Ecole Hassania des Travaux Publics

Le Génie Logiciel: Cycle de vie d'un logiciel

Zineb Rachik
Ecole Hassania des travaux publics
Année universitaire : 2016-2017

Analyse des besoins

- C’est la première grande étape du cycle de vie du logiciel
- Il s’agit du processus visant à établir quelles fonctionnalités le système doit fournir et les contraintes auxquelles il sera soumis.

Conception

- Dans les grands systèmes, la phase de la conception se divise en trois phases:
  - ✓ Spécification précise des composants logiciels.
  - ✓ Conception globale (conception architecturale): mettre le point sur les relations entre les composants logiciels.
  - ✓ Conception détaillée de ces divers composants.
- Il n'y a pas d'ordre clair entre ces trois phases, le concepteur procédant par itération.

Spécification des logiciels

- De la spécification des besoins à la spécification fonctionnelle:
- La fonction principale de la phase d'analyse des besoins est de préciser les services qui seront rendus par le logiciel à l'utilisateur. Les spécifications des besoins sont donc orientées utilisateur.
- En revanche, les spécifications fonctionnelles sont destinés au concepteur du logiciel. Le document des spécifications fonctionnelles doit alors contenir les définitions abstraites des composants et non pas la définition des services destinés à l'utilisateur.
- Une séparation claire entre la spécification fonctionnelle et la conception est difficile à établir; la spécification est souvent vue comme une partie du processus de la conception. Elle est sa représentation abstraite.

Conception architecturale

- Identification des sous systèmes constituant le système.
- Chaque sous système doit être décomposé en un certain nombre de composants; la spécification de ces sous système revient alors à définir les opérations de ces composants
- Définition les interactions entre les composants

➢ Résultat: une description de l'architecture du logiciel.

Conception détaillée

- Cette conception s'exprime en fonction d'abstractions de plus bas niveau, que l'on peut facilement traduire en code exécutable.
- Les algorithmes utilisés dans chaque composant doivent être détaillés.
- des tests unitaires sont définis pour s'assurer que les composants réalisés sont conformes à leurs descriptions.
- Résultat : pour chaque composant, le résultat consiste en :
  - ✓ Un dossier de conception détaillée.
  - ✓ Un dossier de tests unitaires.
  - ✓ Un dossier de définition d'intégration logiciel.

Qu’est ce qu’une bonne conception?

- Il n’existe pas de critère définitif permettant de définir une bonne conception.
- Le critère le plus décisif peut être l’efficacité du code produit et la maintenabilité du produit: une bonne conception facilite la maintenance et le cout des changement est minimal.
- Ces objectifs peuvent être atteints lorsque la conception apporte à la fois un haut degré de cohésion et un couplage faible.

Cohésion ?

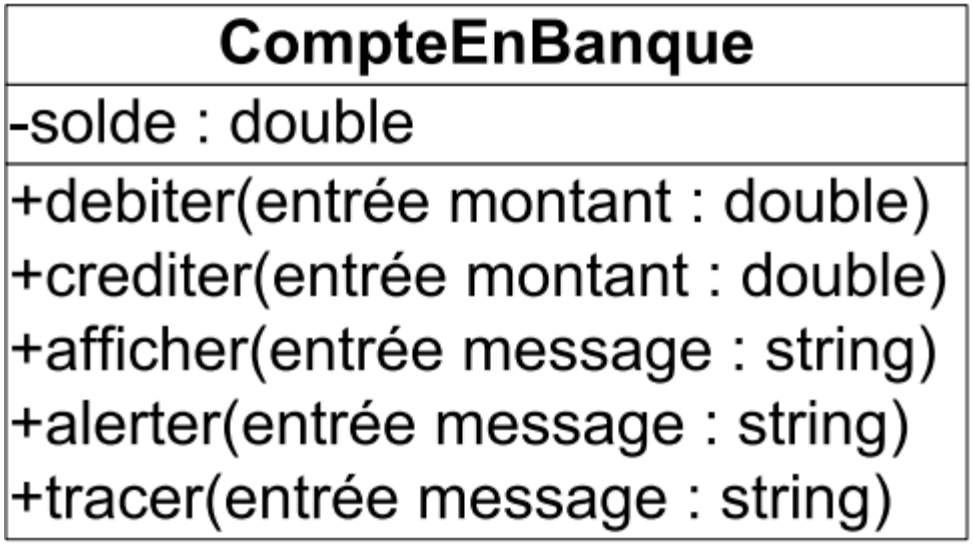
- On dit qu'une unité de programme fait preuve d'un haut degrés de cohésion si ses sous éléments remplissent des fonctions très proches; cela signifie que chaque sous élément est crucial pour que cette unité remplisse son rôle.
- En terme de classes, l'idée est de vérifier que nous rassemblons bien dans une classe des méthodes cohérentes, qui visent à réaliser des objectifs similaires.
- Des éléments qui sont regroupés dans une même unité car par exemple ils s'exécutent au même temps, ont un faible degré de cohésion.

Cohésion: mauvais exemple 1

- Imaginons une classe UserManager qui permet à l'utilisateur de récupérer ses informations (nom, prénom…) d'une base de donnée à partir d'un nom d'utilisateur et d'un mot de passe. Une telle classe est n'est pas cohésive car elle effectue des oppérations qui n'ont pas un lien direct entre elles: lecture des donnés, validation, récupération de la base…
- On peut rendre cette classe fortement cohésive en la découpant en plusieurs classes, une classe pour la vérification , une classe pour récupérer les données de la base….

Cohésion: mauvais exemple 2

- Mauvais exemple : il serait mal venu d'implémenter une méthode "Afficher()" ou "Tracer()" puisque l'objectif de cette classe est de représenter le modèle d'un compte en banque et non celui d'une fenêtre graphique ou d'un service de journalisation.



Couplage?

- Le couplage est une indication de la force des connexions entre unités.
- Une entité (fonction, module, classe, package, composant) est couplée à une autre si elle dépend d'elle.
- Les système à couplage fort ont des connexions fortes entre unités qui dépendent les unes des autres.
- Le couplage faible désigne une relation faible entre plusieurs entités (classes, composants), permettant une grande souplesse de programmation, de mise à jour. Ainsi, chaque entité peut être modifiée en limitant l'impact du changement au reste de l'application.

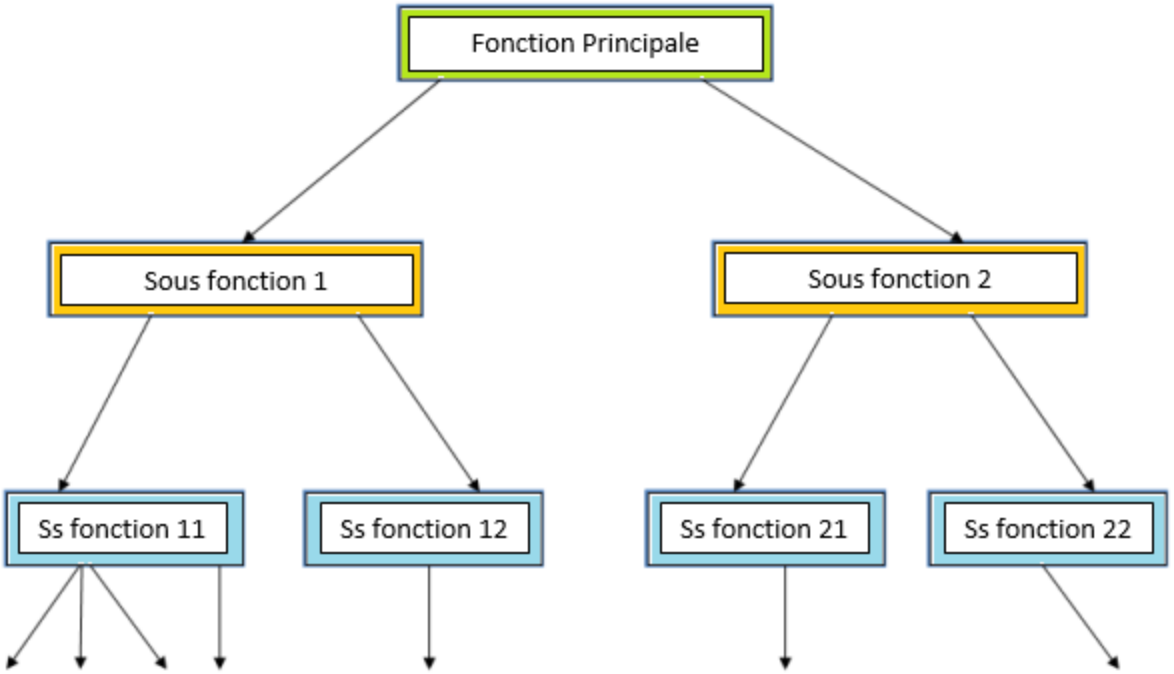
Systèmes à Forte cohésion et au couplage faible

- L’avantage de tels systèmes est qu’il est possible de remplacer une unité quelconque par une unité équivalente avec peu de changement dans les autres unités du système.

Conception; approche fonctionnelle

- Cette approche dérive des langages de programmation procéduraux. Au départ nous avons un grand problème que nous décomposons en plusieurs sous-problèmes. On effectue ensuite la décomposition des sous-problèmes en d'autres sous-problèmes. On itère le processus jusqu'à obtention de problèmes simples à comprendre donc à programmer. C'est donc un ensemble de fonctions précises en interaction les unes avec les autres.
- Dans cette approche, c'est la fonction principale (la solution au problème) qui décrit l'architecture du système. C'est une stratégie extrêmement ordonnée, logique et elle permet une réduction considérable de la complexité. Mais ce qu'on lui reproche c'est qu'elle ne permet pas une réutilisation des composants déjà définis et aussi il n'est pas évident d'assurer l'évolution du logiciel.

Conception; approche fonctionnelle



Conception; approche objet

- Contrairement à l'approche fonctionnelle, l'approche orienté objet tient sa forme de la structure du système. Dans cette approche, le logiciel est considéré comme un ensemble d'objets possédant des caractéristiques spécifiques et qui interagissent entre eux. Une caractéristique d'un objet peut être un attribut ou une méthode.
- Pour pouvoir l'utiliser, il est impératif de connaître les concepts ***d'encapsulation, d'agrégation, d'héritage, de polymorphisme, de généralisation, de spécialisation …***
- L'approche orientée objet est très rigoureuse et surtout moins intuitive que l'approche fonctionnelle.

Codage et tests unitaires

- Il s’agit de l’écriture du code source du logiciel, et tester son comportement afin de vérifier s’il réalise les responsabilités qui lui sont allouées

Intégration et validation

- Assembler le code source du logiciel (éventuellement partiellement) et dérouler les tests d'intégration: les différents sous-ensembles sont mis ensemble pour former le logiciel complet.
- On vérifie ainsi les interactions entre les sous-ensemble.
- Construire le logiciel complet exécutable. Dérouler les tests de validation sur le logiciel complet exécutable.

Maintenance

- Maintenance corrective :
  - Corriger les erreurs : défauts d'utilité, d'utilisabilité, de fiabilité...
    - Identifier la défaillance, le fonctionnement
    - Localiser la partie du code responsable

    Maintenance adaptative
- Maintenance adaptative :
  - Ajuster le logiciel pour qu'il continue à remplir son rôle compte tenu de l'évolution des Environnements d'exécution
    - Fonctions à satisfaire
    - Conditions d'utilisation
- Maintenance perfective: donne lieu à de nouvelles versions

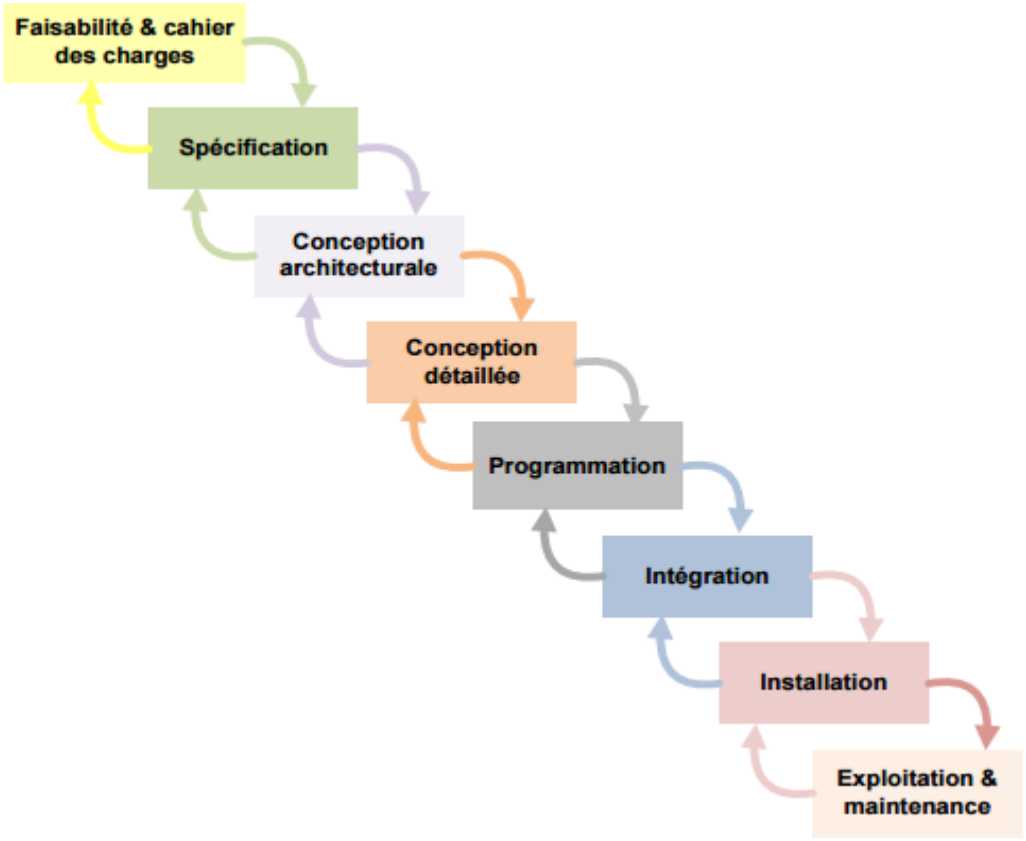
Modèles de cycles de vie

- Afin d'être en mesure d'avoir une méthodologie commune entre le client et la société de service réalisant le développement, des modèles de cycle de vie ont été mis au point définissant les étapes du développement ainsi que les documents à produire permettant de valider chacune des étapes avant de passer à la suivante. A la fin de chaque phase, des revues sont organisées avant de passer à la suivante.

Modèles de cycles de vie: modèle en cascade

- Le modèle en cascade (Waterfall model) est le plus classique des cycles de vie.
- Cycle de vie linéaire sans aucune évaluation entre le début du projet et la validation
- Le projet est découpé en phases successives dans le temps
- A chaque phase correspond une activité principale bien précise produisant un certain nombre de livrables
- Chaque phase ne peut remettre en cause que la phase précédente

Modèles de cycles de vie: modèle en cascade

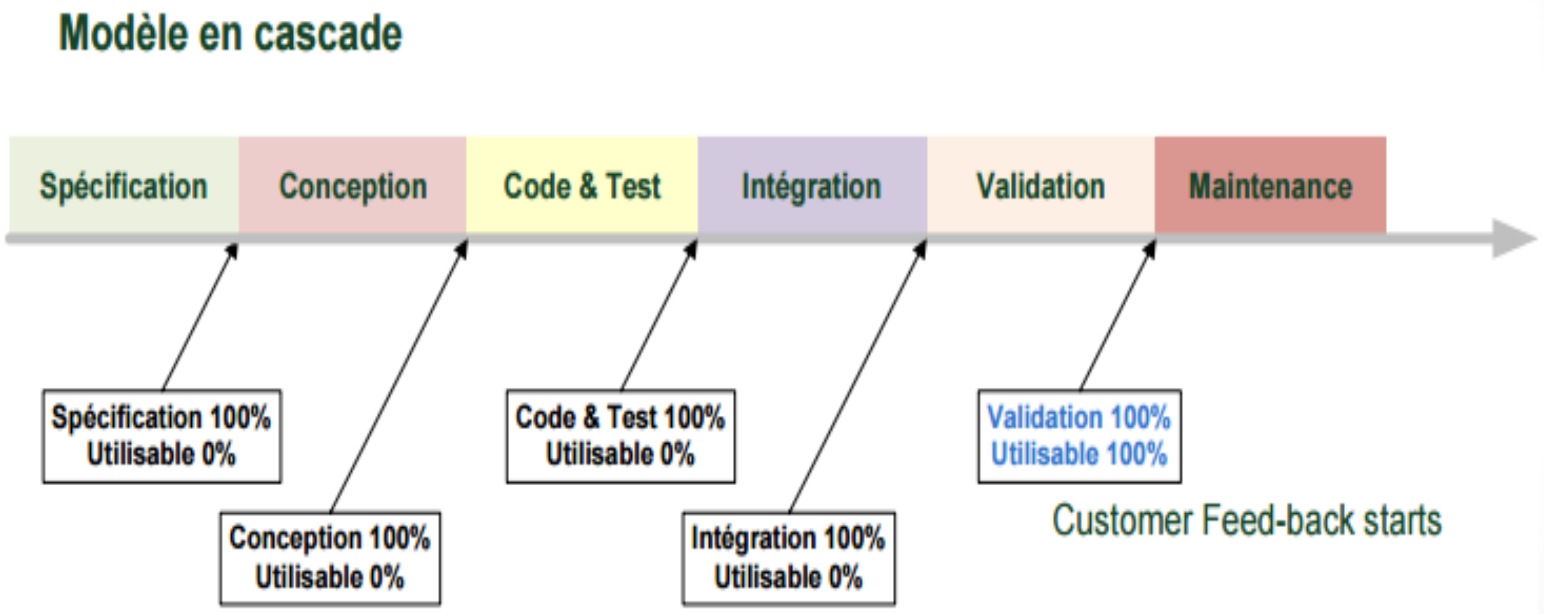


Modèle en cascade: inconvénients

- Peu réaliste:
  - Les vrais projets suivent rarement un développement séquentiel
  - Établir tous les besoins au début d'un projet est difficile
- Aucune validation intermédiaire (Aucune préparation des phases de vérification):
  - Obligation de définir la totalité des besoins au départ
  - augmentation des risques car validation tardive : remise en question coûteuse des phases précédentes
- Sensibilité à l'arrivée de nouvelles exigences : refaire toutes les étapes
- Très faible tolérance à l'erreur (les anomalies sont détectées tardivement) qui induit automatiquement un coût important en cas d'anomalie.

❖ Bien adapté lorsque les besoins sont clairement identifiés et stables

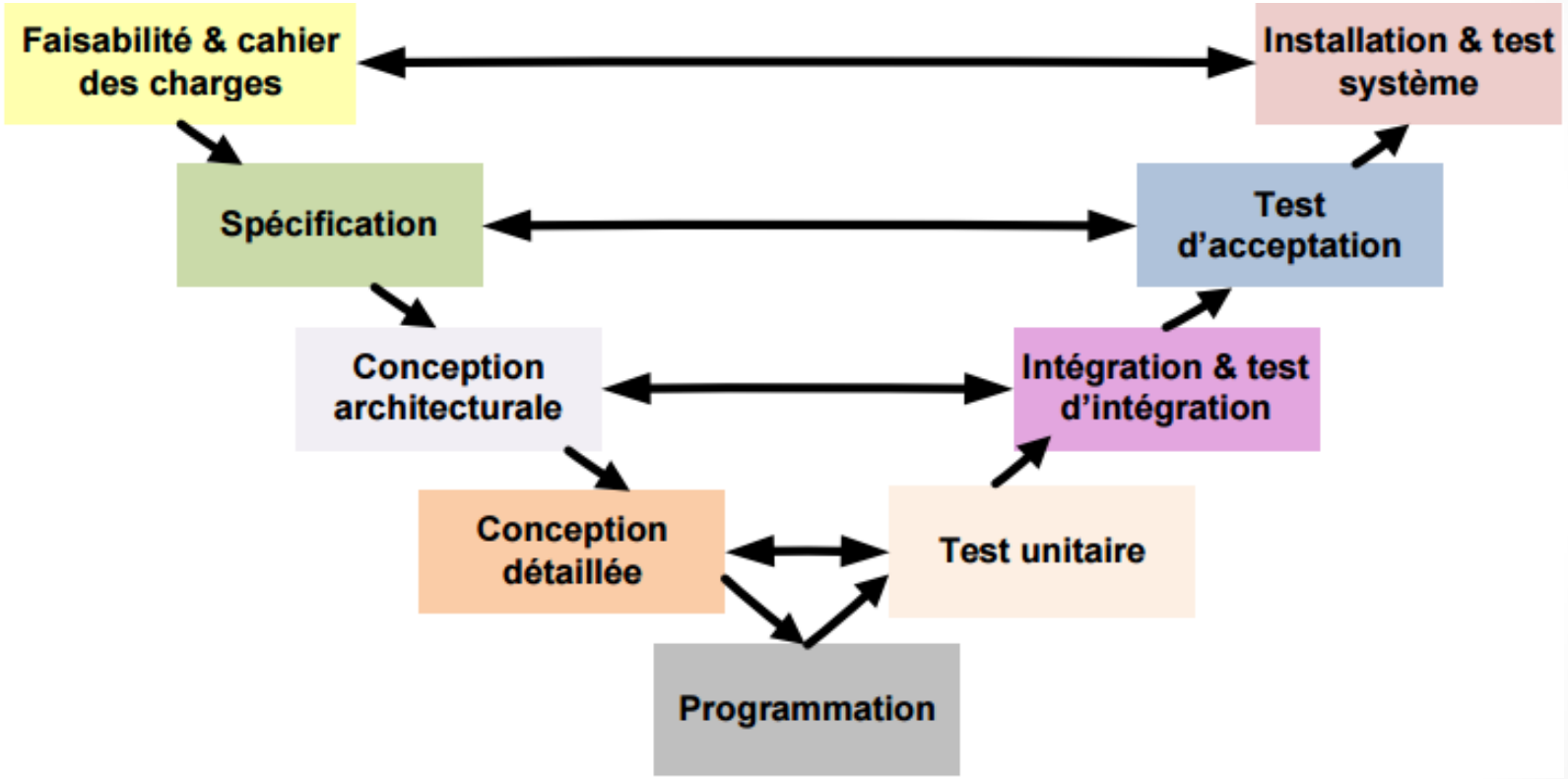
Modèle en cascade: inconvénients



Modèles de cycles de vie: Cycle de vie en V

- Face aux problèmes de réactivité que pose l'approche en cascade, l'industrie informatique a adopté le cycle en v.
- Le modèle de cycle de vie en V part du principe que les procédures de vérification de la conformité du logiciel aux spécifications doivent être élaborées dès les phases de conception.
- A **chaque étape de conception correspond une phase de test** ou de validation

Modèles de cycles de vie: Cycle de vie en V



Modèles de cycles de vie: Cycle de vie en V

- Les phases du cycle en V sont:
- Etude et analyse : l'analyse ou la définition des besoins, la rédaction des spécifications, la conception architecturale, la conception détaillées
- Codage : développement de l'application
- Tests et validations : tests unitaires, test d'intégration, tests de validation et maintenance corrective.
- Remarque: plus on avance dans l'étude plus le niveau de détail est précis, ensuite on code, et plus on avance dans les tests moins le niveau de détails est précis.

Cycle de vie en V: avantages

- La stricte structure en V permet d'espérer que le livrable final sera parfait, puisque les étapes de test sont aussi nombreuses que les étapes de réflexion.
- Il est facile de prévoir les tests à réaliser au moment où l'on conçoit une fonctionnalité ou une interface, le travail s'enchaîne donc de façon assez naturelle.

Cycle de vie en V: inconvénients

- Il faut attendre longtemps pour savoir si on a construit le bon logiciel.
- Difficile d'impliquer les utilisateurs lorsqu'un logiciel utilisable n'est disponible qu'à la dernière phase
- Idéal quand les besoins sont bien connus, quand l'analyse et la conception sont claires

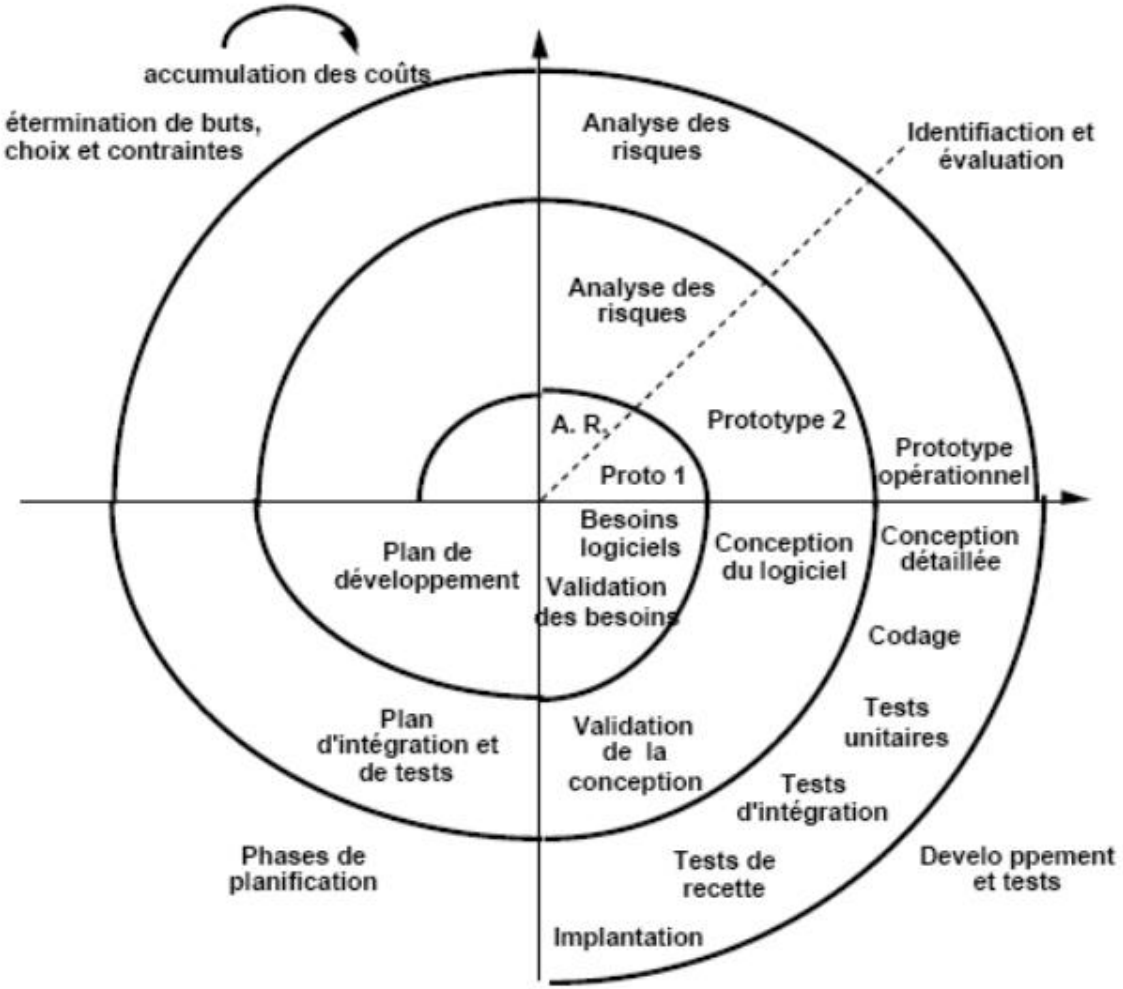
Modèles de cycles de vie: Cycle en spirale

- **le cycle en spirale reprend les étapes du cycle en V**, mais prévoit l'implémentation de **versions successives**, ce qui permet de mettre l'accent sur la **gestion des risques**, la première phase de chaque itération étant dédiée à ce poste. A ce point il est nécessaire de définir la notion de **prototype**.

Prototype?

- Il s’agit des versions incomplètes du produit.
- Il peut s’agir d’une simple maquette sans aucune fonctionnalité (on parle alors de prototype horizontal)
- ou bien de sites partiellement fonctionnels : telle version implémentera la navigation de base, la suivante ajoutera l’espace membres, puis la zone de téléchargements… on parlera alors de prototype vertical.
- Avantage :
  - Les efforts consacrés au développement d’un prototype sont le plus souvent compensés par ceux gagnés à ne pas développer de fonctions inutiles

Cycle en spirale



Cycle en spirale

- Chaque cycle de la spirale se déroule en quatre phases :
- détermination, à partir des résultats des cycles précédents, ou de l'analyse préliminaire des besoins, des objectifs du cycle, des alternatives pour les atteindre et des contraintes ;
- analyse des risques, évaluation des alternatives et, éventuellement maquettage ;
- développement et vérification de la solution retenue, un modèle « classique » (cascade ou en V) peut être utilisé ici ;
- revue des résultats et vérification du cycle suivant.

Cycle en spirale: avantages

- Le but premier de ce modèle étant la gestion des risques, ceux-ci sont logiquement limités.
- L'expertise du client croit à chaque itération du cycle, l'apprentissage se fait par touche et pas d'un seul bloc.
- Enfin, ce modèle est très adaptatif : si chaque prototype apporte des fonctionnalités indépendantes, il est possible de changer l'ordre de livraison des versions.

Cycle en spirale: Inconvénients

- Il n’est adapté qu’aux projets suffisamment gros, inutile de prévoir la livraison de 5 ou 6 prototypes pour un petit projet.
- L’évaluation des risques en elle-même et la stricte application du cycle de développement peut engendrer plus de coûts que la réalisation du logiciel.
- Enfin, ce type de cycle de développement est complexe, entre les étapes prévues en théorie et celles mises en pratique il y a une grande différence.

Processus de développement unifiés

- Un processus unifié est un processus construit sur UML (Unified Modeling Language).
- Il se distingue par les caractéristiques suivantes (ROQUES, 2008) :

➢ Itératif: permet le développement d'une solution effective de façon incrémentale.

➢ Piloté par les risques

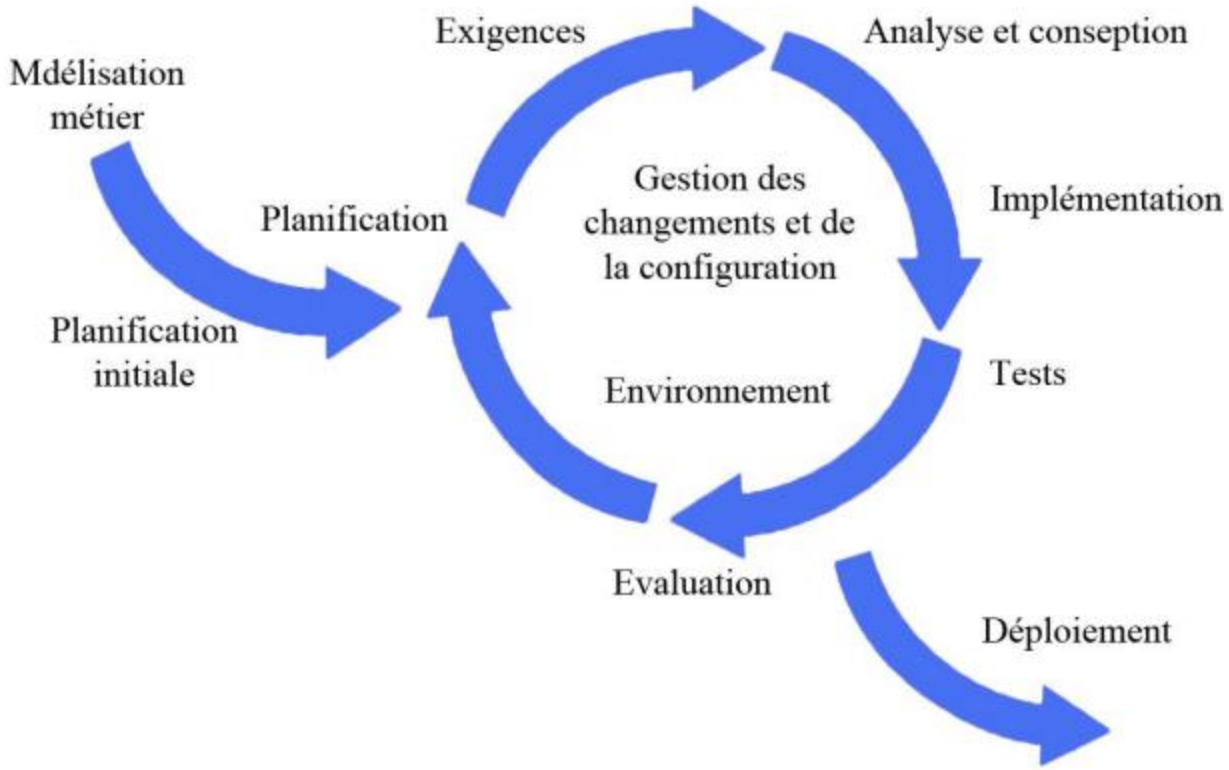
➢ Centré sur l'architecture: choix de l'architecture logicielle est fait lors des premières phases.

➢ Conduit par les cas d'utilisation: orienté par les besoins utilisateurs.

RUP

- Rational Unified Process
- Dérivée de UP (UP a été créée en 1996)
- Commercialisée par IBM en 1998
- Méthode de développement logiciel

RUP: Cycle de vie



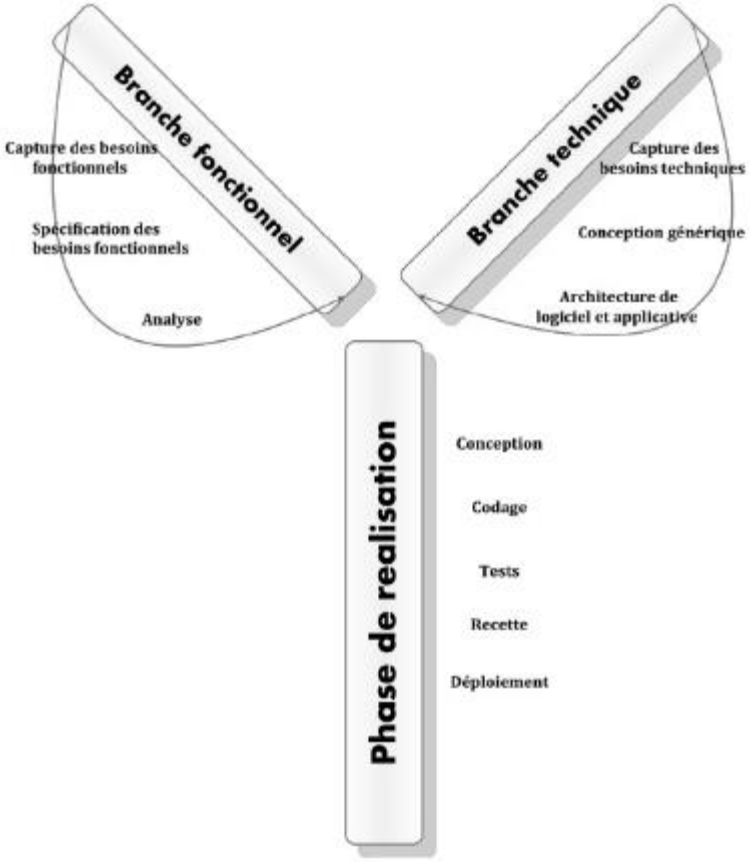
Forces et faiblesses de la méthode RUP

- Forces:
- Traçabilité à partir des Uses Cases jusqu'au déploiement Approche basée sur l'architecture Gestion des risques dans les projets Cadre propice à la réutilisation
- Faiblesses:
- Coût de personnalisation souvent élevé Très axé processus Vision non évidente ni immédiate

2TUP

- C’est un processus unifié.
- Propose un cycle de développement qui sépare les aspects techniques des aspects fonctionnels et propose une étude parallèle des deux branches : fonctionnelle (étude de l’application) et la technique (étude de l’implémentation).
- 2TUP s’articule autour de trois branches :
  - Une branche technique
  - Une branche fonctionnelle
  - Une branche de conception et réalisation

2TUP

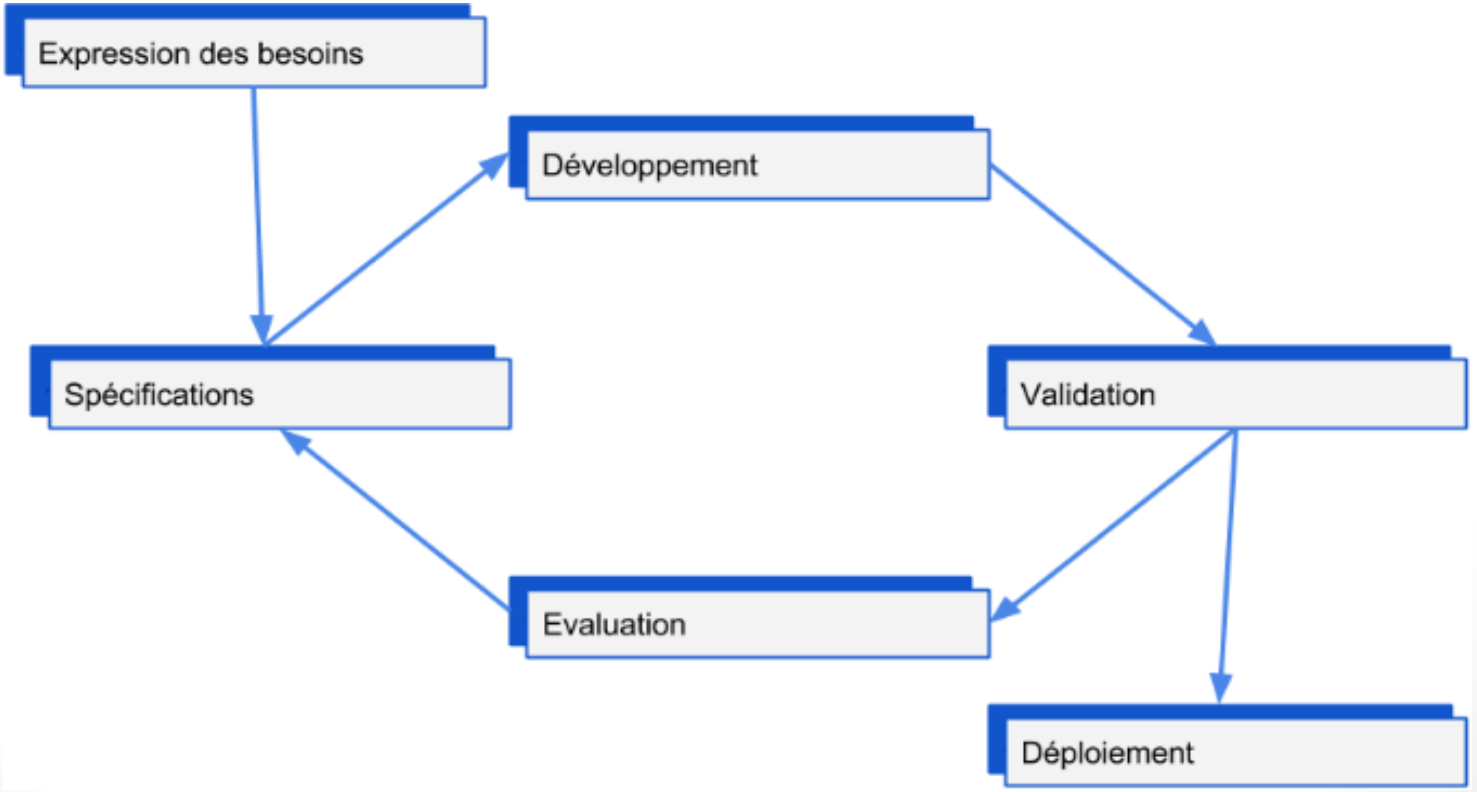

Branche fonctionnel
Capture des besoins fonctionnels
Spécification des besoins fonctionnels
Analyse
Branche technique
Capture des besoins techniques
Conception générique
Architecture de logiciel et applicative
Phase de realisation
Conception
Codage
Tests
Recette
Déploiement

2TUP

- Branche fonctionnelle:
  - capture des besoins fonctionnels
  - définir La frontière fonctionnelle entre le système et son environnement.
  - Les activités attendues des différents utilisateurs par rapport au système.
- Branche technique
  - capture des besoins techniques
  - conception générique
- Phase conception - réalisation
  - produire le modèle de conception système. Ce dernier organise le système en composants, délivrant les services techniques et fonctionnels, Ce qui induit le regroupement des informations des branches technique et fonctionnelle.
  - conception détaillée
  - L'étape
  - de codage
  - valider les fonctionnalités du système développé.

Cycle itératif

- Simplifions un peu le modèle précédent en réduisant le nombre d'étapes du cycle et séparons les activités des artéfacts (c'est à dire les produits issus de ces activités). Nous arrivons logiquement au modèle itératif



Cycle itératif: Principes

- Développez de petits incréments fonctionnels livrés en courtes itérations
- Un incrément fonctionnel est un besoin du client.
- Pour chaque version à développer après la 1ère version livrée, il faut arbitrer entre :
  - Les demandes de correction,
  - les demandes de modification
  - les nouvelles fonctionnalités à développer

Cycle itératif: avantages et inconvénients

- **Avantages du modèle itératif :** Ce type de cycle de développement est plus souple que les cycles classiques: chaque itération permet de s'adapter à ce qui a été appris dans les itérations précédentes et le projet fini peut varier du besoin qui a été exprimé à l'origine. Comme dans le cycle en spirale, la mise à disposition de livrables à chaque cycle permet un apprentissage de l'utilisateur final en douceur.
- **Inconvénients du modèle itératif :** la confiance qui amène bien souvent à négliger les test d'intégration. Ainsi les développeurs livrent une nouvelle fonctionnalité sans se rendre compte qu'ils ont cassé une chose qui fonctionnait dans les cycles précédents. Il faut donc que le chef de projet soit particulièrement vigilant lors de la phase de tests.

AGL: ateliers de génie logiciel

- Un AGL est un logiciel aidant à la réalisation de logiciels.
- Il intègre des outils adaptés aux différentes phases de la production d'un logiciel et facilite la communication et la coordination entre ces différentes phases.
- Un AGL est basé sur des méthodologies qui formalisent le processus logiciel, et à l'intérieur de ce processus, chacune des phases qui le composent.

Les outils ``CASE''

- Les AGL intègrent différents outils d'aide au développement de logiciels, appelés outils CASE: éditeurs de texte, de diagrammes ,outils de gestion de configuration, SGBD, compilateurs, debuggers, outils pour la mise en forme, la génération de tests…
- Ces différents outils interviennent lors d'une ou plusieurs phases du cycle de vie du logiciel:
  - conception (éditeurs de texte, de diagrammes, ...),
  - programmation (éditeurs de texte, compilateurs…)
  - Certains outils, concernant notamment la gestion de configurations, la gestion de projet, interviennent durant la totalité du processus logiciel

L'intégration d'outils CASE: Intégration des données

- Un AGL intègre différents outils CASE, de manière à les faire coopérer de façon uniforme. Cette intégration s'effectue à trois niveaux:
- **Intégration des données:**
- Les outils CASE manipulent des données: Différents outils sont amenés à partager une même donnée: les tables générées par un éditeur de diagrammes sont utilisées par un SGBD…

L'intégration d'outils CASE: Intégration des données

- Un AGL doit alors prendre en charge la communication de ces données entre les différents outils. Cette intégration peut être simplement physique:
  - tous les outils de l'AGL utilisent un seul format de représentation des données. Ceci implique que tous les outils de l'AGL connaissent la structure logique des fichiers qu'ils sont amenés à utiliser: il est nécessaire de normaliser la structure logique des fichiers.
  - L'intégration des données peut se faire également au niveau logique en utilisant un système de gestion des objets qui gère automatiquement les différentes entités et leurs interrelations
- Un AGL devrait également gérer la cohérence entre les différentes versions de ces données (gestion de configuration).

L'intégration d'outils CASE: interface utilisateur et activités

- **Intégration de l'interface utilisateur:** tous les outils intégrés dans l'AGL communiquent avec l'utilisateur selon un schéma uniforme, ce qui facilite leur utilisation
- **Intégration des activités:** un AGL peut gérer le séquencement des appels aux différents outils intégrés, et assurer ainsi un enchainement cohérent des différentes phases du processus logiciel. Cet aspect implique que l'on dispose d'un modèle du processus de développement bien accepté (ce qui relève un peu de l'utopie !!!).

Les différents types d'AGL

- On distingue essentiellement deux types d'AGL selon la nature des outils intégrés:
- *Les environnements de conception (upper-case):*

➢ ces ateliers s'intéressent plus particulièrement aux phases d'analyse et de conception du processus logiciel.

➢ Ils intègrent généralement des outils pour l'édition de diagrammes (avec vérification syntaxique), des dictionnaires de données, des outils pour l'édition de rapports, des générateurs de (squelettes de) code, des outils pour le prototypage, ...

➢ Ces ateliers sont généralement basés sur une méthode d'analyse et de conception (Merise, ...) et utilisés pour l'analyse et la conception des systèmes d'information.

Les différents types d'AGL

- *Les environnements de développement (lower-case):*
- ces ateliers s'intéressent plus particulièrement aux phases d'implémentation et de test du processus logiciel. Ils intègrent généralement des éditeurs (éventuellement dirigés par la syntaxe), des générateurs d'interfaces homme/machine, des SGBD, des compilateurs, optimiseurs, debuggers, ...
- WinDev est un environnement de développement.

Les différents types d'AGL

- Certains environnement, plus évolués, sont dédiés à un langage particulier. Il existe par exemple des environnements dédiés à InterLisp, Smalltalk, Loops... Ces différents environnements proposent des bibliothèques de composants, une interface graphique, des éditeurs dédiés au langage, des interprètes, debuggers, ... Ces environnements permettent un développement rapide et convivial.
- En revanche, l'application développée est intégrée dans (et généralement inséparable de) l'environnement, ce qui peut poser des problèmes de portabilité et de coût.

Les différents types d'AGL

- Enfin, il existe des générateurs d'environnements de programmation: Mentor, Gandalf, Synthesizer Generator, ...
- A partir de la description formelle d'un langage (à l'aide de grammaires attribuées, de la logique), ces différents systèmes génèrent un environnement de programmation dédié au langage, contenant un éditeur dédié au langage, un debugger, un interpréteur, ...

Quelques AGL

- „**Windev** de PCSoft.
- **PowerDesigner** de Sybase.
- „**PowerBuilder** de Sybase (PowerSoft).
- **Objecteering** de SoftTeam
- „**Win'Designde** CECIMA. „
- **Oracle Developer** de Oracle Corporation.
- „**Rational Suite Development Studio** de Rational Software.
- **Eclipse** (http://www.eclipse.org)

Qualité d’un logiciel

- Un logiciel de qualité est un logiciel capable de répondre aux besoins du client sans défaut d’exécution.
- La qualité logicielle est définie par un ensemble de règles et de procédures à suivre au cours du développement d’un logiciel.
- La qualité d’un logiciel se reflète non seulement dans les processus de développement mais aussi dans la qualité des éléments qui le constituent.

Aptitude d'un produit ou d'un service à satisfaire les besoins des utilisateurs. (AFNOR - juillet 1982)

Qualité d'un logiciel

- Il est donc nécessaire

  - De définir ce que l'application doit faire et comment elle doit le faire, tant d'un point de vue fonctionnel que d'un point de vue technique
  - De décider quelles techniques utiliser pour évaluer chacune des caractéristiques

Mesurer quoi?

- Les processus du développement du logiciel
- Les produits livrables ou documents qui résultent d’un processus
- Les ressources exigées par un processus

Mesurer quoi?

- Chaque entité des trois classes processus , produits et ressources possède :

- Des attributs internes : attributs mesurables sur l'entité indépendamment de son environnement
- Des attributs externes : attributs mesurables par rapport aux liens avec son environnement

Mesurer quoi?

- Les attributs internes de produits sont souvent utilisés pour prédire les attributs externes
- Ces prédictions permettent de contrôler le développement

Il est très difficile de définir objectivement des mesures qui dépendent de beaucoup d'autres mesures

Méthodes de mesure

- ISO/IEC 9126 propose les grandes lignes pour un processus d'évaluation de la qualité - ISO/IEC 14598 propose un cadre plus précis pour l'évaluation du produit logiciel
- Le projet SCOPE définit un cadre complet pour l'évaluation
- La méthode GQM permet de choisir les indicateurs adéquats et QIP de les améliorer

Processus d'évaluation (9126)

- Processus d'évaluation en trois étapes

➢ La définition des exigences de qualité

- L'objectif de cette première étape est de spécifier les exigences en termes de caractéristiques de qualité. Ces exigences peuvent varier d'un composant du produit à un autre

➢ La préparation de l'évaluation

- L'objectif majeur est d'initier l'évaluation. Ceci est fait en trois sous- étapes

Processus d'évaluation (9126)

- La sélection des indicateurs de qualité. Ces dernières doivent correspondre aux caractéristiques énumérées
- La définition des taux de satisfaction. Les échelles de valeurs doivent être divisées en niveaux de satisfaction des exigences
- La définition des critères d'appréciation. Ceci inclut la préparation de la procédure de compilation des résultats par caractéristique. Il est possible aussi de prendre en compte dans cette procédure des aspects de gestion, tels que le temps ou les coûts.

Processus d'évaluation (9126)

- La procédure d'évaluation
- Mesurer: Les indicateurs sélectionnés sont appliqués au produit, donnant ainsi des valeurs
- Notation: Pour chaque valeur mesurée, une note (de satisfaction) est attribuée
- Appréciation: En utilisant les critères d'appréciation, un résultat global de l'évaluation du produit est obtenu. Ce résultat est confronté aux aspects de gestion (temps et coûts) pour la prise de décision

Directives complémentaires (14598)

- L'objectif de cette norme est de fournir
  - Les directives d'identification, d'implantation et d'analyse des indicateurs nécessaires au processus d'évaluation du produit final
  - Les directives de définition des indicateurs qui permettent des évaluations partielles pendant le cycle de développement

Directives complémentaires (14598)

- Cette norme donne entre autres
- Des informations générales sur des indicateurs de qualité des logiciels
- Des critères de sélection de ces indicateurs
- Des directions pour l'évaluation des résultats des mesures (données)
- Des directions pour l'amélioration du processus de mesure
- Des exemples de types de graphes d'indicateurs
- Des exemples d'indicateurs qui peuvent être utilisés pour les caractéristiques de qualité de ISO/IEC 9126

Un cadre d’évaluation, SCOPE

- SCOPE est un projet européen ESPRIT (1989-93) (Software CertificatiOn Programme in Europe)
- Objectifs :
- Définir des procédures d'attribution d'un label de qualité à un logiciel quand celui-ci satisfait un certain ensemble d'attributs de qualité
- Développer des technologies nouvelles et efficaces d'évaluation, à des coûts raisonnables, permettant l'attribution de ce label
- Promouvoir l'utilisation des technologies modernes de l'ingénierie des logiciels dans l'attribution du label

Processus SCOPE

- Documents produits
  - Les critères d'évaluation
  - La spécification de l'évaluation
  - Le plan de l'évaluation
  - Le rapport d'évaluation

Comment choisir la bonne mesure?

- Le choix de la mesure dépend de l'objectif des mesures
- L'une des approches les plus utilisées pour le choix des mesures est GQM (Goal – Question – Metrics)

GQM : introduction

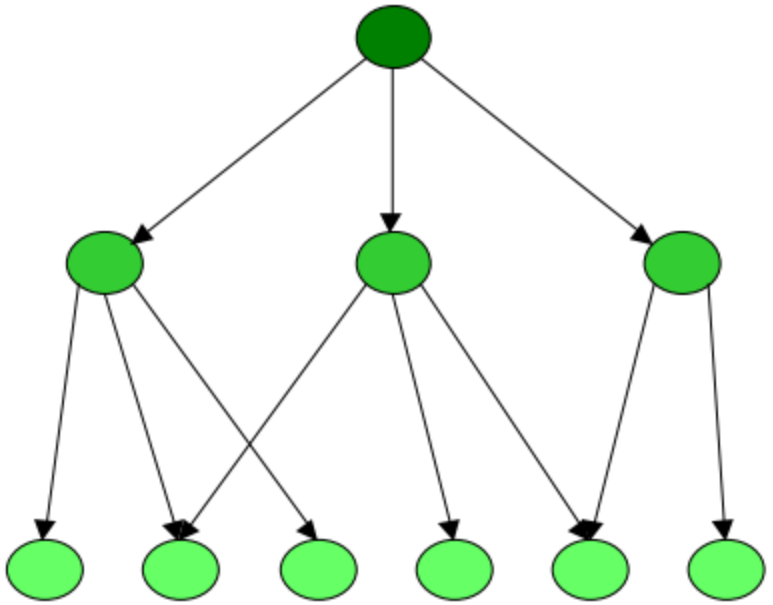
- Approche introduite par Basili et al.
- GQM propose un cadre en trois étapes
- – Énumérez les objectifs principaux du projet de développement ou de maintenance
- – Dérivez de chaque objectif, les questions dont les réponses permettent de déterminer si le but est atteint
- – Décidez de ce qui doit être mesuré afin de pouvoir répondre aux questions

GQM: introduction

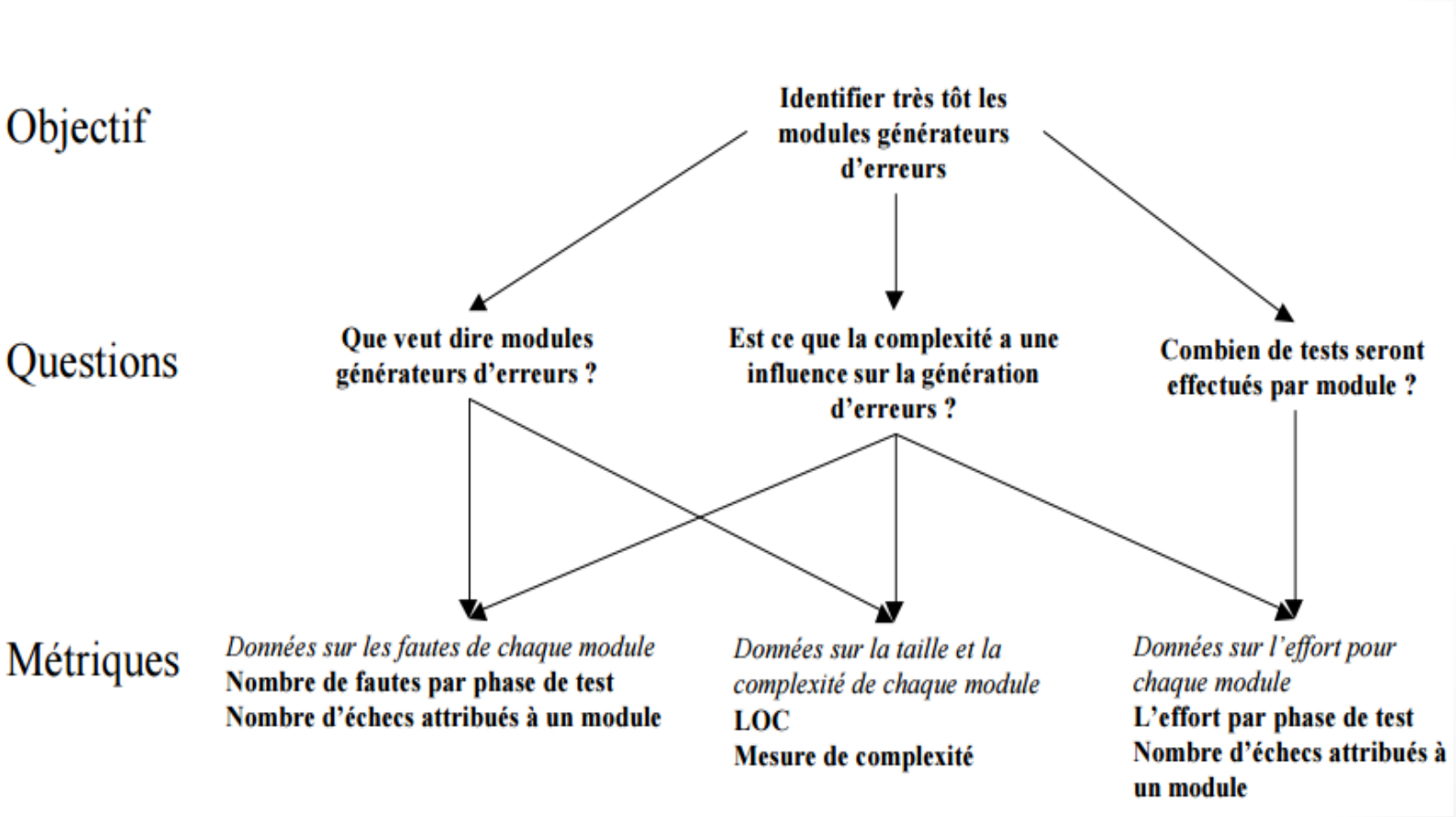
Objectif (*goal*)

Questions

Indicateurs (*metrics*)



GQM: exemple


Objectif
Identifier très tôt les modules générateurs d'erreurs
Questions
Que veut dire modules générateurs d'erreurs ?
Est ce que la complexité a une influence sur la génération d'erreurs ?
Combien de tests seront effectués par module ?
Métriques
Données sur les fautes de chaque module
Nombre de fautes par phase de test
Nombre d'échecs attribués à un module
Données sur la taille et la complexité de chaque module
LOC
Mesure de complexité
Données sur l'effort pour chaque module
L'effort par phase de test
Nombre d'échecs attribués à un module

GQM: composantes de l'approche

- Paradigme : définit les principes à suivre
- Plan : décrit l'objectif des mesures, les questions dérivées et les mesures qui en découlent + définit quelles mesures utiliser et pourquoi
- Méthode : donne les lignes directrices pour initier et exécuter des programmes de mesure

Paradigme GQM

- Le paradigme de GQM est basé sur l'idée que la mesure doit être guidée par un objectif
- Toute collecte de données dans un programme de mesure doit être basée sur un raisonnement explicitement documenté
- Avantages

➢ Aide dans l'identification des indicateurs utiles et appropriés et dans l'analyse et l'interprétation des données collectées

➢ Permet une évaluation de la validité des conclusions tirées et évite les rejets des programmes de mesure

Paradigme GQM

- Principes

- La tâche d'analyse à exécuter doit être spécifiée avec précision et de manière explicite (objectif explicite de la mesure)
- Chaque indicateur doit avoir une justification explicitement documentée ; cette justification est utilisée pour expliquer la collecte des données et pour guider l'analyse et l'interprétation de ces données
- Les personnes qui définissent l'objectif de la mesure doivent être complètement impliquées dans l'initiation et l'exécution du programme de mesure

Plan GQM

- Le plan décrit en détail l'analyse basée la mesure
- Il comporte trois niveaux de raffinement

– Niveau conceptuel : un objectif est définit pour une entité, en fonction d'un modèle de qualité, par rapport à une point de vue dans un environnement donné

– Niveau opérationnel : un ensemble de questions est utilisé pour définir quantitativement l'objectif et spécifier comment cet objectif sera interprété

– Niveau quantitatif : un ensemble de données est associé à chaque question pour permettre d'y répondre de manière quantitative

Plan GQM

- Un objectif doit préciser
  - Quelle entité est analysée
  - L'objectif de l'analyse
  - Quelle caractéristique est analysée
  - Le point de vue qui doit guider l'analyse
  - L'environnement de l'analyse

Méthode ou processus GQM

- Il n’existe pas une façon standard d’appliquer l’approche GQM
  Un exemple de processus est celui en 7 étapes

– Caractérisation de l’organisation et du projet
– Identification des objectifs de la mesure
– Production du plan GQM
– Production du plan de mesure
– Collecte et validation des données
– Analyse des données et interprétation
– Stockage des résultats à fins de réutilisation

La mesure pour l'amélioration

- Mesure = Amélioration systématique
- Relation entre mesure et amélioration

– La mesure décrit quantitativement l'état courant

– La connaissance de l'état courant permet de définir des objectifs quantitatifs réalistes d'amélioration

– La connaissance de l'état actuel permet d'identifier les points forts et les points faibles du processus utilisé

– La connaissance des points faibles du processus permet d'identifier les changements à faire pour l'améliorer

– L'impact d'un changement ne peut être mesuré que s'il existe une base quantitative permettant la comparaison

La mesure pour l'amélioration

- L'approche GQM fait partie d'une approche globale appelée QIP (Quality Improvement Paradigm)
- Processus d'amélioration en 6 étapes

– Caractérisation

– Définition des objectifs

– Choix du processus

– Exécution

– Analyse

– Consolidation

Les tests logiciel

Tester: exécuter le programme dans l'intention d'y trouver des anomalies ou des défauts

G. J. Myers (The Art of Software Testing, 1979)

- Le test est une méthode dynamique visant à trouver des bugs
- Le coût de test est de 30 à 40% du coût de développement.

➢ Mais un bug peut coûter encore plus cher.

Les tests logiciel

- Les questions à se poser dans la phase des tests:

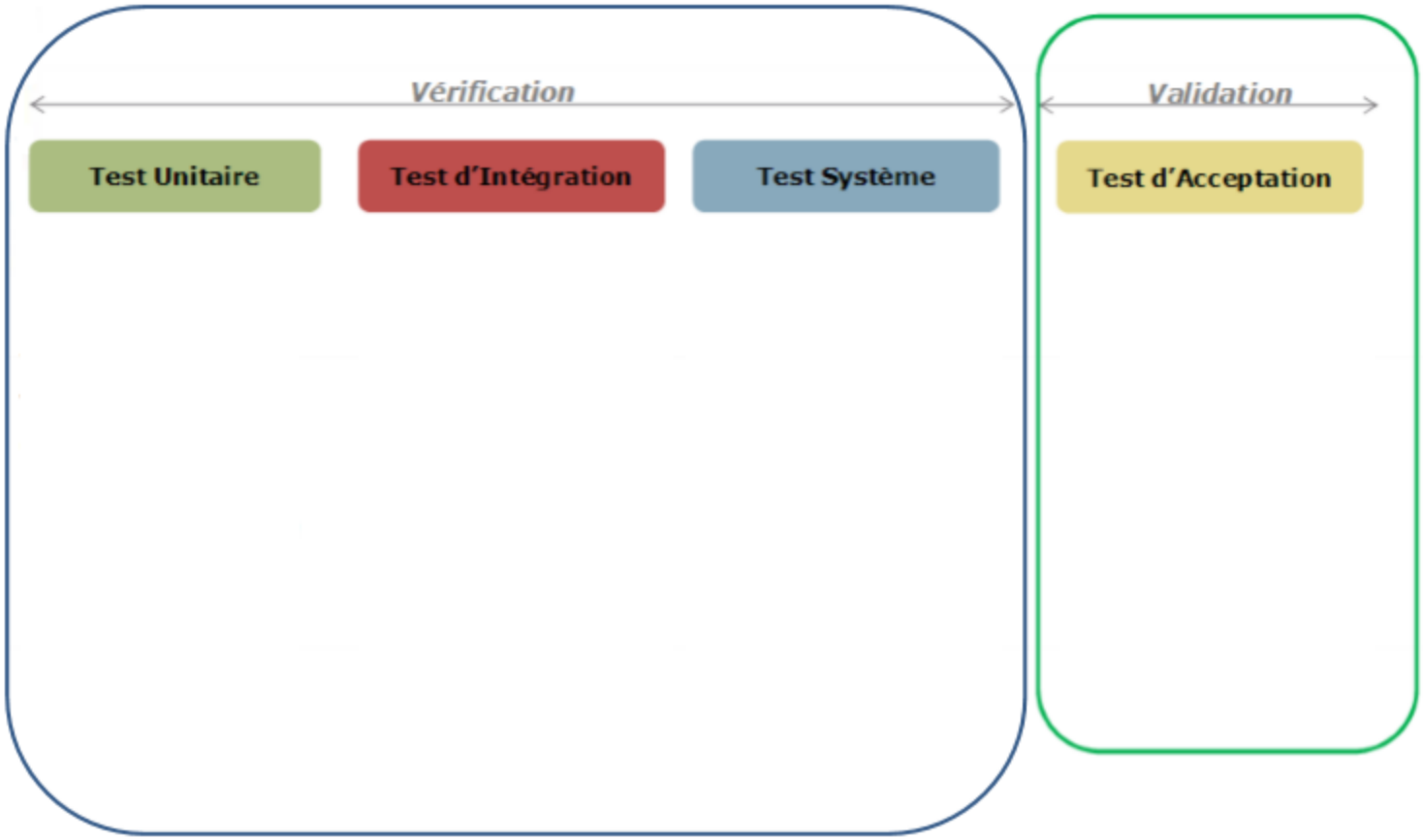
- Est-ce que le logiciel fonctionne correctement ?

  Vérification

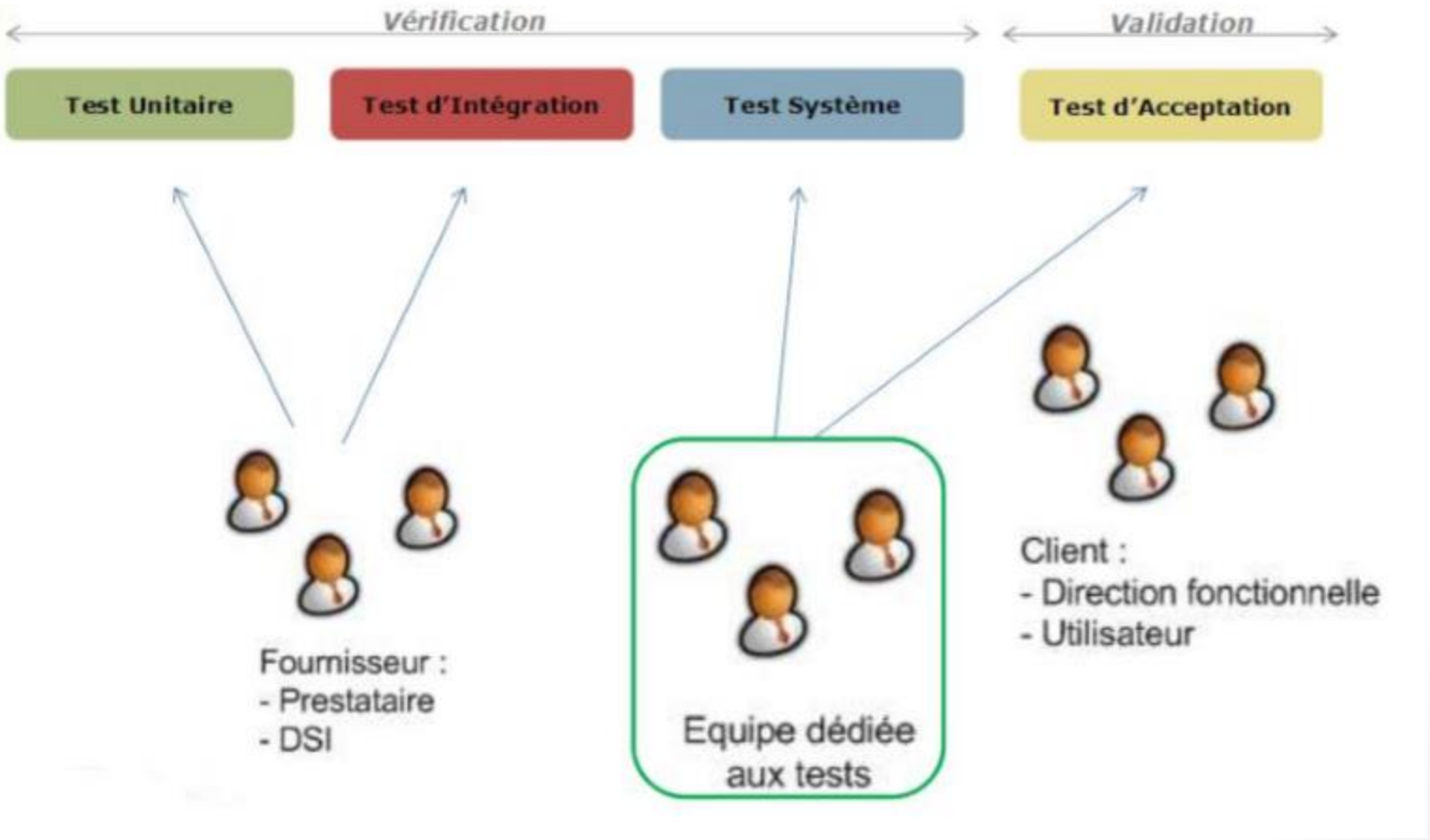
- Est-ce le logiciel fait ce que le client veut ?

  Validation

Phases des tests



Phases des tests



Période des tests

- Dépend de la méthodologie employée

❑Cascade , cycle en V..

- A la fin du projet

❑ Itérative (exemple : Scrum, RUP ...)

- Tout au long du projet

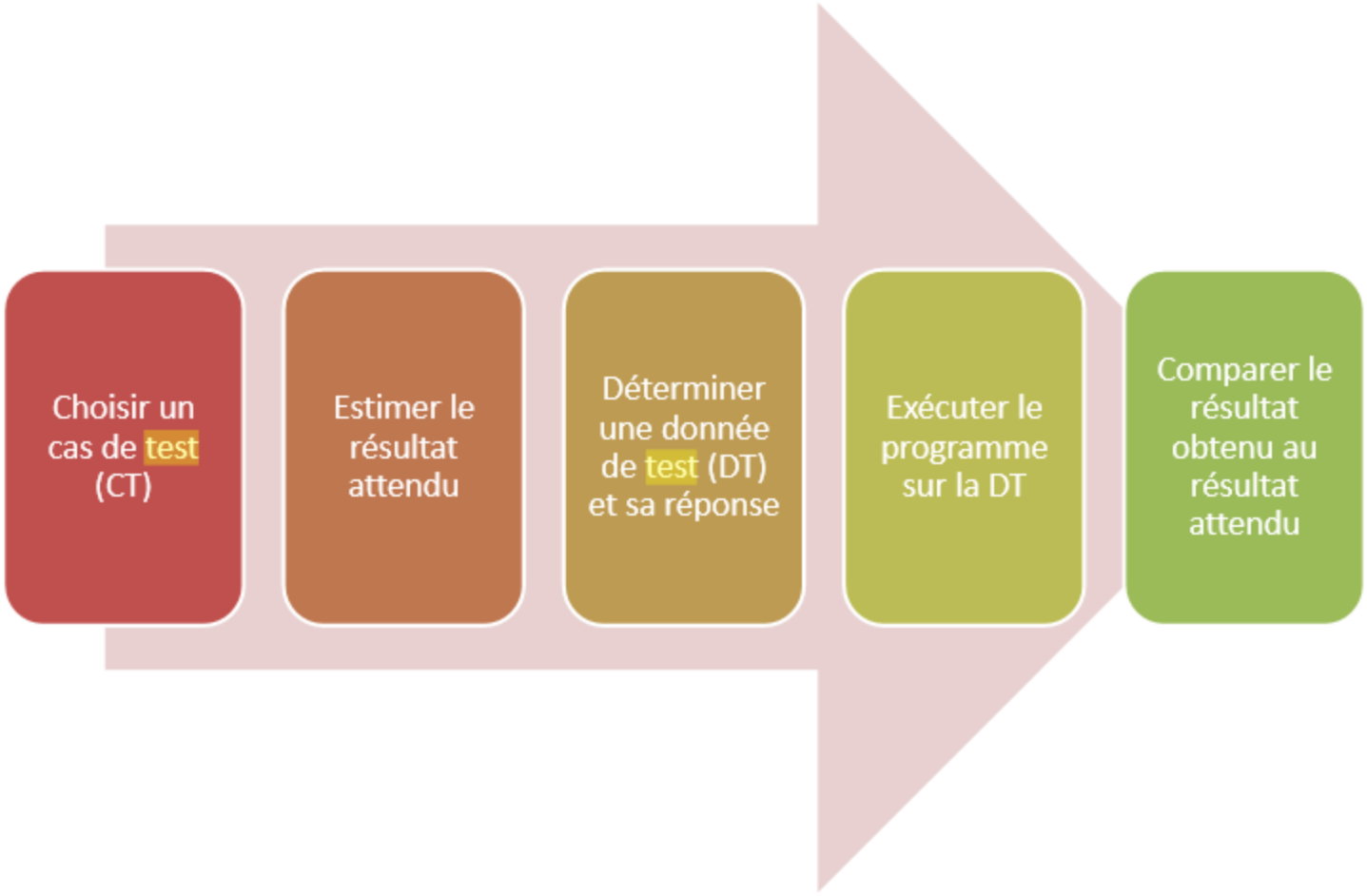
Contraintes des tests

- Base de données : volume des données, intégrité
- Web: disponibilité, multi-navigateur, liens
- Compilateur: test du langage d'entrée, des optimisations, code généré
- Interface graphique: multi-threading, séquences d'actions
- Code embarqués: avionique, nucléaire, téléphone mobile
- OS

Méthodes de tests

- Il y a différentes méthodes de tests:
- Test boite noire-blanche
- Prédiction d'erreur (on sait ce qu'on cherche)
- Tests automatiques, ...

Processus du test



Critères des tests

- Boite Noire À partir de spécifications
- Boite Blanche À partir du code
- Boite Grise Domaines des entrées + arguments statistiques

Boite noire

- Les tests en « boite noire » consistent à examiner uniquement les fonctionnalités d'une application, c'est-à-dire si elle fait ce qu'elle est censée faire, peu importe comment elle le fait. Sa structure et son ,fonctionnement interne ne sont pas étudiés.
- Le testeur doit donc savoir quel est le rôle du système et de ses fonctionnalités, mais ignore ses mécanismes internes. Il a un profil uniquement « utilisateur ».

Boite noire

- Simplicité: ces tests sont simples à réaliser, car on se concentre sur les entrées et les résultats. (+)
- Rapidité: en raison du peu de connaissances nécessaires sur le système, le temps de préparation des tests est très court. (+)
- Impartialité: on est ici dans une optique « utilisateur » et non « développeur ». (+)
- Superficialité: ces tests ne permettent pas de voir, en cas de problème, quelles parties précises du code sont en cause. (-)
- Redondance: il y a une forte chance de tester qu'une action a tendance à être inclus dans celui d'autres tests. (-)

Boite blanche

- Les tests en « boîte blanche » consistent à examiner le fonctionnement
- d’une application et sa structure interne, ses processus, plutôt que ses
- fonctionnalités. Sont ici testés l’ensemble des composants internes du
- logiciel ou de l’application, par l’intermédiaire du code source,
- principale base de travail du testeur.

Boite blanche

- Anticipation: ce test au cours du développement d'un programme permet de repérer des erreurs (problèmes dans le futur). (+)
- Optimisation: étant donné qu'il travaille sur le code, le testeur peut également
- profiter de son accès pour optimiser le code. (+)
- Exhaustivité: étant donné que le testeur travaille sur le code, il est possible de vérifier intégralement ce dernier. (+)
- Complexité: ces tests nécessitent des compétences en programmation, et une connaissance accrue du système étudié. (-)
- Durée: à cause de la longueur du code source, ces tests peuvent être très longs. (-)
- Industrialisation: il est nécessaire de se munir d'outils tels que des analyseurs de code, des débogueurs. (-)
- Cadrage: très compliqué de cadrer le projet. (-)
- Intrusion: être risqué de laisser son code à la vue d'une personne externe à son entreprise, (-)

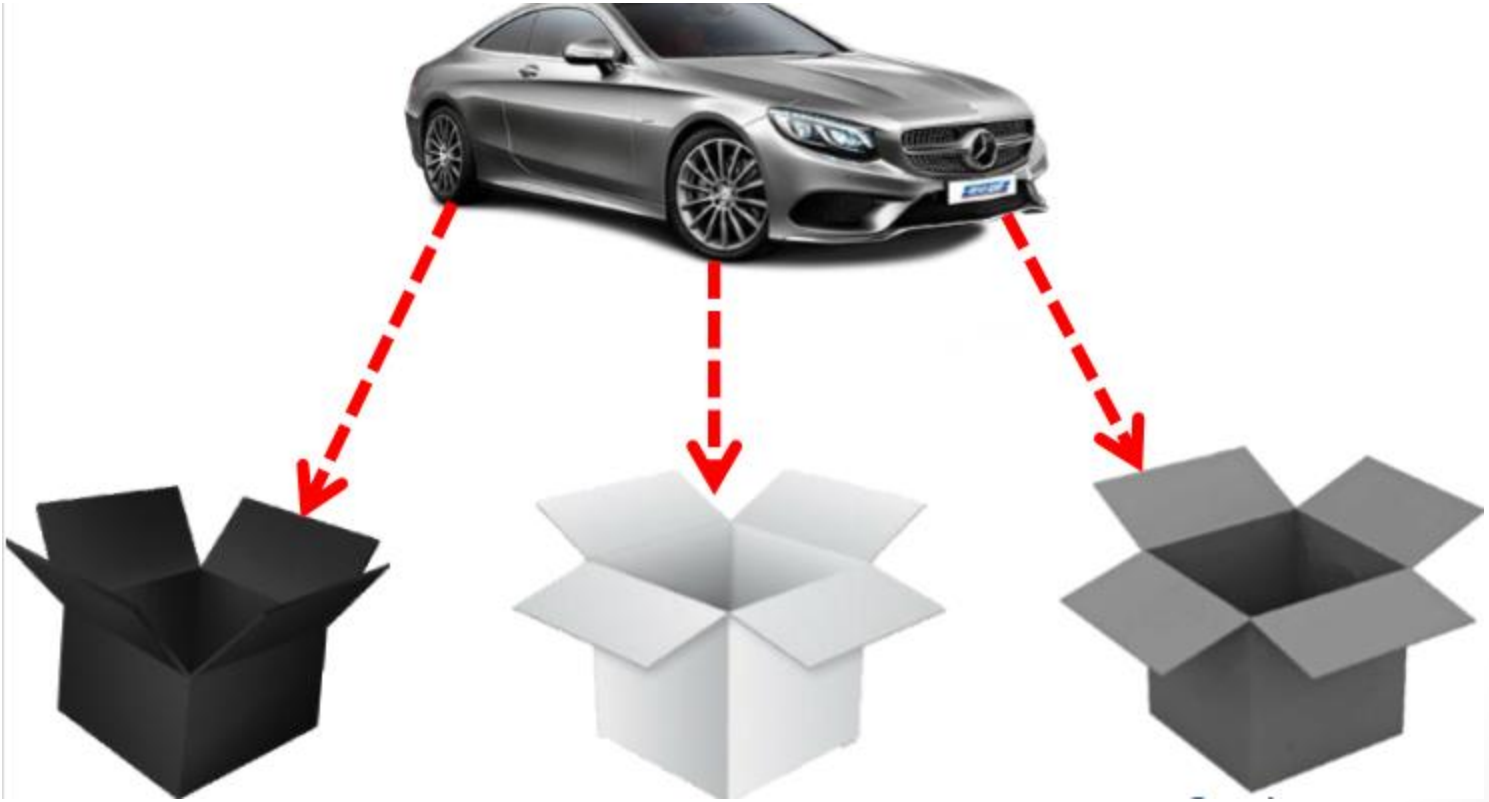
Boite Grise

- Les tests en « boîte grise » compilent ces deux précédentes approches :ils éprouvent à la fois les fonctionnalités et le fonctionnement d'un système. C'est-à-dire qu'un testeur va par exemple donner une entrée (input) à un système, vérifier que la sortie obtenue est celle attendue, et vérifier par quel processus ce résultat a été obtenu.

Boite grise

- Impartialité: les tests gardent une démarcation entre le développeur(s) et le testeur(s). (+)
- Intelligence: en connaissant la structure interne du programme, un testeur peut créer des scénarios plus variés et intelligents. (+)
- Non exhaustivité: étant donné que le code source n'est pas accessible, il est impossible avec ce genre de tests espérer avoir une couverture complète du programme. (-)

Exercice



Phases de Test

- Test unitaire
- Test d’intégration
- Test de conformité / système
- Test de validation / Acceptation
- Test de régression

Test Unitaire

- Test d’un bloc (unité) de programme (classe, méthode, etc.) en isolation.
- Tous les langages de programmation et tous les frameworks de développement dignes de ce nom proposent des outils de testing unitaire.
- Plus le code est modulaire, plus il est facile à tester unitairement
- Diviser pour mieux régner

Test Unitaire

- Jeu de test
- Trace pour le contrôle (le vérificateur vérifie que le testeur a effectué les tests du jeu de test)
- Trace entre la conception et l'exécution du test (entre le moment de la spécification et celui ou le code est écrit et peut être testé)
- Séparer l'exécution des tests de l'analyse des résultats (ce n'est pas au moment où on fait le test qu'il faut juger de la pertinence des sorties, ce n'est pas la même personne qui le fait la plupart du temps)

Test Unitaire

- Qui peut faire le test unitaire ?
- Pour les logiciels de faible criticité (état d'un milieu), elle peut être réalisée par l'équipe de développement (mais pas par le développeur ayant codé la pièce de code).

Test Unitaire

- Outils XUnit:
- Ensemble d'environnements pour programmer des tests unitaires
- JUnit pour Java
- Cunit pour le C
- CppUnit pour le C++
- NUnit pour le C#
- PyUnit pour le Python
- JsUnit pour le JavaScript
- FlexUnit pour Adobe Flex / ActionScript (Flash. . .)
- PHPUnit pour PHP

Tests d’intégration

- L'intégration, c'est simplement le fait d'assembler plusieurs composants logiciels élémentaires pour réaliser un composant de plus haut niveau.
- Par exemple, le fait d'utiliser une classe Client et une classe Produit pour créer un module de processus commande sur un site marchant, c'est de l'intégration !
- Un test d'intégration vise à s'assurer du bon fonctionnement de la mise en œuvre conjointe de plusieurs unités de programme, testés unitairement au préalable.
- Vérification et validation des sous-systèmes logiciels entre eux

Tests d'intégration

- Qui peut faire le test d'intégration ?
- Toujours par une équipe de tests indépendante de l'équipe de développement.

Tests de validation

- Ce type de test intervient à la fin du processus d'intégration. Il permet de tester le système dans son ensemble avec des données réelles.
- Vérifier la conformité du logiciel à la spécification du logiciel

Tests de validation

- Qui peut faire le test de validation ?
- Toujours par une équipe de tests indépendante de l‘équipe de développement.

Test de (non) Régression

- Chaque fois que le logiciel est modifié, s’assurer que « ce qui fonctionnait avant fonctionne toujours ».
- Vérifier que la correction des erreurs n'a pas infecté les parties déjà testées. [Cela consiste à systématiquement repasser les tests déjà exécutés]

Types de tests

- Fonctionnels
- Est-ce que le logiciel est conforme à la spécification ? Liés à la spécification, à la qualité, à la performance, à l'interfaçage, ...
- Structurel
- Est-ce que le codage est correct ? Fautes d'implémentation, ou fonctions non prévues ….

Tests fonctionnels

- Vérifier que l'algorithme mis en œuvre permet de résoudre le problème posé
- Vérifier le comportement d'un logiciel / spécification (fonctions non conformes ou manquantes, erreurs d'initialisation ou de terminaison du logiciel)
- Vérifier le respect des contraintes (performances, espace mémoire, etc.) et des facteurs qualité associés au logiciel (portabilité, maintenabilité, etc.)

Tests structurels

- Valide l'implémentation d'un algorithme. Elle est automatisable
- Détecter les fautes d'implémentation.
- Vérifier que le logiciel n'en fait pas plus que sa spécification et qu'il n'existe pas de cas de plantage.
- Critère d'arrêt : lié à la structure du code et non à la fonctionnalité du logiciel.