CVDA

Enseignante : Catarina Ferreira Da Silva
Catarina.ferreira@univ-lyon1.fr

Ce cours est basé sur celui d'Amélie Cordier des années précédentes.

Table des matières :

I. Objectifs de ce module

- Distinguer les différents types d'Ateliers de Génie Logiciel (AGL) et savoir manipuler quelques-uns (PowerAMC, Netbeans,…)
- Apprendre à choisir un AGL pour répondre à des besoins spécifiques
- Savoir effectuer de la veille technologique, c'est-à-dire chercher des outils, les comparer, et les choisir en fonction des besoins
- Comprendre la notion de cycle de vie d'un logiciel
- Connaître les différents modèles de cycle de développement de logiciels : traditionnel en cascade, cycle en V, cycle en spirale, *quick and dirty*, semi-itératif, *Rapid Application Development* (RAD), *eXtreme Programming* (XP) et Agile
- Connaître et utiliser des outils de gestion de versions
- Connaître et utiliser des outils de test de logiciels
- Connaître et utiliser des outils de debugging

II. Modalités d'évaluation

- Un exposé de veille technologique en groupe : 25 %
- Un QROC 1 contenant environ la 1er moitié de la matière : 35%
- Un QROC 2 contenant environ la 2ème moitié de la matière : 35%
- La participation dans les séances de cours et éventuellement la sélection d'un des TP à retourner à la fin d'une des séances : 5%

III. Introduction

AGL signifie Atelier de Génie Logiciel (en anglais, CASE, Computer Aided Software Engineering). Un AGL est un ensemble d'outils de Génie Logiciel regroupés sous un « chapeau » commun, ce qui assure une certaine homogénéité d'utilisation entre les différents outils. En d'autres termes, un AGL est un logiciel qui aide à fabrique r d'autres logiciels. Pour comprendre ce qu'est un AGL, il est donc nécessaire de définir ce qu'est le Génie Logiciel, et donc de revenir sur la définition de logiciel. L'objectif de ce cours est d'acquérir le vocabulaire et les connaissances nécessaires pour comprendre ce qu'est un AGL et pour pouvoir choisir le ou les AGL adaptés à une situation donnée, lorsqu'il s'agit de développer un logiciel (par exemple, dans le cadre du projet tuteuré).

IV. Génie Logiciel

Le Génie Logiciel (Software Engineering) recouvre tous les aspects qui ont trait à la conception et au développement de logiciels. Le GL s'intéresse donc aux phases d'analyse, de conception, de développement et de maintenance d'un logiciel. L'objectif du GL est de proposer des méthodes et des outils permettant de rationnaliser le développement de logiciels. Il propose également un ensemble de « bonnes pratiques ».
Quelques considérations historiques : au début de l'ère « informatique », le développement des logiciels était un processus très peu supervisé et relativement chaotique. Les développeurs produisaient des logiciels sans suivre de méthode particulière. Mais avec la complexification du domaine, le développement de logiciels est devenu une tâche plus complexe laissant moins de place aux erreurs (augmentation de la puissance de calcul des ordinateurs, apparition des interfaces homme-machine, utilisation des bases de données, débuts de la programmation concurrente, problématique du temps réel, apparition d'applications multiutilisateurs, multiplication du nombre de concepteurs et de développeurs, travail à distance, utilisation du Web, etc.). Il est alors devenu nécessaire d'organiser le développement logiciel. Le terme « Génie Logiciel » est apparu pour la première fois en 1968, lors d'une conférence internationale sur le développement du logiciel. Une des préoccupations principales était alors la baisse de qualité observée au niveau des logiciels produits. C'est suite à ces observations et à cette conférence que la discipline scientifique « Génie Logiciel » est apparue, avec pour vocation affirmée de diminuer les problèmes identifiés.

Les retards de développement, les bugs, les coûts de maintenances, etc. restent toujours des problèmes importants dans le domaine de la création de logiciels, mais l'application raisonnée d'une approche Génie Logiciel permet néanmoins de les minimiser.

Ainsi, le GL peut être considéré comme une science qui vise à rationaliser, structurer et systématiser le développement des logiciels. Il propose des méthodes, des procédures et des outils pour supporter la conception et le développement des logiciels.

Le GL a pour but de permettre à la fois un gain de temps de développement et un gain de qualité sur le produit fini (et donc un gain financier). Il offre par ailleurs un cadre rigoureux qui permet aux développeurs de travailler dans des conditions plus confortables.
Il est important d'observer que le GL n'est pas limité aux outils (que l'on va survoler dans ce cours), mais qu'il comprend également les théories, les méthodes, les langages et les techniques permettant de guider les concepteurs et les développeurs dans la réalisation de logiciels de qualité.

V. Logiciel

Le terme logiciel permet de parler de façon générique de l'ensemble des programmes que l'on peut trouver sur un dispositif numérique. Un logiciel est donc un composant informatique permettant d'automatiser les traitements qui peuvent être effectués sur des données. Le logiciel peut être configuré, manipulé et utilisé par l'humain, ou fonctionner de façon autonome. Le terme logiciel est très généraliste : il recouvre à la fois les applications utilisateur (suite bureautique), les jeux, les applications système, les composants, les plugins, les drivers, les systèmes d'exploitation, etc.

VI. Cycle de vie du logiciel

Le cycle de vie d'un logiciel (software lifecycle) désigne l'ensemble des étapes par lesquelles passe un logiciel depuis sa phase de conception jusqu'à sa maintenance une fois en phase de production. Ainsi, le *cycle de vie d'un logiciel* désigne toutes les étapes du développement d'un logiciel, de sa conception à sa disparition. L'objectif d'un tel découpage est de permettre de définir des jalons intermédiaires permettant la validation du développement logiciel, c'est-à-dire la conformité du logiciel avec les besoins exprimés, et la vérification du processus de développement, c'est-à-dire l'adéquation des méthodes mises en œuvre.
L'origine de ce découpage provient du constat que les erreurs ont un coût d'autant plus élevé qu'elles sont détectées tardivement dans le processus de réalisation. Une bonne maîtrise du cycle de vie permet de détecter les erreurs au plus tôt et ainsi de maîtriser la qualité du logiciel, les délais de sa réalisation et les coûts associés.

Le cycle de vie du logiciel comprend généralement les étapes suivantes[1] :

- **Définition des objectifs** : Cet étape consiste à définir la finalité du projet et son inscription dans une stratégie globale.
- **Analyse des besoins et faisabilité** : C'est-à-dire l'expression, le recueil et la formalisation des besoins du demandeur (le client) et de l'ensemble des contraintes, puis l'estimation de la faisabilité de ces besoins.
- **Spécifications ou conception générale** : Il s'agit de l'élaboration des spécifications de l'architecture générale du logiciel.
- **Conception détaillée** : Cette étape consiste à définir précisément chaque sous-ensemble du logiciel.
- **Codage (Implémentation ou programmation)** : C'est la traduction dans un langage de programmation des fonctionnalités définies lors de phases de conception.
- **Tests unitaires** : Ils permettent de vérifier individuellement que chaque sous-ensemble du logiciel est implémenté conformément aux spécifications.
- **Intégration** : L'objectif est de s'assurer de l'interfaçage des différents éléments (modules) du logiciel. Elle fait l'objet de tests d'intégration consignés dans un document.
- **Qualification (ou recette)** : C'est-à-dire la vérification de la conformité du logiciel aux spécifications initiales.
- **Documentation :** Elle vise à produire les informations nécessaires pour l'utilisation du logiciel et pour des développements ultérieurs.
- **Mise en production** : C'est le déploiement sur site du logiciel.
- **Maintenance** : Elle comprend toutes les actions correctives (maintenance corrective) et évolutives (maintenance évolutive) sur le logiciel.

La séquence et la présence de chacune de ces activités dans le cycle de vie dépendent du choix d'un modèle de cycle de vie entre le client et l'équipe de développement. Le cycle de vie permet de prendre en compte, en plus des aspects techniques, l'organisation et les aspects humains.

VII. La relation entre le client et les équipes de développement

Traditionnellement on parle de **maîtrise d'ouvrage et maîtrise d'œuvre**.

Maître d'ouvrage (MOA) : le MOA est une personne morale (entreprise, direction etc.), une entité de l'organisation. Ce n'est jamais une personne.

[1] Adapté de http://laurent-audibert.developpez.com/Cours-UML/html/Cours-UML005.html

Maître d'œuvre (MOE) : le MOE est une personne morale (entreprise, direction etc.) garante de la bonne réalisation technique des solutions. Il a, lors de la conception de la solution informatique, un devoir de conseil vis-à-vis du MOA, car la solution informatique doit tirer le meilleur parti des possibilités techniques.

Le MOA est client du MOE à qui il passe commande d'un produit nécessaire à son activité.

Le MOE fournit ce produit ; soit il le réalise lui-même, soit il passe commande à un ou plusieurs fournisseurs (autres entreprises) qui élaborent le produit sous sa direction.

La relation MOA et MOE est définie par un contrat qui précise leurs engagements mutuels.

Lorsque le produit est compliqué, il peut être nécessaire de faire appel à plusieurs fournisseurs. Le MOE assure leur coordination ; il veille à la cohérence des fournitures et à leur compatibilité. Il coordonne l'action des fournisseurs en contrôlant la qualité technique, en assurant le respect des délais fixés par le MOA et en minimisant les risques.

Le MOE est responsable de la qualité technique de la solution. Il doit, avant toute livraison au MOA, procéder aux vérifications nécessaires (« recette usine »).

Plusieurs façons d'organiser les étapes de conception et de développement d'un logiciel sont disponibles. Ces approches sont détaillées dans la section suivante.

VIII. Les modèles de conception et de développement

Les principales méthodes et modèles sont :

- Le modèle en cascade
- Le cycle en V
- Le cycle en spirale
- Le développement itératif et incrémental
- L'approche *Quick-and-dirty*
- Le cycle semi-itératif
- La méthode RAD
- La méthode de l'eXtreme Programming
- Les méthodes agiles

Le **modèle en cascade** (on ne peut pas construire le toit avant les murs). Dans ce modèle, on considère qu'une modification en amont du cycle a des conséquences majeures sur le reste, on développe donc chacune des phases en séquence (l'une après l'autre). On valide l'avancée des développements à chaque fin de phase et on ne passe pas à la phase suivante si la validation n'est pas satisfaisante (Figure 1). Le problème de ce modèle est qu'il est lent :

si une phase est bloquée, le reste du projet ne peut pas progresser (ce qui est un problème dans le cas d'un développement réparti, comme on en fait de plus en plus...). Ce modèle n'est pas vraiment compatible avec les développements actuels : il ne permet pas d'intégrer suffisamment de phases de tests dans la boucle de conception.

Les différentes étapes du modèle en cascade sont :

- Étude préalable
- Prototypage (rapide jetable ou lent récupérable)
- Spécification
- Conception générale
- Conception détaillée
- Écriture des programmes, vérification individuelle
- Validation globale
- Diffusion, mise en place
- Exploitation, maintenance

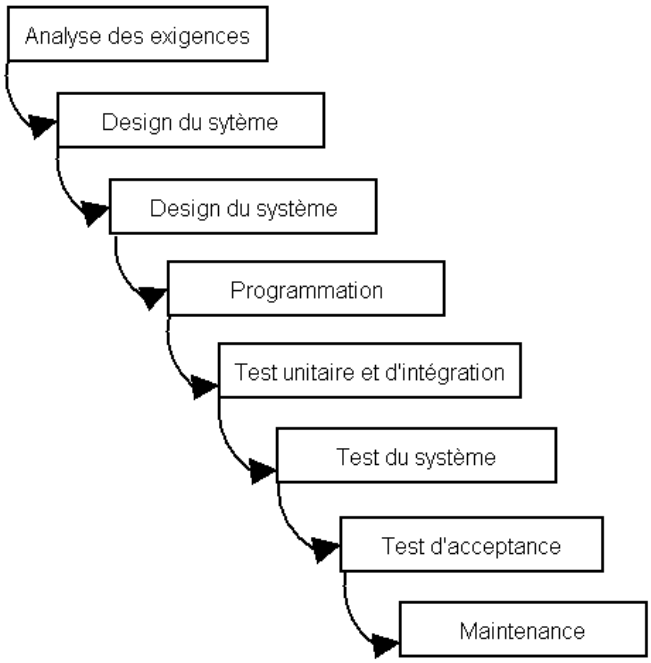


Figure 1. Le modèle en cascade

Le **cycle en V** est une amélioration du modèle en cascade pour essayer de le rendre plus réactif. C'est aussi le modèle de développement le plus connu et encore très utilisé. Ce modèle associe à chaque étape une autre de test ou de validation au même niveau d'abstraction (Figure 2). Il s'agit d'un modèle dans lequel le développement des tests et du logiciel sont effectués de manière synchrone. La vérification de la conformité doit être

anticipée dès la phase de conception. Ce modèle permet d'éviter de trop gros retours en arrière en cas d'anomalie.

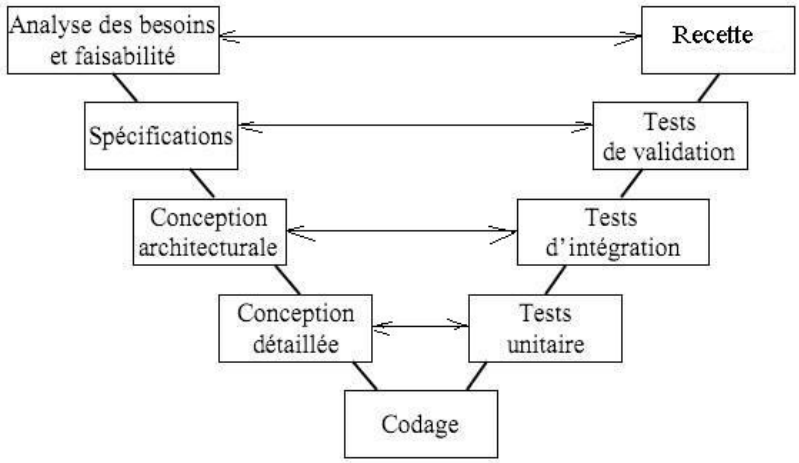


Figure 2. Le modèle en V

Cependant, ce modèle souffre toujours du problème de la vérification trop tardive du bon fonctionnement du système.

Ces deux modèles classiques souffrent de deux maux qui ne sont pas récents :

- Elles considèrent que tous les besoins ont été exprimés au début et sont parfaitement compris de toutes les parties prenantes du projet;
- Les besoins ne peuvent pas évoluer au cours du projet.

Le **cycle en spirale** introduit des changements importants pour remédier aux faiblesses des modèles précédents, surtout leur manque de résilience face au changement. Le cycle en spirale tente d'adresser les risques progressivement en répétant le modèle de cascade en une série de cycles ou itérations. Il met l'accent sur l'activité d'analyse des risques. Dans cette approche, on commence par un jeu restreint de fonctions, puis on augmente la complexité à chaque itération. Les produits issus des itérations sont donc de plus en plus riches en fonctionnalités à chaque incrément.

Proposé par B. Boehm en 1988, ce modèle est beaucoup plus général que les précédents. Chaque cycle de la spirale se déroule en quatre phases :

1. Déterminer les objectifs : détermination, à partir des résultats des cycles précédents, ou de l'analyse préliminaire des besoins, des objectifs du cycle, des alternatives pour les atteindre et des contraintes ;
2. Evaluer les alternatives : analyse des risques, évaluation des alternatives et, éventuellement maquettage ;

3. Développement du produit : développement et vérification de la solution retenue, un modèle « classique » (cascade ou en V) peut être utilisé ici ;
4. Planifier le cycle suivant : revue des résultats et vérification du cycle suivant.

L'analyse préliminaire est affinée au cours des premiers cycles. Le modèle utilise des maquettes exploratoires pour guider la phase de conception du cycle suivant et permet l'implication du client. Ce dernier peut évaluer les résultats au fur et à mesure qu'ils sont produits. Cependant c'est un modèle relativement complexe, comme on peut voir dans la Figure 3, et difficile à gérer.

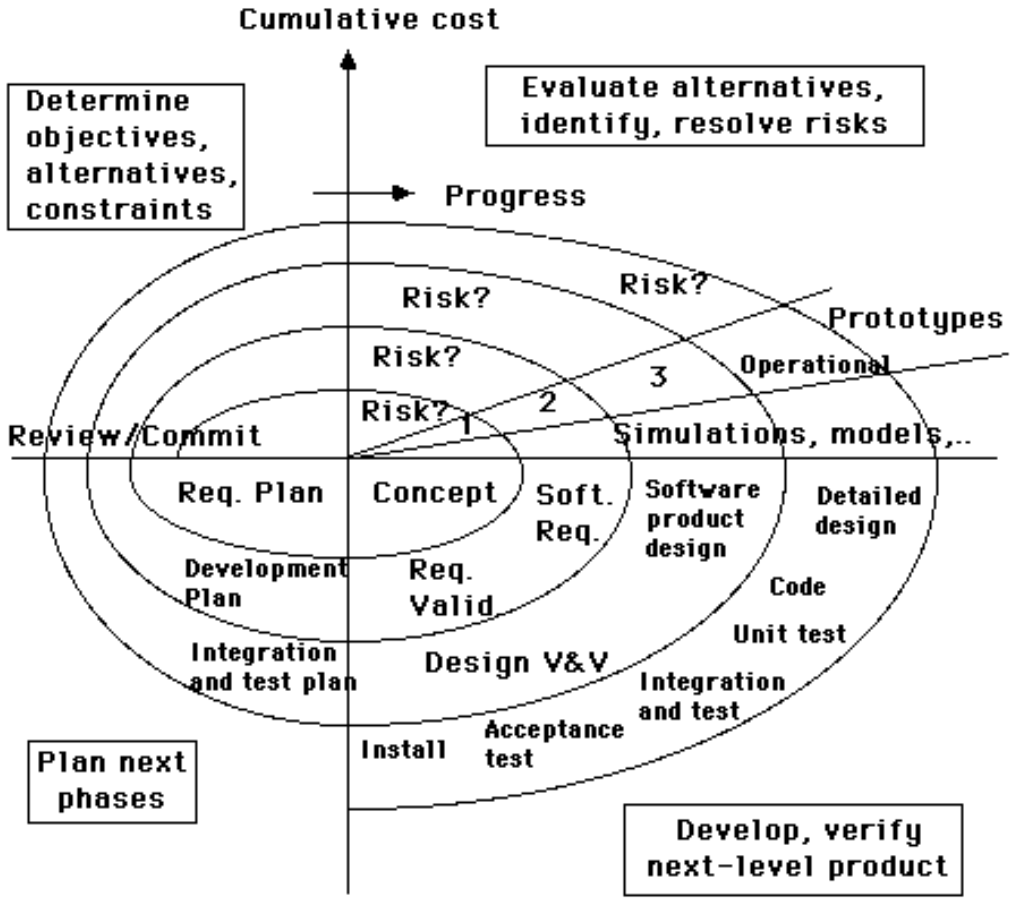


Figure 3. Le modèle en spirale tel que proposé par Boehm

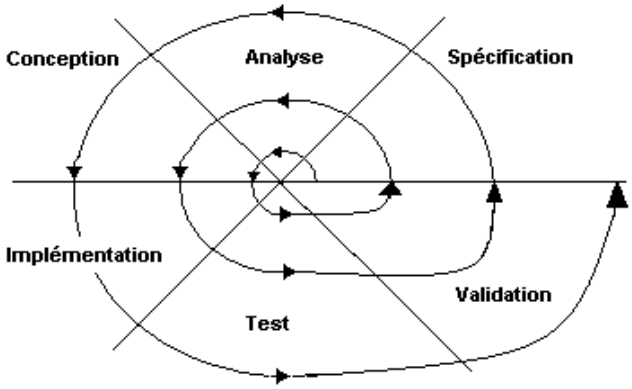


Figure 4. Une variante simplifiée du modèle en spirale

Le **développement itératif et incrémental** est proposé par Ivar Jacobson, Grady Booch et James Rumbaugh dans l'Unified Process[2]. Ces auteurs caractérisent ce processus en de la manière suivante :

1. Spécification pilotée par les cas d'utilisation (use-case driven) qui doit décrire complètement les fonctionnalités du logiciel à développer. Cette étape remplace la spécification traditionnelle. Chaque cas d'utilisation doit répondre à la question « qu'est-ce que le logiciel doit faire pour chaque utilisateur ? ». Ces cas d'utilisation conduisent le processus de développement, puisque ensuite les développeurs construisent et implémentent pour réaliser ces cas d'utilisation. Les tests doivent assurer que les cas d'utilisation sont correctement implémentés.
2. Développement orientée architecture : l'architecture logicielle représente les aspects les plus importants statique et dynamique du système (la plate-forme sur laquelle le logiciel sera installé, les composants réutilisables, les considérations de déploiement, les systèmes existants et les exigences non fonctionnelles). L'architecte choisit les cas d'utilisation qui représentent les principales fonctions du système (5% à 10% de tous les cas d'utilisation), les spécifie en détail, et les réalise en termes de sous-systèmes, de classes et de composants.
3. Développement itératif et incrémental : le projet de développement de logiciel est divisé en mini-projets, dont chacun est une itération qui se traduit par une augmentation (ou incrément). Chaque itération traite des risques les plus importants et réalise un groupe de cas d'utilisation. Dans les premières phases, une conception générale pourrait être remplacée par une autre plus détaillée. Dans les phases ultérieures, les incréments sont généralement additifs et améliorent la convivialité du produit.

[2] A Rational Approach to Software Development Using Rational Rose 4.0

Les incréments ainsi que leurs interactions doivent donc être spécifiés globalement, au début du projet. Les incréments doivent être aussi indépendants que possibles, fonctionnellement mais aussi sur le plan du calendrier du développement.

L'approche ***Quick-and-dirty*** (littéralement rapide et sale, traduit vite fait-mal fait) : c'est une méthode de programmation souvent utilisée pour réaliser des prototypes et des maquettes, elle est utilisée en particulier en vue de présenter rapidement au client un brouillon du logiciel. Il est rare que l'on conserve du code *quick-and-dirty* dans une application en version de production.

Le cycle **semi-itératif**. Il est à la base de la méthodologie RAD (*Rapid Application Development*). Les deux premières phases sont classiques : expression des besoins et conception de la solution. Mais la troisième phase change : on revient à des itérations courtes pour le développement, ce qui permet de faire avancer le code rapidement, et par morceaux, tout en assurant la possibilité de faire de nombreux tests. Cette méthode met en œuvre un cycle itératif, incrémental et adaptatif. L'idée est d'impliquer le plus possible l'utilisateur final dans le développement, en lui faisant valider les avancées de l'application le plus souvent possible. Les cycles de développement étant très courts, ils facilitent également la mise en place de tests simples mais efficaces.

Dans la méthode de l'**eXtreme Programming** (XP), les activités d'analyse, de programmation, de test et de validation sont effectuées continuellement et parallèlement, selon un cycle qui comporte de très nombreuses et fréquentes itérations avec à chaque fois un jeu restreint de fonctionnalités. Ce procédé, appelé intégration continue, implique une forte coopération de l'utilisateur, qui est considéré comme co-auteur du logiciel. L'XP est une sorte de RAD, mais il dispose de sa méthodologie propre.

Principe de méthodes **agiles**. Agile est un qualificatif de divers procédés de développement en rupture avec les procédés d'ingénierie classiques. Ces procédés mettent l'accent sur les changements constants du cahier des charges et du code source des logiciels, une collaboration étroite et une forte implication de l'utilisateur final, et un cycle de développement en spirale avec de nombreuses et courtes itérations. La programmation Agile regroupe les méthodes RAD et XP. Il existe un manifeste de la programmation Agile qui énonce les principes de ces nouveaux paradigmes de programmation, très en vogue.

Les méthodes agiles reposent sur une structure commune : itérative, incrémentale et adaptative.

Il s'agit là de fonctionner selon un schéma itératif pour arriver à la solution finale.
Ainsi l'analyse des besoins va déboucher sur une liste de spécifications qu'il faudra classer par lot de réalisation et à l'intérieur de ces lots par priorité d'implémentations (haute,

moyenne, faible). On gardera à l'esprit bien sûr que certaines fonctionnalités sont dépendantes d'autres. Un point qui sera pris en compte lors de la définition des priorités.

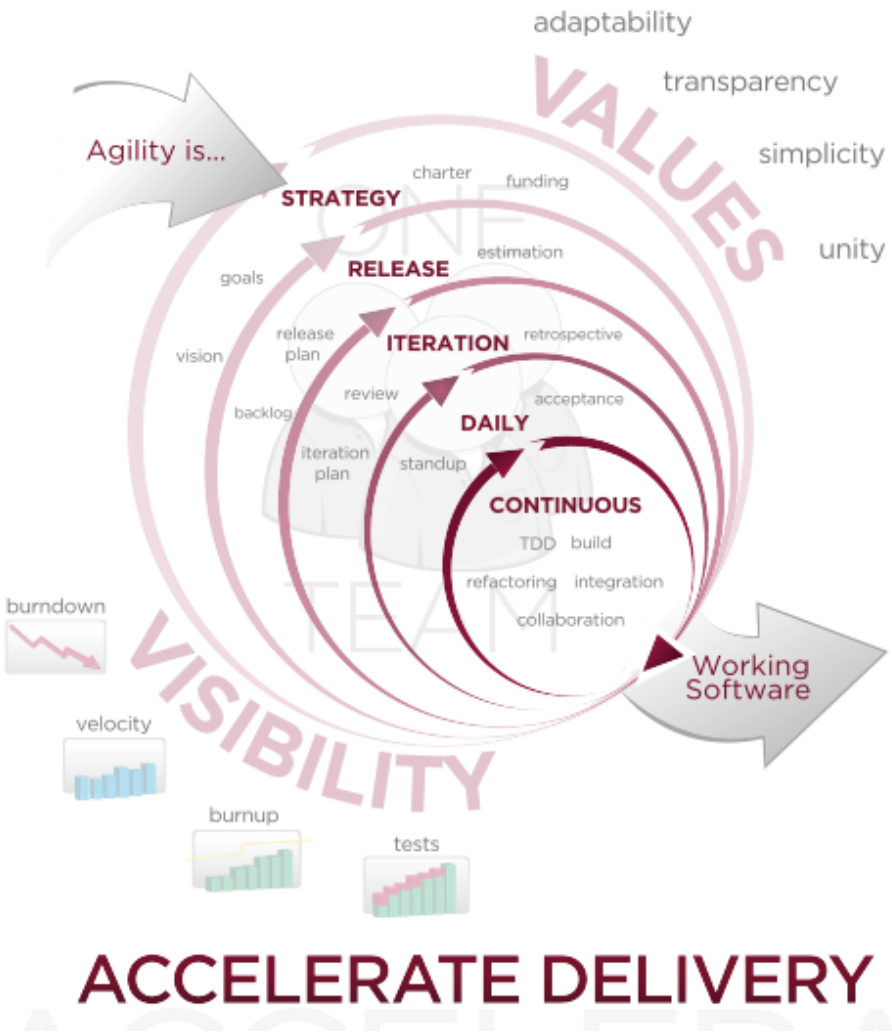
On va donc lancer un certain nombre d'**itérations** basées sur ces lots. Une itération est constituée par l'implémentation d'une liste de spécifications plutôt courte. L'application est alors développée, testée et livrée aux utilisateurs pour qu'ils puissent faire leur retour. Les demandes de modifications sont prises en compte et implémentées pour donner lieu à une autre livraison.

La définition du nombre d'itérations permet de garder la maîtrise du déroulement du projet tant en terme de charge que de planning.

Les méthodes agiles ont plusieurs avantages[3] :

- Elles sont bien adaptées aux clients qui veulent rapidement disposer de l'application. La livraison même partielle est rassurante car montre l'avancement du projet;
- Elles permettent de prendre en compte une expression de besoin floue ou incomplète. Car c'est devant l'application que l'utilisateur prendra conscience des oublis ou des erreurs dans le cahier des charges qu'il a pu commettre;
- L'interaction permanente entre les utilisateurs et l'équipe de développement. Pas d'effet tunnel, où les utilisateurs doivent attendre de longues semaines la livraison de l'application. Une meilleure compréhension des besoins des utilisateurs par les développeurs.

[3] http://philippe.scoffoni.net/conduite-projet-demarches-agile-vs-demarche-monolithique/



Source image : https://commons.wikimedia.org/wiki/File:Agile_Software_Development_methodology.svg

Et juste pour rire, la **méthode R.A.C.H.E**. Rapid Application Conception and Heuristic Extreme-Programming, visible ici : http://www.risacher.com/la-rache/

IX. Qualité d'un logiciel

La **qualité** d'un logiciel est un aspect très important et trop souvent négligé.
La qualité du logiciel est définie par son aptitude à satisfaire les besoins des utilisateurs.
Dans le domaine du logiciel, satisfaire les besoins de l'utilisateur suppose une démarche qualité qui prenne en compte :

1. la qualité du processus de développement (coûts, délais, méthodes, organisation, personnel, techniques, outils),
2. la qualité intrinsèque du produit (modularité, simplicité, ...),
3. la qualité du service fourni par le logiciel en exploitation.

La qualité du processus de développement est basée sur l'utilisation de méthodes de développement et de gestion de projet généralement définies dans le Manuel Qualité de l'entreprise rédigé au cours de la mise en place d'une politique d'assurance qualité.
L'évaluation de la qualité intrinsèque du logiciel est effectuée sur le produit en développement en fonction des facteurs de qualité attendus, définis lors de la commande (spécifications).
Celle du service porte sur le logiciel en exploitation chez l'utilisateur (ou client) et consiste notamment à vérifier son adéquation aux exigences spécifiées.
Pour plus de détails voir http://mariepascal.delamare.free.fr/IMG/pdf/coursQualite.pdf

Donc, la qualité dépend entièrement de la conception et de la réalisation du logiciel (le logiciel ne « s'use » pas...). Les AGL peuvent aider à améliorer la qualité des logiciels.

Liste des critères de qualité (attention, certains critères sont relatifs aux concepteurs et d'autres sont plus de l'ordre des préoccupations utilisateurs).

- Validité (répond aux besoins exprimés dans le cahier des charges).
- Fiabilité (capacité à fonctionner dans des conditions anormales).
- Extensibilité (facilité avec laquelle il est possible d'ajouter des fonctionnalités au logiciel).
- Réutilisabilité (aptitude à être réutilisé, même partiellement, dans d'autres applications).
- Compatibilité (capacité à être combiné avec d'autres).
- Efficacité (utilisation optimale des ressources matérielles et du temps).
- Portabilité (facilité avec laquelle le logiciel peut être utilisé sur d'autres plateformes).
- Vérifiabilité (facilité avec laquelle on peut vérifier et valider la qualité du logiciel, tests).
- Intégrité (protection du code et des données).
- Facilité d'emploi (aspect utilisateur, ergonomie. L'ergonomie est une exigence grandissante chez les utilisateurs).

X. Les tests

Pour assurer la qualité d'un logiciel, il est nécessaire de faire des **tests** fréquents et complets. Au niveau de la conception, on ne parlera pas de « test » mais de « validation » des choix de conception effectués. Les méthodologies relatives à cet aspect ne relèvent pas de ce cours.

Au niveau développement, les tests visent essentiellement à chasser les **bugs**, quels qu'ils soient (comportement incorrect, inattendu ou manquant). Les bugs peuvent provenir des erreurs de conception ou de programmation. La gravité du dysfonctionnement peut aller de très mineure (apparence légèrement incorrecte d'un élément d'interface graphique), à des évènements en passant par des pertes plus ou moins grandes de données, et, plus rarement, par une détérioration du matériel.

XI. Ateliers de Génie Logiciel : similarités et différences

Proposition de définition : un AGL est un ensemble d'outils intégrés couvrant une partie significative du cycle de vie d'un logiciel, et qui permettent de mettre en œuvre les principes du Génie Logiciel.

AGL orientés conception :

- Spécification des besoins (support de communication avec le client et/ou le chef de projet).
- Aide à la conception (réalisation de diagrammes UML par exemple). C'est un gain de temps par rapport au papier crayon, et cela permet également la réutilisation d'éléments dans d'autres phases du cycle de vie (génération de code à partir du modèle UML par exemple).
- Aide à la documentation de conception (production de rapports de conception, comme avec PowerAMC). Dans ce cas, les AGL sont de véritables outils de communication, avant tout chose.
- Outils de gestion de projet (liste des tâches, temps de travail, diagrammes de **Gantt**, etc.).
- Outils de **rétro-conception** (reverse engineering).
- Générateurs de code (production du script de création d'une BD à partir de PowerAMC).
- Outils de prototypage rapide (réalisation de maquettes et de code quick-and-dirty).

AGL orientés développement :

- Aide à la documentation de code (génération de Javadoc).
- Outils de reverse engineering.
- Editeur de code (indentation automatique, coloration syntaxique, auto-complétion).
- Outils de **profiling de code** (place mémoire, temps d'exécution, code non utilisé, etc.).
- Debugger, optimiseur de code.

- Outil de **gestion de versions** (cet aspect fera l'objet d'un mini-cours dédié).
- Outil de configuration et de suivi des tests, génération de jeux de tests.
- Gestionnaire de bugs.
- Suivi et maintenance (demande d'ajouts de nouvelles fonctionnalités, **tests de non régression**).

XII. Remarques et conclusions sur cette partie

Le terme AGL recouvre une gamme vaste et imprécise d'outils visant à aider les concepteurs et les développeurs à réaliser des logiciels de qualité, dans de bonnes conditions. Autrement dit, un AGL doit permettre d'améliorer la qualité du produit fini, la productivité de ses réalisateurs, et le confort de travail.
Certains AGL permettent de donner plusieurs points de vue sur la réalisation du logiciel : cycle de vie, état d'avancement, connexion avec d'autres composants, etc. D'autres se contentent de donner un point de vue unique : la modélisation par exemple.
Le choix de l'AGL à utiliser doit être fait en fonction des objectifs du projet, et non l'inverse. Il est donc très important de savoir identifier les caractéristiques pertinentes d'un projet qui serviront de critères de choix de l'AGL. Les objectifs de la partie suivante sont d'une part d'apprendre à mieux connaître les AGL existants et à en repérer les fonctionnalités, et d'autre part, d'apprendre à choisir un AGL pour un objectif donné.

Les questions à se poser lorsque l'on doit choisir un AGL :

- Temps de prise en main admissible ?
- A-t-on déjà des connaissances sur l'outil ?
- Cherche-t-on un outil de conception ou de développement ?
- Quelles fonctionnalités sont disponibles ?
- Un AGL est-il imposé par la structure dans laquelle se déroule le projet ?
- Doit-on choisir une solution partagée par plusieurs utilisateurs ?
- La solution retenue est-elle difficile à déployer à l'échelle de la structure ?

Les critères de choix :

- Orienté conception ou orienté développement ?
- L'outil est-il simple à prendre en main ou bien riche en fonctionnalités ?
- Léger / efficace / rapide ?
- Quels langages de programmations sont supportés ?
- Sur quelles plateformes tourne la solution ?
- Gratuit ou payant ? Libre ? Version d'évaluation ?
- Supporte le travail collaboratif ?

- Réputation, qualité des mises à jour ?
- Documentation ? Communauté d'utilisateurs ?