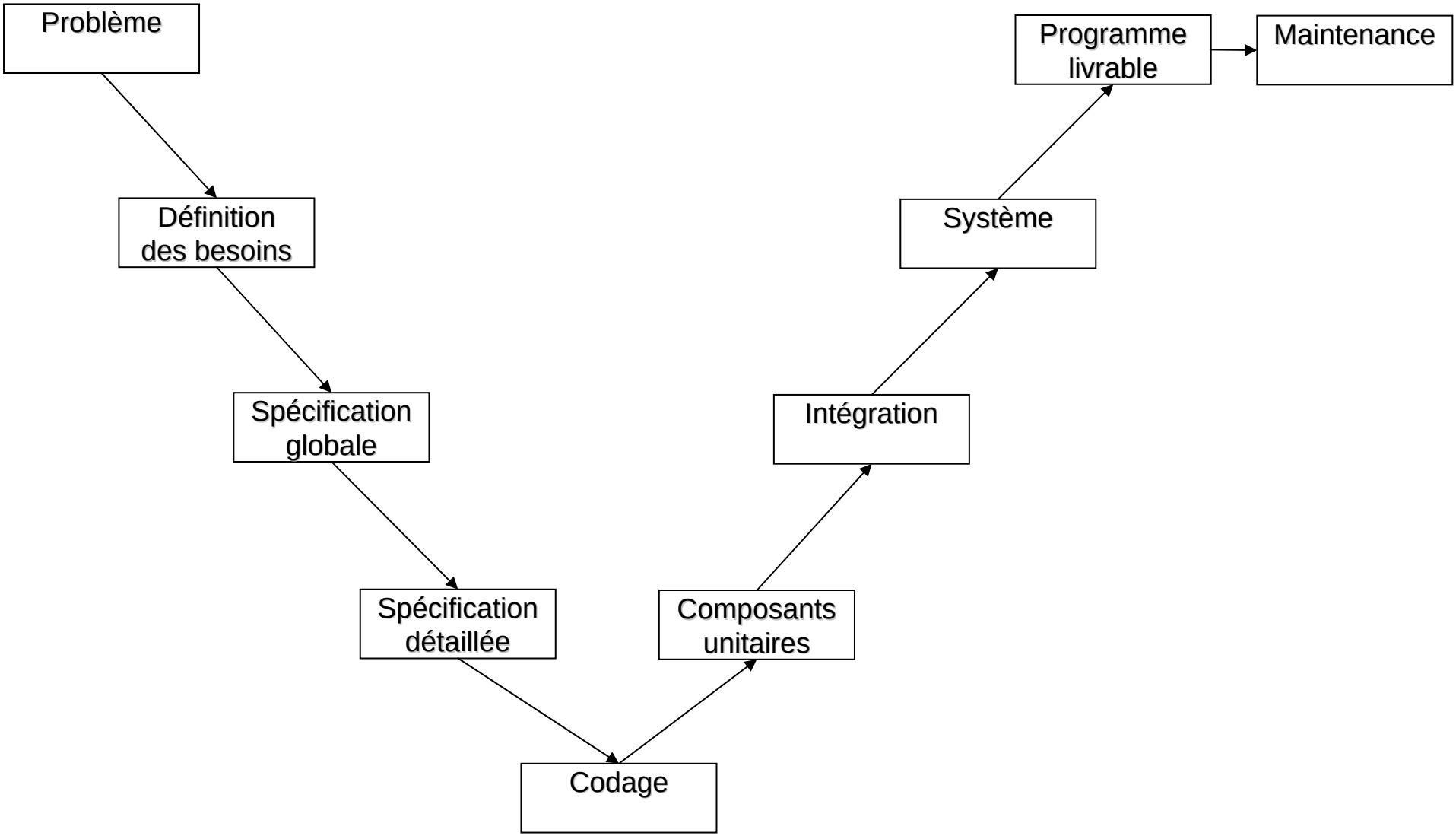
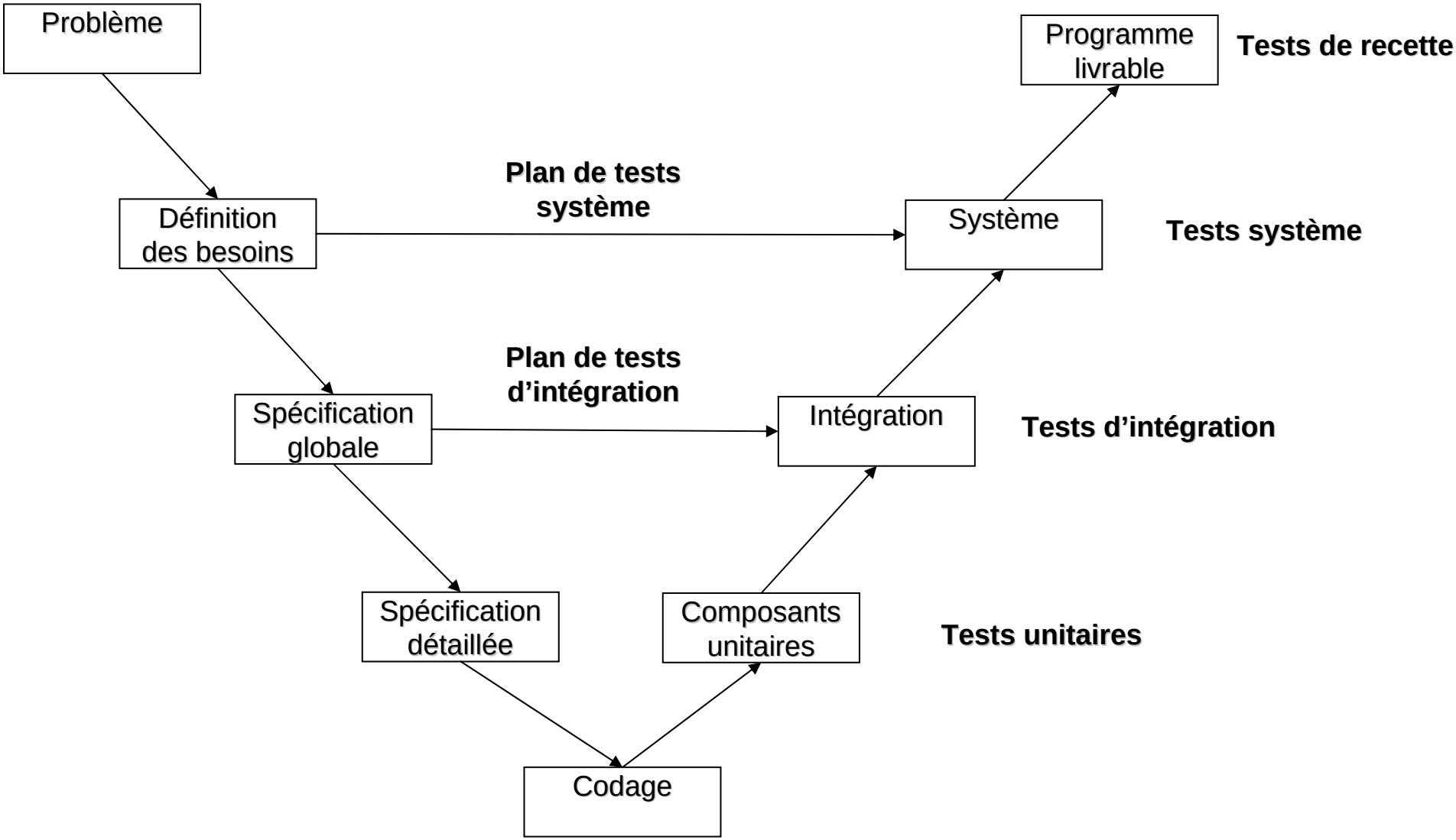
4: Le test et cycle de développement

Cycle de développement en V



Hiérarchisation des tests



Le test dans le cycle de développement

tester dès que possible

- tester dès l'analyse

processus de développement itératif

- tester à chaque incrément
- test de régression (ou non-régresssion): nouveau test du système après une modification pour vérifier qu'elle n'a pas apporté d'autres fautes;

eXtreme programming: on écrit les tests puis on code

- formellement déconseillé d'utiliser les tests comme spécification !

test de montée en charge

- test des performances;

test de recette

- pour obtenir l'approbation du client avant la livraison.

Test unitaire

Test d'une **unité logicielle** : test de base réalisé par le programmeur au fil du développement

- dépend fortement du paradigme de programmation utilisé :
  - procédural : la procédure
  - OO : la classe et ses méthodes

    On écrit pour chaque classe testée une ou plusieurs classes contenant une suite de test, avec pour chaque méthode un ou plusieurs cas de test.
- pour détecter des fautes dans son comportement individuel;
- que faire si le comportement dépend d'autres unités ? Différents points de vue :
  1. on ne teste pas (unitairement);
  2. on teste en utilisant les autres unités si elles sont disponibles, mais alors c'est plutôt du test d'intégration;
  3. on simule le comportement des autres unités par des bouchons [ou stubs].
- que faire si le comportement dépend d'éléments non contrôlables ? (ex réseau, base de données, etc) : on utilise des bouchons.
- attention aux inter-dépendances entre méthodes.
- en boîte blanche ou boîte noire.

Test unitaire

Les réticences :

- ça prend trop de temps d'écrire des tests !
- ça prend trop de temps d'exécuter des tests !
- ce n'est pas mon boulot de tester mon code !
- je suis payé pour écrire du code, pas pour le tester !

Écrire des programmes testables :

- La classe de test est extérieure à la classe testée;
- Il faut aussi tester les méthodes privées;
- Il faut pouvoir :
  - accéder à l'état d'un objet;
  - amener un objet dans un état propice au test.

Ne pas hésiter devant le refactoring

- Tester amène souvent à revoir son code en l'améliorant…

Test unitaire : Que tester ?

Vient avec l'expérience…mais quelques repères [Hunt Thomas] :

1. Au préalable, découper le comportement de la méthode en sous comportements (classes d'équivalences) qu'il faudra tester individuellement :
   - dans tel cas, la méthode doit lancer une exception;
   - dans tel autre cas, elle doit retourner ça;
   - dans tel autre cas encore, elle doit retourner autre chose;
   - etc.

2. Le résultat est-il juste ?
   - S'assurer qu'on a bien testé le retour d'une fonction/la levée des exceptions,
     [en général c'est le plus facile et c'est ce qu'on fait en premier]

3. Conditions aux limites : C'est souvent les conditions aux limites qui posent pb dans une application !
   - **Conformance** : Est-ce que la donnée est conforme à un format pré-défini ? (Ex. sur le traitement d'une adresse email : ? @ ? . ?
   - **Ordering** : Dans le cas où on travaille sur une collection ordonnée :
     - si on cherche une valeur : vérifier qu'on la trouve bien en tête/milieu/fin de collection
     - si une méthode prend une collection en entrée : le code présuppose-t-il un ordre particulier pour cette collection ?
     - si une donnée interne doit être maintenue triée, le vérifier ;

Test unitaire : Que tester ? (suite)

- **Range**
  - Cas où une variable peut prendre ses valeurs dans un intervalle donné, souvent plus grand que celui qui nous intéresse (un age codé sur un entier par exemple);
    - éviter le codage sur un type simple "trop grand", créer son propre type à la place, gardé par des assertions
    - utiliser intensivement des pré-cond et des invariants de classe ;
    - penser à tester les valeurs litigieuses : une valeur nominale, mais aussi la plus petite valeur, et la plus grande.
- **Reference**
  - Cas où votre méthode référence d'autres méthodes ou classes : dans quelles conditions peuvent-elles être utilisées ?
    - regarder scrupuleusement les documentations à la recherche de pré-post condition explicites ou non.
- **Cardinality**
  - Quand il faut compter... par exemple si on doit maintenir et publier un top-ten des meilleures ventes :
    - peut-on publier un top-ten vide ? à un élt ? à moins de 10 elts ? à 10 elts ?
    - et si la société ne vend que 5 articles ? 0 ?
    - et si brusquement on passe à un top-5 ?
- **Time**
  - problèmes de gestion du temps réel (quel calendrier, changements d'heures, etc)

Test unitaire : Que tester ? (suite)

3. Check Inverse RelationShips
   - Symétrie et fonction inverse : Si on calcule une racine carrée, vérifier que le résultat élevé au carré donne la valeur initiale.

4. Cross-checking with other means
   - Re-calculer un résultat en utilisant une autre version (version plus ancienne abandonnée car moins efficace mais déjà testée par ex).
   - Vérifier au moyen d'invariants les choses du style "si j'emprunte un livre j'en ai un de plus emprunté, un de moins libre, au total toujours le même nombre".

5. Force Error Conditions
   - Vérifier le comportement de la méthode dans les mauvais cas qui finissent toujours par se produire :
     - plus de mémoire, ou d'espace disque;
     - erreur réseau, plus de réseau;
     - base de données plantée;
     - etc.

6. Performance characteristics

Test d'intégration

Test d'un **ensemble d'unités** qui coopèrent;

- But : détecter des erreurs dans leur interopérabilité, la mauvaise utilisation d'une interface;
- interconnexion de composants (niveau macro);
- commence très tôt en objet (niveau micro): une classe est typiquement composée d'objets d'autres classes;
- bien repérer l'inter-dépendance des classes pour choisir un ordre d'intégration :
  - si les dépendances forment un arbre (un ordre partiel), alors on peut intégrer simplement de bas en haut;
  - s'il y a un cycle de causalité (A dépend de B qui dépend de A), fréquent :
    - on émule une des classes (par ex A);
    - on teste B avec l'émulation de A;
    - on teste A avec B;
    - on reteste B avec le vrai A.
- typiquement en boîte noire.

Test système

- L’application à tester est complètement intégrée dans son environnement :
  - inclut les autres applications utilisées,
  - l'environnement opérationnel (par exemple la JVM).
- on teste les scénarios intéressants déterminés lors de l'analyse (use cases, sequence diagrams);
- en boîte noire uniquement :
  - Le spécification est alors le seul critère de référence.

Des règles de bon sens

Concernant la forme des cas de test :

- inclure dans un cas de test des entrées pour le programme mais aussi le résultat attendu (sortie calculée, émission d'une exception, impression d'un message, etc);
- toujours déterminer le résultat attendu par rapport à la spécification du programme (pas au code);
- stocker les cas de tests pour pouvoir les exécuter à nouveau;
- soigner la traçabilité des tests.

Concernant le processus de test :

- Si possible faire tester par un autre développeur que celui du code sous test;
- examiner très attentivement les rapports de test, les stocker aussi;
- À chaque modification : relancer tous les cas de tests (non régression).

Concernant le choix des objectifs de test :

- vérifier que le programme se comporte bien dans les cas attendus comme dans les cas invalides;
- si exception levée : vérifier qu'elle l'est;