Vérification et Validation

Génie Logiciel Master 1 II

Mihaela Sighireanu

Objectifs

I. Introduire la vérification et la validation (V&V) du logiciel et comprendre leurs différences.

II. Définir le plan de V&V et ses composantes.

III. Introduire le test de logiciel et ses composantes.

IV. Décrire le processus d'inspection de code et son rôle dans le V&V.

V. Décrire l'analyse statique comme une technique de vérification.

I: Vérification versus Validation

- Vérification:
  - « Avons-nous construit le produit bien? »
  - i.e., le logiciel doit être conforme à sa spécification

- Validation:
  - « Avons-nous construit le bon produit? »
  - i.e., le logiciel doit faire ce que l'utilisateur a besoin

Processus de V&V

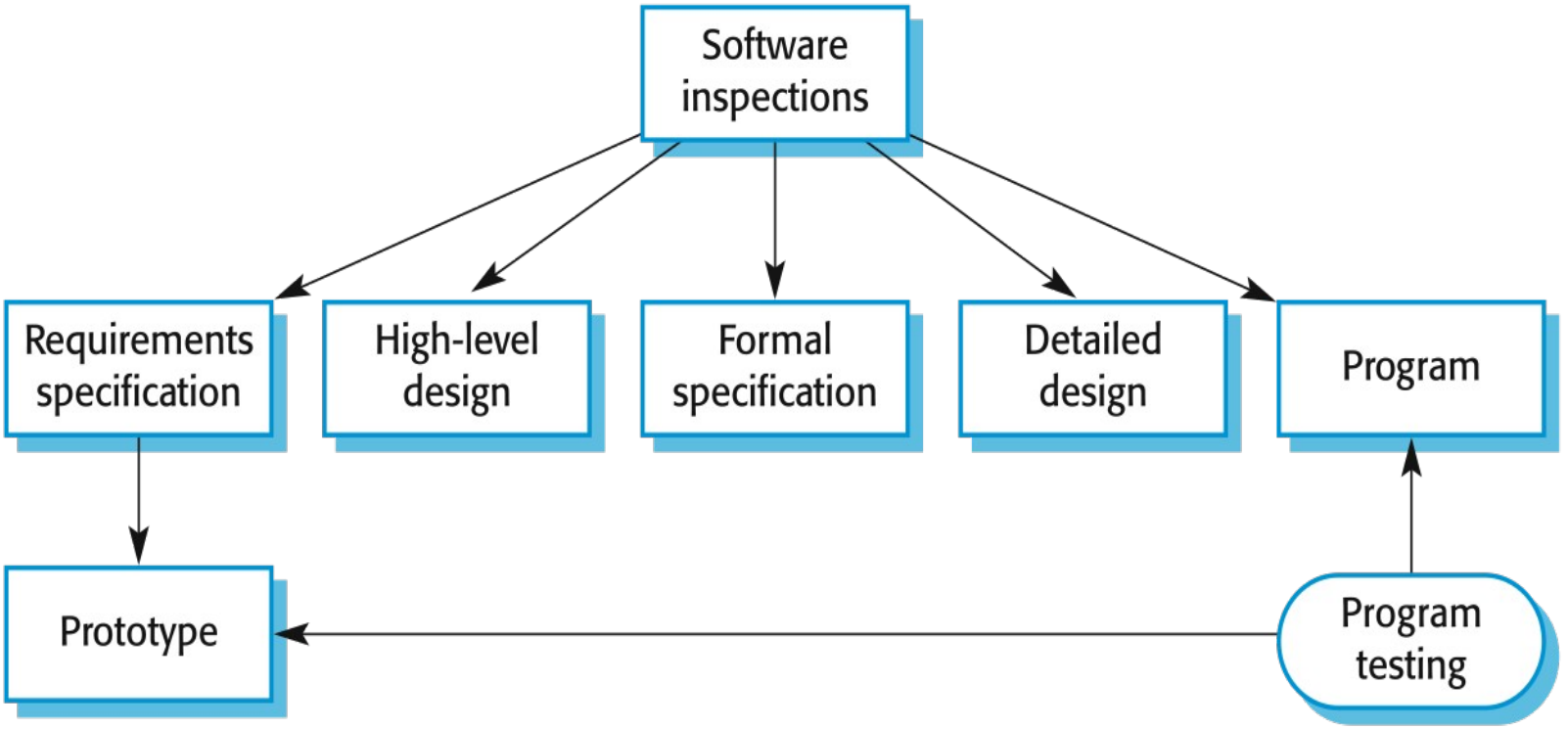
- Le processus de V&V doit être appliqué à chaque étape du développement.
- Ses objectifs:
  - Montrer que le système n'a pas de défauts.
    - Principalement pour les systèmes critiques de niveau A.
  - Découvrir des défauts dans le système.
    - Ceci ne veut pas dire TOUS les défauts mais plutôt augmenter la confiance dans ce système.
  - S'assurer que le système construit est utile et utilisable en pratique.

Types de V&V

- Statique = inspection/analyse statique (i.e., sans exécution) du système pour découvrir des problèmes ou prouver sa correction.
  - Sans exécution, peut être manuelle ou automatique

- Dynamique = expérimenter et observer le comportement à l'exécution du système.
  - Le système est exécuté sur des données de test et son exécution est observée (manuellement ou automatiquement).

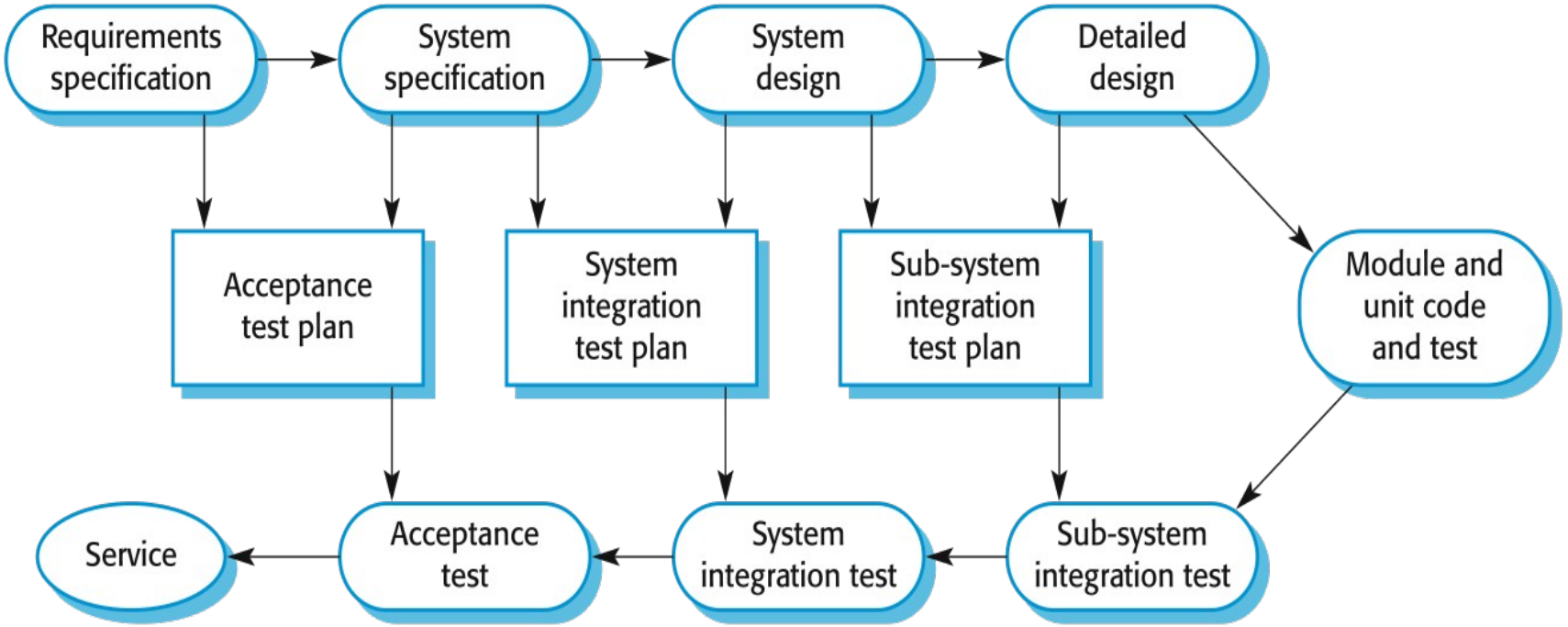
V&V statiques et dynamiques



II: Plan de V&V

- Le plan doit être fait avec attention pour obtenir le plus de résultats en test et inspection.
- Il doit être fait très tôt pour commencer tôt dans le processus de développement.
- Doit établir le rapport entre la part de la V&V statique et dynamique.
- Le plan de test doit indiquer les politiques à suivre pour écrire les tests plutôt que les tests eux même.

V&V et développement en V



Structure d'un plan de test

- Processus de test
  - Description des principales phases du test.
- Traçabilité des charges
  - Comment les charges ont été testées (tests en annexes).
- Composants testés
  - Liste de composants testés (tests en annexes).
- Agenda des tests
  - Agenda de tests, les ressources nécessaires et son intégration avec l'agenda du développement.
- Procédure d'enregistrement des tests
- Besoins logiciels et matériels
- Contraintes diverses
  - e.g., insuffisance en testeurs, etc.

III: Test du logiciel

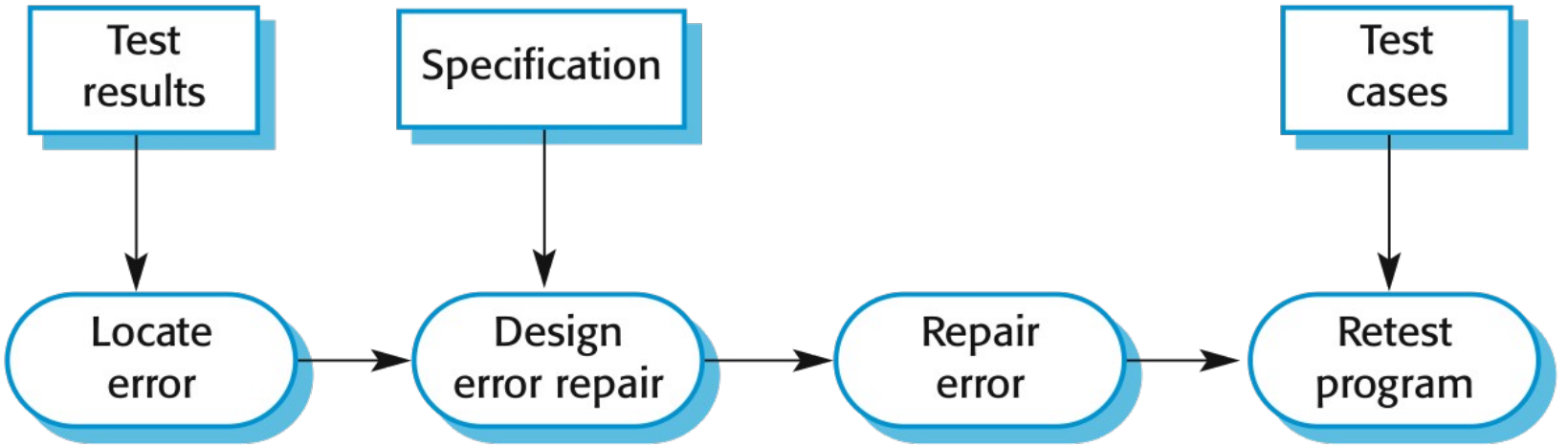
- Peut révéler la présence d'erreurs et NON leur absence.
- La technique utilisable actuellement pour valider les charges non-fonctionnelles car elle est basée sur une exécution du logiciel.
  - L'analyse statique est au niveau de la recherche pour ce type de charges.
- Doit être fait en conjonction avec l'inspection statique pour améliorer la couverture du processus de V&V.

Types de tests

- Test de défaut
  - Utilisé pour trouver les défauts du système.
  - Un test de défaut bon est celui qui découvre des défauts dans le système.
- Test de validation
  - Utilisé pour montrer que le logiciel satisfait les charges.
  - Un test de validation bon est celui qui montre que la charge sous test est bien implémentée.

Test versus dépannage

- Test de défaut et dépannage (debug) sont deux processus différents:
  - Le test a comme objectif de révéler la présence de défauts dans un logiciel
  - Le dépannage a comme objectif de localiser et réparer ces défauts.
  - Le dépannage formule des hypothèses sur la cause du défaut et utilise le test pour les valider.



Test dans ce cours

(a) Décrire les principes du test de système et test de composant.

(b) Donner quelque stratégies pour écrire des tests.

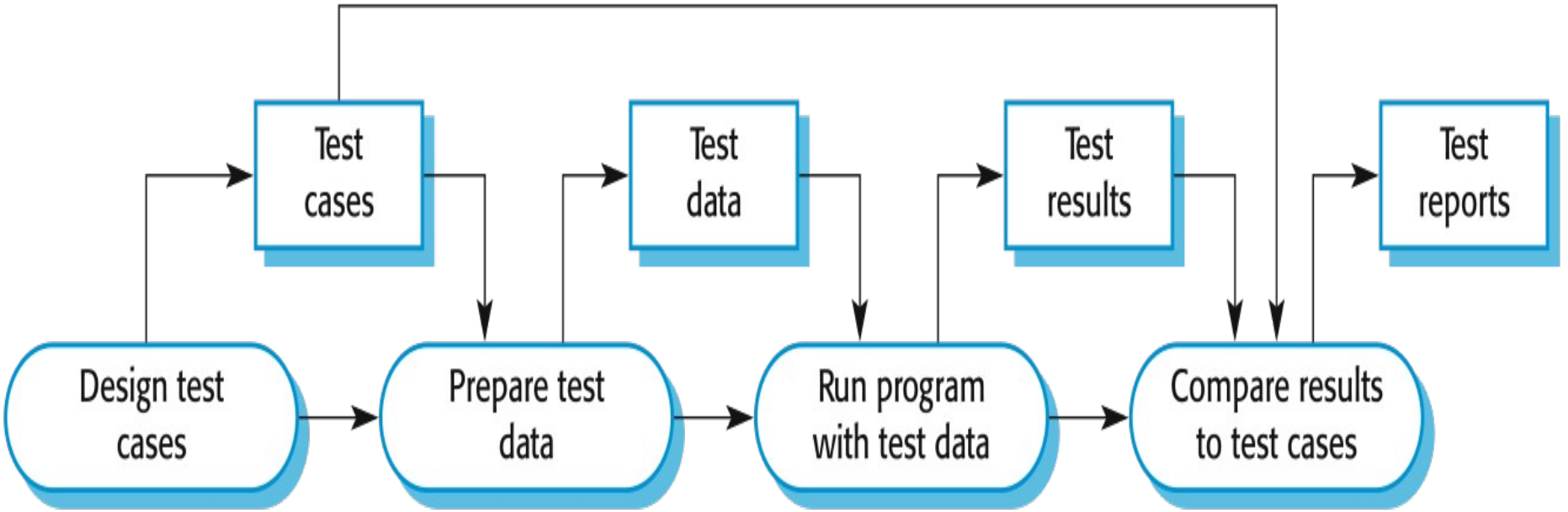
(c) Décrire une chaîne d'outils nécessaires pour l'automatisation du test.

- … voir plus en M2 LC+LP: Méthodes de test

III(a): Processus de test: classes

- Test de composant
  - Concerne les briques de base du logiciel
  - En général, fait pas le programmeur du composant
  - ... sur la base de son expérience avec le programme.
- Test de système
  - Concerne des groupes de composants intégrés pour créer un système ou un sous-système.
  - En général, fait par une équipe indépendante
  - … sur la base de la spécification du système.

Processus de test: étapes



Processus de test: politique

- Le test exhaustif est impossible à réaliser...
- La politique de test définit l'approche utilisée pour sélectionner les tests à écrire et exécuter.
- Exemple:
  - Tester toutes les fonctions accessibles par des menus.
  - Tester la combinaison (en séquence) des fonctions accessibles par les menus.
  - Tester les fonctions avec entrées utilisateur ayant des valeurs correctes et incorrectes.

Test de système

- Demande la composition de briques de base pour construire un sous-système ou le système.
- Peut concerner une distribution partielle ou une extension/incrément d'une distribution.
- Deux phases:
  - Test d'intégration: l'équipe a accès au code et le test est fait au fil de l'intégration des composantes.
  - Test de distribution (release): l'équipe teste le système complet comme une boîte noire (i.e., sans regarder le code).

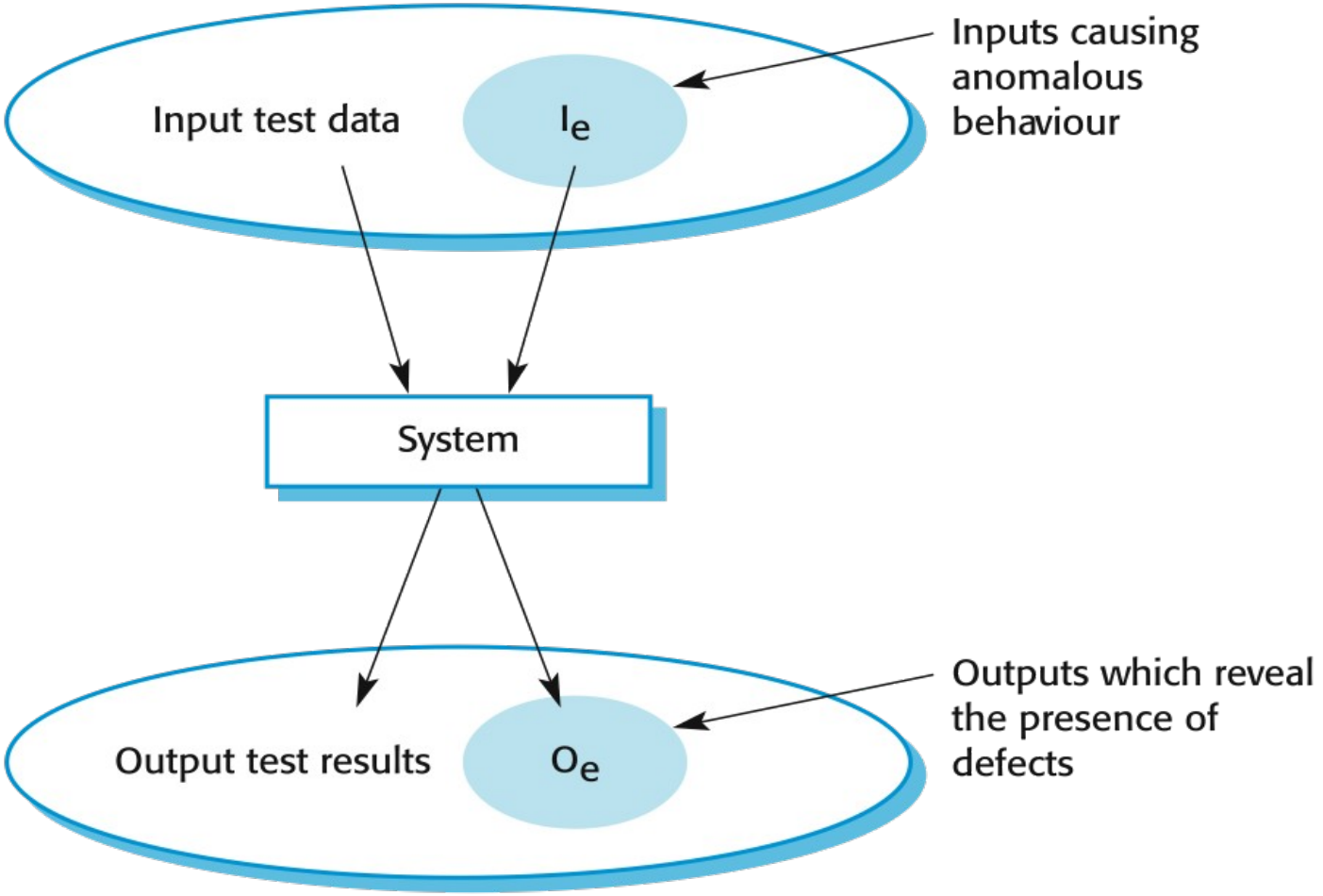
Test d'intégration

- Compose les briques et teste pour défauts provenant de l'interaction de ces briques.
- Pour faciliter la localisation des défauts, l'intégration doit être incrémentale.
- Approches:
  - Top-down: construire le squelette fonctionnel et puis le décorer avec des briques.
    - Facilite la validation de l'architecture et la démonstration précoce d'un prototype.
  - Bottom-up: intégrer les briques par groupes et puis attaquer les fonctionnalités.
    - Facilite l'implémentation du test.
- Du code supplémentaire peut être nécessaire pour observer les résultats des tests.

Test de distribution

- But: accroître la confiance des clients
- Test de validation, i.e., montre que le système satisfait ses charges.
- Approche « boîte noire »:
  - Tests basés sur la spécification uniquement, sans connaissance des choix d'implémentation.
  - On teste les fonctionnalités, les performances et le comportement aux limites (stress) du logiciel.

Test en « boîte noire »



Test en « boîte noire »: politiques

- Comment choisir les tests pour révéler des défauts:
  - Choisir les entrées qui génèrent toutes les messages d'erreur.
  - Définir des entrées qui provoquent le « buffer overflow ».
  - Répéter les mêmes entrées ou séries d'entrées plusieurs fois.
  - Forcer la génération des sorties invalides.
  - Forcer des sorties trop courtes ou trop larges.
  - Couvrir tous les séquences d'utilisation avec des scénarios positifs et négatifs.
  - etc.

Exemple: LIBSYS

- Tester le mécanisme de login avec un utilisateur correct et incorrect pour tester que les utilisateurs corrects sont acceptés et ceux incorrects sont rejetés.
- Tester la fonctionnalité de recherche avec 0, 1, 2 ou plusieurs sources sélectionnées et avec des articles existants ou non-existants.
- Tester l'interface d'affichage des documents avec différents formats.
- Tester le mécanisme de téléchargement.
- Tester l'envoi d'email pour confirmer le téléchargement correct du document.

Test de performance

- Teste les propriétés non-fonctionnelles comme le temps de réponse et la fiabilité.
  - Tests avec une charge croissante de tâches;
  - Tests en présence d'erreurs de transmission ou de temps de réponse très grands;
  - etc.

Tests aux limites

- Exécute le système en utilisant des entrées très proches ou au delà des limites prévues.

- Permet de tester le comportement du système en cas de panne et ainsi s'assurer qu'il ne provoque pas des pertes de service ou de données inacceptables.

- Très utilisés dans les systèmes distribués (sites web) pour exhiber des comportements anormaux en cas de charge du réseau.

Test de composant

- = test unitaire, processus qui teste les composantes en isolation.
- C'est un test de défaut.
- La composante sous test (IUT - Implementation Under Test) peut être:
  - Des fonctions ou des méthodes d'un objet;
  - Des classes contenant champs et méthodes;
  - Des classes composites avec des fonctionnalités accessibles par une classe abstraite ou interface.

Test de classe

- Pour couvrir par test une classe
  - Tester toutes les méthodes de la classe;
  - Tester les setters et getters de tous les champs;
  - Tester les objets dans tous les états possibles de la classe.
- L'héritage rend cette tâche plus difficile...
- Exemple: la bibliothèque LASER (le voir sous Eclipse et JUnit)

Test d'interface

- Trouver des défauts à cause d'erreurs dans les interfaces ou d'hypothèses invalides sur les méthodes de l'interface.
  - Un composant appelle une méthode de l'interface d'un autre composant avec un ordre inversé des paramètres ou avec des pointeurs null.
  - La vitesse de réponse d'une méthode d'interface ne correspond pas au critères d'utilisation.
  - L'activation d'un composant ne se fait pas dans le bon ordre et casse les hypothèses de son utilisation.
  - etc.

III(b): Ecriture des tests

- Vocabulaire:
  - Cas de test (TC) : une exécution du programme déclenchée par des données de test (DT).
  - Suite de tests (TS) : un ensemble de DT.
  - Objectif de test (TO) : comportement à tester.
  - Système sous test (SUT ou IUT) : système/composant à tester.

- But: écrire un ensemble de tests avec un « bon » couvrement de V&V.
- Approches:
  - En partant des charges: requirement-based
  - En partant des entrées: partition testing
  - En partant du code: structural testing ou en « boîte blanche »

Test des charges

- Un bon principe pour l'écriture des charges est que les charges doivent être testables.
- C'est un test de validation:
  - Pour chaque charge écrire une TS.
  - Une spécification complète des charges est très utile à ce niveau.
- XP: spécifier les charges en écrivant leurs tests.

Exemple: charges LIBSYS

- L'utilisateur doit être capable de rechercher un article dans un ensemble des DB ou dans un sous-ensemble qu'il a sélectionné.
- Le système doit fournir les programmes de visualisation pour tous les formats de documents.
- A chaque commande soit être associé un numéro unique que l'utilisateur doit pouvoir le copies dans son compte.

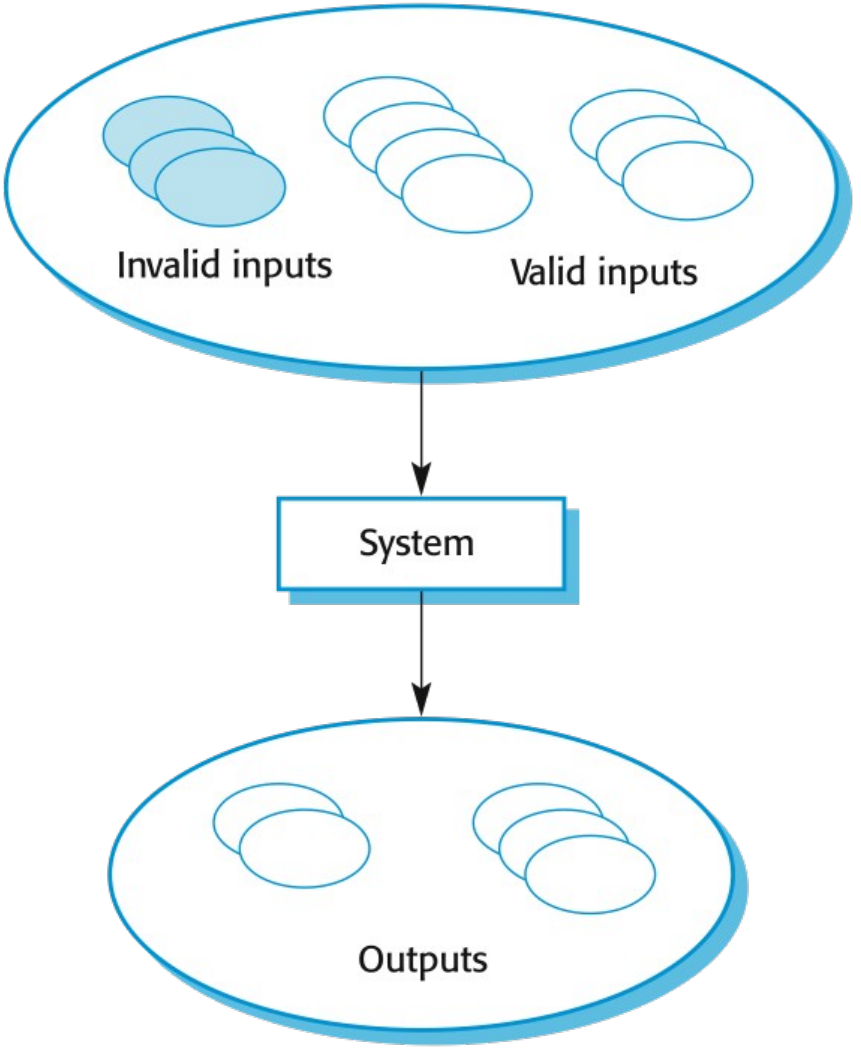
Exemple: tests LIBSYS

- Tester des recherches pour des articles connus et absents avec un ensemble de 1 DB.
- Tester des recherches pour des articles connus et absents avec un ensemble de deux DB.
- Tester des recherches pour des articles connus et absents avec un ensemble de plus que deux DB.
- Tester la sélection d'une DB et lancer la recherche pour des articles présents et absents.
- Tester la sélection de plusieurs DB et lancer la recherche pour des articles présents et absents.
- etc.

Test des entrées/sorties

- Les entrées et les résultats peuvent être partagés en classes dont les éléments concernent le même comportement du IUT.

- Une telle classe = classe d'équivalence ou partition. Les TC doivent être choisis dans chaque classe.
- On part de la spécification!



Exemple: search

Procedure Search

(Key:ELEM, T: SEQ of ELEM) returns

(Found: BOOLEAN, L: ELEM_INDEX)

Pre-condition:

- La séquence T a au moins un élément

Post-condition:

- L'élément est trouvé à la position L
  - (FOUND and T[L]=Key)
- Ou Key n'est pas dans le tableau
  - (not FOUND and
    not (!i. 0<=i<=size(T) et T[i]=Key))

- Classes d'équiv des entrées:
  - Conformes à la pre-cond
  - Invalides pour la pre-cond
    - e.g., T vide
  - Key dans T
  - Key absent du T

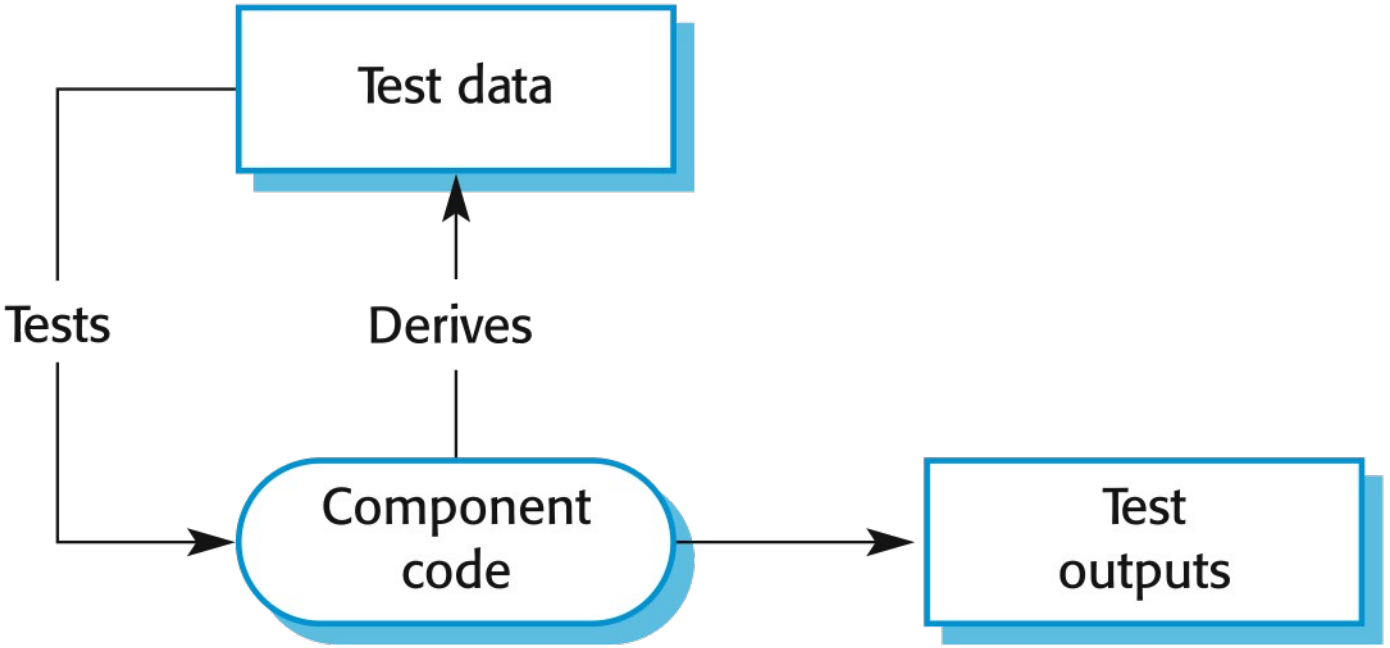
Exemple: search

Sequence	**Element**
Single value	In sequence
Single value	Not in sequence
More than 1 value	First element in sequence
More than 1 value	Last element in sequence
More than 1 value	Middle element in sequence
More than 1 value	Not in sequence

Input sequence (T)	**Key** (Key)	**Output** (Found, L)
17	17	true, 1
17	0	false, ??
17, 29, 21, 23	17	true, 1
41, 18, 9, 31, 30, 16, 45	45	true, 7
17, 18, 21, 23, 29, 41, 38	23	true, 4
21, 23, 29, 33, 38	25	false, ??

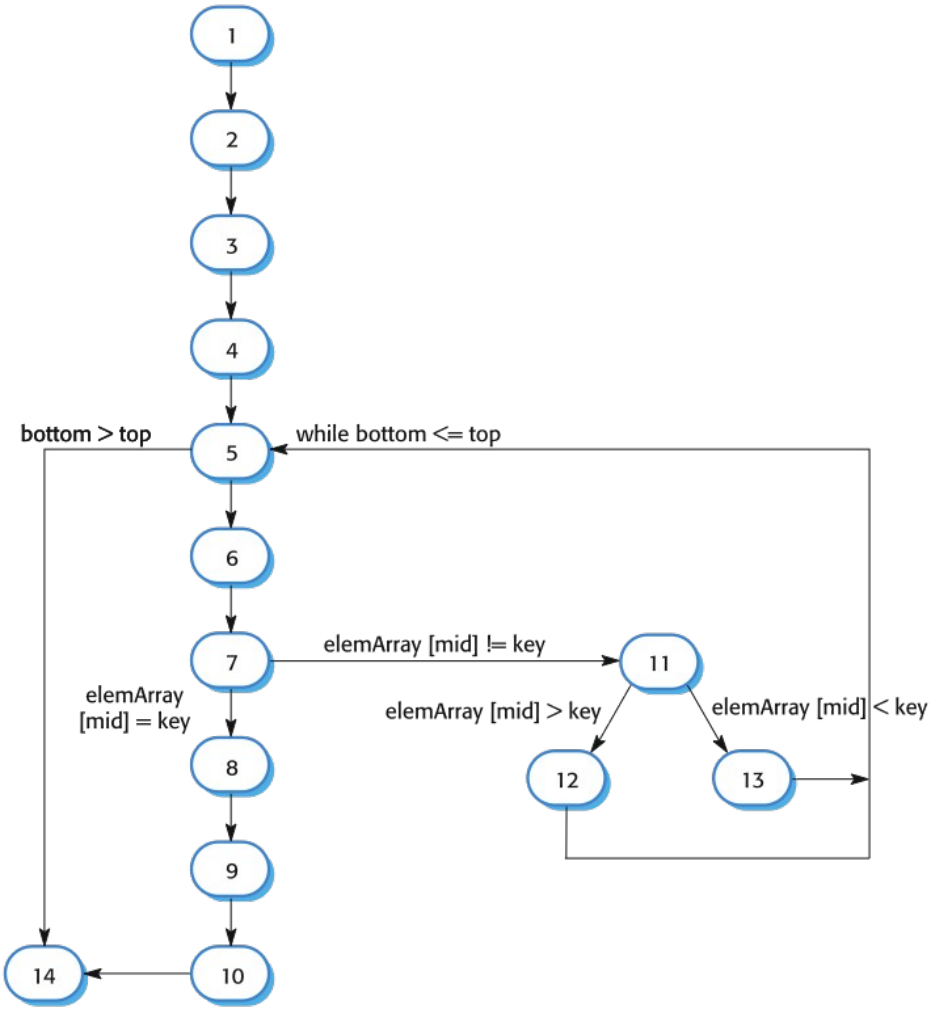
Test structurel

- = en « boîte blanche » car les tests sont écrits en partant de la structure du code.
- But: générer des DT qui exécutent le programme pour couvrir toutes les instructions/branches/etc.



Principe du test structurel

- Partir du graphe de contrôle du programme:
  - Les noeuds sont des instructions ou des décisions.
  - Les arcs sont des changement de contrôle.
  - Exemple: binary search
- Définir des TC tels que des chemins indépendants sont exécutés:
  - 1 2 3 4 5 6 7 8 9 10 14
  - 1 2 3 4 5 14
  - 1 2 3 4 5 6 7 11 12 5 …
  - 1 2 3 4 6 7 2 11 13 5 ...



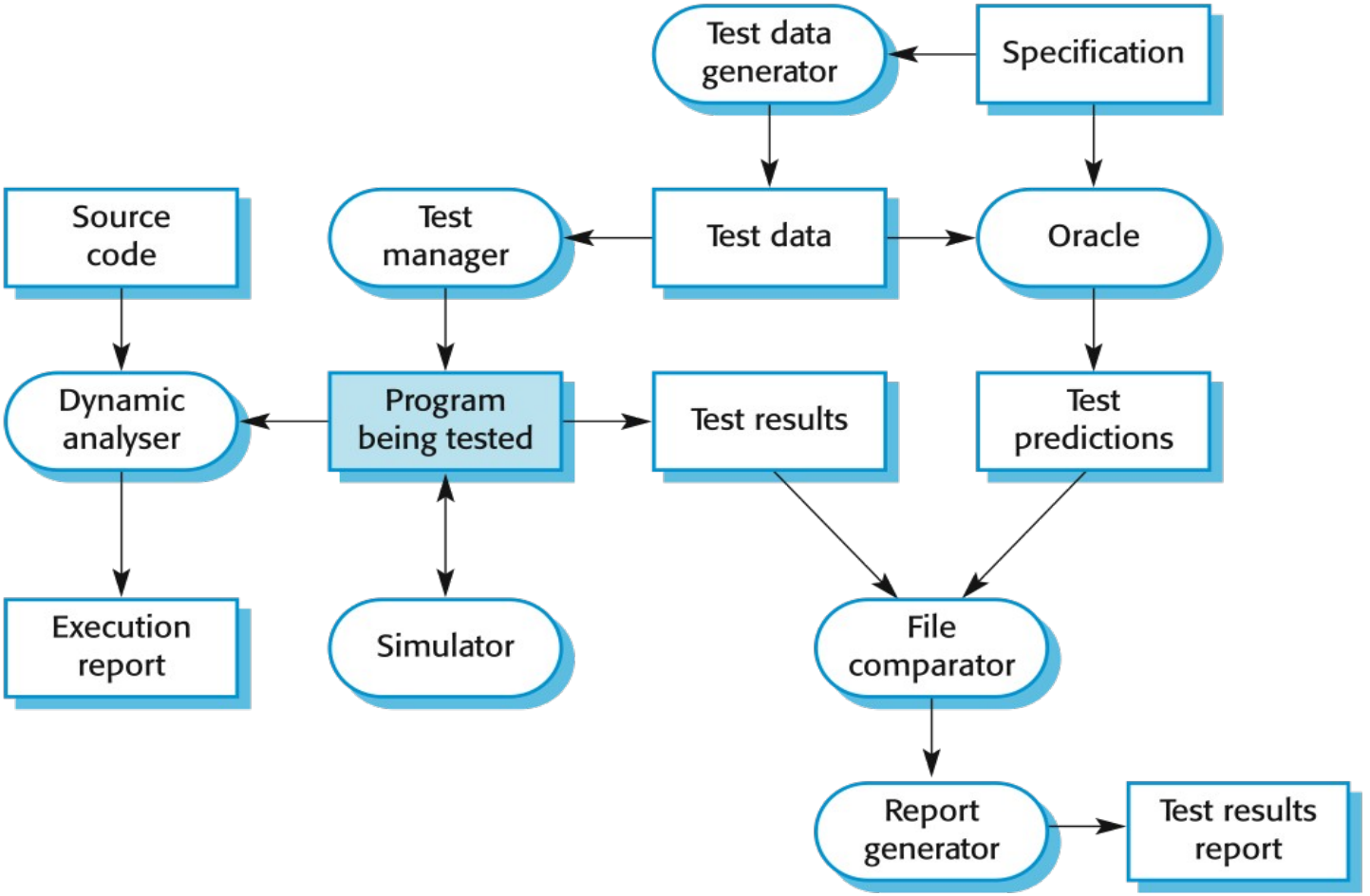
Test structurel: politiques

- Critères de couverture du code:
  - S: Toutes les instructions
  - D: Toutes les arcs (décisions)
  - C: Toutes les conditions atomiques
  - DC: Toutes les combinaisons D et C
  - MC/DC: Toutes les combinaisons booléennes de C pour une D
  - All-uses: tous les chemins entre la définition et l'utilisation d'une variables
  - ...

III(c): Automatisation du test

- Le test est très coûteux: jusqu'à 50% du coût du logiciel.
- Des outils d'aide au test existent pour:
  - L'exécution automatique des tests: suite Xunit
  - Le calcul de la couverture
  - La comparaison des résultats
- … mais il faut souvent les adapter au logiciel développé.

Un cadre automatique idéal



IV: Inspections de code

- Consiste à examiner le code, sans l'exécuter, soit par des humains (inspection) soit par un logiciel (analyse statique).
- Comme l'exécution du code n'est pas requise, ces méthodes peuvent être appliquées avant d'avoir une implémentation (même compilable).
- Appliquable à tout autre représentation que du code (cahier de charges, design, suite de tests, etc.).
- Le but: découvrir des défauts ou des anomalies.
- Le principe: plusieurs défauts peuvent être révélés par une lecture de code par des relecteurs qui ont une expertise du domaine/du langage de programmation/du formalisme de spécification/du design.

Inspection et test

- L'inspection et le test sont des méthodes complémentaires; les deux peuvent être utilisés pendant le processus V&V:
  - L'inspection vérifie la conformité à une spécification mais pas à une charge utilisateur.
  - L'inspection ne peut pas vérifier des charges non-fonctionnelles comme les performances, l'ergonomie, etc.
  - Le test demande une implémentation exécutable.

Inspection de code

- But: détecter des défaut mais pas le corriger!
- Défauts recherchés (exemples) :
  - Erreurs logiques
  - Mauvaise utilisation des variables, par exemple sans initialisation, avec une sémantique différente...
  - Non respect des standards de codage, etc.

Pré-conditions de l'inspection

- Une spécification précise doit être disponible.
- Les inspecteurs doivent connaître des standards de codage de l'entreprise.
- Des exemples de code correct doivent être disponibles.
- Une liste d'erreurs (checklist) à inspecter doit être prête.
- Avoir l'acceptation de la hiérarchie pour le coût supplémentaire au stade précoce du développement.
- Ne pas utiliser l'occasion pour évaluer le personnel.

Processus d'inspection

1. Planification (moment, participants, code)
2. Amorçage (distribution des tâches)
3. Préparation individuelle
4. Réunion d'inspection
5. Conclusions et traitement des résultats
6. Suivi des conclusions

Procédure d'inspection

- Présentation du système à inspecter aux Insp.
- Distribution du code et des documents aux Insp.
- Inspection avec enregistrement des défauts trouvés.
- Modification du code pour corriger les défauts.
- Re-inspection pour s'assurer des résultats (non-obligatoire).

Participants

- Auteur ou propriétaire du code
  - Le programmeur responsable à produire le code et les documents nécessaires à son inspection. Il est aussi responsable à corriger les défauts trouvés lors de l'inspection.
- Inspecteur
  - Trouve des défaut, anomalies, omissions, inconsistances dans le code et/ou le document; peut même attaquer des choix de conception.
- Lecteur
  - Présente le code aux inspecteurs lors de la réunion d'Insp.
- Secrétaire de séance
  - Enregistre les résultats de la réunion.
- Médiateur et/ou responsable d'inspection
  - Gère la réunion et fait un rapport à la hiérarchie; améliore le processus d'inspection, les listes d'erreurs et les standards de codage.

Checklist

- Liste des erreurs les plus communes à surveiller pendant l'inspection.
- Les défauts répertoriés dépendent du langage de programmation utilisé.
- Plus le typage est faible, plus longue sera la liste.
- Exemples: initialisation mauvaise/absente des variables, nommage incorrect des constantes, terminaison de boucle, etc.

Exemples de checklist

- Erreurs de données
  - Toutes les variables sont initialisées avant utilisation?
  - Toutes les constantes utilisées sont nommées?
  - La taille des tableaux est Size ou Size-1?
  - Les chaînes de caractères ont un caractère de fin?
  - Les tampons peuvent déborder?
- Erreurs de contrôle
  - Les conditions des IF sont correctes?
  - Toute boucle termine?
  - Les expressions/instructions composées sont correctement parenthésées?
  - Tous les case sont traités dans un CASE? Le break est présent?

Exemple de checklist

- Erreurs d'entrée/sortie
  - Tous les variables d'entrée sont utilisées?
  - Toutes les variables de sortie sont initialisées avant la sortie?
  - Les valeurs d'entrée non prévues peuvent corrompre le système?
- Erreurs d'interface
  - Tous les appels de méthode/fonction ont le nombre correct de paramètres?
  - Les paramètres actuels et formels ont la même sémantique?
  - Les paramètres sont dans le bon ordre?
- Erreurs d'accès à la mémoire
  - Le changement d'une structure modifie correctement les liens?
  - La mémoire dynamique est bien allouée et initialisée? Est-elle libérée?
- Gestion des exceptions
  - Toutes les conditions d'erreurs ont été gérées (par des exceptions/messages)?

Performances de l'inspection

- En vitesse:
  - 500 instructions/heure en présentation
  - 125 instructions/heure/personne en préparation
  - 90/125 instructions/heure en réunion
- Coût très élevé: 40h*homme pour 500 lignes
- Mais très efficace et flexible: 80% des erreurs en moyenne pour tout type de code.

IV: Analyse statique

- = (Semi-)Algorithmes d'analyse de code pour:
  - Découvrir des défauts potentiels
  - Synthétiser des propriétés invariantes du code
- N'impose pas que le code soit exécutable et peut s'appliquer sur une partie du code.
- Basée sur des heuristiques/approximations sûres (mais pas exactes), ses résultats doivent être interprétés car présence des « faux positifs ».
- Une aide importante pour les inspecteurs, mais ne remplace pas complètement l'inspection de code.
- Techniques utilisées dans les compilateurs.
- Moins efficace pour les langages structurés/fortement typés (par exemple Java).

Exemples d'analyses

- Défauts de données:
  - Variables utilisées avant initialisation
  - Variables déclarées mais pas utilisées
  - Variable non lue entre deux affectations
  - Violation des bornes de représentation
  - Division par 0
- Défauts de contrôle
  - Code mort
  - Boucle infinie
- Défauts I/0
  - Double affichage sans affectation
- Défauts d'interface
  - Paramètres en mauvais nombre/type
  - Résultat de fonction non utilisé
  - Fonctions non utilisées
- Défauts de gestion mémoire
  - Mémoire non initialisé/libérée
  - Arithmétique de pointeurs complexe

Types d'analyse statique

- Analyse du flot de contrôle: code mort, boucles avec points d'entrée/sortie complexes
- Analyse d'utilisation des données: utilisation de variables, conditions de test redondantes ou inutilisées, etc.
- Analyse d'interfaces: conformité entre déclaration et utilisation de procédures.
- Analyse du flot d'information: calcule les relations entre les variables d'entrée et de sortie.
- Analyse de chemins: identifie tous les chemins d'exécution possibles et leurs instructions.
- Voir cours en M2

Exemples d'outils

- Lint ou Splint pour C
- Jlint pour Java
- Py-lint pour Python
- etc.

Vérification et méthodes formelles

- Les méthodes formelles d'analyse
  - Preuve
  - Model-checking

peuvent être utilisées quand une spécification formelle du système existe.

- = Analyse statique exacte (quand c'est décidable).

Pro méthodes formelles

- Réaliser une spécification formelle demande une analyse approfondie du système ce qui peut révéler des défauts, incohérences, etc.
- Aide à l'inspection de code.
- Permet une génération de code/test/certification automatisée.

- Exemple: Cleanroom

Contre méthodes formelles

- Demande des notations/maths qui ne sont pas comprises par les experts du domaine.
- Très coûteux (experts) pour produire une spécification formelle et encore plus pour prouver que l'implémentation satisfait la spéc.
- On peut atteindre le même niveau de confidence en utilisant des techniques de V&V moins coûteuses.

Conclusion

- Génie logiciel
  - Travail important mais difficile car...
  - il n'existe pas UNE solution générale...
  - mais un ensemble de principes, techniques et outils...
  - qui évolue en permanence avec les nouveaux types de logiciel.
  - Demande de la rigueur, de l'abstraction, des connaissances approfondies en programmation et maths, des facilités d'écriture et expression.

Projet

- Date limite du rendu: vendredi 28 mai à 16h
  - au secrétariat de l'UFR (papier) ou par mail
- Contenu du rendu:
  - Cahier de charges: prendre en compte les commentaires faits au premier rendu.
  - Spécification formelle: B ou ADT
  - Design: diagrammes UML
  - Plan de test avec suites de test d'acceptation, d'intégration, de fonctionnement.