Génie Logiciel - Support de cours

Régis WITZ – rwitz@unistra.fr

Table des Matières

Définitions

Le projet

Un projet est un processus **unique**, qui consiste en un ensemble d'**activités** coordonnées et maîtrisées comportant des dates de début et de fin, entrepris dans le but d'atteindre un objectif conforme à des exigences spécifiques de **qualité** avec des contraintes de **coûts** et de **délais**.

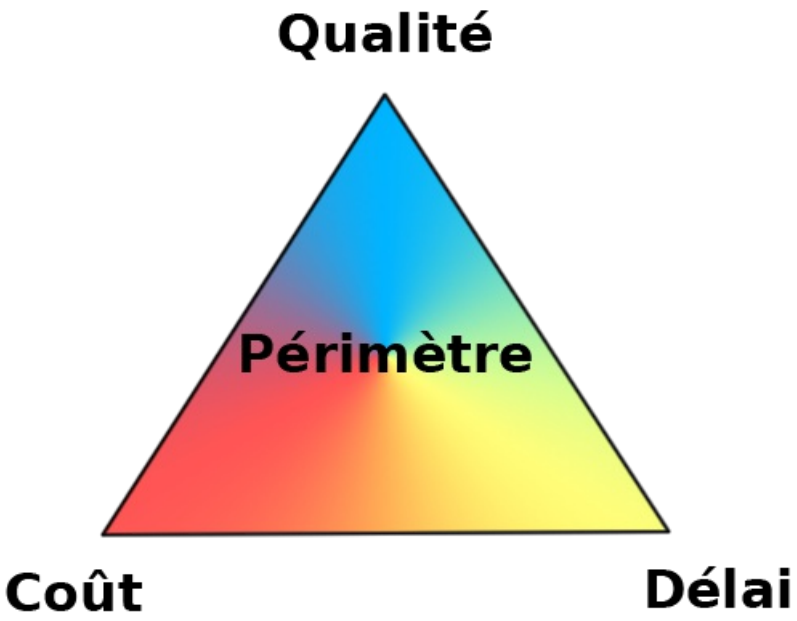


Figure 01:Qualité, Budget, Délai sont interdépendants

Ces contraintes de qualité, de coûts et de délais fixent le **périmètre** du projet. Elles sont interdépendantes : "gagner" sur l'une fait souvent "perdre" sur l'une ou l'autre.

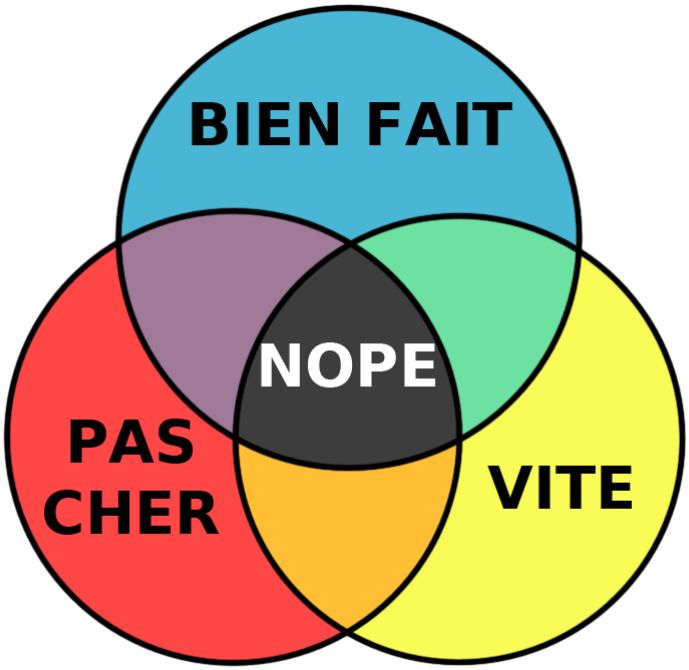


Figure 02:« Je veux le meilleur, le moins cher et je ne veux pas attendre ! »

Les activités d'un projet informatique créent des **livrables** : outils, méthodes ou services informatiques.

Les acteurs et leurs rôles

Le projet est réalisé avant tout par des hommes et des femmes. Ces personnes physiques sont réunies en *entités* qui sont autant de personnes morales (une entreprise, un service ou une équipe particuliers, …).

Chacune de ces entités assume une ou plusieurs responsabilités (planifier, concevoir, développer, tester, …) qui, si elles sont correctement menées, aboutiront au succès du projet.

Durant la phase de réalisation d'un projet, on peut distinguer deux entités particulières :

- le **client**: aussi appelé *Maîtrise d'ouvrage* (*MOA*) ou *Project owner*, il s'agit de celui qui **exprime le besoin**.
- le **fournisseur**: aussi appelé *Maîtrise d'œuvre* (*MOE*), il s'agit de celui qui **répond au besoin**.

En plus du client et de son fournisseur, d'autres entités peuvent être **partie prenante** dans le projet. Il s'agit des utilisateurs finaux de la solution, des organismes finançant le projet, etc.

La Maîtrise d'OuvrAge (MOA)

La maîtrise d'ouvrage est à l'origine du projet. C'est elle qui en fixe les objectifs que sont le **besoin** à remplir, ainsi que les **coûts** et les **délais** du projet.

C'est elle qui détient la **connaissance fonctionnelle** du domaine dans lequel s'inscrit le projet, tel que le cadre dans lequel doit exister le projet, et l'attente des fonctionnalités qu'il aportera.

Il s'agit du **propriétaire** au final du projet. La MOA est donc **celui qui paye**.

La Maîtrise d'Œuvre (MOE)

La maîtrise d'œuvre fournit le produit attendu par la maîtrise d'ouvrage. C'est elle qui assure la qualité, les coûts et les délais.

Elle peut se faire aider par un ou plusieurs fournisseurs. C'est alors à elle qu'incombe la responsabilité de coordoner et de superviser ses prestataires.

La MOE est **celui qui réalise**.

Qualité d'un logiciel

Lorsqu'on parle de la qualité d'un logiciel, on parle en fait de plusieurs **caractéristiques** différentes, elles-mêmes décomposées en sous-caractéristiques (ou attributs).

Toutes les caractéristiques d'un logiciel n'ont pas forcément la même importance pour une organisation donnée. C'est pourquoi chaque organisation définit son propre **modèle de qualité**, avec parfois plusieurs modèles applicables à ses différents produits logiciels. En particulier, chaque organisation définit ses propres méthodes (métriques) pour évaluer le **degré de présence** des différents attributs de qualité.

Les principales **normes** utilisées pour la définition de modèle(s) de qualité : ISO/IEC 9126 (ancienne version) et ISO/IEC 25010 (nouvelle version).

Dans ces normes, le terme *logiciel* est employé au sens large. Il peut désigner à la fois le code source, les exécutables, les librairies, les documents d'architecture, et ainsi de suite. En conséquence, la notion d'*utilisateur* peut elle aussi s'appliquer, suivant la caractéristique et/ou le logiciel considéré, à un client ou à un développeur, par exemple.

Caractéristiques

Capacité fonctionnelle (Functional suitability)
Est-ce que le logiciel répond aux besoins fonctionnels exprimés ?

Pertinence (Suitability / Functional appropriateness)
Est-ce que toutes les fonctionnalités présentes sont adéquates ?

Exactitude (Accuracy / Functional Correctness)
Est-ce que les résultats ou effets fournis sont corrects ?

Complétude (Functional completeness)
Est-ce que les résultats ou effets fournis sont suffisants ?

Conformité (Functionality compliance)
Est-ce que le logiciel respecte les normes et règlements en vigueur ?

Fiabilité (Reliability)
Est-ce que le logiciel maintient son niveau de service dans des conditions précises et pendant une période déterminée ?

Maturité (Maturity)
Est-ce que la fréquence d'apparition des incidents est faible ?

Tolérance aux pannes (Fault tolerance)
Est-ce que l'apparition d'un incident dégrade le comportement du logiciel ?

Facilité de récupération (Recoverability)
Quel est la capacité du logiciel à retourner dans un état opérationnel complet après une panne ?

Disponibilité (Availability)
Est-ce que les fonctionnalités du logiciel sont disponibles en permanence ?

Conformité (Reliability compliance)
Est-ce que le logiciel respecte les normes et règlements en vigueur ?

Ergonomie (Usability)
Effectivité, efficacité et satisfaction avec laquelle des utilisateurs spécifiés accomplissent des objectifs spécifiés dans un environnement particulier.

Facilité de compréhension (Understandability / appropriateness recognizability)
Est-ce qu'il est (intuitivement) facile pour un utilisateur de comprendre et retenir comment réaliser une opération ?

Facilité d'apprentissage (Learnability)
Est-ce qu'il est facile de former de futurs utilisateurs à l'utilisation du logiciel ?

Facilité d'exploitation (Operability)
Est-ce qu'il est facile pour l'utilisateur d'accomplir sa tâche ?

Protection contre les erreurs d'utilisation (User error protection)
Comment réagit le logiciel si son utilisateur a un comportement inatendu ?

Accessibilité (Accessibility)
Est-ce que le logiciel est utilisable par des personnes en situation de handicap ?

Attrait (Attractiveness / user interface aesthetics)
Est-ce que l'utilisateur a envie d'utiliser l'interface ?

Conformité (Usability compliance)
Est-ce que le logiciel respecte les normes et règlements en vigueur ?

Rendement (Performance Efficiency)
Est-ce que le logiciel requiert un dimensionnement rentable et proportionné de la plate-forme d'hébergement en regard des autres exigences ?

Comportement temporel (Time behaviour)
Est-ce que le temps de réponse du logiciel varie en fonction de son usage ?

Utilisation des ressources (Resource utilization)
Est-ce que les resources (mémoire, processeur, …) sont correctement utilisées par le logiciel ?

Capacité (Capacity)
Est-ce que les ressources fournies par le logiciel le sont au bon moment et en quantité adéquate ?

Conformité (Efficiency compliance)
Est-ce que le logiciel respecte les normes et règlements en vigueur ?

Maintenabilité (Maintainability)
Est-ce que faire évoluer le logiciel est aisé en cas d'anomalie ou de nouveaux besoins ?

Facilité d'analyse (Analyzability)
Est-ce qu'il est facile d'identifier les déficiences du logiciel ou leur cause ?

Facilité de modification (Modifiability)
Quel est l'effort nécessaire pour remédier à ces déficiences ?

Testabilité (Testability)
Quel est l'effort nécessaire pour valider le comportement du logiciel ?

Modularité (modularity)
Est-ce que modifier un composant du logiciel impacte peu les autres composants ?

Ré-utilisabilité (Reusability)
Est-ce qu'il est facile de combiner des composants existants pour obtenir une nouvelle fonctionnalité ?

Conformité (Maintainability compliance)
Est-ce que le logiciel respecte les normes et règlements en vigueur ?

Portabilité (Portability)
Est-ce que le logiciel peut être transféré d'une plate-forme ou d'un environnement à un autre ?

Facilité d'adaptation (Adaptability)
Est-ce qu'il est facile d'adapter le logiciel à des changements d'environnement opérationnel ?

Facilité d'installation (Installability)
Est-ce qu'il est facile d'installer le logiciel dans un environnement prévu ?

Interchangeabilité (Replaceability)
Est-ce qu'il est facile d'utiliser ce logiciel à la place d'un autre dans un même environnement ?

Conformité (Portability compliance)
Est-ce que le logiciel respecte les normes et règlements en vigueur ?

Compatibilité (Compatibility) Est-ce que le logiciel fonctionne bien avec d'autres ?

Coexistence (Co-existence)
Est-ce que le fonctionnement du logiciel altère ou est altéré par un autre ?

Interopérabilité (Interoperability)
Est-ce que le logiciel peut communiquer avec d'autres ?

Sécurité (Security) Est-ce qu'on peut faire confiance au logiciel ?

Confidentialité (Confidentiality)
Est ce que les données ne sont accessibles qu'à ceux qui y sont autorisés ?

Intégrité (Integrity)
Est-ce que les donnés sont garanties contre une modification non autorisée ?

Non-répudiation (Non-repudiation)
Est-ce que le logiciel peut prouver que chaque action a bien eu lieu ?

Responsabilité (Accountability)
Est-ce que le logiciel peut relier chaque action à son auteur ?

Authentification (Authenticity)
Est-ce que le logiciel peut empêcher une identité d'être usurpée ?

La « Sur-qualité »

On appelle parfois « sur-qualité » le fait de réaliser une fonctionnalité alors que celle-ci n'est pas spécifiée.

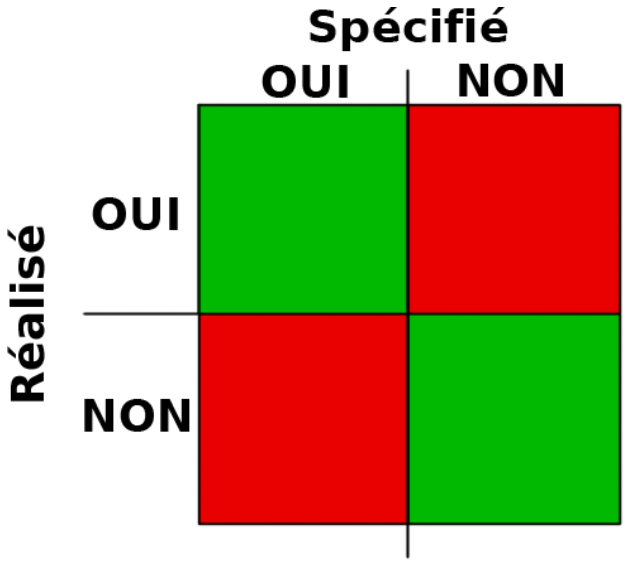


Figure 03:« Sur-qualité »

L'interprétation habituelle est alors la suivante :

- Réaliser une fonctionnalité demandée est normal : c'est ce en quoi consiste la qualité logicielle.
- Ne pas réaliser une fonctionnalité alors qu'elle est demandée dégrade la qualité du logiciel.
- Ne pas réaliser une fonctionnalité qui n'est pas demandée est normal.
- Mais réaliser une fonctionnalité alors que celle-ci n'est pas demandée est une perte ou un manque à gagner pour le fournisseur, étant donné que le client ne va pas payer pour obtenir quelque chose qu'il n'a pas demandé !

Bien que tout à fait défendable, cette façon de résumer la « sur-qualité » est limitative. Il est possible d'interpréter la situation de manière plus nuancée, en se posant la question : la fonctionnalité considérée est elle *utile* ou *inutile* pour le client ? Cette question exprime parfaitement la différence entre le besoin tel qu'il est *exprimé* par le client et le besoin *réel* du client (voir la phase d'analyse du besoin).

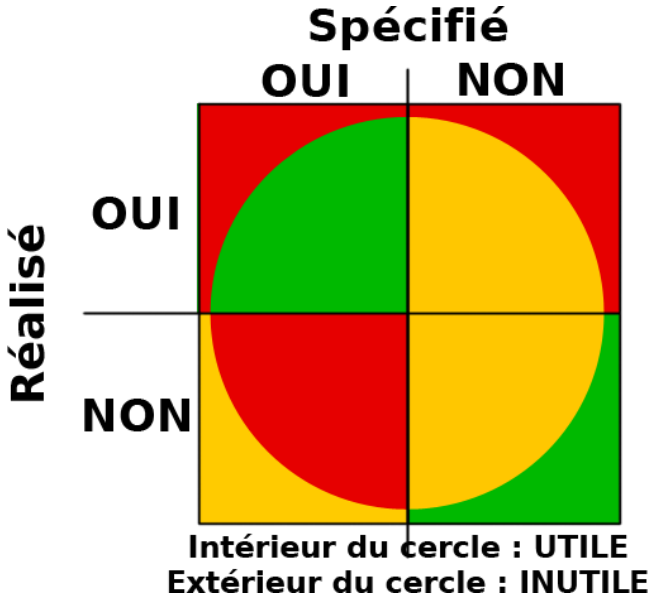


Figure 04:Une vision alternative de la « Sur-qualité »

On a alors l'interprétation suivante :

- Réaliser une fonctionnalité demandée est normal *si celle-ci est utile*.
- Réaliser une fonctionnalité demandée mais *inutile* peut signifier un manque lors de la phase d'analyse du besoin.
- Ne pas réaliser une fonctionnalité alors que celle-ci est demandée est une faute du fournisseur.

    Cependant, ci la fonctionnalité non réalisée s'avère inutile pour le client, celui-ci ne sera

peut-être pas gêné par son absence.

Ne pas réaliser une fonctionnalité qui n'est pas demandée est normal.

Mais si la fonctionnalité non réalisée aurait été finalement utile au client ? Il s'agit-là d'une opportunité d'amélioration du produit …

Réaliser une fonctionnalité alors que celle-ci n'est pas demandée est:

une perte pour le fournisseur, effectivement, si la fonctionnalité en question est inutile.

une manière d'aller au devant du besoin du client, et donc une preuve de la qualité du service fourni par le fournisseur, qui apporte la preuve qu'il a mieux saisi le besoin réel du client que le client lui-même.

Avec ce point de vue alternatif, la « sur-qualité » peut donc devenir un atout commercial.

Phases d'existence

Analyse du besoin / Faisabilité / Cahier des charges

Cette phase permet de collecter les données nécessaires pour rendre compte de la situation réelle.

Il est d'abord nécessaire de **comprendre le contexte**, en utilisant toutes les sources d'information pertinentes. La principale source d'information est le client, auquel il faut poser toute question utile, en n'hésitant pas à approfondir : certains éléments vitaux sont parfois tellement évidents/implicites pour le client qu'il ne pense pas à les mentionner.

Ensuite, mettre en perspective les besoins exprimés par avec le contexte permet de redéfinir ou valider les besoins **réels**. Ces besoins réels devraient être ordonnés par degré d'importance et peuvent être regroupés par thèmes.

À cette phase sont aussi établies les différents paramètres de conception, et en particulier les **contraintes** qui peuvent empêcher d'atteindre toute ou partie des objectifs de départ. Les contraintes techniques ou économiques ne sont pas les seules à considérer ; certains domaines sont aussi touchés par des contraintes légales, sociales, environnementales, et ainsi de suite.

La phase d'analyse aboutit à la validation par la MOA d'un **cahier des charges** réaliste et convenant à ses besoins réels. Ce document précise clairement les tâche à effectuer ainsi qu'une estimation du temps nécessaire pour chacune.

Cette phase permet aussi à la MOE d'évaluer qu'elle a accès à toutes les compétences nécessaires au projet, ou si le recours à des ressources externes est nécessaire.

Étude préalable

En amont de la phase d'analyse du besoin, on peut trouver une phase encore antérieure d'étude préalable. Tandis que l'analyse du besoin vise à être précise et exhaustive, l'étude préalable vise simplement à dresser un premier tour d'horizon et à fixer le périmètre.

La phase d'étude préalable ne vise pas l'exhaustivité, mais à orienter les phases suivantes (y compris l'analyse détaillée du besoin).

Exemple d'outils d'analyse

QQOQCCP

4 questions : Qui ? Quoi ? Où ? Quand ?

Complétées par 3 modalités : Comment ? Combien ? Pourquoi ?

Un équivalent anglais est le Five W's : Who, What, Where, When, Why ? Utilisé par les journalistes pour la rédaction de leurs articles.

Méthode des 5 pourquoi

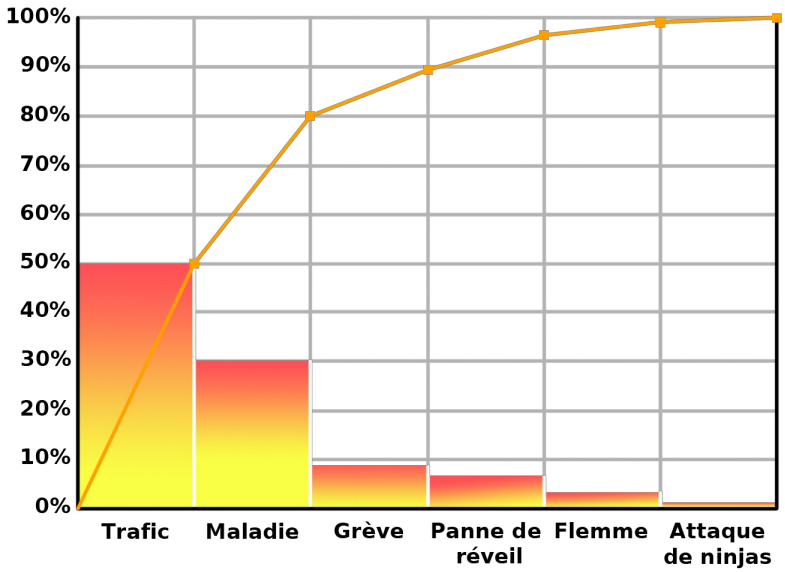
Poser successivement cinq questions commençant par *Pourquoi* peut permettre de mieux cerner la cause réelle d'un problème.

Diagramme de Pareto

Un tel diagramme représente sous forme graphique les différentes causes d'un problème, ordonnées par leur importance.

C'est une application de la loi des 80/20 (ou principe de Pareto) : 20% des causes produisent 80%

des effets.

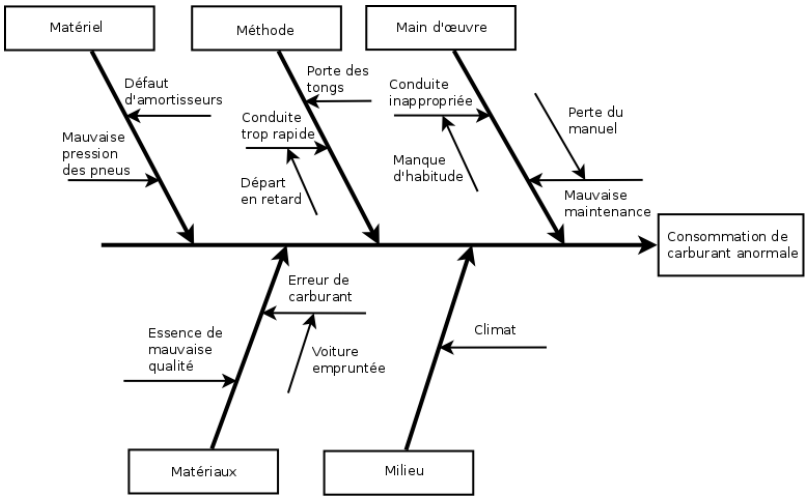


Exemple de diagramme de Pareto:Causes de retards en cours

Diagramme d'Ishikawa

Aussi appelé 5M ou arêtes de poisson, ce diagramme permet de mettre en relation les causes et les effets.

- Matériaux entrant en jeu
- Matériel : équipement, machines, logiciels, technologies
- Méthode opératoire, logique
- Main d'œuvre humaine
- Milieu : environnement, contexte



Exemple de diagramme d'Ishikawa:Causes d'une consommation de carburant anormalement élevée

Spécification

Lors de la phase de spécification, le besoin qui a été analysé précédemment est décrit avec plus de détail, sous forme d'**exigences** que la solution doit impérativement satisfaire.

Un document de spécification peut être de deux types principaux :

- Une spécification **fonctionnelle** décrit les **processus métier** dans lesquels la solution intervient. Par exemple, les unités utilisées, les règles de calcul ou d'interaction, etc. La spécification fonctionnelle représente le **but à atteindre**.
- Une spécification **technique** décrit l'**environnement technique** dans lequel la solution s'inscrit.

> Par exemple, le design architectural, le format des données d'échange avec les composants déja présents, les langages de programmation utilisés, le format des bases de données, le système hôte, … peuvent être fixées dans ce document. La spécification technique représente le **moyen d'atteindre le but** fixé par la partie fonctionnelle.

Cette phase débouche souvent sur plus d'un document de spécification. Notamment, les exigences peuvent être raffinées de plus en plus au cours de cette phase jusqu'à atteindre un niveau de détail satisfaisant : on peut alors créer des spécifications générales, puis plus détaillées.

Le plus souvent, c'est la MOA qui est à l'origine des spécifications générales. Il peut cependant être pertinent que les spécifications détaillées soient plutôt écrites par la MOE.

Puisqu'elle décrit aux futurs utilisateurs et développeurs à quoi ressemblera le produit fini, la spécification permet de faire les estimations de coût et de durée. Elle sert donc de base pour établir le planning du projet.

La spécification sert aussi de base contractuelle. Après cette phase, toutes les fonctionnalités qui sont hors-spécification n'ont pas à être ni demandées, ni payées par le client.

Conception

Tandis que la phase de spécification a pour but de décrire la solution vue de l'extérieur, la conception la décrit vue de l'intérieur. Tandis que la phase de spécification décrit les contraintes, la conception apporte les solutions.

C'est le travail de la MOE.

Comme toute documentation, elle peut être raffinée de plus en plus, par exemple en un documents de conception préliminaire/architecturale, puis détaillée.

Implémentation / Développement

Cette phase consiste en la **réalisation** de la solution telle qu'elle a été conçue.

Tests

Tester le logiciel tel qu'il est implémenté a pour objectif d'améliorer la qualité ou de connaître le degré de présence d'une qualité particulière.

Un **test** consiste en la vérification *partielle* du logiciel. Il correspond à la combinaison de trois choses:

- des données en entrée
- un objet à tester
- une situation attendue

Si la situation attendue correspond à la situation observée lors du test, c'est un signe de la qualité du logiciel.

Intégration

Durant la phase d'**intégration**, chaque module du logiciel est intégré et testé dans l'ensemble.

Aussi appelé **tests fonctionnels**, cette phase a pour but de vérifier l'aspect fonctionnel (incluant performances, stabilité, etc), parfois non détectable par des tests de plus bas niveau.

Validation

Durant la phase de **validation**, le système est testé dans son ensemble, et dans un environnement se rapprochant au maximum de l'environnement final. Le but est d'évaluer sa conformité avec les exigences spécifiés.

Un type particulier de validation est la **recette**. Elle se déroule en présence de tous les acteurs (MOA et MOE). Elle précède souvent un jalon important de la vie du projet, comme une livraison.

Déploiement

Le déploiement d'un logiciel consiste à sa **mise en production**, c'est à dire à le rendre disponible et utilisable pour le client, ainsi que pour ses utilisateurs finaux.

On peut décomposer cette phase en plusieurs étapes qui s'appliqueront (ou pas) à un projet particulier.

Livraison (*release*, *packaging*)
Les différents composants de la solution sont préparés afin de les rendre utilisables.

Activation (*install*, *activation*)
La solution est rendue utilisable dans son environnement de production. Ses différents composants sont installés et configurés.

Désactivation (*uninstall*, *deactivation*)
Une solution précédente peut avoir à être totalement ou partiellement désinstallée ou désactivée pour permettre à la nouvelle solution de la remplacer.

Mise à jour (*update*)
Le nouvelle solution peut nécessiter une version plus récente de dépendances déjà présentes dans son environnement de production. Elle peut aussi faire partie d'un système plus grand, qui doit alors être mis à jour pour permettre l'activation de la solution.

Le déploiement d'une même solution peut être effectué à plusieurs reprises. Cette phase rend indispensable l'utilisation d'un **gestionnaire de version** ainsi que d'un **gestionnaire de configuration**.

Exploitation / Maintenance

Un logiciel peut être amené à évoluer même après avoir été livré, au cours d'actions de **maintenance**.

Une maintenance peut être de plusieurs types :

La **maintenance corrective** consiste à résoudre une anomalie constatée

maintenance curative
Elle corrige l'anomalie de manière permanente.

maintenance paliative
Elle empêche l'anomalie d'endommager le système on l'environnement client, tout en permettant au logiciel de continuer à remplir tout ou partie de ses fonctionnalités. Cependant, étant donné que son impact est forcément négatif à un certain degré, ce type de maintenance est souvent de nature temporaire.

La **maintenance préventive** consiste à intervenir sur un logiciel avant qu'une anomalie ne survienne. Ce type de maintenance peut être *systématique* ou *conditionnel*.

La **maintenance évolutive** permet de mieux répondre au besoin ou de répondre à de nouveaux besoins, en modifiant le logiciel existant ou en développant de nouvelles fonctionnalités.

La maintenance se différencie des autres phases en ce que le logiciel considéré est déjà en production.

Modèles de développement

Un modèle de développement ordonne de manière structurées les activités de construction du logiciel. Le détail des activités qui se dérouleront dépend du projet.

Caractéristiques

Clairement défini et implémenté

Compréhensible par les acteurs du projet

Accepté par les acteurs du projet

Observable de l'extérieur (autres acteurs, parties prenantes, …)

Il doit permettre de détecter les problèmes avant que le produit ne soit mis en service

Un unique problème imprévu ne doit pas stopper toute la réalisation

Au sein d'un modèle de développement, chaque activité doit détailler :

Les Tâches à réaliser et leurs auteurs

Les Décisions à prendre (le cas échéant)

Les artefacts livrables.

Documents

Sources

Binaires (exécutables, librairies, …)

Il y a au minimum une activité de début et une activité de fin. Il doit y avoir un chemin reliant chaque activité à celle de fin.

Deux activités sont séparées par au moins un artefact. Une activitée ne peut être commencée tant que ses artefacts d'entrée n'existent pas.

En cascade

Ce modèle linéaire se base sur deux idées :

- modifier une étape a des conséquences sur les étapes suivantes, et donc
- une étape ne peut pas être débutée avant que la précédente ne soit achevée

Ce modèle comporte un nombre **fixe et prédéfini** de phases. Chacune des phases produit un certain nombre de livrables, eux aussi définis à l'avance. Chaque phase commence et termine à une date fixe. On ne peut passer à la phase suivante que lorsque les livrables de la phase courante sont validés.

Si une anomalie est détectée, on remonte d'une ou plusieurs phases en arrière.

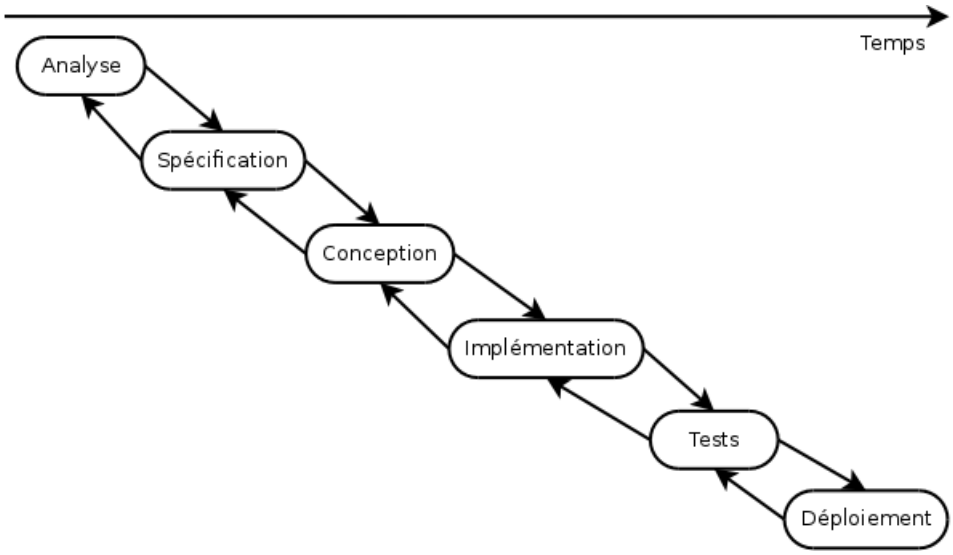


Figure 01:Cascade

Ce modèle suppose que l'on connaisse et maîtrise la plupart des exigences au lancement du projet.

Il nécessite d'accorder une attention très importante à la **documentation**. En particulier, il faut livrer proprement chaque document, puis attendre les retours et les commentaires sur cette livraison, puis faire évoluer ces documents pour y intégrer ces commentaires, et ainsi de suite, jusqu'à ce que chaque document soit accepté par toutes les parties. … Et ce, à chaque étape.

Avantages

Tout est **prévisible** : Les acteurs savent précisément ce qui doit être livré, à quelle date et ce que cela entraîne.

Inconvénients

- Le **temps nécessaire** pour obtenir un logiciel testable est important.
- Les phases les plus **risquées** (tests …) arrivent **à la fin** du cycle. Ce modèle est donc dans les faits très peu tolérant aux erreurs.
  - Que se passe-t'il si un besoin a été mal interprété ?
  - Et si un détail de conception s'avère inadapté lors de l'implémentation ou du déploiement ?
- La durée de vie d'un projet étant souvent de plusieurs années. Pourtant, ce modèle est très **intolérant** aux changements.
  - Que se passe-t'il si le besoin évolue ?
  - Et si la nature du marché change ?

Domaines d'application

Ce modèle peut néanmoins être adapté dans certains cas :

Les domaines où il est impossible ou très coûteux de revenir en arrière. Par exemple, c'est le monde du BTP qui a donné naissance à ce modèle (peut-on construire un bâtiment avant d'avoir spécifié le terrain et conçu les plans ?).

Les projets dont le périmètre est faible et la durée très courte. Dans de tels petits projets, le risque de retour en arrière est à priori faible.

Il est à déconseiller pour les nouveaux systèmes en raison des nombreux problèmes de spécification et de conception que la nouveauté entraîne.

En "V"

Ce modèle linéaire tente de mettre en évidence la **complémentarité** entre certaines phases. Chaque phase d'étude et d'analyse est **couplée** avec un phase de tests qui la valide.

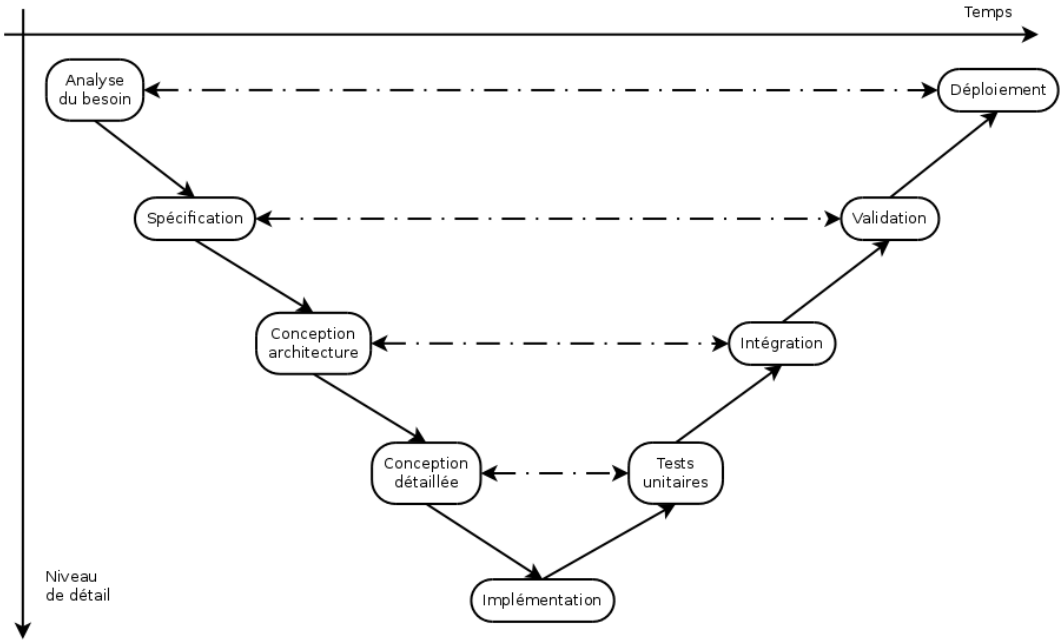


Figure 02:Cycle en V

Ce modèle est calqué sur la production industrielle classique. Il est donc de loin le plus utilisé dans le **domaine industriel**.

Avantages

Modèle **éprouvé** :

Vaste taux d'usage en entreprise depuis les années 80

Supporté par de nombreux standards

Appuyé par de nombreux outils

Les phases de test sont aussi importantes que les phases de réflexion. De plus, il est plus facile de décrire de manière exhaustive comment tester une fonctionnalité au moment où celle-ci est conçue. La synergie entre analyse et description du test est profitable.

Puisqu'il favorise la décomposition hiérarchique et fonctionnelle, il permet l'organisation du travail, des équipes et la maîtrise des coûts (voir la méthode COCOMO). Cela lui offre une bonne visibilité. Le suivi de projet est facilité.

Inconvénients

Ne fait qu'amenuiser les inconvénients du modèle en cascade sur lequel il est basé. Le principal problème restant le **manque de souplesse**.

Différence entre la théorie et la pratique. Les phases durant lesquelles le niveau de détail est accru, en particulier celles de spécification détaillées et d'implémentation, permettent parfois de se rendre compte que les analyses issues des phases précédentes sont incomplètes ou carrément irréalisables en l'état.

Domaines d'application

Ses inconvénients persistants en dépit de son vaste taux d'utilisation font que le cycle en V est davantage un **idéal** vers lequel certains aimeraient tendre mais est rarement appliqué tel quel.

Il est donc en général utilisé pour de grands projets industriels avec plus ou moins de bonheur.

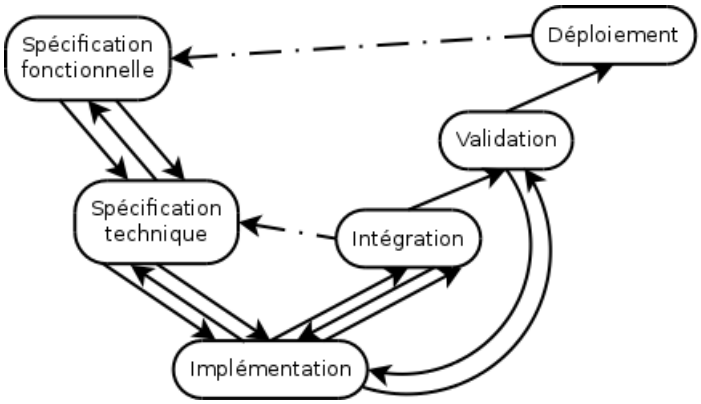


Figure 03:Cycle en V, trop fréquemment

En spirale (de Boehm)

Inspiré par la Roue de Deming, ce modèle itératif met l'accent sur la **gestion des risques**. Il reprend les étapes du cycle en V, mais prévoit la création de versions successives au cours de **cycles** de développement.

Chaque cycle peut être découpé en 4 étapes distinctes représentées par l'acronyme PDAC (*Plan-Do-Check-Act*) :

- Planifier
  - Analyse des besoins
  - Détermination des objectifs
  - **Analyse des risques**
  - Analyse des alternatives
- Développer
  - Spécification
  - Conception
  - Implémentation
  - Tests unitaires
- Contrôler
  - Intégration
  - Validation
- Ajuster
  - Livraison
  - Définition du prochain cycle

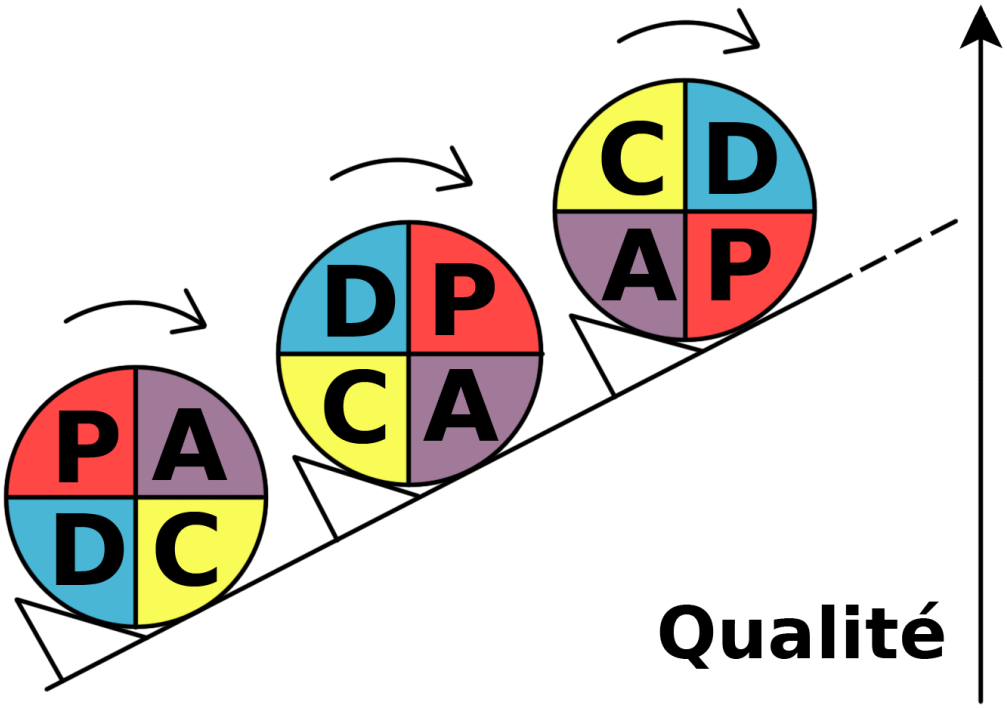


Figure 04.1:Roue de Deming

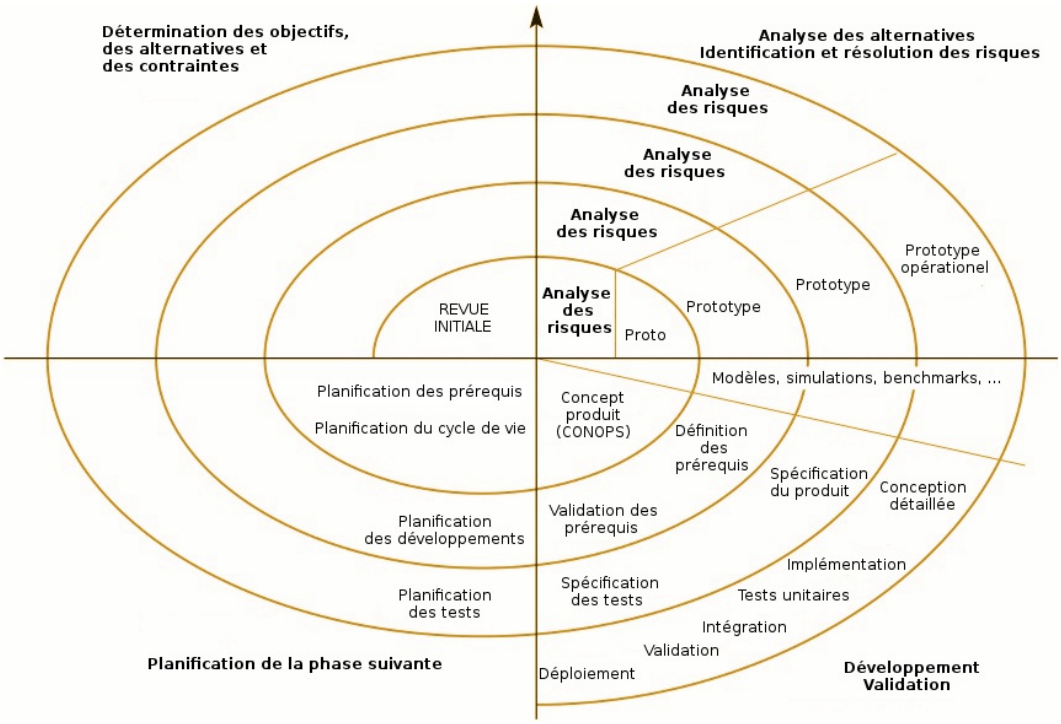


Figure 04.2:Exemple de développement en spirale

Chaque cycle ne se base pas forcément sur les artefacts livrés par le cycle précédent : différents risques peuvent être addressés l'un après l'autre.

Comme dans tout modèle itératif, le nombre de cycles n'est pas déterminé à l'avance.

Avantages

Maîtrise des risques : on se concentre sur les aspects les plus incertains du développement.

Mise en avant des objectifs de qualité.

Intègre maintenance et développement.

Possibilité d'intervertir l'ordre de certains cycles indépendants

Compatible à de nombreuses approches et outils existants.

Inconvénients

Nécessite une vraie expertise sur l'évaluation des risques : la nature des risques peut être différente d'un cycle à l'autre.

Mettre en place ce modèle assez complexe nécessite une grande expérience.

Les premiers cycles de la spirale ne produisent en général pas de solution exploitable.

Verbosité inadaptée pour les petits projets ou des domaines suffisamment connus. Dans le pire des cas, la stricte application de ce modèle (en particulier l'évaluation des risques) engendre un coût plus élevé que la réalisation du projet elle-même.

Domaines d'application

Ce modèle est logiquement approprié aux projets où le périmètre et le risque sont importants.

Exploratoire

Le principe de ce modèle itératif est de perfectionner à plusieurs reprises (on parle d'itérations) la spécification du projet, ainsi que son développement et sa validation. L'objectif est de **collaborer** avec le client et de **raffiner** de plus en plus la solution.

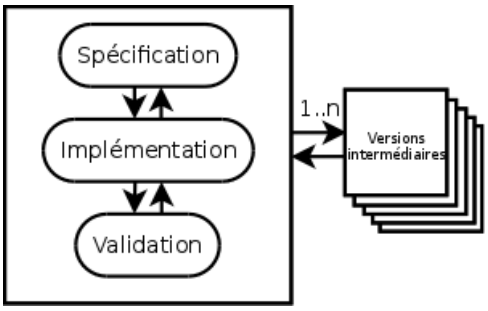


Figure 05:Modèle exploratoire

En général, c'est le délai accordé au projet qui décide du nombre d'itérations.

Avantages

Minimise le risque pour les nouvelles applications : il est possible d'explorer certaines spécificités du systèmes, de les évaluer afin d'opter pour la meilleure stratégie.

Le résultat est souvent pleinement compréhensible et satisfaisant pour le client.

Théoriquement livrable à chaque itération :

Favorise les tests et les validations intermédiaires.

Il est possibile d'arrêter le processus n'importe quand.

Inconvénients

Tentation d'abréger le processus et de se contenter d'une solution incomplète.

Peu structuré, donc impossible à appliquer tel quel à grand échelle.

Faible visibilité pour les intervenants extérieurs.

Domaines d'application

Le domaine d'application de ce modèle est donc les petits systèmes (ou parties d'un système) interactifs dont le résultat peut être visible rapidement. En particulier, les IHM.

Évolutif

Ce modèle itératif vise a réaliser une version provisoire de la solution aux besoins connus afin de pouvoir la mettre en exploitation au plus vite. De **nouvelles versions** seront ensuite déployées, chacune remplaçant la version précédente. Chaque version apportera de nouvelles fonctionnalités ou modifiera les fonctionnalités existantes.

Les versions intermédiaires sont réalisés avec tous les principes de qualité d'une version finale : ce

ne sont donc *pas* des prototypes jetables !

Modèle incrémental

La première version constitue un système partiel. Chaque nouvelle version ajoute une nouvelle fonctionalité complète.

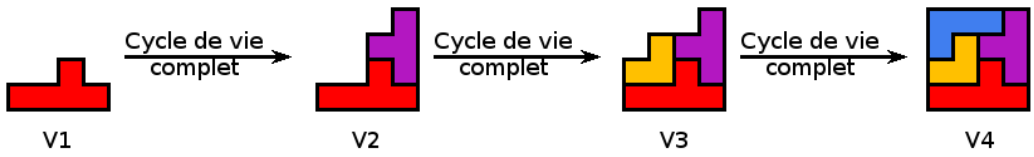


Figure 07.1:Incrémental, par prototypage vertical

Modèle itératif

Le système doit être dès le départ découpé en fonctionalités bien définies. La première version constitue une coquille complète du système. Chaque fonctionnalité qui n'est pas implémentée est remplacée par un **bouchon**. Chaque nouvelle version modifie ou améliore une fonctionalité.

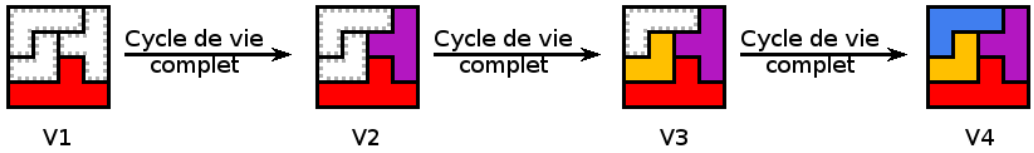


Figure 07.2:Évolutif, par prototypage horizontal

Avantages

- Augmente la **compétitivité** en réduisant le temps de mise sur le marché.
- Formation précoce des utilisateurs.
- Détection précoce des imprévus.

Inconvénients

- Le processus de deploiement de chaque version doit être maîtrisé.
- Risque d'aboutir à un parc hétérogène :
  - Assurer la tracabilité de chaque version.
  - Assurer la tracabilité de chacun de leurs composants.
- La manière d'obtenir un retour des utilisateurs doit être maîtrisée.
- Risque de remise en cause du noyau assurant les fonctionnalités de base.

Domaines d'application

Ce modèle est approprié à tout domaine **fortement concurrentiel**, et où les utilisateurs sont disposés à utiliser un produit incomplet.

Agile

En génie logiciel, le terme d'agilité couvre plusieurs modèles de développement distincts, qui offrent tous au moins les trois caractéristiques suivantes :

- **itératifs** : les différents intervenants travaillent par périodes de durée variable (quelques semaines),
- **incrémentaux** : chaque itération travaille sur le résultat des précédentes
- **flexibles** : chaque itération n'est planifiée qu'à son début, et non pas au lancement du projet

Les principes de l'agilité à été formalisées dans le *Manifeste Agile* (2001).

Différents implémentations de ces principes ont vu le jour : *RAD*, *Crystal Clear*, processessus

unifiés (modèle en "Y" ou *2TUP*, *RUP*, *AUP*, *XUP*), … Cependant, les implémentations les plus utilisées sont aujourd'hui *Extreme Programming* (*XP*) et *Scrum*.

Le modèle agile voit le produit comme la somme de ses **fonctionnalités**. Chaque itération (ou **sprint**) produit un certain nombre de fonctionnalités.

Le processus s'arrête quand les fonctionnalités implémentées satisfont le client. Le nombre de fonctionnalités final n'est pas connu à l'avance ; à l'inverse, la situation est examinée à la fin de chaque itération.

- Expression du besoin. Ce besoin peut varier à chaque itération.
- Déroulement de l'itération. Chaque itération doit se concentrer sur l'essentiel ; quelles fonctionnalités sont essentielles pour cette itération dépend du besoin exprimé.
  - Planification conjointe entre le client et l'équipe de développement
    - Décision de quelles fonctionnalités vont être développées, et desquelles vont être laissées de coté.
    - Traduction du besoin en langage technique.
  - Développement : réalisation de ce qui a été spécifié.
    - Implémentation
    - Tests
  - Inspection
    - Validation client : vérification que le résultat de l'itération (les *artefacts* produits) est conforme au besoin exprimé.
    - Déploiement : mise à disposition du client des artefacts qu'il a validés.
    - Éventuellement, évaluation : comprendre l'état actuel du projet
      - Analyse des difficultés rencontrées
      - Plan d'amélioration.

Avantages

- Pragmatisme : chaque itération se consacre sur l'essentiel. Ce processus évite de perdre du temps sur des tâches sans valeur ajoutée pour le client. Cela permet de s'approcher d'un mode de fonctionnement optimal.
- Réactivité offerte par des itérations courtes.
- Met l'accent sur la satisfaction du client.

Inconvénients

- Le coût en temps pour le client n'est pas à négliger.
- Moins facile à mettre en place qu'il n'y parait.
- Comment justifier des tâches nécessaires à l'organisation mais sans bénéfice immédiat pour le client ?
- Un nombre étonnant de freins d'ordre organisationnel ou personnel peuvent apparaître. Cette méthode de travail peut en effet être difficile d'approche pour certains (résistance au changement, difficulté à communiquer, etc).
- Difficile à adapter aux équipes trop grandes, souffrant d'un turnover trop important, ou composées de membres trop spécialisés dans leurs domaines respectifs.
- Difficile à adopter si le code du produit est mal maitrisé ou insuffisamment testé.

Domaines d'application

Les modèles implémentant ce processus nécessitent que le besoin client soit clairement exprimable. Il est adapté aux entreprises ouvertes d'esprit et où la communication inter- et intra-équipe est bonne.

Deux exemples de modèles agiles

L'*XP* et *Scrum* sont aujourd'hui les deux méthodes agiles les plus connues et utilisées en entreprise. Elles sont compatibles entre elles.

Planification

XP et *Scrum* font toutes les deux un usage intensif de « TODO lists » appelés**backlogs**.

Le *Product backlog* est la liste de toutes les activités à réaliser pour que le projet soit terminé. Les activités y sont triées par ordre de priorité.

Le *Sprint backlog* est la liste des activitées à réaliser dans le sprint en cours. Au début de chaque sprint, cette liste est peuplée avec des activités issus du product backlog au cours d'une réunion appelée le *planning poker* (ou *planning game*), qui réunit l'équipe de développement mais aussi son client. En général, ce sont les activités les plus prioritaires qui sont choisies. La durée prévue pour réaliser chaque activité placée dans le sprint backlog doit être estimée (chiffrée). Cette durée est représentée sous forme de « coût » abstrait de l'activité par rapport aux autres activités déjà réalisées lors des sprints précédents.
La somme des coûts des activités réalisées lors des sprints précédents donne la **vélocité** respective de l'équipe de développement. C'est en se basant sur ces vélocités que l'équipe de développement peut déterminer de manière réaliste combien d'activités elle peut s'attendre à pouvoir réaliser au cours du sprint qui débute.

La liste *« Doing »* regroupe les activités en cours de réalisation. Il est important de sortir ainsi les activités du sprint backlog afin que plusieurs personnes ne travaillent pas sur la même chose.

La liste *« Done »* regroupe les activités du sprint en cours qui sont terminées.

Extreme Programming (XP)

XP est autant un style qu'une discipline de développement. C'est aussi une voie d'**amélioration continue**. Comme son nom l'indique, cette méthode est quasiment uniquement axée sur développement logiciel. Il s'agit avant tout d'un assortiment de principes simples qui, combinés et poussés à l'extrême, permettent d'atteindre un excellent niveau de qualité logicielle.

Principes

Étant admis que parler un vocabulaire commun est le moyen le plus efficace de se comprendre, *XP* encourage la définition et l'usage des mêmes **métaphores** par les différents acteurs.

Étant admis que le meilleur moyen de gagner du temps est de faire les choses simplement, une **solution simple** doit toujours être privilégiée par rapport à une solution compliquée.

Étant admis que les besoins du clients sont changeants, des **itérations courtes** sont à privilégier.

Étant admis que les tests sont utiles, ils faut développer et exécuter des **tests systématiques**, en particulier en pratiquant le TDD.

Étant admis que l'intégration est une phase cruciale, il est indispensable de faire de l'**intégration continue**.

Étant admis que la revue de code est utile, celle-ci sera faite systématiquement, via la **programmation en binôme**.

Étant admis qu'une bonne architecture logicielle est indispensable, la conception sera améliorée systématiquement, via la pratique de **refactoring** systématique.

Test Driven Development (TDD)

Le *TDD* est une discipline de développement qui oblige à écrire systématiquement et de manière exhaustive le code testant la fonctionnalité avant même de développer la fonctionnalité en elle-même. Outre la volonté de ne pas « oublier » de tester ce qu'on implémente, cette discipline offre aussi l'avantage de déboucher sur une meilleure conception logicielle, qui découle du fait que, dès le départ, le code est envisagé du point de vue de l'appelant.

Voici le workflow typique d'un développeur travaillant en TDD :

1. Écrire un test
2. Éxécuter le test, et vérifier que celui-ci est en échec.
3. Écrire le code source nécessaire et suffisant pour que le test passe.
4. Vérifier que le test passe.
5. Refactorer le code source.

Refactoring

La discipline de refactoring consiste à remanier/réécrire le code afin de l'améliorer. Refactorer un code consiste en particulier à accorder l'attention nécessaire eux points suivants :

Est-ce que que quelqu'un lisant le code dispose au même endroit de toute l'information nécessaire à la compréhension du code ? Entre autres :

- Pour quelle raison ce code a t'il été écrit ?
- Quelles en sont les entrées et les sorties ?
- Quelles sont les principes algorithmiques entrant en jeu ?
- Les commentaires de code, si ils sont présents, sont-ils synchronisés avec le code source ?

Toute redondance est-elle supprimée ?

- Factorisation du code pour en éviter la duplication.
- Suppression du code mort.
- Suppression des commentaires inutiles.

Les algorithmes sont ils écrits de la manière la plus compréhensible possible ?

- Extraction de méthodes nommées et éventuellement commentées de manière descriptive.
- Formattage du code.

Le meilleur compromis est-il fait entre nombre minimal de classes et complexité propre à chaque classe ?

Scrum

Scrum est une méthode initialement empirique d'organisation et de suivi du projet. Elle a la particularité de laisser l'équipe de développement s'auto-organiser, ce qui la rend par exemple compatible avec l'*Extreme Programming*.

Ses valeurs sont celles de **transparence** (envers toutes les parties concernées, et par le biais d'un langage commun), d'**inspection** des livrables, et d'**adaptabilité**.

Rituels

Scrum comporte certaines réunions, ou rituels, pour promouvoir ces valeurs :

la **réunion de planification** (ou _planning poker) qui a lieu en début de sprint

la **mêlée quotidienne** (ou *standup meeting*), qui réunit quotidiennement tous les membres de l'équipe durant 15 minutes au maximum, et offre à chacun l'opportunité de confier rapidement aux autres :

- ce qu'il a fait depuis la denière mêlée
- ce qu'il prévoit de faire d'ici à la prochaine mêlée
- le cas échéant, les problèmes qu'il a rencontré ou qu'il prévoit que l'équipe risque de rencontrer à l'avenir

la **revue de sprint** est axée produit, et réunit à la fin de chaque sprint tous les intervenants, avec pour objectifs de :

- valider l'incrément apporté au produit par le sprint
- mettre à jour les différents backlogs

la **rétrospective** est propre à l'équipe de développement, et lui permet de revenir sur le ou les derniers sprints dans une optique d'**amélioration continue**.

Rôles

Scrum reconnait exactement trois rôles (ni plus, ni moins) pour les membres d'une équipe projet.

Le **product owner** est le propriétaire du produit, et le client de l'équipe de développement. C'est lui qui détient la **vision** de ce que doit être le produit. C'est donc aussi à lui qu'appartient le product backlog : il fixe la priorité entre les différentes activités et qui s'assure que leur spécification est comprise de tous. *Scrum* recommande que le product owner soit **toujours présent** sur le même site que l'équipe de développement, qui peut avoir besoin de son retour ou de ses explications à tout moment pendant le sprint.

Les membres de l'**équipe de développement** sont collectivement responsables de la livraison du sprint, que cela soit au niveau de la qualité comme des délais. Ce sont eux qui estiment le coût de chaque activité lors du *planning poker*.
Chacun des membres de l'équipe de développement est considéré comme **non-spécialisé** / pluridisciplinaire. Cela signifie que n'importe quel membre peut intervenir sur n'importe quelle

activité du sprint backlog.
L'équipe de développement fonctionne en auto-organisation : *Scrum* n'impose rien quant à la façon dont les développements se déroulent ; elle se contente de fournir un cadre pour que la qualité et les délais soient respectés.
L'équipe de développement est **mono-produit** et ne prend ses ordres que du product owner.
Scrum recommande une taille de 3 à 9 personnes composant l'équipe de développement.

Le **scrum master** est responsable de l'adhésion de tous à la méthode *Scrum* et de sa mise en œuvre. Il a un rôle de **facilitateur**, de **formateur** et de **coach**.
C'est aussi lui qui participe au « scrum de scrums ».

Planification

Le rôle principal d'un chef de projet est d'organiser l'usage des différentes ressources à sa disposition (charge de travail des équipes, …) en les découpant en différentes activités, ou tâches, dont l'enchainement garantit que le périmètre du projet soit respecté.

Une tâche est un ensemble d'actions qui, lorsqu'elles sont toutes entièrement réalisées, marquent un avancement significatif du projet. Le projet est lui-même terminé lorsque toutes les tâches qui le composent sont réalisées.

Le périmètre de chaque tâche doit être correctement défini :

si une tâche n'est pas assez détaillée, la visibilité ne sera pas bonne et la gestion de projet ne sera guère facilitée.

si une tâche est trop détaillée, le trop grand nombre de tâches obstruera à son tour la visibilité.

Le diagramme PERT/CPM et le diagramme de Gantt sont deux techniques largement utilisées pour planifier. Ces deux types de diagrammes permettent de coordonner des tâches plus ou moins séquentielles avec pour objectif d'obtenir une prévision réaliste des délais et des coûts.

PERT/CPM

PERT/CPM permet de mettre en évidence :

les dépendances entre tâches

la parallélisation des tâches

trois estimations (optimiste, pessimiste, réaliste) de la durée des tâches

la marge des tâches et du projet qu'elles composent

Caractéristiques d'une tâche

Chaque tâche `T` offre les caractéristiques suivantes :

Durée : Durée `D(T)` nécessaire à la réalisation de la tâche considérée.

Prédécesseurs : Autres tâches devant être entièrement réalisées avant que `T` ne puisse débuter.

Successeurs : Autres tâches ayant `T` comme prédecesseur.

DO : Date de Début au plus tôt `DO(T)` (*ES: Earliest Start Time*). `T` ne peut pas débuter avant cette date.

FO : Date de Fin au plus tôt `FO(T)` (*EF: Earliest Finish Time*). `T` ne peut pas être terminée avant cette date.

DA : Date de Début au plus tard `DA(T)` (*ES: Latest Start Time*). `T` doit commencer avant cette date.

FA : Date de Fin au plus tard `FA(T)` (*ES: Latest Finish Time*). `T` doit être terminée avant cette date.

Marge : La marge `M(T)` est le retard maximal admissible (*Slack*/*Float*).

DO	Durée	FO
	Nom de la tâche	
DA	Marge	FA

Figure 01:Tâche PERT/CPM

`FO(T) = DO(T) + D(T)` : une tâche ne peut finir avant qu'elle ait été entièrement réalisée.
`DA(T) = FA(T) - D(T)` : une tâche doit être entièrement réalisée avant de finir.
`DO(T) = max(FO(prédécesseurs(T)))` : une tâche ne peut débuter que lorsque ses prérequis sont finis.
`FA(T) = min(DA(successeurs(T)))` : les successeurs d'une tâche ne peuvent débuter que lorsque cette tâche est finie.
`M(T) = FA(T) - FO(T)` : la marge est égal au délai entre la date à laquelle elle *doit* terminer et la date à laquelle elle *peut* terminer.

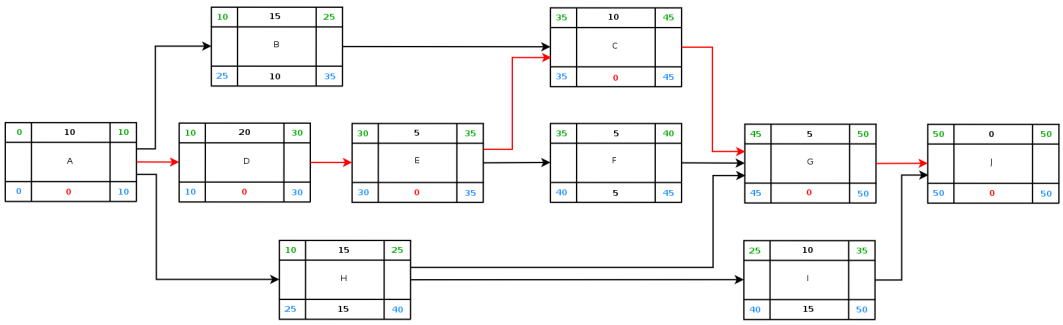


Figure 02:Diagramme PERT/CPM

Chemin critique

Une tâche dont la marge est égale à 0 est considérée comme **critique**. Cela signifie que le moindre retard dans la réalisation de cette tâche retardera d'autant le projet complet.

Le **chemin critique** est le chemin le plus long du diagramme de PERT. Toutes les tâches du chemin critique ont la même marge (éventuellement 0, mais pas forcément). Si une tâche du chemin critique accuse davantage de retard que sa marge, tout le projet sera retardé d'autant.

Jalons

Il peut être pertinent de créer dans un diagramme de PERT des **jalons**. Un jalon est typiquement une tâche "virtuelle" dont la durée est égale à 0. Elle marque (et permet de contrôler) un moment important de l'avancement d'un projet.

Topologie

Y a t'il exactement un tâche de début et une tâche de fin ?

Toutes les tâches ont-elles au moins un prédecesseur et un successeur ?

Tous les liens sont-ils représentatifs ?

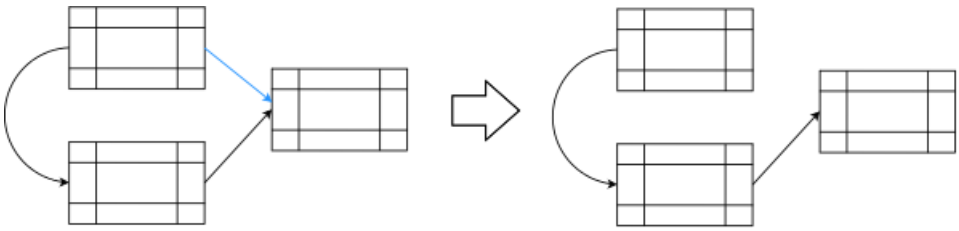


Figure 03.1:Transition redondante

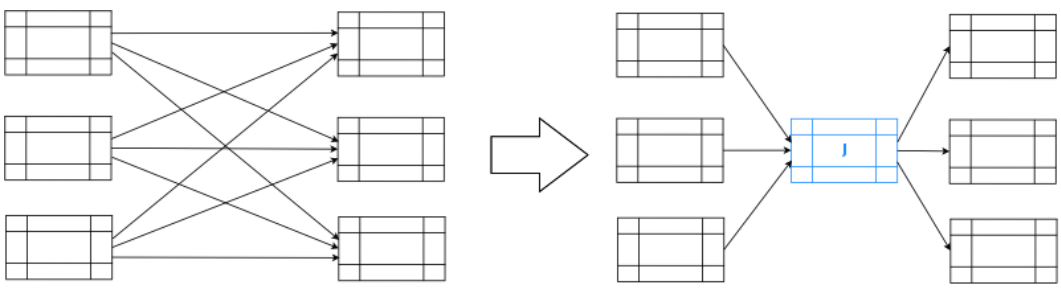


Figure 03.2:Création d'un jalon

Démarche pas à pas

1. **Identifier** *toutes* les tâches à réaliser, sans négliger certaines tâches "annexes" :

Le projet nécessite-t-il des technologies non maîtrisées en interne (besoins en formation, en procédure d'achat, …) ?

La charge est-elle trop grande pour les équipes actuelles (besoins en recrutement, …) ?

Tout est-il présent pour assurer la qualité du projet (besoin en documentation, en tests, …) ?

Une tâche particulièrement complexe nécessite-t-elle d'être gérée comme un sous-projet ?

2. **Estimer la durée** de chacune des tâches. Étant donné qu'il est impossible, lors de la phase de planification, de connaître la durée réelle de chaque tâche, on l'estime comme une variable, de manière probabiliste.

   Temps moyen estimé `TM` : temps de travail réaliste, nécessaire dans des conditions "normales"

   Temps optimiste `TO` : temps de travail dans des conditions "idéales" (aucune interruption ni difficulté imprévue, …)

   Temps pessimiste `TP` : temps de travail dans les "pires" conditions (déficit d'effectif, multiples imprévus, …)

   La durée est donc une durée "espérée", obtenue grâce à la formule suivante: `D = (TO + 4×TM + TP)÷6`

   C'est lors du calcul de la durée qu'on choisit (ou non) de s'accorder un délai de précaution. Après, il est trop tard ; en particulier, le calcul ultérieur de sa marge n'est pas là pour ça !

3. **Ordonnancer** les tâches

   organiser leur séquencement chronologique

   optimiser en envisageant les parallélismes possibles

   déterminer le chemin critique

4. **Planifier** les différentes tâches

   affectation des ressources humaines nécessaires

   affectation des ressources matérielles nécessaires

   affectation de davantage de (ou de meilleures) ressources sur les tâches critiques

Gantt

Comparé à un diagramme de PERT/CPM, un diagramme de Gantt est peut-être plus simple à appréhender. Il a comme avantage de permettre de clairement visualiser les dates de début et de fin, le pourcentage de complétion de chaque tâche, ainsi que leur chevauchement éventuel. En revanche, il met moins l'accent sur les dépendances entre tâches et ne prend pas en compte la marge ou le risque.

Dans un diagramme de Gantt, un tâche n'a qu'une date de début ou une date de fin, ainsi qu'une durée. Les tâches s'enchainent ensuite de manière logique, en utilisant l'un des quatre types de liaison :

de Fin à Début (FD) : une tâche ne peut pas commencer avant que la précédente ne soit terminée

de Début à Début (DD) : une tâche ne peut pas commencer avant que la précédente ne commence

de Fin à Fin (FF) : une tâche ne peut pas se terminer avant que la précédente ne soit terminée

de Début à Fin (DF) : une tâche ne peut pas se terminer avant que la précédente ne commence

Une tâche peut évidemment avoir plusieurs autres tâches la précédant. De même, une tâche peut précéder plusieurs autres tâches.

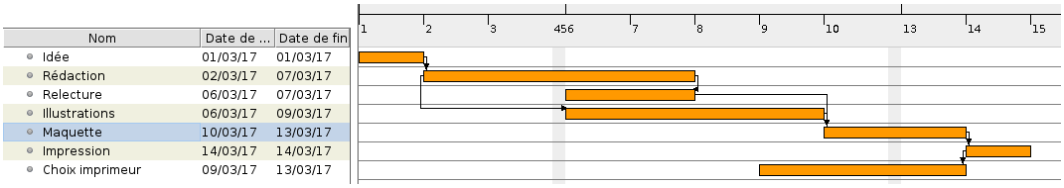


Figure 04:Exemple de diagramme de Gantt : Rédaction d'une documentation

Modélisation UML

Diagrammes statiques

Les diagrammes de cas d'utilisation ainsi que les diagrammes de classes et les diagrammes d'objets sont des **diagrammes statiques**.

Ces diagrammes modélisent des éléments de manière statique. Ils ne s'occupent pas des aspects dynamiques (déroulement du temps, enchaînement d'opérations, …).

Quand ce qui est à modéliser est complexe, il vaut mieux dresser plusieurs diagrammes complémentaires, chacun se focalisant sur un aspect particulier. Par exemple: la structure hiérarchique des différents éléments, leur regroupement logique, l'état du système à un instant T, la manière dont les éléments collaborent entre eux, et ainsi de suite.

Souvent, même se focaliser sur un aspect particulier est trop complexe pour résumer cet aspect en un seul diagramme.

Cas d'utilisation (use cases)

Un **diagramme de cas d'utilisation** est utilisé pour comprendre et structurer des besoins. Il ne vise pas l'exhaustivité (forte abstraction, faible niveau de détail), mais la clarté et l'organisation des besoins. Par exemple, il permet de trier les besoins des utilisateurs finaux entre besoins avérés et besoins qui sont en fait couverts par d'autres besoins ou besoins qui n'en sont pas vraiment. Définir les besoins réels permet de fixer le périmètre du projet en identifiant les fonctionnalités principales du système en cours de conception. Ce sont ces objectifs qu'il faut garder en tête tout au long du projet.

Le diagramme de cas d'utilisation classe les différents acteurs et la façon dont ils utilisent (interagissent avec) le système.

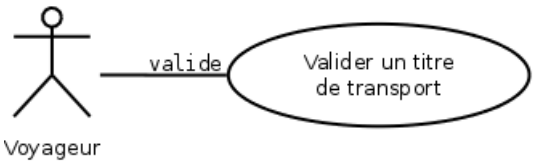


Figure 01:Cas d'utilisation simple

Éléments UML

Relation

Lien unissant deux éléments ou plus. La relation peut être de différentes natures : utilisation, extension, dépendance, agrégation, etc. On peut détailler la nature d'une relation en utilisant un **stéréotype** (`<< stereotype >>`).

En ce qui concerne les diagrammes de cas d'utilisation, UML reconnait trois types de relation : inclusion, extension et généralisation.

Une relation d'**inclusion** représente la décomposition d'une tâche en plusieurs sous-tâches.

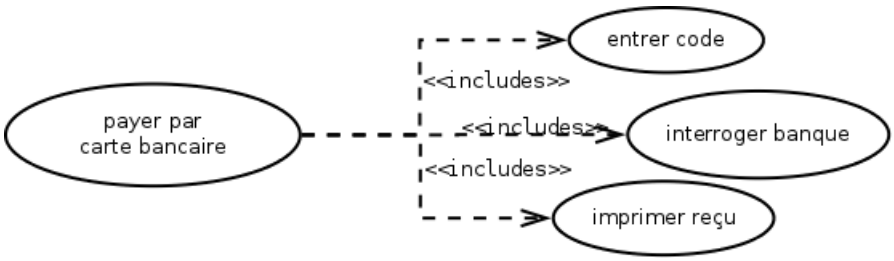


Figure 02.1:Relation d'inclusion

Une relation d'**extension** représente le fait qu'une tâche puisse en appeler une autre.

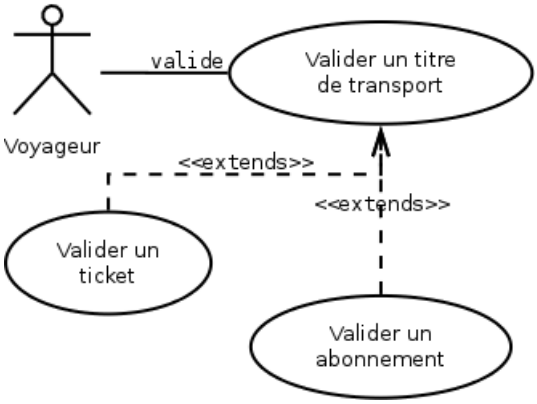


Figure 02.2:Relation d'extension

Une relation de **généralisation** représente le fait qu'une tâche puisse être substituée par une autre, plus spécialisée.

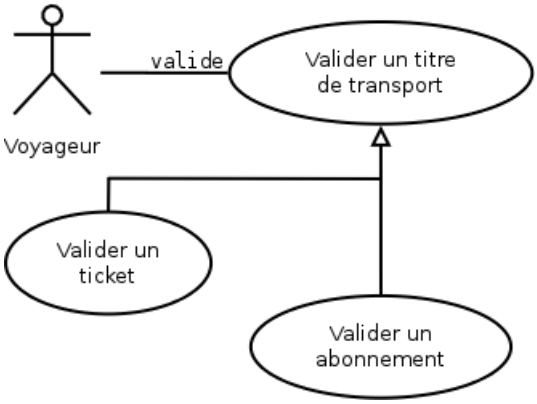


Figure 02.3:Relation de généralisation

Acteur

Entité extérieure au système, mais qui agit sur lui. Ce peut être une personne physique, un autre système déjà en place, etc. En réponse à chaque action d'un acteur, le système accomplit une action en retour qui correspond au besoin de l'acteur.

Une même personne physique peut exercer plusieurs rôles différents dans un même projet. C'est pourquoi les acteurs sont surtout définis par leur rôle dans un diagramme UML.

Cas d'utilisation

Action du système en réponse au besoin exprimé par un acteur. Plusieurs cas d'utilisation peuvent être relié entre eux (relations de dépendance, de hiérarchie). Pour faciliter la compréhension et la réutilisation, les cas d'utilisation peuvent être organisés en **packages**.

Package (paquet)

Élément d'organisation qui rassemble d'autres éléments en un groupe **logique**. Les éléments qu'il contient sont accessibles via une **interface** d'accès. Il sert aussi d'**espace de noms**.

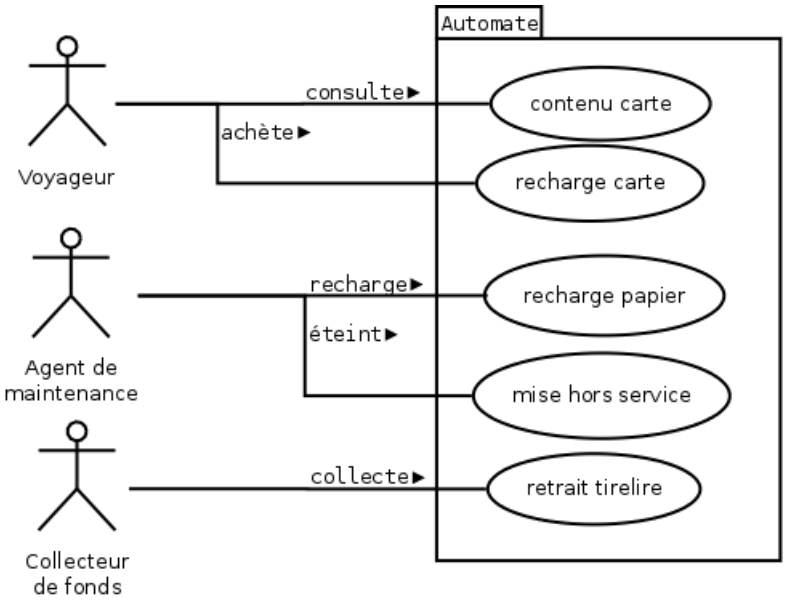


Figure 04:Regroupement en package

Un package peut dépendre d'un ou plusieurs autres packages.

Classes et Objets

UML permet de modéliser des **diagrammes de classe** ainsi que des **diagrammes d'objets**.

> Un petit rappel de POO ?
>
> Une classe, c'est un nom, des attributs et des méthodes.
>
> Un objet, c'est une instance de classe.

Les notations sont les suivantes.

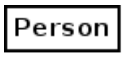


Figure 05.1:Classe `Person`.

Figure 05.2:Classe abstraite `Œuvre`.

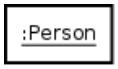


Figure 05.3:Instance anonyme de la classe `Person`.

Figure 05.4:Instance nommée de la classe `Person`.

Figure 05.5:Instance nommée d'une classe anonyme.

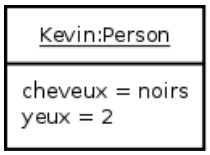


Figure 05.6:Instance nommée de la classe `Person`. Les valeurs de certains attributs *pour cette instance* sont spécifiés.

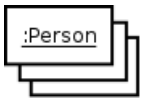


Figure 05.7:Collection d'instances anonymes de la classe `Person`.

Documentation d'une classe

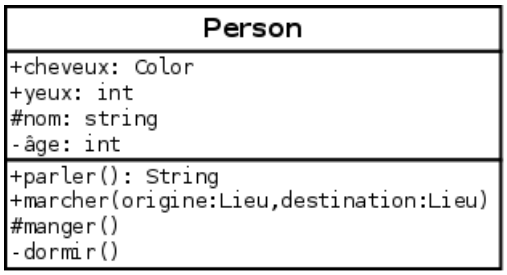


Figure 05.8:Documentation plus complète de la classe `Person`.

- Les attributs dont la visibilité est **publique** sont précédés d'un `+`
- Les attributs dont la visibilité est **protégée** sont précédés d'un `#`
- Les attributs dont la visibilité est **privée** sont précédés d'un `-`

> Ce n'est pas parce que la représentation d'une classe au sein d'un diagramme donné ne contient pas certains attributs ou méthodes qu'elle ne les possède pas.
>
> De même, la visibilité de certains attributs peut être omise.
>
> La représentation d'un élément UML est une **abstraction** qui contient plus ou moins de détails suivant ce que nécessite le diagramme qui la contient pour être compréhensible. Le but d'un diagramme donné est d'être **clair**. L'exhaustivité, quand à elle, est atteinte en combinant tous les diagrammes d'un document de conception.

Associations

Deux classes ou objets sont associés lorsqu'ils sont connectés d'une manière ou d'une autre.

Le sens de lecture privilégié d'une association peut être précisé à coté de son libellé.

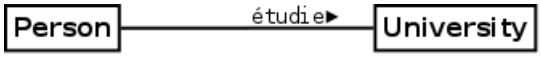


Figure 06.1:Sens de lecture

Plus de deux éléments peuvent évidemment être associés.

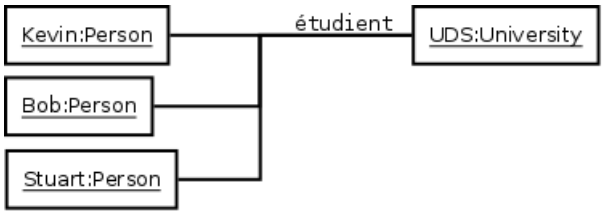


Figure 06.2:Association multiple

Rôle

Préciser le **rôle** de chaque élément dans une association est parfois indispensable quand la relation va dans les deux sens.

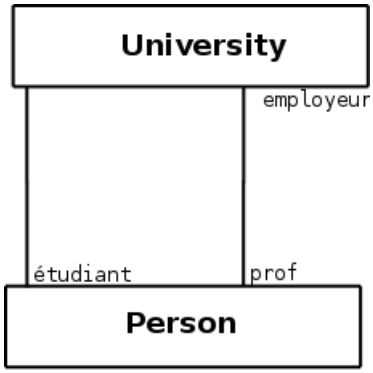


Figure 06.3:Rôles

Cardinalité

La **cardinalité** est le nombre d'objets qui participent à une association.

- `1` Exactement un, ni plus ni moins
- `*` Un nombre indéfini
- `1..n` Entre un (inclus) et `n` (inclus)
- `n..m` Entre `n` (inclus) et `m` (inclus)

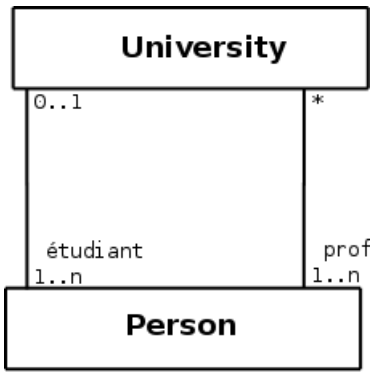


Figure 06.4:Cardinalités

> Attention, il n'est utile de modéliser que ce qui est requis, pas la vie réelle ! Évidemment, dans la réalité, une personne peut évidemment être inscrite à plusieurs universités, mais le système que ce diagramme modélise ne semble pas avoir besoin de ce niveau de détail …

Contraintes

Les **contraintes** permettent de préciser la portée d'un autre élément. Par exemple, un contrainte peut préciser un rôle ou restreindre un nombre d'instances.

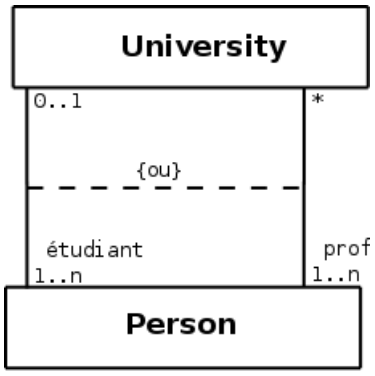


Figure 06.5:Contraintes

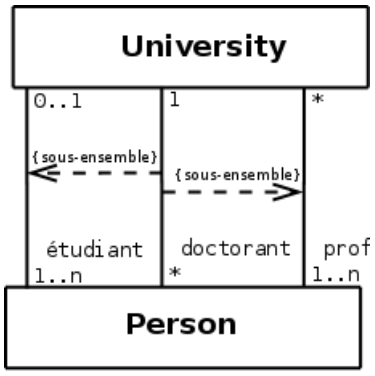


Figure 06.6:Contraintes

Relation à sens unique

Par défaut, une association est navigable dans les deux sens. Mais parfois, il est nécessaire d'indiquer qu'un élément n'en "connait" pas un autre.

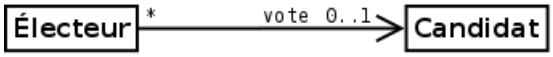


Figure 06.7:Relation à sens unique

Relation interdite

Parfois, il est utile d'indiquer qu'une association entre deux éléments est à proscrire.

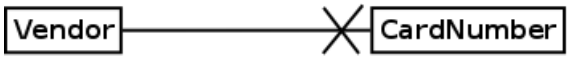


Figure 06.8:Relation interdite

Relation n-aire et classe d'association

Une **classe d'association** permet de réaliser la relation entre les éléments.

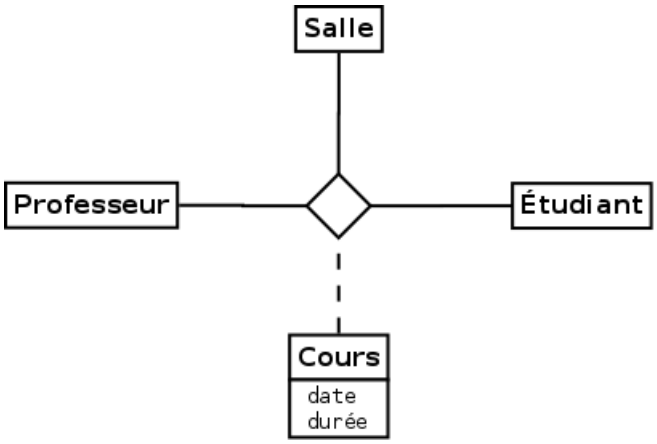


Figure 06.8:Relation n-aire avec classe d'association

> De telles associations peuvent être difficiles à déchiffrer. Il peut être préférable de limiter leur utilisation, et de modéliser d'autres relations.

Interfaces

TODO

Héritage

Sert à modéliser la hierarchie qui relie les différentes classes. Cette modélisation peut être :

ascendante : partir des objets les plus concrets, et **généraliser** progressivement en factorisant leur propriétés (attributs et méthodes) dans des super classes.

descendante : partir des objets génériques, et les **spécialiser** en étendant leurs propriétés dans des classes plus spécifiques.

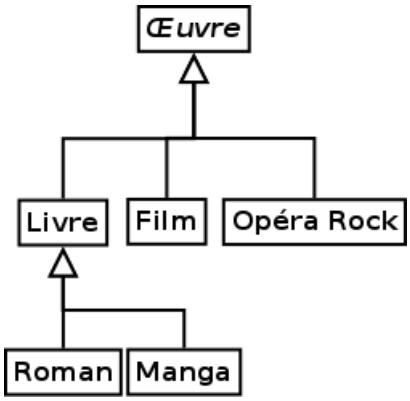


Figure 07:Héritage

> La manière de hiérarchiser des classes est souvent subjective. Cependant, les principes de base les plus connus sont représentés par l'acronyme **SOLID** :
>
> **S**: Responsabilité unique (Single responsibility)
> Une classe doit avoir une et une seule responsabilité, qu'elle doit remplir complètement.
>
> **O**: Ouverture/fermeture (Open/closed)
> Une classe doit être ouverte à l'extension (c'est à dire : facilement spécialisable), mais fermée à la modification (son comportement ne doit pas pouvoir être altéré).
>
> **L**: substitution de Liskov (Liskov substitution)
> Une instance de type T doit pouvoir être remplacée par une instance de type G, tel que G sous-type de T, sans que cela ne modifie la cohérence du programme.
>
> **I**: ségrégation des interfaces (Interface segregation)

> Il vaut mieux créer plusieurs interfaces, chacune adaptée à un client, plutôt qu'une seule interface générale.
>
> **D**: inversion des Dépendances (Dependency inversion)
> Les abstractions ne doivent pas dépendre des détails d'implémentation.
>
> Le "God object" est un contre-exemple (anti-pattern).

Agrégation

Tandis que l'héritage est une relation de type "Être" ("Is a"), l'agrégation est une relation de type "Avoir" ("Has a"). C'est une relation non symétrique. Elle exprime une relation de **subordination**.

Une agrégation peut vouloir dire qu'un élément *fait partie* d'un autre (l'autre est un "agrégat" de l'un). Elle peut aussi signifier qu'un changement d'état ou une action sur l'un a des conséquences sur l'autre.

Un élément peut être agrégé dans plusieurs autres.

Le cycle de vie d'un agrégat peut être différent de celui des éléments qu'il agrège. En d'autre termes, une instance d'élément agrégé peut exister sans agrégat, et inversement.

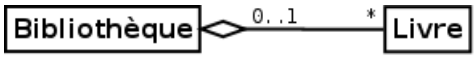


Figure 08.1:Agrégation

Composition

La composition est une agrégation forte.

Un composant ne peut être lié qu'à un seul composé.

Les cycle de vie des éléments agrégés (les **composants**) et de l'agrégat (le **composite**) sont liés. En d'autre termes, si l'agrégat est détruit, ses composants le sont aussi (mais l'inverse n'est pas forcément vrai).

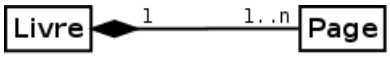


Figure 08.2:Composition

Syntaxe des messages

Chaque message peut être assortis d'attributs. Ces attributs permettent entre autres d'indiquer, pour un message donné :

- son numéro d'ordre
- la condition de son envoi
- ses arguments
- sa récurrence

```
[pre"/"] [["["guard"]"] [number] ["*"["||"]["["times"]"]] ":"] [var":="] label"("[params]")"
```

- `label` libellé du message
- `params` paramètres du message, le cas échéant
- `var` permet d'affecter le retour du message à une "variable". Cela permet par exemple d'utiliser ce retour comme paramètre (`params`) d'un autre message.
- `pre` Liste de numéros d'ordre (`number`) séparés par des virgules. Le message n'est envoyé que lorsque tous les messages précédents l'ont été. Cet attribut permet la **synchronisation** de différents messages.
- `guard` Expression booléenne. C'est la condition d'envoi du message, appelée **garde** (*guard*) en UML. Le message n'est envoyé que si cette condition est vérifiée. La condition elle-même peut être exprimée en langage naturelle, par une expression mathématique, etc.
- `number` Numéro d'**ordre** du message dans la séquence de message modélisée par le diagramme. Ce numéro peut être un simple entier, ou suivre un indiçage plus complexe.
- `times` permet d'envoyer autant de fois un message. Par défaut, l'envoi est séquentiel. Combiné avec `||`, les envois se font en parallèle. Le nombre `times` peut être remplacé par

`*` pour figurer un nombre indéfini d'envois. `*||` figure un nombre indéfini d'envoi parallèles.

Exemples

```
1: hi()
2: lmao()
3: kthxbye()
```

Une séquence de trois messages numérotés `1`, `2`, `3`.

```
1: humeur := comment_ca_va()
[humeur = "bien"] 2.a: content_pour_toi()
[humeur = "pas top"] 2.b: compatir()
```

On se sert du résultat du premier message pour conditionner l'envoi du message suivant. Notez que les numéros d'ordre ne sont ni forcément des entiers, ni n'ont tous exactement le même format.

```
4,5.1 / [3]||[i := 1..3]: message()
```

Le message `message` est envoyé 3 fois, en parallèle. Ces envois parallèles n'ont lieu qu'après que les message `4` et `5.1` ont été eux-mêmes envoyés.

```
[disque plein] 4.2.a * : deleteTmpFile()
[disque plein] 4.2.b   : reduceSwap(20%)
```

Ces messages sont envoyés simultanément si la condition `disque plein` est vérifiée. `reduceSwap` est envoyé une seule fois ; `deleteTmpFile` peut être envoyé plusieurs fois.

Types de message

Un message simple  ne spécifie aucune caractéristique particulière utile à la compréhension du diagramme. Attention, ça n'est pas pour ça qu'il n'en a pas !

Un message synchrone bloque son expéditeur jusqu'à réception de la réponse de son destinataire.

Un message avec timeout  bloque son expéditeur pendant un certain temps. La durée de blocage précise peut être spécifiée par une contrainte. L'expéditeur est bloqué, soit :

- jusqu'à réception de la réponse du destinataire
- jusqu'à la fin du timeout

Un message asynchrone n'interrompt pas le flot d'exécution de l'émetteur. L'émetteur n'a à priori aucune confirmation de la prise en compte du message par son destinataire. C'est pour cela que le retour d'un message asynchrone devrait toujours être modélisé explicitement.

Un message dérobant  est un message qui ne déclenche une action de la part de son destinataire que si celui-ci s'est préalablement mis en attente de ce message.

Un élément peut tout à fait s'envoyer un message à lui même. Cela représente une activité interne à l'élément, comme une période de calcul par exemple.

Diagramme de collaboration

Ce diagramme permet de modéliser les interactions entre objets, dans un contexte donné. Le contexte des interactions est donné par l'état des objets qui interagissent.

Ce diagramme est aussi appelé diagramme de communication.

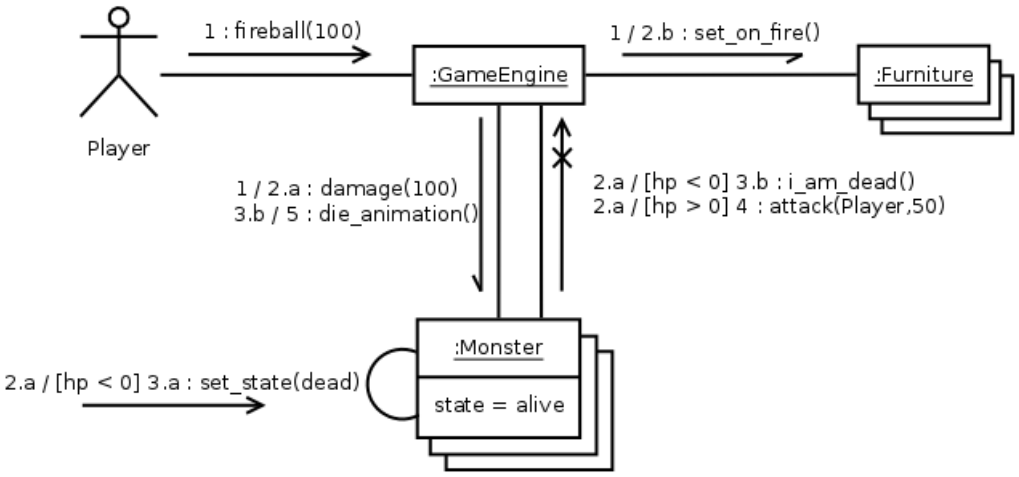


Figure 09:Exemple de diagramme de collaboration

Diagramme de séquence

Ce diagramme permet de modéliser les interactions entre objets au cours du temps. Il met l'accent sur la chronologie des envois de message.

Sur un diagramme de séquence, le temps s'écoule verticalement. Les événements qui surviennent en premier sont en haut du diagramme, et ceux plus tardifs se rencontrent chronologiquement, au fur et à mesure qu'on progresse vers le bas.

L'ordre horizontal des éléments (acteurs, objets, …) n'a par contre aucune importance.

Un diagramme de séquence peut illustrer un cas d'utilisation de manière dynamique.

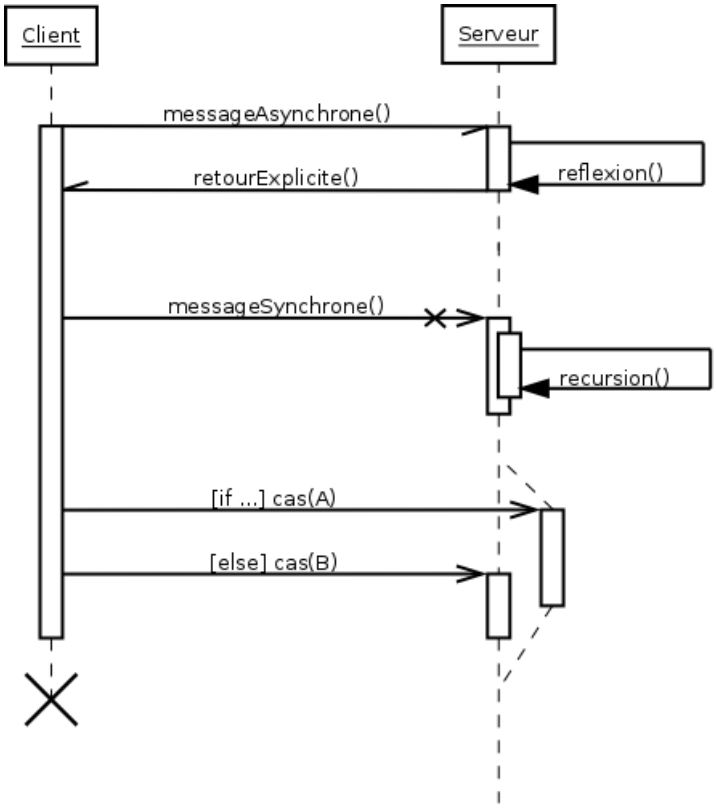


Figure 10:Diagramme de séquence

Période d'activité

Un diagramme de séquence permet de représenter une période d'activité d'un objet à l'aide d'une zone rectangulaire représentant la période de la vie de l'objet durant laquelle cet objet est actif (au sens du diagramme considéré).

Cette zone rectangulaire permet aussi de représenter une récursion.

Conditions et itérations

Pour représenter une exécution conditionnelle sur un diagramme de séquence, il est possible de

dédoubler la ligne de vie de l'objet concerné. Cela n'est cependant pas nécessaire : on peut en effet exprimer les conditions dans le descriptif d'un message.

Il est aussi possible de représenter une itération.

Destruction d'un objet

Il est possible de représenter la fin du cycle de vie d'un objet grâce à une croix en bas de sa ligne de vie.

Diagramme d'activité

Un diagramme d'activité permet de relier différentes activités entre elles. Passer d'une activité à une autre se fait via une transition. Une transition reliant deux activités est utilisée automatiquement dès que la première activité se termine.

Figure 12.1:Transition d'un diagramme d'activité

Le début d'un diagramme d'activité est matérialisé par un rond noir plein. Il est indispensable.

La fin d'un diagramme d'activité est matérialisée par un rond noir entouré d'un cercle. Elle est facultative, car un diagramme d'activité est terminé quand il n'y a plus d'activité en cours.

Pour organiser et faciliter la lecture d'un diagramme d'activité, il est possible de matérialiser des "couloirs d'activité".

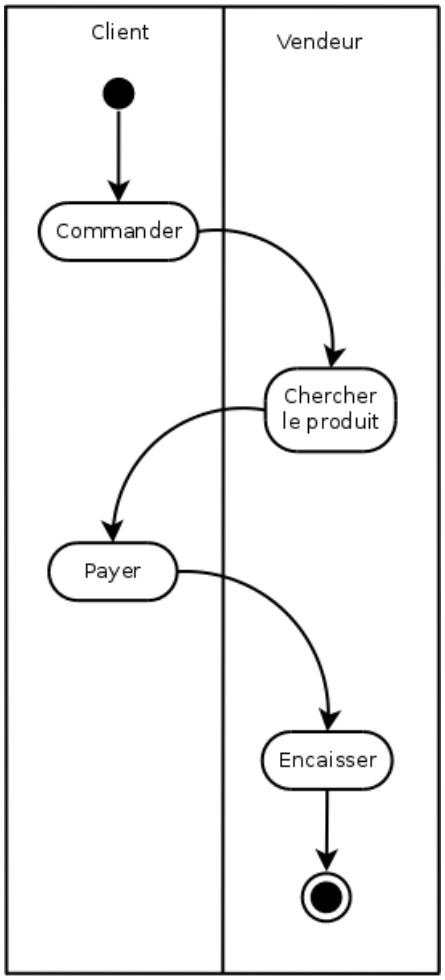


Figure 12.2:Couloirs d'activité

Une transition conditionnelle est représentée en utilisant un ou plusieurs losanges ainsi que des gardes.

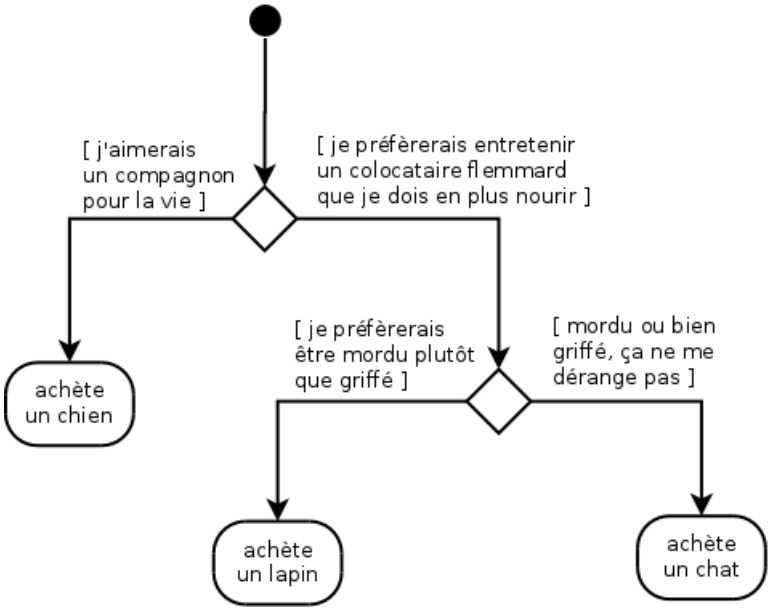


Figure 12.3:Transitions conditionnelles

Il est possible de paralléliser plusieurs activités grâce à des barres de synchronisation. Les barres de synchronisation respectent les règles suivantes :

- une barre n'est franchie que lorsque toutes les activités qui y arrivent sont terminées
- les activités partant d'une barre démarrent toutes en même temps

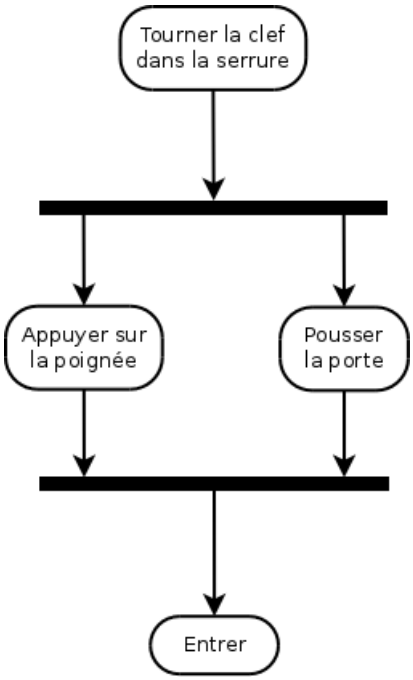


Figure 12.4:Parallélisation

Tests

Tester un produit a pour but d'en garantir sa qualité. Un test logiciel fournit une vue objective et indépendante de l'état d'implémentation de la fonctionalité correspondante. Il permet entre autres d'évaluer que le produit :

- réagit correctement face à toutes sortes d'interactions qu'il est possible d'avoir avec lui
- accomplit ses fonctions de la manière désirée par le client : non seulement chaque fonctionnalité doit déboucher sur la situation attendue, mais elle doit le faire dans le temps et l'environnement spécifiés

Il semble évidemment impossible de décrire le moindre cas de test sans disposer de la spécification correspondante. Spécifications et tests sont donc entre autres intrinsèquement liés dans la plupart des modèles de développement, comme le cycle en V.

Analyse statique

L'**analyse statique** du code (*sans* l'exécuter) permet d'obtenir une idée du comportement qu'il aura lors de son exécution. Certaine méthodes formelles d'analyse statique de code existent (par exemple, la logique de Hoare) ; mais aucune ne permet de dire à coup sûr si le code produira des erreurs lors de son exécution. Cependant, appliquer un certain nombre de principes lors du codage permet de réduire le *risque d'erreur* lors de l'exécution du code.

Les principes à appliquer peuvent parfois être empiriques. Ils dépendent en général du langage de programmation utilisé (toujours initialiser une variable avant de l'utiliser, limiter le nombre de structures de contrôle, …).

L'analyse statique du code est aujourd'hui souvent intégrée dans les outils de développement de base des langages de programmation que sont les compilateurs et les IDE. Une solution comme SonarQube est un autre exemple d'outil d'analyse statique.

Revue de code

La revue de code est une méthode empirique d'analyse statique qui consiste à faire lire le code source d'un développeur par une autre développeur. Le but est d'obtenir un point de vue additionel sur la conception et l'implémentation, et ainsi d'augmenter la qualité du code.

Niveaux de granularité

Un **test unitaire** teste individuellement une unité de code ; il s'agit en général d'une fonction ou d'un objet. On s'intéresse à la *plus petite partie testable* d'un logiciel.

Un **test d'intégration** combine différents modules logiciels et les teste comme un groupe fonctionnel.

Un **test système** est conduit sur un système *complet* et *intégré*. Il évalue la manière dont les différents groupes fonctionnels communiquent.

Un **test de validation** vérifie que le système correspond au besoin exprimé.

Puisque pour tester le logiciel à un niveau toujours plus élevé il s'agit de combiner/intégrer ses différents composants les uns avec les autres, il est légitime de se demander dans quel ordre intégrer ces composants.

Il existe pour répondre à cette qustion différentes approches.

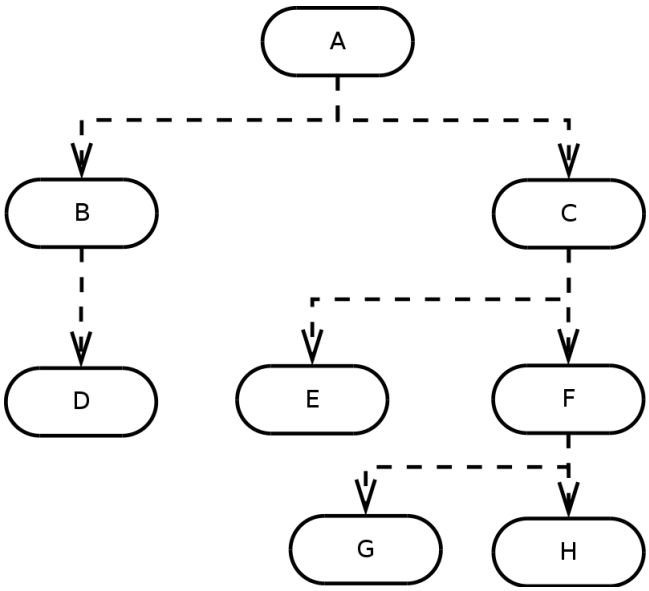


Figure 01:Exemple de dépendances

Approche « bottom-up »

Avec l'approche « **bottom-up** », on intègre les composants les uns avec les autres en commencant par ceux qui ont le moins de dépendances.

1. D'abord l'ensemble des composants sans aucune dépendance
2. Ensuite, l'ensemble des modules qui dépendent d'un ou plusieurs composants de l'ensemble précédent

Et ainsi de suite, jusqu'à intégrer le (ou les) composant(s) du plus haut niveau.

Dans la figure précédente, on intègrerait avec l'approche « bottom-up » d'abord les composants *D*, *E*, *G* et *H*, puis seulement *F* et *B*, puis ensuite *C*, et enfin *A*.

Avantages

Étant donné qu'on ne commence les tests d'intégration d'un composant particulier que lorsque les tests d'intégration de *toutes* ses dépendances ont réussi, cela signifie qu'une intégration échouée d'un composant indique que l'erreur se trouve dans ce composant particulier, et pas dans une de ses dépendances.
Cette approche offre donc une bonne **localisation des erreurs**.

Cette approche étant intuitive et correspondant bien souvent à la réalité de la phase de développement (où les dépendances d'un composant sont développées avant le composant lui-même), les tests d'intégration d'un composant peuvent être réalisés **juste après son développement**.

Pour la même raison, cette approche tire parti de la **réutilisabilité** des composants : chaque composant est testé avec ses dépendances réelles, elles-mêmes déjà développées et intégrées.

Inconvénients

Les composants de plus haut niveau, qui sont bien souvent ceux qui remplissent directement le besoin métier, et donc les seuls qui intéressent vraiment le client, sont testés **en dernier**. La qualité du système n'est donc garantie qu'à la toute fin.

De même, cette approche est tardive à valider la conception du logiciel. En effet, plus tardive est le test d'intégration d'un composant, plus tardif est le fait de vérifier que celui-ci fonctionne comme le prévoit le schéma d'architecture.

Enfin, cette approche nécessite évidemment la présence de modules **sans dépendances**.

Approche « top-down »

Avec l'approche « **top-down** », on intègre les composants les uns avec les autres en commencant par celui ou ceux de plus haut niveau.

1. D'abord le composant de plus haut niveau
2. Puis l'ensemble des composants dont le précédent dépend directement
3. Puis l'ensemble des dépendances directes de l'ensemble des composants précédent

Et ainsi de suite, jusqu'à intégrer le (ou les) composant(s) qui n'ont aucune dépendance.

Les dépendances d'un composant particulier sont, avec cette approche, remplacés par des bouchons (voir la description du modèle de développement évolutif - itératif).

Dans la figure précédente, on intègrerait avec l'approche « bottom-up » d'abord le composant *A*, puis *B* et *C*, puis seulement *D*, *E* et *F*, puis enfin *G* et *H*.

Avantages

Puisqu'on peut vérifier la qualité des composants les plus importants (fonctionnellement parlant) dès le début, cette approche permet un **prototypage rapide** de la solution.

La **conception est mise à l'épreuve**, elle aussi, plus rapidement : les besoins des composants de plus haut niveau sont implémentés et éprouvés au fur et à mesure. En cas d'inadéquation entre la réalité et la conception d'origine, cette dernière peut être améliorée sans remettre en cause trop importante de l'existant. Elle offre donc une **flexibilité d'implémentation** appréciable.

De par le recours systématique au bouchonnage, chaque composant est testé **en isolation**, ce qui offre une bonne localisation des erreurs.

Inconvénients

Le recours systématique aux bouchons peut être vu comme imposant une **charge de développement et de maintenance** un peu plus importante.

Le risque est de **négliger les composants de plus bas niveau** :

> Il est nécessaire de s'assurer que ceux-ci soient entièrement développés et testés, même si ce sont les derniers à l'être. En effet il faut garder à l'esprit que même si les tests des modules de plus haut niveau sont réussis, rien ne garantit qu'il fonctionneront correctement

lorsque les bouchons seront remplacés par leurs dépendances réelles.

Il est parfois facile d'oublier de prendre en compte la réutilisabilité des composants.

Autres approches

L'approche en « **sandwich** » tente de maximiser les avantages des deux approches précédentes en intégrant en même temps et indépendamment les composants de haut et de bas niveau. En contrepartie, le niveau médian est parfois négligé.

L'approche « **big bang** » intègre tous les composants en même temps. Pour des raisons évidentes de complexité, cette approche est difficile à adopter pour les gros projets.

Niveaux d'accessibilité

Tester un logiciel en mode « **boîte noire** » consiste à en tester le comportement sans avoir aucune connaissance concernant son fonctionnement interne. Un test « boîte noire » connaît uniquement *ce que* doit faire le logiciel, et pas *comment* il le fait.
En mode boite noire, le testeur n'a à priori pas besoin de savoir programmer. Il peut donc apporter un regard différent, mais peut potentiellement fournir des efforts inutiles.

Un test en mode « **boîte blanche** » nécessite de connaître le fonctionnement interne d'un logiciel ainsi que les structures de données mises en jeu. Un tel test accorde autant d'intérêt aux mécanismes internes d'un programme qu'aux résultats qu'il produit.
L'analyse statique fonctionne évidemment toujours en mode boîte blanche.

Tests fonctionnels vs tests non-fonctionnels

Un test fonctionnel a pour objectif de vérifier la bonne implémentation d'un cas d'utilisation. Il est donc fortement lié à un **besoin**.

Un test non fonctionnel a pour objectif de vérifier le respect d'une **contrainte**. Cette contrainte est la plupart du temps technique.

Types de tests

On peut citer de nombreux types de tests différents. Même si elle a vocation à mettre en évidence la diversité des tests possibles à réaliser, la liste suivante n'est pas exhaustive.

tests de **déploiement** / d'**installation** du logiciel dans son environnement de production

test de **compatibilité** du logiciel avec d'autres composants existants

smoke tests / tests de **démarrage** : vérifier que le logiciel démarre bien et accomplit ses fonctionnalités minimales

soak tests / **stress tests** / tests de **charge** : vérifier que le logiciel continue de remplir le besoin à grande échelle, ou en le privant d'une partie de ses ressources

sanity tests : vérifier si on peut se permettre d'exécuter le test suivant

test de **non-regression** par rapport aux fonctionnalités existantes du logiciel

tests **alpha** puis **beta** / **pilote** : sur un panel d'utilisateurs pré-sélectionnés

tests **destructifs** : vérifier si on arrive à détruire tout ou partie des fonctionnalités du logiciel

tests de **performance** : vérifier si les contraintes de performance du logiciel sont remplies

tests de **sécurité** : vérifier la présence d'aucune faille de sécurité

tests de **concurrence** : vérifier le comportement du logiciel durant une phase d'activité normale

tests d'**usabilité**, tests d'**accessibilité** : voir la qualité d'ergonomie

tests de **localisation** : vérifier que la qualité du logiciel est maintenue pour toutes les cultures

tests de **conformité** par rapport à une norme, par exemple en activant tous les warnings du compilateur

tests **continuels** : voir le chapitre sur l'intégration continue

Caractéristiques

Un logiciel étant constitué de la somme de tous ses composants, si chacun de ces composants est testé de manière indépendante, exhaustive et conforme à sa spécification, la qualité de tout le logiciel est garantie.

Afin de se rapprocher au maximum de l'exhaustivité, les tests d'une application doivent être les plus détaillés possibles.

Idéalement, chaque test devrait être indépendant des autres. Par extension, cela signifie que le périmètre de chaque test devrait être limité. Cependant, au sein de ce périmètre, le test doit être le plus détaillé possible.

Déroulement

Situation initiale (« *given* »)

Il s'agit de l'état dans lequel se trouve le système avant d'appliquer le comportement décrit par le test.

La situation initiale d'un test correspond à ses **pré-conditions**: les données en entrées et/ou une suite de commandes de **mise en place** (*setup*).

Action (« *when* »)

Il s'agit d'une suite d'événements (actions utilisateur, message réseau, …) appliqués au système à partir de la situation initiale.

Ce comportement appliqué se traduit souvent par une commande (ou une suite de commandes) visant à amener le système de la situation initiale vers une situation attendue correspondante.

Situation attendue (« *then* »)

Il s'agit de l'état dans lequel doit se trouver le système après avoir appliqué le comportement décrit à partir de la situation initiale.

Si la situation réelle (ou situation observée) est différente de la situation attendue, le test *doit* échouer.

Qualités

Isolation

Isolation de l'élément testé

Afin de minimiser les effets de bords indésirables lors des tests, il est préférable que l'élément à tester le soit en isolation ; c'est à dire, en remplaçant ses dépendances par des objets spécifiquement créés pour le test. Ces objets spécifiques sont à priori bien plus simples, et donc exempts de dysfonctionnements, que leurs équivalents « réels ».

Cette manière de tester est similaire à l'approche « top-down » vue précédement. En pratique, il est cependant souvent nécessaire de trouver le bon compromis en ce qui concerne le niveau d'étanchéité des tests :

> Une étanchéité totale des tests isole parfaitement chaque élément testé. Un échec à un tel test localise automatiquement l'anomalie dans l'élement en question, puisqu'aucune de ses dépendances n'est utilisée. Cependant, en raison de la multiplicité des objets de tests spécifiques, le code de test correspondant peut être difficile à maintenir.
>
> Ne tester aucun composant en isolation garantit à priori de gagner du temps, puisque tout les éléments de code produits sont utilisés en production. Cependant, en cas de test(s) en échec, l'analyse est complexe, puisque l'anomalie peut se trouver non seulement dans l'objet testé, mais aussi dans n'importe laquelle de ses dépendances, ou encore dans les interactions entre ces différents éléments.

Objets spécifiques à un contexte de test

> Un *dummy* est un objet sans aucun comportement, qui peut être utilisé par exemple pour compléter une liste de paramètres d'appel. Le comportement de l'objet remplacé ne doit avoir aucune influence sur le déroulement du test.
>
> Un *stub* (ou *bouchon*) est un objet dont le retour est pré-programmé, et est à priori toujours le même. Le *stub* a un comportement peu complexe.
>
> Un *fake* s'apparente à un *stub* dont le comportement est plus complexe. Le comportement d'un *fake* est censé se rapprocher de celui d'un objet ou d'un groupe d'objets utilisés en production. Cependant, un *fake* est bel est bien un objet à n'utiliser que dans un contexte de tests, et ne doit jamais être utilisé en production !
>
> Un *spy* est un *stub* qui offre des fonctionnalités d'analyse. Il permet par exemple de logger des informations supplémentaires quand on l'utilise, ou d'accumuler certaines métriques sur son utilisation.

Un *mock* est un objet dont le comportement est pré-programmé. Un *mock* provoque une erreur si il est utilisé en dehors de ce comportement.

Isolation des tests entre eux

Chaque test devrait pouvoir être exécuté de manière indépendante, en autonomie. De plus, si plusieurs tests sont exécutés (ie. sous forme de **suite** de tests), leur séquencement devrait pouvoir changer d'une exécution à l'autre sans altérer le résultat de chaque test.

En effet, si un test `T` dépend de l'exécution préalable d'un autre test `P` :

- si `P` et `T` échouent, l'échec de `T` est-il du uniquement à l'échec de `P`, ou à un ou plusieurs autres problèmes propres à `T` ?
- si `P` échoue mais que `T` réussit, la situation est-elle normale ? `T` ne devrait-il pas échouer aussi, puisque sa précondition, `P`, n'est pas remplie ?
- cette dépendance de `T` envers `P` reflète-t-elle vraiment la spécification ?

Toutes ces questions compliquent fortement l'analyse et donc la maintenance du système. Dans la situation précédement décrite, il est préférable de considérer `P`, non pas comme un test indépendant, mais comme faisant partie de la mise en place de `T`.

Fixtures

La plupart des frameworks de test proposent au moins deux fonctions spécifiques, appelées **fixtures** :

- une fonction `setUp` destinée à regrouper les commandes de mise en place d'un contexte (ie. d'une sitation initiale) commun à tous les tests d'une suite donnée. Cette fonction est systématiquement appelée avant chaque test de la suite.
- une fonction `tearDown` destinée à regrouper les commandes de "nettoyage", permettant aux autres tests de la suite de repartir d'une situation initiale saine. Cette fonction est systématiquement appelée après chaque test de la suite.

> Voici certaines opérations qui sont typiquement à faire dans une fonction `setUp` :
>
> - Créer une base de données de test spécifique, différente de la base de données de production. Cela peut permettre de préserver la sécurité du système (la base contient de "fausses" informations), de diminuer le temps d'exécution des tests (la base de test est plus légère que la base de prodution), ou tout simplement de rendre leur exécution possible (si le format des données de production est difficilement reproductible).
> - Créer un certain nombre de fichiers (réels ou virtuels). Si les fichiers sont créés sur le disque, ils devront évidemment être supprimés par la fonction `tearDown`.
> - Formatter un disque et installer un système d'exploitation propre. Cela permet de garantir un contexte d'exécution des tests sain.
> - Acquisition d'un device. Ce device devra évidemment être libéré par la fonction `tearDown`.
> - Préparation des données servant au tests sous forme de fakes/stubs/mocks.

Déterminisme

Le résultat d'un test doit être systématique. Tant qu'aucun changement n'est apporté, un test réussi doit rester réussi, et un test en échec doit rester en échec.

Un test déterministe permet de garantir, via entre autres l'usage de bouchons, de mocks et d'une implémentation en isolation, qu'une anomalie constatée est reproductible, et donc qu'elle peut être corrigée.

> De nombreux éléments peuvent rendre un test non déterministe. Par exemple :
>
> - la génération de variables aléatoires (*random*)
> - les *threads*
> - les appels systèmes : date/heure, internationalisation, …

Exhaustivité

Un test unitaire étant la validation d'un comportement spécifique, il n'est par nature qu'un indicateur partiel de qualité. Seule l'exhaustivité des tests permet donc de garantir la qualité globale d'un système.

Il est en pratique nécessaire de tester *tous* les élements suivants :

- les cas nominaux, testant la capacité fonctionnelle du système en regard de sa spécification
- les cas d'erreur, ou la façon dont le système réagit aux entrées imprévues
- différentes types d'inputs, notamment les cas limites

En pratique, il est indispensable de créer un ou plusieurs nouveaux tests dès qu'une anomalie (eg. une régression) est constatée !

Couverture de code

Tester le niveau de couverture du code offre un bon indicateur sur l'exhaustivité avec laquelle le code de tests vérifie le code source.

Il existe différent niveau de granularité de couverture de code :

- vérifier que toutes les fonctions sont appelées
- vérifier que toutes les instructions sont exécutées
- pour chaque structure de contrôle, vérifier que tous les chemins d'exécutions sont empruntés
- pour chaque expression booléenne, vérifier que toutes les opérandes contribuent à son évaluation

Intégration continue

L'intégration continue permet de builder, tester et, éventuellement, livrer et déployer un ou plusieurs systèmes et ce, de manière **automatique**, **fiable** et **systématique**.

Voici à quoi ressemble le workflow typique d'un développeur travaillant avec un serveur d'intégration continue. Partons du principe que le développeur doit implémenter une nouvelle fonctionnalité `F`.

1. D'abord, le développeur met à jour sa copie locale avec le code issue du dépôt de sources central
2. Ensuite, il implémente `F`, en codant à la fois.
   a. la fonctionnalité `F` proprement dite
   b. le code testant l'implémentation de `F`
3. Ensuite, il effectue un build local. Ce build effectue non seulement la phase habituelle de compilation, mais exécute aussi les tests.
4. Si les tests sont OK, il peut mettre à jour sa copie locale. En effet, d'autres développeurs ont très bien pu modifier le code du dépôt central pendant qu'il implémentait `F`.
   a. Si il y a des nouveautés, le développeur les fusionne (*merge*) dans les siennes. Puis, il retourne à l'étape *3*.
   b. Sinon, il commite son implémentation de `F`.
5. Suite à ce commit, l'intégration continue va automatiquement builder le système (et donc lancer les tests).
   a. Si le build échoue, le développeur doit corriger le problème qui fait que son build passe localement, mais qu'il échoue sur le serveur. Donc : retour à l'étape *2*.
   b. Si le build passe, et seulement alors, `F` peut être considérée comme finie d'être implémentée !

Afin d'en arriver là, il est nécessaire de prendre en compte plusieurs points en termes de configuration du serveur d'intégration continue, du mécanisme de build proprement dit, de la manière dont les développeur vont commiter, et de la visibilité offerte par l'intégration continue.

Configuration

Une intégration continue nécessite l'usage impératif d'un gestionnaire de sources. Le gestionnaire de source choisi dépend des préférences de l'équipe de développement. Cependant, il est important qu'une seule branche de référence soit identifiée. C'est cette branche, appelée *master*, qui servira pour la livraison.

De la même manière, il est nécessaire d'avoir un serveur d'intégration continue. Ce serveur

d'intégration continue doit :

avoir accès à tout le nécessaire pour builder et tester entièrement le projet.

fournir le maximum d'outils aux membres de l'équipe de développement.

Build

Le build doit impérativement être automatique. En d'autre termes, cela signifie que toute modification du système doit systématiquement déclencher un build.

Les bénéfices sont multiples :

Le mécanisme de build peut être très complexe : nécessitant de nombreuses dépendances, découpée en nombreuses phases conditionnelles, visant plusieurs plateformes différentes, intégrant une pipeline de déploiement … Toute automatisation de ce processus est cela de moins à gérer par l'équipe de développement.

Un build automatique est la meilleure manière de se préserver des erreurs humaines.

L'exécution des tests fait partie intégrante du build.

Un serveur d'intégration continue peut être configuré pour ne builder que le nécessaire (c'est à dire uniquement ce qui a été modifié, ainsi que ce qui en dépend). Cela améliore la rapidité avec laquelle le produit peut être livrée.

Commits

Un bon usage quotidien d'un serveur d'intégration se traduit par une discipline de modification du code résumée par l'axiome *« commit early, commit often »* : cela signifie que toute modification devrait être commitée, de la manière la plus immédiate et la plus atomique possible.

Les bénéfices sont les suivants :

Si un ou plusieurs tests sont en échec suite à un commit de quelques lignes, l'erreur est plus facile à analyser et à corriger que si le commit fesait des dizaine de lignes ou davantage.

Se forcer à commiter souvent favorise en général un meilleur découpage du code, et donc une meilleure conception.

Commiter souvent de manière utile permet d'enrichir l'historique maintenue par le gestionnaire de sources, ce qui facilite l'analyse lors de corrections ou d'évolutions futures. Ce dernier point suppose évidemment la présence de commentaires de commits informatifs.

Visibilité

L'intégration continue permet la transparence.

Transparent pour les membres de l'équipe de développement eux-mêmes, qui savent de manière permanente où ils vont, car ils profitent à la fois du feedback du client mais aussi de celui de leur intégration continue.

Cela signifie aussi de se rendre visible à l'extérieur de l'équipe de développement, par exemple en rendant les artefacts disponibles aux endroits appropriés à ceux qui en ont besoin.

Last updated 2018-01-23 17:38:59 UTC