
  - Taux de couverture : nombre d’instructions exécutées / nombre total d’instructions
- Expérience démontre que cette couverture de test est peu efficace

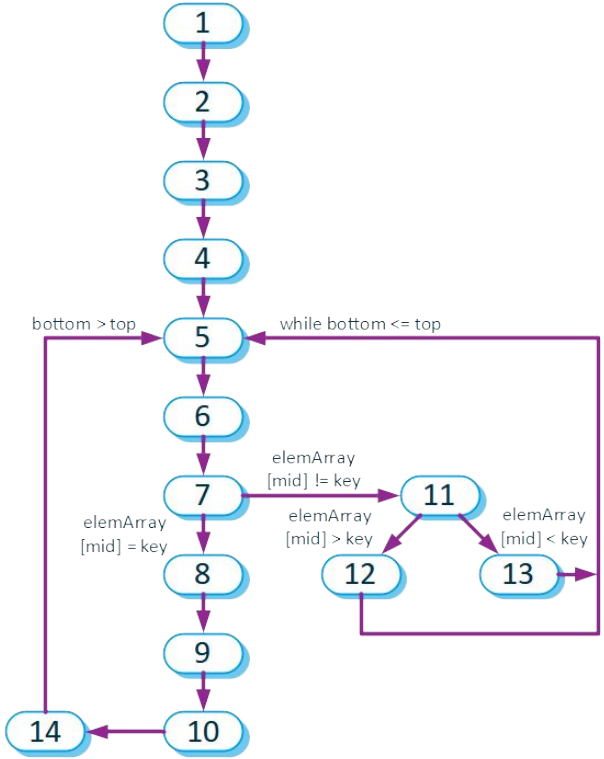
Test de chemins

- Consiste à exécuter un nombre maximum de chemins sur le graphe de contrôle
  - Contrôle dynamique des enchaînements
- En théorie, le type de test structurel le plus efficace
  - Nécessite un graphe de contrôle
  - Taux de couverture = nombre de chemins testés / nombre total de chemins possibles



Test de chemins

- En pratique, le nombre total de chemin du graphe peut être très élevé
  - Explosion combinatoire
- Très difficile à mettre en oeuvre pour des logiciels non-triviaux
  - Taille et complexité importante
- Tous les chemins ne peuvent pas nécessairement être parcourus
  - Prédicats en contradiction
  - Nécessité de détecter ces chemins théoriques non-exécutables



Conception des cas de test

- Différentes approches pour construire les cas de test
  - Basé sur les besoins (boîte fermée)
  - Test de partition (boîte fermée / ouverte)
  - Test structurel (boîte ouverte)

Test basé sur les besoins

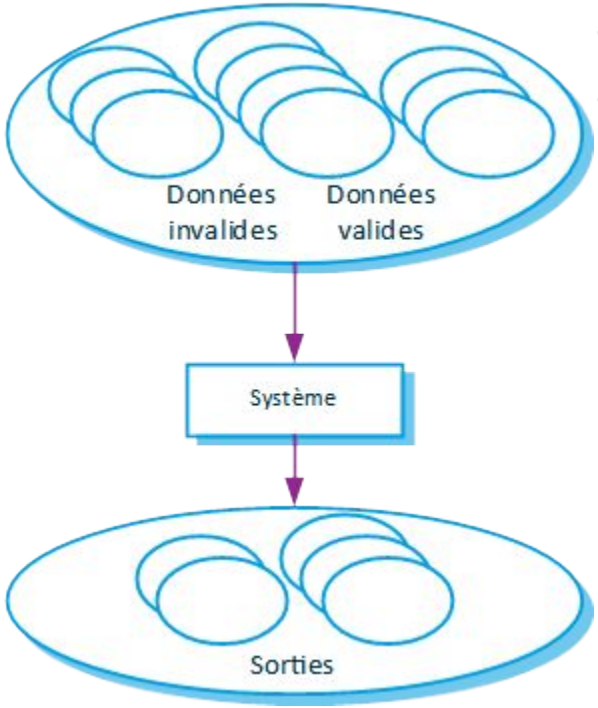
- Les cas de tests sont conçus pour s’assurer de la conformité aux besoins
  - Utilisé au niveau des tests de système
  - Les besoins (spécifications) doivent être décrits de façon claire pour pouvoir construire des cas de tests
  - Pour chaque besoin, on construit un **ensemble** de cas
  - Souvent appelés tests de validation

Test de partition

Domaine des variables d'entrées divisé en plusieurs classes d'équivalence.

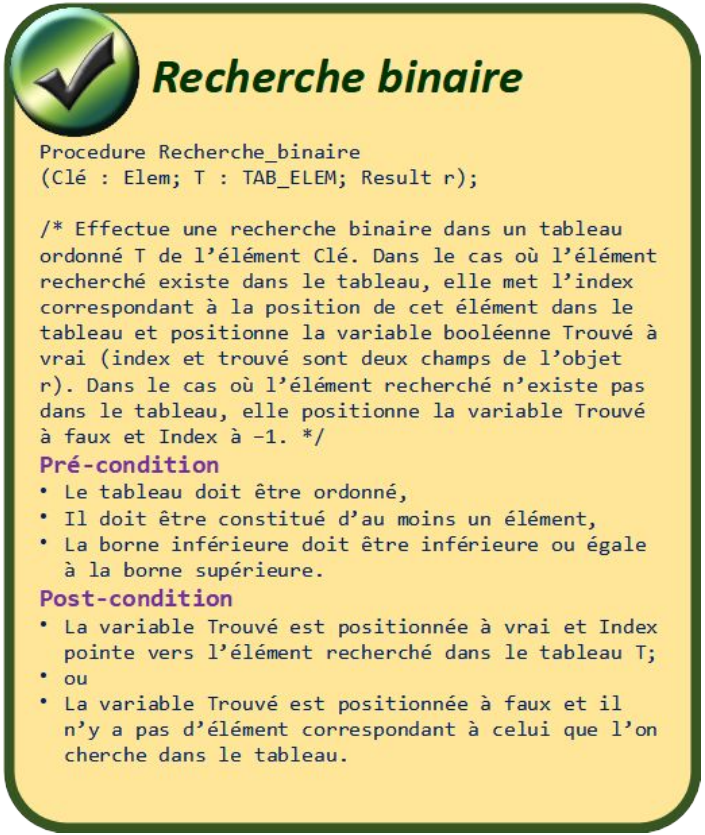
On connaît la description fonctionnelle du programme.

Espace des résultats.



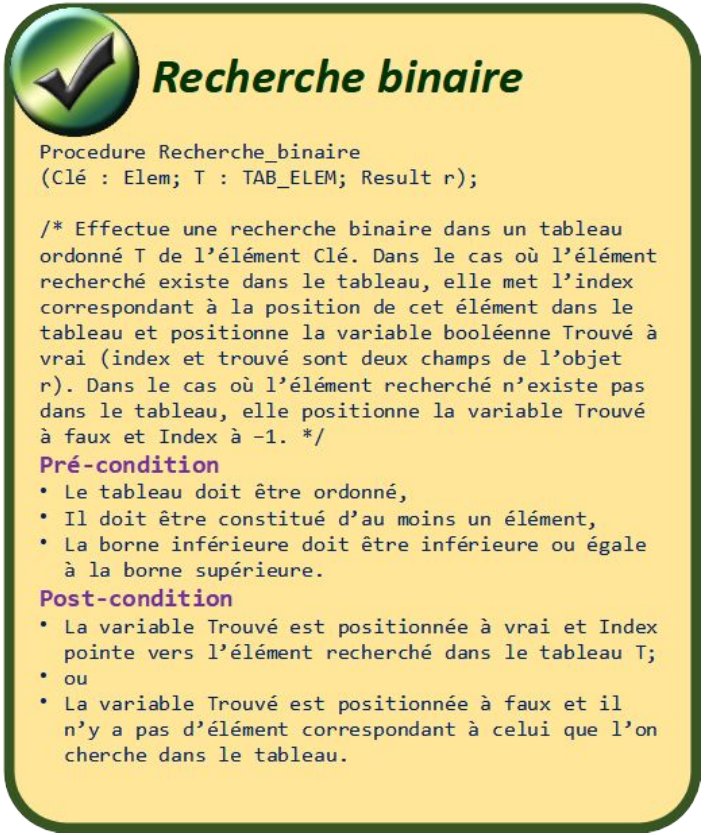
- Permet de réduire le nombre de tests
- On identifie les données possibles en entrées (domaine)
  - Regrouper les données en classes d’ équivalences
  - Le programme doit se comporter de manière similaire pour tous les éléments d’une même classe d’équivalence
  - Choisir des données représentatives de chaque classe

Exemple de test partition



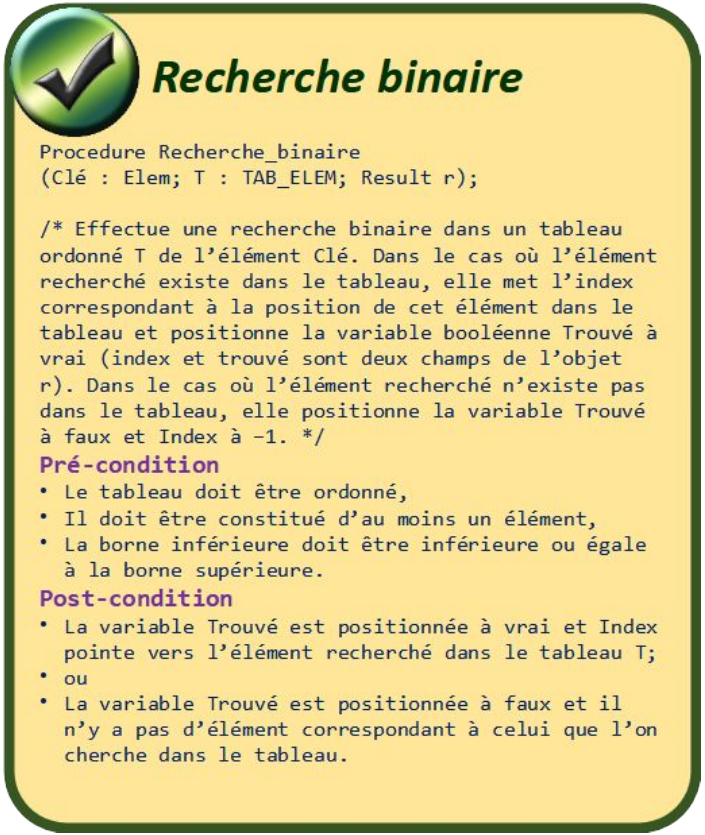
- Composant qui effectue une recherche binaire
  - Cherche un élément dans un tableau ordonné
  - Renvoie l'index correspondant à la position de l' élément dans le tableau
- La figure à gauche donne une spécification du composant à l'aide de pré-conditions et post-conditions

Exemple de test partition



- On peut définir les classe d'équivalence suivantes :
  1. Entrées conformes à la pré-condition
  2. Entrées non-conformes à la pré-condition
  3. Entrées où l'élément recherché existe dans le tableau
  4. Entrées où l'élément recherché n'existe pas dans le tableau

Exemple de test partition



Recherche binaire

```
Procedure Recherche_binaire
(Clé : Elem; T : TAB_ELEM; Result r);

/* Effectue une recherche binaire dans un tableau
ordonné T de l'élément Clé. Dans le cas où l'élément
recherché existe dans le tableau, elle met l'index
correspondant à la position de cet élément dans le
tableau et positionne la variable booléenne Trouvé à
vrai (index et trouvé sont deux champs de l'objet
r). Dans le cas où l'élément recherché n'existe pas
dans le tableau, elle positionne la variable Trouvé
à faux et Index à -1. */
```

Pré-condition

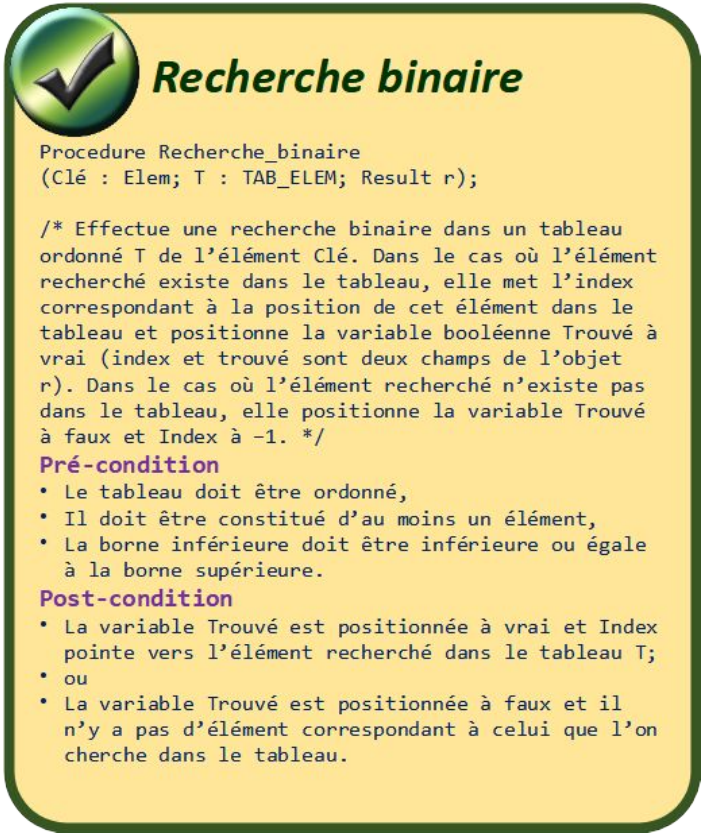
- Le tableau doit être ordonné,
- Il doit être constitué d'au moins un élément,
- La borne inférieure doit être inférieure ou égale à la borne supérieure.

Post-condition

- La variable Trouvé est positionnée à vrai et Index pointe vers l'élément recherché dans le tableau T;
- ou
- La variable Trouvé est positionnée à faux et il n'y a pas d'élément correspondant à celui que l'on cherche dans le tableau.

- On peut également combiner les classes 1 et 2 avec les classes 3 et 4 et obtenir quatre nouvelles classes d'équivalences
  5. Entrées conformes où élément existe
  6. Entrées conformes où élément n'existe pas
  7. Entrées non-conformes où élément existe
  8. Entrées non-conformes où élément n'existe pas

Exemple de test partition



Recherche binaire

```
Procedure Recherche_binaire
(Clé : Elem; T : TAB_ELEM; Result r);

/* Effectue une recherche binaire dans un tableau
ordonné T de l'élément Clé. Dans le cas où l'élément
recherché existe dans le tableau, elle met l'index
correspondant à la position de cet élément dans le
tableau et positionne la variable booléenne Trouvé à
vrai (index et trouvé sont deux champs de l'objet
r). Dans le cas où l'élément recherché n'existe pas
dans le tableau, elle positionne la variable Trouvé
à faux et Index à -1. */
```

Pré-condition

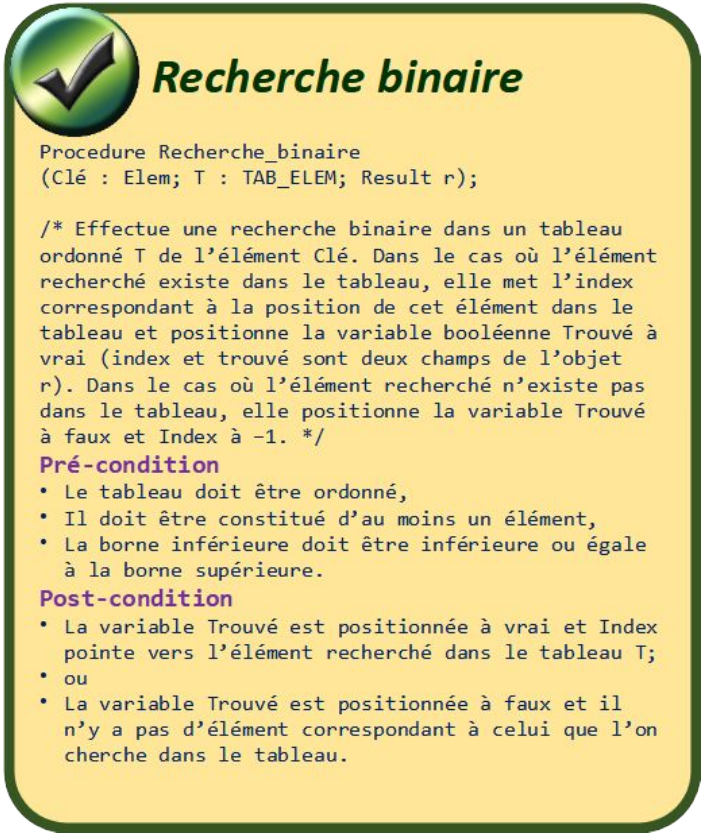
- Le tableau doit être ordonné,
- Il doit être constitué d'au moins un élément,
- La borne inférieure doit être inférieure ou égale à la borne supérieure.

Post-condition

- La variable Trouvé est positionnée à vrai et Index pointe vers l'élément recherché dans le tableau T;
- ou
- La variable Trouvé est positionnée à faux et il n'y a pas d'élément correspondant à celui que l'on cherche dans le tableau.

- On peut aussi partitionner la forme du tableau
  - Le tableau ne possède qu'un élément
  - Le tableau possède un nombre pair d'éléments
  - Le tableau possède un nombre impair d' éléments
- Pour chacune de ces classes, on conçoit un test où l'élément existe ou non

Exemple de test partition



Recherche binaire

```
Procedure Recherche_binaire
(Clé : Elem; T : TAB_ELEM; Result r);

/* Effectue une recherche binaire dans un tableau
ordonné T de l'élément Clé. Dans le cas où l'élément
recherché existe dans le tableau, elle met l'index
correspondant à la position de cet élément dans le
tableau et positionne la variable booléenne Trouvé à
vrai (index et trouvé sont deux champs de l'objet
r). Dans le cas où l'élément recherché n'existe pas
dans le tableau, elle positionne la variable Trouvé
à faux et Index à -1. */
```

Pré-condition

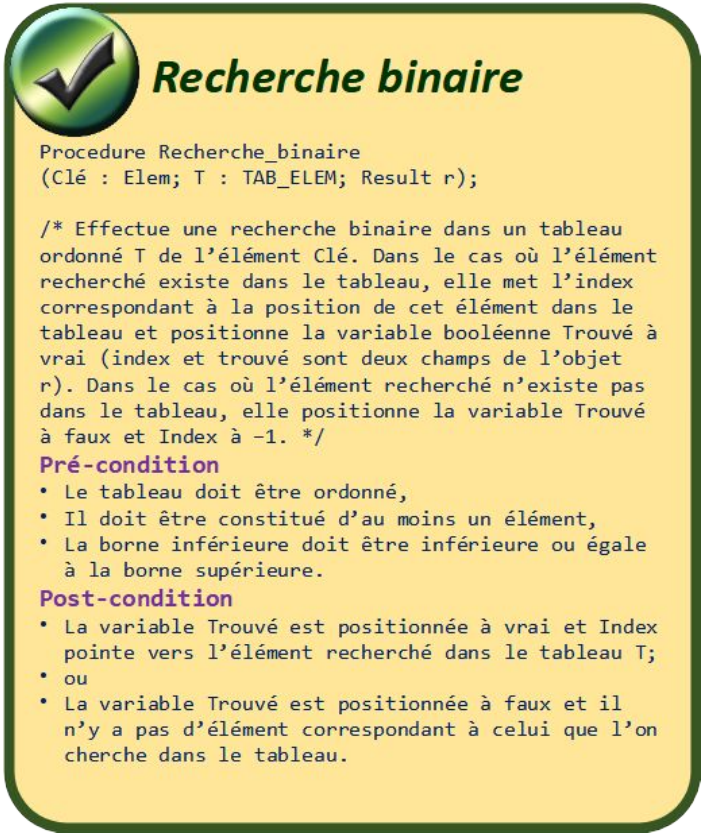
- Le tableau doit être ordonné,
- Il doit être constitué d'au moins un élément,
- La borne inférieure doit être inférieure ou égale à la borne supérieure.

Post-condition

- La variable Trouvé est positionnée à vrai et Index pointe vers l'élément recherché dans le tableau T;
- ou
- La variable Trouvé est positionnée à faux et il n'y a pas d'élément correspondant à celui que l'on cherche dans le tableau.

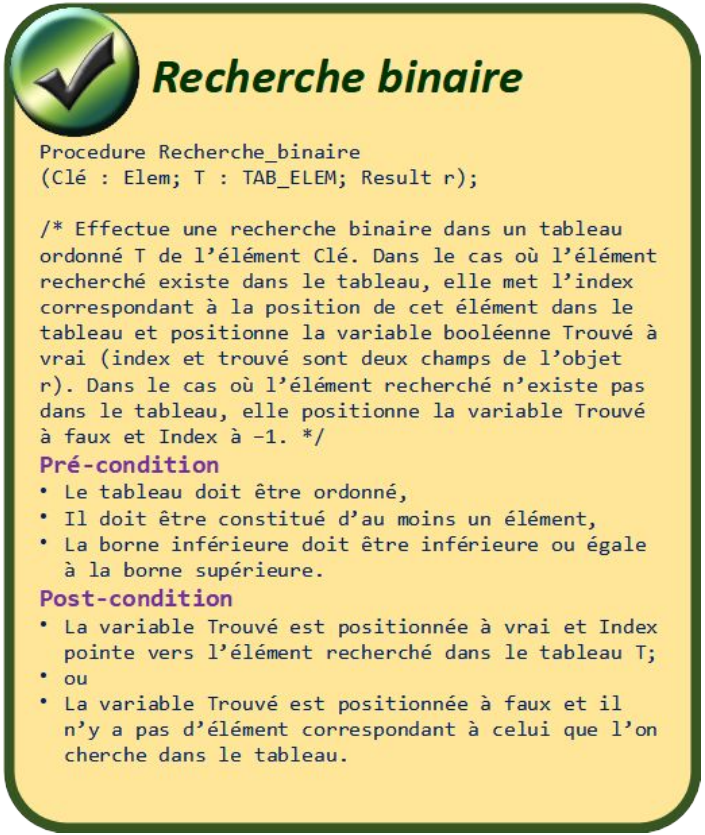
- Finalement, on peut également partitionner sur le résultat en changeant où l'élément recherché se retrouve
  - L'élément est le premier du tableau
  - L'élément est le dernier du tableau
  - L'élément est au centre du tableau

Exemple de test partition



- Après combinaison, on obtient une dizaine de classes d'équivalence
  1. Tableau de taille 1, soit l'élément recherché
  2. Tableau de taille 1, élément absent
  3. Tableau de taille paire, élément en premier
  4. Tableau de taille paire, élément en dernier
  5. Tableau de taille paire, élément absent
  6. Tableau de taille paire, élément au centre
  7. Tableau de taille impaire, élément en premier
  8. Tableau de taille impaire, élément en dernier
  9. Tableau de taille impaire, élément absent
  10. Tableau de taille impaire, élément au centre

Exemple de test partition



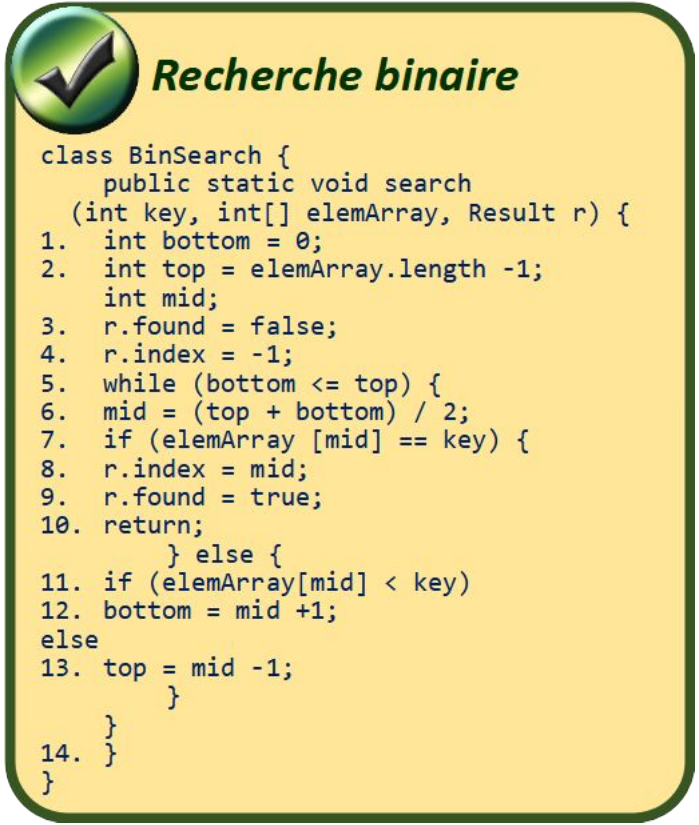
- Quelques exemples de cas de tests pour la procédure de recherche binaire

  - **Entrée** : Tableau = [21]; Clé = 21
  - **Sortie** : Trouvé = vrai, Index = 0

  - **Entrée** : Tableau = [21]; Clé = 15
  - **Sortie** : Trouvé = faux, Index = -1

  - **Entrée** : Tableau = [17, 18, 21, 23, 29, 35]; Clé = 21
  - **Sortie** : Trouvé = vrai, Index = 2

Exemple de test structurel



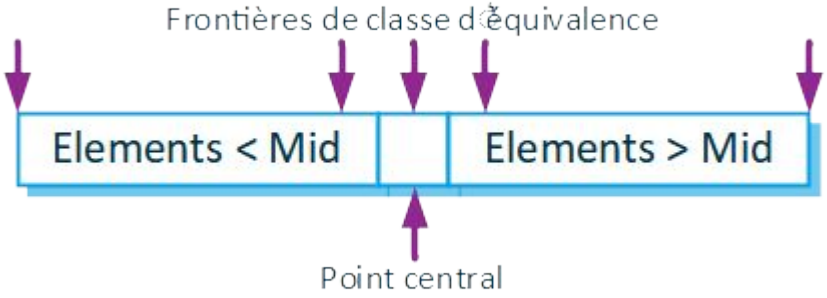
- Analyse du code
- Utilisation des connaissances de la structure pour déduire le domaine

Exemple de test structurel

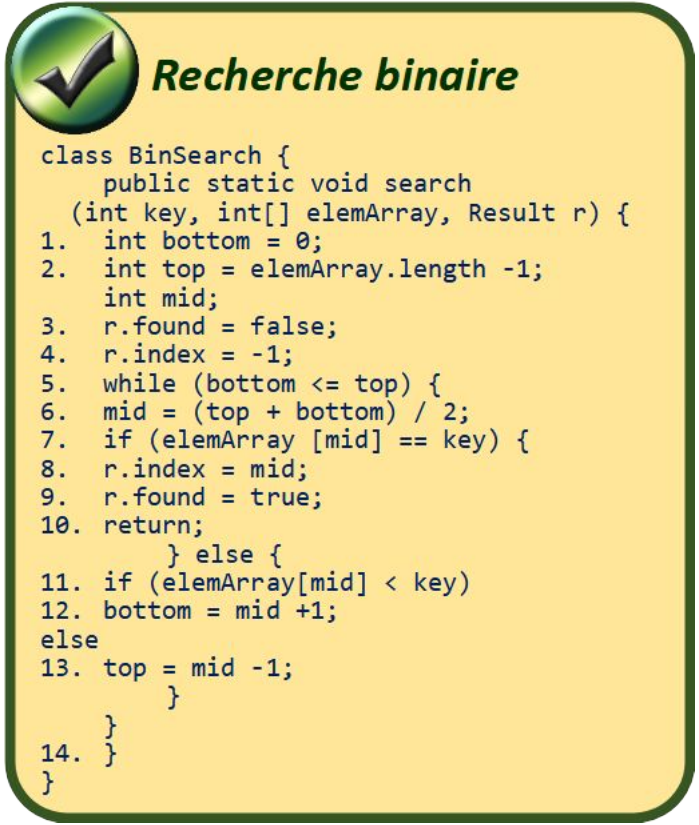
Recherche binaire

```
class BinSearch {
    public static void search
  (int key, int[] elemArray, Result r) {
1.  int bottom = 0;
2.  int top = elemArray.length -1;
    int mid;
3.  r.found = false;
4.  r.index = -1;
5.  while (bottom <= top) {
6.  mid = (top + bottom) / 2;
7.  if (elemArray [mid] == key) {
8.  r.index = mid;
9.  r.found = true;
10. return;
        } else {
11. if (elemArray[mid] < key)
12. bottom = mid +1;
else
13. top = mid -1;
        }
    }
14. }
}
```

- On peut, en premier lieu, affiner les classes d'équivalences identifiées durant la conception des tests fonctionnels
- On ajoute de nouveaux cas à l'ensemble des tests retenus basé sur le code



Exemple de test structurel

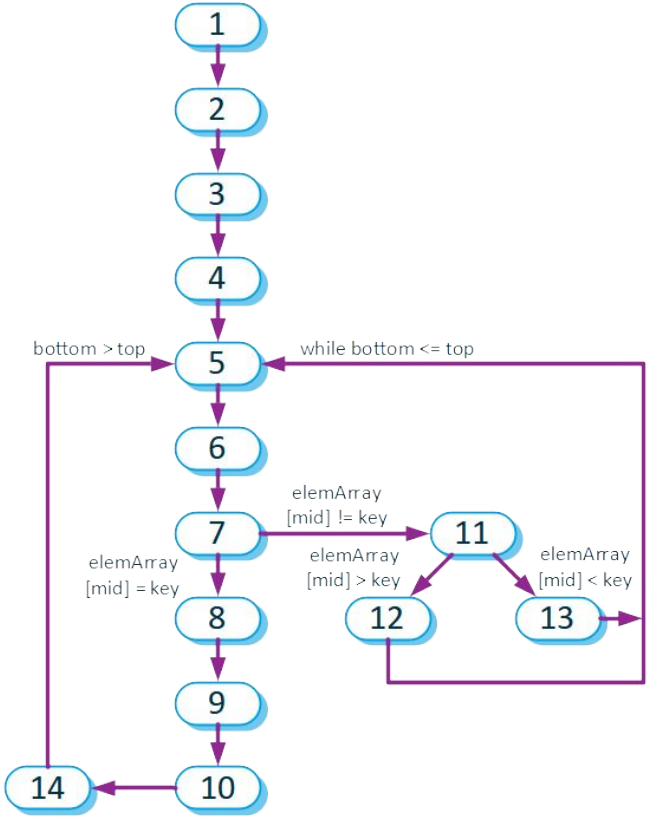


Recherche binaire

```
class BinSearch {
    public static void search
  (int key, int[] elemArray, Result r) {
1.  int bottom = 0;
2.  int top = elemArray.length -1;
    int mid;
3.  r.found = false;
4.  r.index = -1;
5.  while (bottom <= top) {
6.  mid = (top + bottom) / 2;
7.  if (elemArray [mid] == key) {
8.  r.index = mid;
9.  r.found = true;
10. return;
        } else {
11. if (elemArray[mid] < key)
12. bottom = mid +1;
else
13. top = mid -1;
        }
    }
14. }
}
```

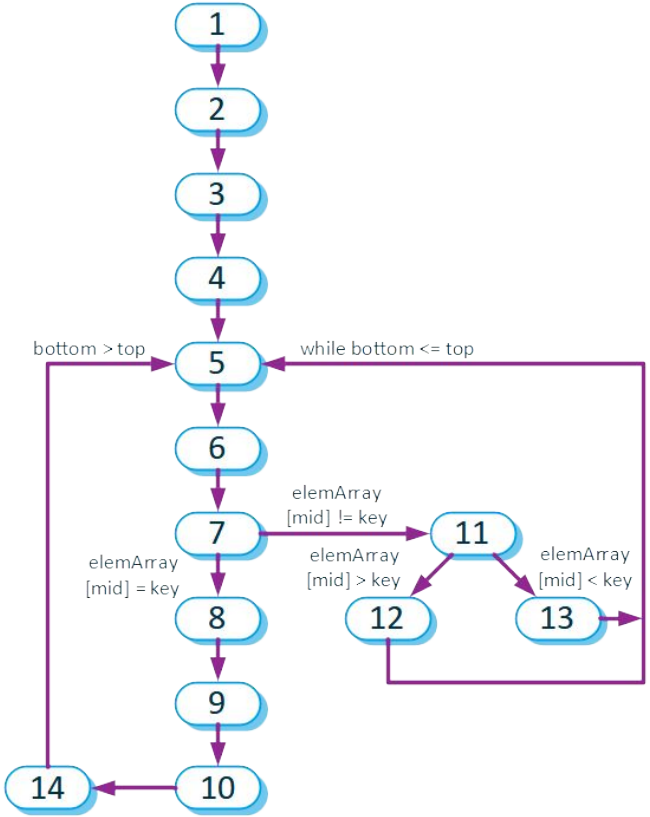
- Le test de chemins consiste à passer par tous les chemins d’exécutions du composant
  - On s’assure que toutes les instructions ont été exécutées au moins une fois
  - Et que les instructions conditionnelles ont été testées à la fois dans le cas où le résultat est vrai et où il est faux

Exemple de test structurel



- La complexité cyclomatique (McCabe, 1976) permet d'identifier le nombre de chemins possibles
  - Nb de chemins = nb de conditions simples + 1
- Pour calculer la complexité cyclomatique exacte, il faut décomposer les conditions composites en conditions simples

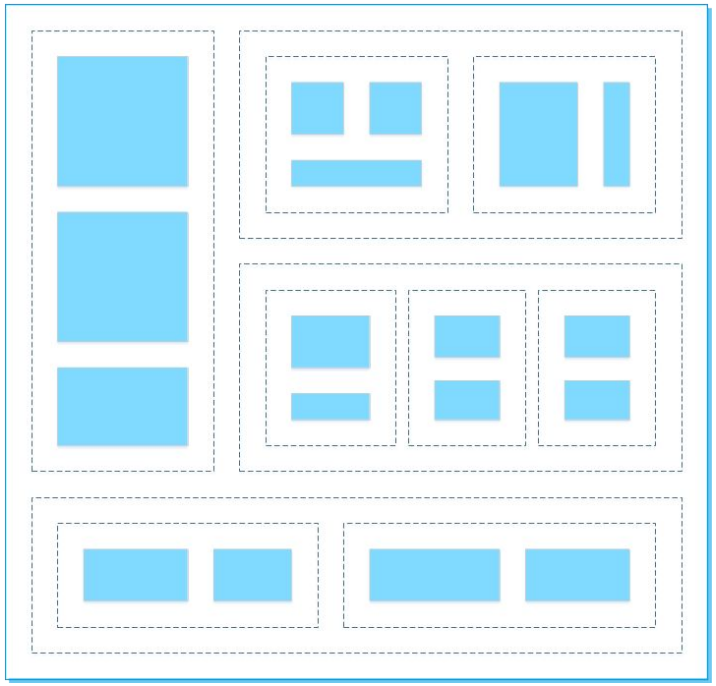
Exemple de test structurel



- On vise à concevoir un cas de test pour chaque chemin
- Certains outils d'analyse de programme dynamique permettent de suivre l'exécution du programme (instrumentation du code)
  - Met en évidence les chemins testés et non testés

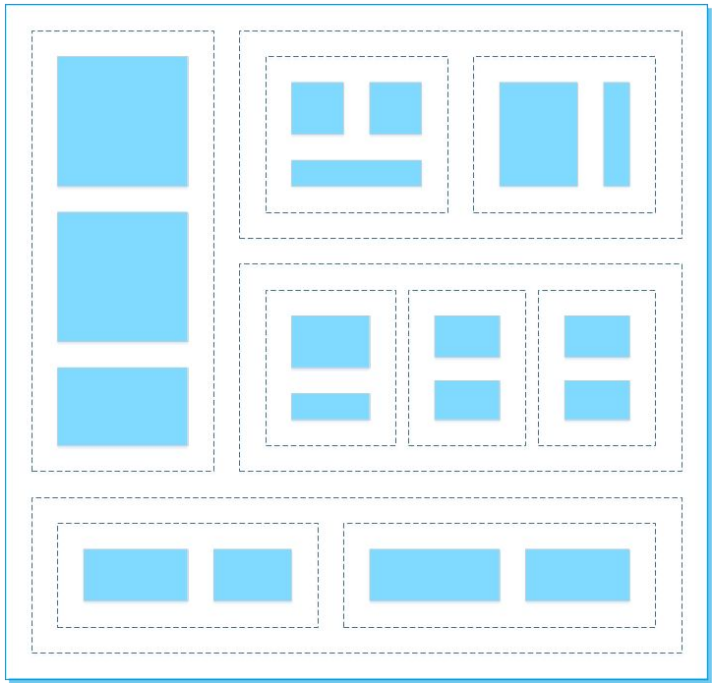
Étapes du processus de test

- Les programmes de taille importante demandent un traitement spécial
- Il n’est pas réaliste de vouloir tester un programme comme une seule unité
  - Difficulté
  - Efficacité
  - Coûts
- Le test sera structuré



Étapes du processus de test

- Comme pour les processus de conception, les tests se font par étapes
- Chaque étape constitue la suite logique de l’étape précédente
- On distingue plusieurs étapes dans ce processus
  - Tests unitaires
  - Tests d’intégration
  - Tests de système



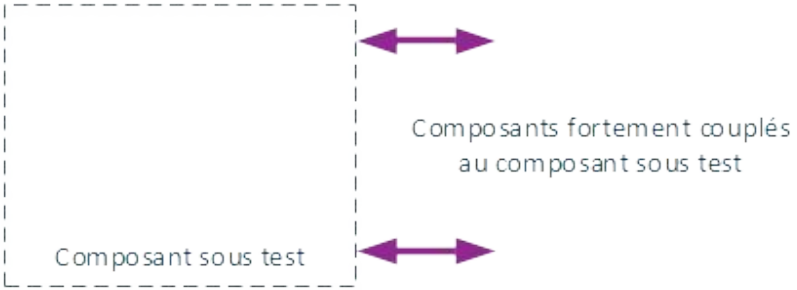
Tests unitaires

- Processus qui a pour but de tester chaque composant du logiciel de manière isolée
  - Sous-système
  - Module
  - Procédures et fonctions
- Dans un système bien conçu, chaque fonction doit avoir une spécification clairement établie
  - Il est relativement facile de concevoir des tests permettant de valider l'adéquation entre le composant et ses spécifications

Tests unitaires

- Les tests unitaires présentent plusieurs avantages
  - Concentration de l’effort de test
  - Découverte des erreurs plus facile
  - Parallélisme durant la campagne de test
  - Souvent effectué par celui ayant programmé l’unité

Tests unitaires



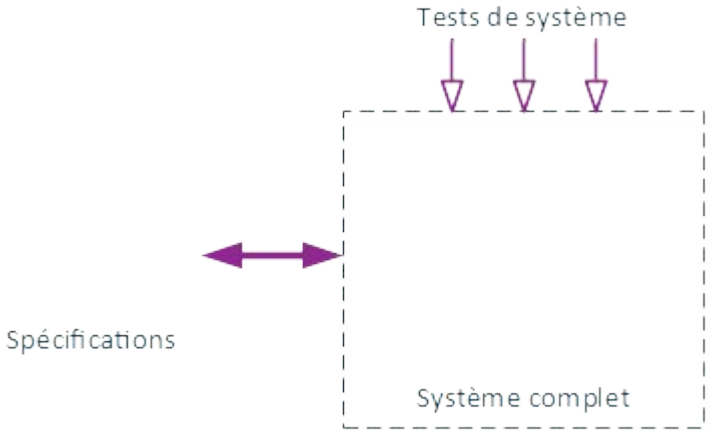
- On peut rencontrer des difficultés lorsque le couplage entre les différents composants du programme est important
- Il est dans notre intérêt de réduire le couplage durant le processus de développement pour faciliter les tests!

Tests d’intégration

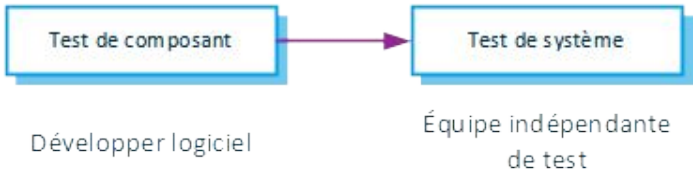
- Les différents composants du logiciel qui ont été testés individuellement sont progressivement assemblés pour construire un programme exécutable qui va être testé
- Les tests d’intégration permettent de vérifier la bonne utilisation des interfaces entre composants
- Vérifie que le système, dans son ensemble, réalise bien les fonctions spécifiées

Tests de système

- Interviennent à la fin du processus d'intégration
- Permettent de tester le système dans son ensemble avec des données réelles
- Mettent en évidence des erreurs dans un logiciel ne satisfaisant pas les niveaux de fonctionnalité ou de performance attendus par l'utilisateurs (spécifications)
- Aussi appelés tests pilotes



Étapes du processus de test



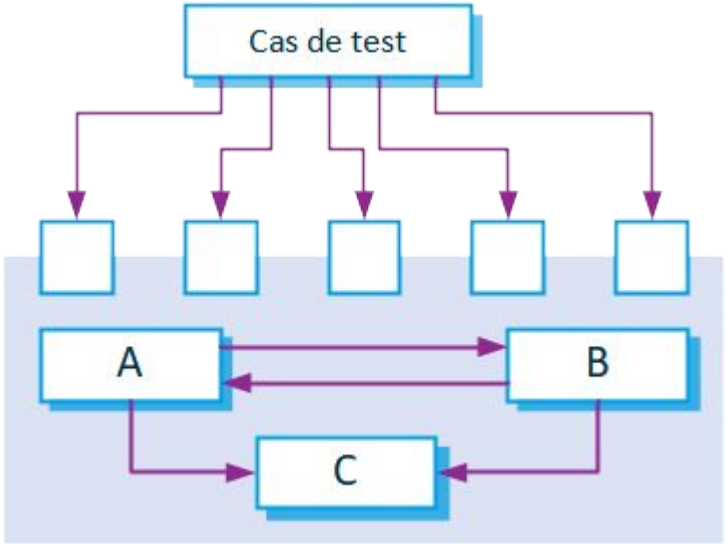
- Le modèle de processus de test (unitaire, intégration, système) est approprié pour le développement de système important
- Pour les petits systèmes ou systèmes développés selon un processus basé réutilisation, le nombre d’étapes peut être inférieur
- Les activités fondamentales sont le test de composant et le test de système

Test de composant

- Processus de test des composants individuels dans le système
  - Test de défaut
- Le développeur du composant est responsable de son test
- Différents types de composants peuvent être testés à cette étape
  - Fonction
  - Méthode
  - Classe d’objet
  - Composite
- On peut se baser sur les besoins, le test de partition et le test structurel

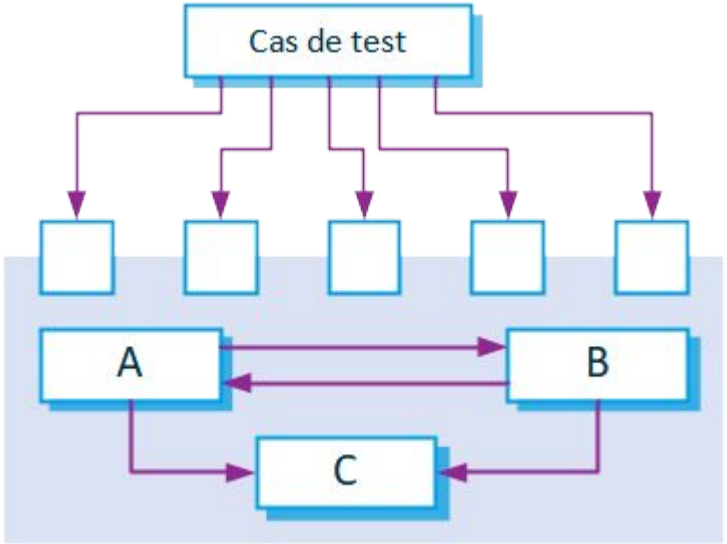
Test d’interface

- Certains composants sont des composites
  - L’accès à leur fonctionnalités se fait vie leur interface
- Tester ces composants revient à s’assurer que les interfaces se comportent conformément à leur spécification
- On les teste à partir de leurs interfaces



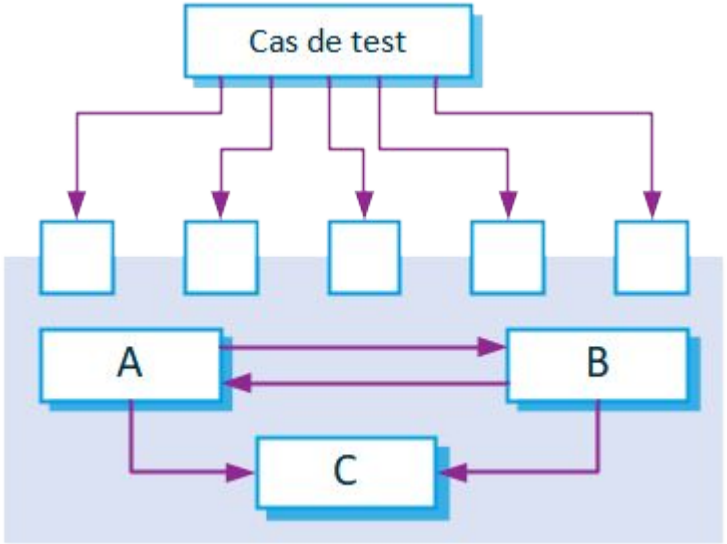
Test d’interface

- Les erreurs d’un composite peuvent être détectées en raison de l’interaction entre leurs différents composants
- On retrouve différents types d’erreurs liés aux interfaces
  - Interfaces de paramètres
  - Interfaces de mémoire partagée
  - Interfaces procédurales



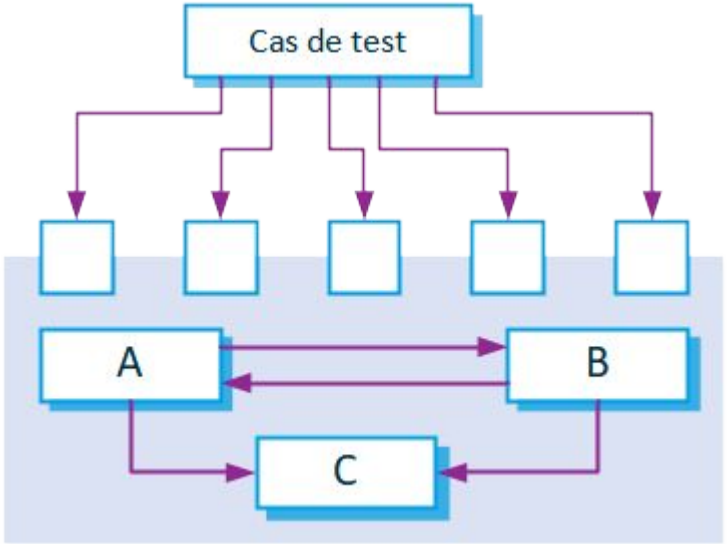
Test d'interface

- On peut subdiviser ces erreurs en trois catégories
  1. Mauvaise interface
     - Erreur d'utilisation de l'interface lors de son appel
  2. Mauvaise compréhension de l'interface
     - Mauvaise compréhension de la spécification du composant appelé
  3. Erreur de synchronisation
     - Dans les système en temps réel, les processus fonctionnent à des vitesses différentes



Test d’interface

- Certaines fautes ne se manifestent que lors d’une utilisation inhabituelle
- Les techniques de validation statiques sont plus efficaces que le test pour découvrir des erreurs d’interface



Tests de système

- Plusieurs composants sont intégrés pour implémenter les fonctionnalités d’un système
  - Le test de système consiste à tester cet ensemble
- Dans le cas du processus de développement itératif, cela revient à tester un incrément
- Dans le processus cascade, on parle de tester le système entier!

Étapes du test de système

- Test d’intégration
  - Permet la découverte d’erreur
- Test de livraison
  - Permet de validation le système
  - Satisfaction des besoins fonctionnels et non-fonctionnels
  - Boîte fermée

Tests d’intégration

- Les composants peuvent prendre plusieurs formes
  - Produits COTS
  - Composants réutilisables
  - Composants développés sur mesure
- La plupart des systèmes importants comportent plusieurs types de composants

Tests d’intégration

- L’intégration de système implique l’identification de paquets de composants qui délivrent certaines fonctionnalités du système
- Il existe différentes stratégies d’intégration de composants
  - Ascendente
  - Descendante
  - Hybride

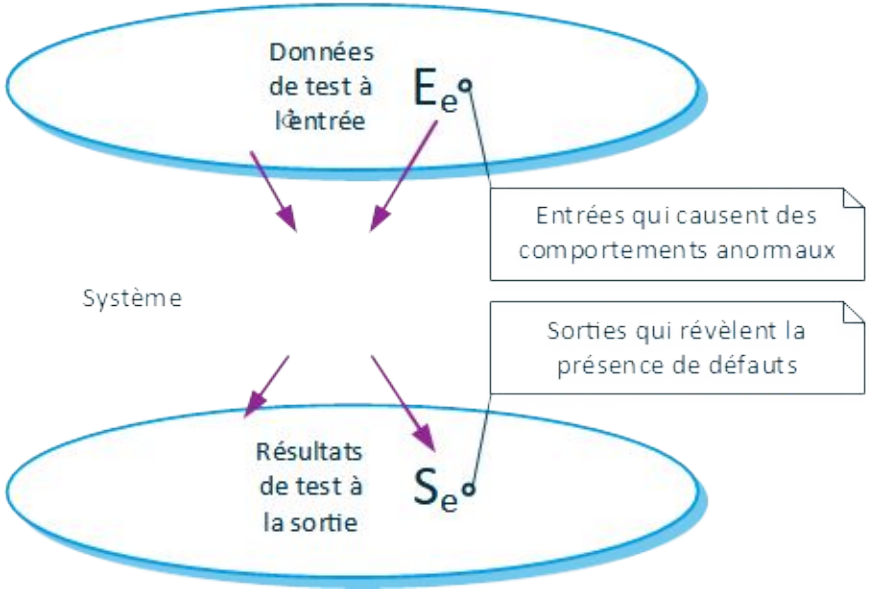
Stratégies incrémentales

- Lors du processus d'intégration, on doit choisir un ordre d'intégration des différents composants
- Cet ordre dépend de plusieurs critères
  - En programmation extrême (XP) ou Scrum, dépend des priorités fixées par le client
  - Dans d'autres approches, l'équipe de développement peut décider de l'ordre
    - Dépendances
    - Fonctionnalité
    - Etc.

Stratégie incrémentales

- On doit intégrer en premier les composants qui implémentent les fonctionnalités les plus utilisées
  - Ces fonctionnalités devront être plus fortement testées
- L'intégration d'un composant peut provoquer des erreurs
  - La correction de ces erreurs risque d'avoir un impact sur le reste des composants
  - Nécessité de développer des tests de régression
  - Les tests de régressions doivent être automatisés, sinon sont trop coûteux

Tests de livraison



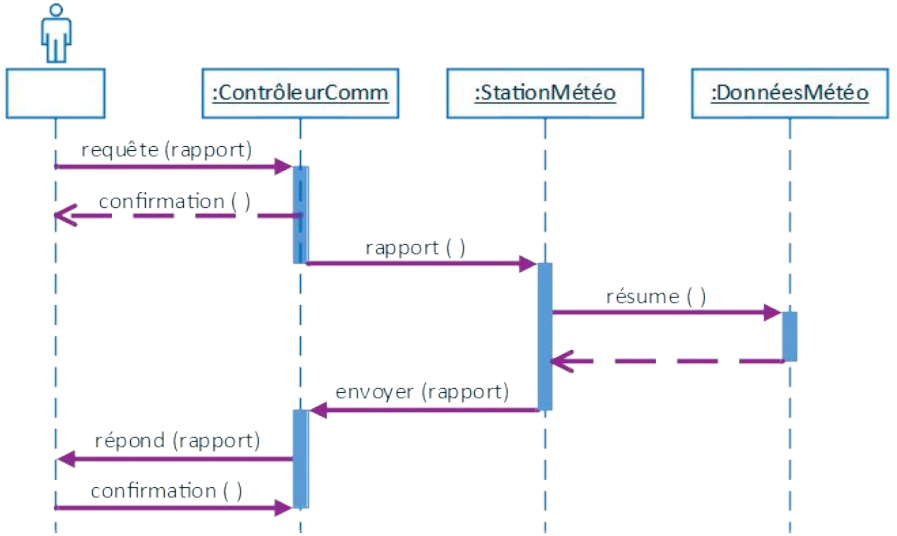
- Visent à tester la livraison qui sera distribuée aux clients
  - Augmenter la confiance
  - Démontrer la satisfactions aux exigences
  - Support des fonctionnalités
  - Performance
  - Fiabilité
  - Etc.
- Processus en boîte fermée
  - Construits à partir des spécifications

Tests de livraison

- Directives de test de livraison selon Whittaker (2002)
  - Choisir des entrées qui forcent le système à générer tous les messages d’erreur
  - Concevoir des entrées qui provoquent le débordement des buffers d’entrée
  - Forcer les sorties invalides
  - Forcer les résultats à être trop grand ou trop petit

Tests de livraison

- Pour chaque cas de test, on veut un ensemble de tests avec des entrées valides et non-valides
- Organiser ces tests basés sur des scénarios probables
- Les cas d'utilisation et diagrammes (séquence, collaboration) peuvent être utiles durant ce processus



Tests de performance

- Visent à s’assurer que le système supporte la charge pour laquelle il a été conçu
- Permet également de démontrer que le système satisfait les besoins non-fonctionnels
  - Et de découvrir des problèmes et des défauts dans le système

Tests de performance

- Pour les tests de performance, on construit un profil opérationnel
  - Ensemble de tests reflétant le comportement voulu en termes de performance
- Le test de performance, ou test de stress, consiste à concevoir des cas de test qui vont au delà de la charge maximale du système
- Ce test a deux intérêts
  - Tester le système en cas de panne afin de s'assurer qu'il n'y aura pas de dégâts irréparables
  - Faire apparaître des défauts qui ne se seraient pas manifestés autrement
- Particulièrement adapté aux systèmes distribués
  - Ces systèmes accusent de fortes dégradations en cas de surcharge

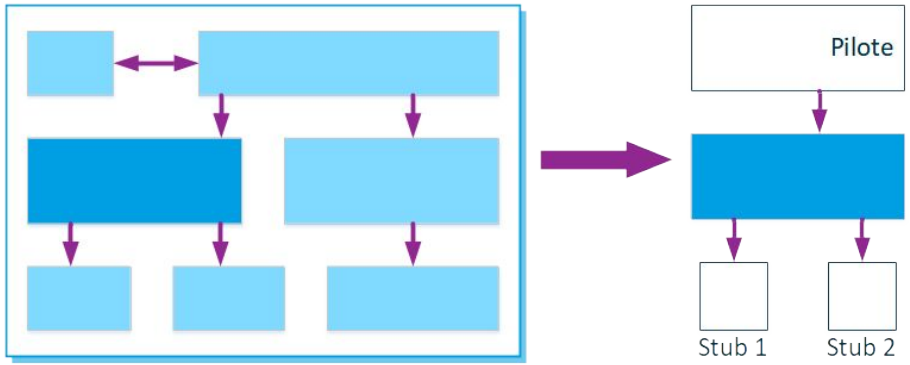
Stratégies de test

- Élément important à prendre en compte durant le processus
- Ordre dans lequel les différents composants seront testés et combinés pour construire un programme exécutable
- On se pose la question : « *Doit-on tester séparément tous les composants, puis les combiner pour tester tout le programme, ou les tester séparément en combinant le prochain composant à tester à l'ensemble des composants déjà testés?* »
- Problème de détermination du séquencement de tests
    - Étroitement lié à la stratégie d'intégration retenue

Stratégies de test

- On distingue deux principales stratégies
  - Non-incrémentale
  - Incrémentale

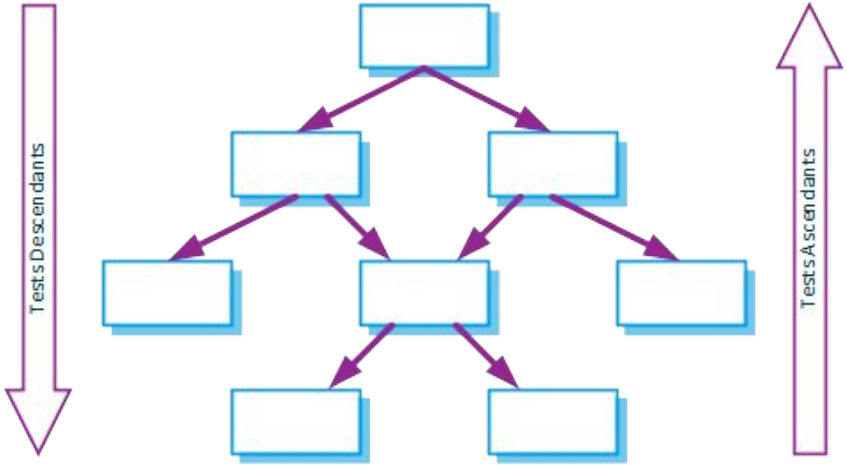
Stratégies non-incrémentales



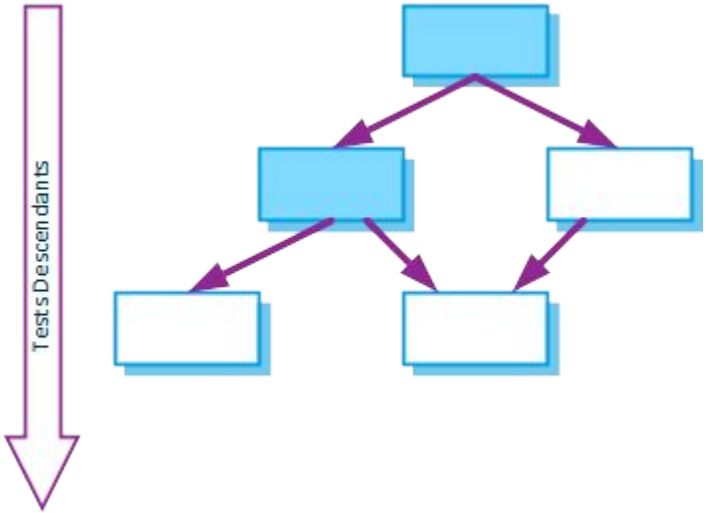
- Consiste à tester de façon indépendante tous les composants du logiciel puis à les intégrer
- Implique le développement d'un moniteur de test (*driver*) et de simulateurs de composants (*stub* ou *mock*) pour chaque composant appelé
- Approche coûteuse et peu utilisée pour des systèmes complets

Stratégies incrémentales

- Consiste à ne pas tester tous les composants isolément, mais intégrés aux composants déjà testés
  - Avantage de limiter le nombre de moniteurs et simulateurs
- Deux approches existent
  - Descendante
  - Ascendante

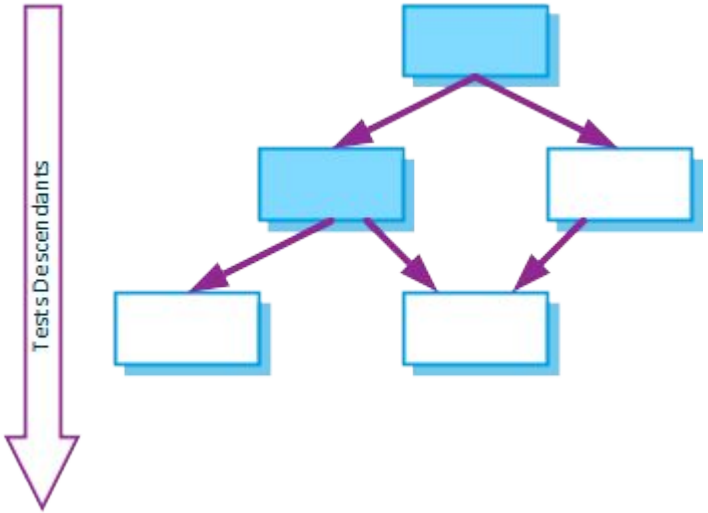


Approche descendante



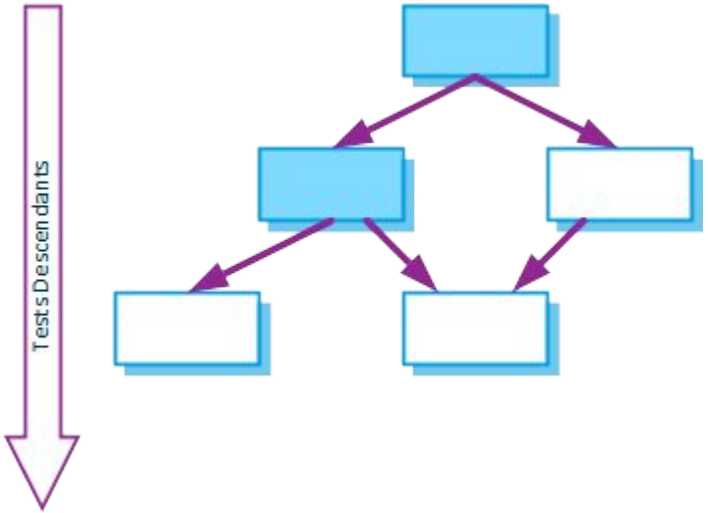
- Généralement associé à un processus de composant descendant
  - Un composant peut être testé dès qu'il a été codé
- Dans cette approche, les composants de plus haut niveau sont testés en premier

Approche descendante



- Un test de composant peut s'effectuer sans que les composants qu'il appelle aient été codés
  - Effort important au niveau de la réalisation des simulateurs de composants
- Avec cette méthode, on dispose très vite d'un système opérationnel
  - Bien que limité dans ses fonctions

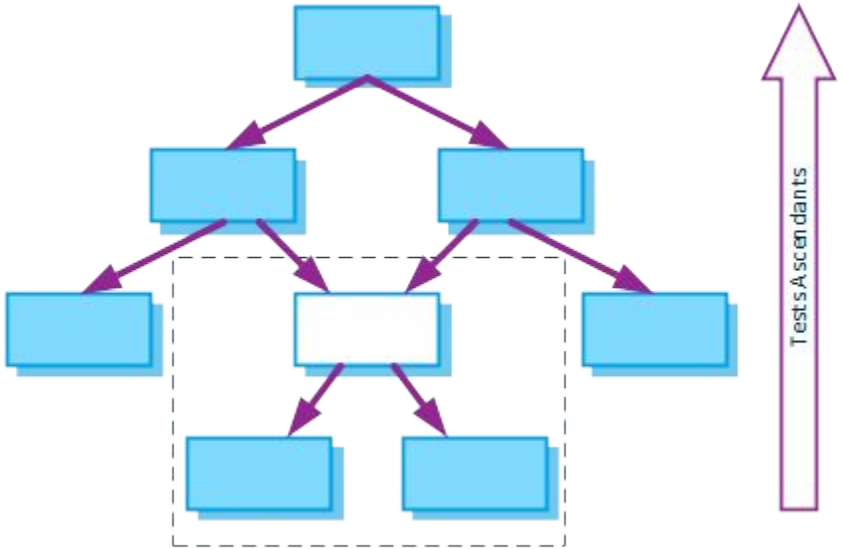
Approche descendante



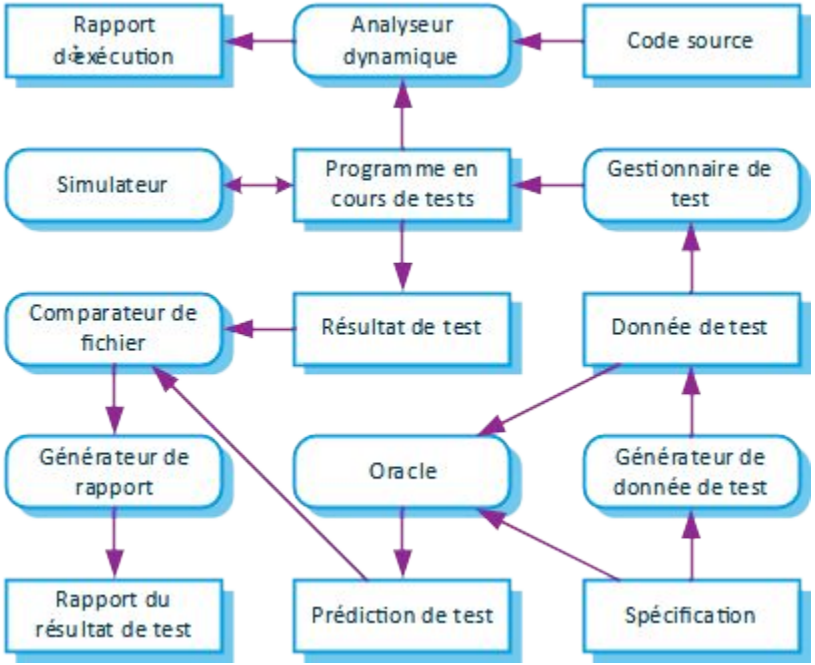
- Cependant, les tests descendants peuvent être extrêmement coûteux
  - Version provisoires de chaque fonction doivent simuler les couches basses du logiciel

Approche ascendante

- Pour réaliser des tests ascendants, il faut réaliser des programmes provisoirs qui sont des moniteurs de test
  - Permettent d'appeler les composants à tester
  - Plus facile d'observer les résultats des tests
- En général préférable
  - Il est relativement plus facile de développer des moniteurs que des simulateurs

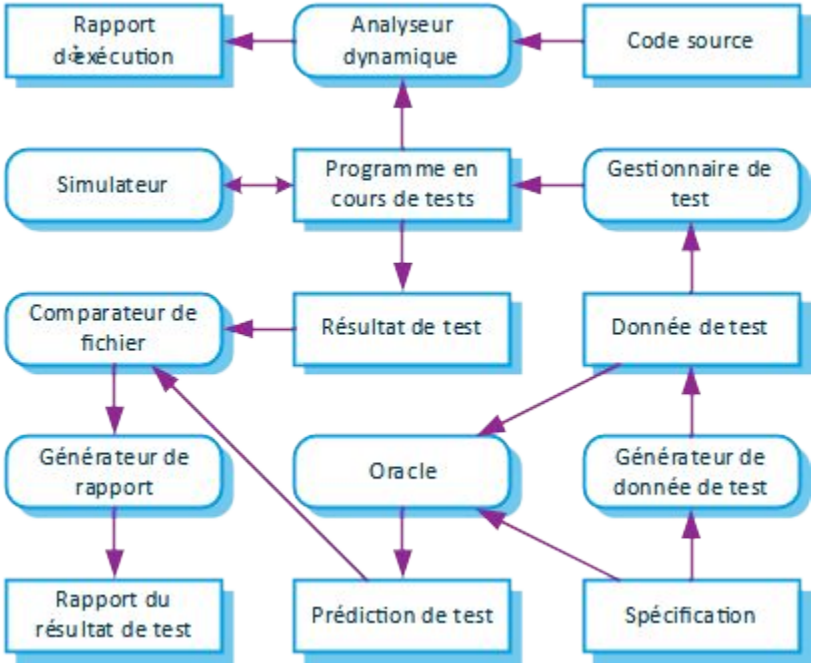


Automatisation du test



- Le test est une phase coûteuse et laborieuse du processus de développement
  - Les outils de test furent parmi les premiers à être développés
- Ces outils disposent de fonctionnalités permettant de réduire le coût du test
  - E.g., JUnit, NUnit, MSTest
- Un environnement de test est un ensemble intégré d'outils supportant le processus de test

Automatisation du test



- Gestionnaire de test
- Générateur de données de test
- Oracle
  - Programme qui, étant donné une spécification, peut prédire les résultats que donnerait le système donné s'il était conforme à ses spécifications
- Comparateur de fichier
- Générateur de rapports