Histoirique

Les années 1990 ont vu fleurir des propositions de modèle de cycle de vie de développement logiciel qui privilégiaient la réactivité par rapport à la planification, qui mettaient le test et la fonctionnalité en bonne place, et qui introduisaient des considérations psycho-sociologiques pour voir le programmeur comme une personne plutôt que comme un exécutant. Ces approches s'appelaient RAD (*Rapid Application Development*), *Extreme Programming*, *Scrum*, etc. Assez vite, les porteurs de ces propositions se sont rendu compte qu'elles avaient beaucoup en commun, et ils l'ont formalisé dans un manifeste, le Manifeste agile (*Manifesto for Agile Software Development*). Aujourd'hui, l'approche Scrum sert souvent de référence. Elle a stabilisé et fait connaître un vocabulaire qui est utilisé même en dehors de/une application stricte de l'approche.

On a d'abord cru que ces approches étaient réservées aux petits projets démarrant *ex nihilo*, et dans des relations contractuelles pas trop contraignantes. Cela excluait les gros projets des SSII/ESN (sociétés de services en ingénierie informatique/entreprises de services au numérique), souvent en mode de maintenance. D'autres pratiques de ces entreprises, comme la délocalisation de sous-projets semblaient aussi s'opposer aux aspects les plus sociaux des approches agiles.

Cependant, l'approche agile s'est lentement diffusée jusqu'à être appliquée aujourd'hui de façon très généralisée dans toutes sortes de cadres de développement logiciel. Elle n'est pas toujours appliquée dans tous ses aspects, mais elle fait désormais partie du vocabulaire standard du développement logiciel.

Agilité et anticipation des risques

L'anticipation des risques suggère une posture de doute, de vérification continue, d'expérimentation et de remise en cause. L'intérêt de l'agilité est qu'elle intègre cette posture dans le cycle de vie. En fait, l'agilité est rarement présentée sous cet angle, mais plutôt comme un moyen de maximiser la valeur du logiciel produit pour le client. Cependant, les deux objectifs sont compatibles, et penser anticipation des risques permet de intégrer de étapes qui ne sont pas immédiatement productives pour le client, mais qui sont nécessaires pour avancer.

Le Manifeste Agile

Le Manifeste Agile met en avant 4 valeurs et 12 principes.

4 valeurs *[les parties entre crochets sont nos commentaires]* **:**

V₁ : S'appuyer sur les **individus** et leurs **interactions** et pas seulement sur les processus et les outils.

V₂ : Valoriser le **logiciel démontrable** et pas seulement sa documentation.

V₃ : Collaborer avec le **client** *[pendant tout le projet]* et pas seulement négocier un contrat avec lui *[avant de lancer le projet]* pour ne vérifier l'exécution à la fin.

V₄ : S'adapter au **changement**, et pas seulement suivre un plan préétabli.

Dans l'original anglais les valeurs sont rédigées en suivant le schéma *X over Y*, et un commentaire spécifie qu'il ne faut pas lire **X plutôt que Y**, mais plutôt **Y ne suffit pas et X est nécessaire**. Il n'est pas non plus exprimé que **X plus Y est suffisant**.

12 principes *[les parties entre crochets sont nos commentaires]* **:**

- Le meilleur moyen de contenter le client *[au sens large, utilisateurs, donneurs d'ordre, …, et pas seulement celui qui paye]* est de lui fournir des fonctionnalités à grande valeur ajoutée *[c'est le client qui décide de la valeur ; V₂, V₃]*.
- Le client peut exprimer des **changements de besoins** à tout moment *[ça peut être une valeur ajoutée pour lui ; V₃, V₄]*.
- Le client doit recevoir des versions opérationnelles du logiciel *[qu'il puisse utiliser dans son rôle de client ; V₂, V₃, V₄]*, selon une périodicité **la plus courte possible**.
- Le client et l'équipe **collaborent quotidiennement** *[V₂, V₃]* .
- Fournira à l'équipe un environnement **motivant** et on lui fera **confiance** *[V₁]*.
- On privilégiera le **face à face** *[éventuellement à distance ; V₁]*.
- On mesure **l'avancement en fonctionnalités** délivrées *[et pas en lignes de code ou autres métriques, …]* ; *V₂]*.
- L'effort de **développement sera soutenable**. Les parties prenantes doivent pouvoir maintenir un effort constant *[sans « coups de bourre » ; V₂, V₃]* aussi longtemps que nécessaire.
- On vise toujours **l'excellence technique** *[voir la charte de l'ingénieur IESF ; V₁]*.
- On **évitera tout travail inutile** *[qui ne contribue pas à la valeur ajoutée démontrable ; V₂]*.
- On laisse l'équipe **s'auto-organiser** pour analyser les besoins du client et trouver la meilleure façon d'y répondre *[V₁]*.
- L'équipe **s'introspecte** régulièrement en quête d'amélioration continue *[V₁, V₄]*.

La méthode Scrum identifie des rôles et des pratiques qui sont entrés dans le vocabulaire commun. Dans la suite, ces termes sont présentés en anglais et sans effort de traduction, car c'est ainsi qu'ils sont utilisés.

Le vocabulaire de la méthode Scrum

Le ***product owner*** (**PO**) est le client, ou son représentant. Dans d'autres contextes, on aurait pu l'appeler le **maître d'ouvrage** ou l'**assistant à la maîtrise d'ouvrage**. En mode cascade ou **V**, le client est supposé formuler ses exigences en début de cycle et sans ambiguïté. En mode agile, il est supposé collaborer intimement avec l'équipe de développement. Le PO définit les fonctionnalités attendues sous la forme de ***user stories***. L'ensemble des *user stories* non traitées constitue le ***backlog***. Le PO peut modifier le *backlog* comme il le veut. C'est un élément important de l'agilité qui permet au client de faire évoluer ses besoins, et aux développeurs de s'y adapter sereinement.

L'équipe de développement est représentée auprès du client par le ***scrum master*** (**SM**). Il organise le travail de l'équipe en ***sprints*** de durée fixée, ex. 1 mois. Un *sprint* se donne pour objectif de réaliser en sélection de *user stories* du *backlog* qui ont été choisies d'un commun accord avec le PO et l'équipe de développement.

Pour y parvenir, l'équipe estime la **charge** de travail (exprimée en **points d'effort**) représentée par chaque *user story* et le PO leur attribue une **priorité**. Les deux peuvent être exprimés par des nombres mais sans socle de fidélité métrique. Ex. 4 est plus prioritaire que 2, mais pas 2 fois plus prioritaire. Pour éviter cette tentation on peut utiliser des échelles non linéaires, ex. la suite de **Fibonacci** (1, 2, 3, 5, 8, …). L'estimation des points d'effort est bien sûr difficile. Elle pourra utiliser des techniques de *brainstorming* comme le ***poker planning***, qui permet aux membres de l'équipe de s'exprimer tout en évitant certains artefacts de la communication de groupe. La **planification** d'un *sprint* (***sprint planning***) devra être cadrée à maximiser le service rendu au PO, tout en respectant la capacité de travail de l'équipe.

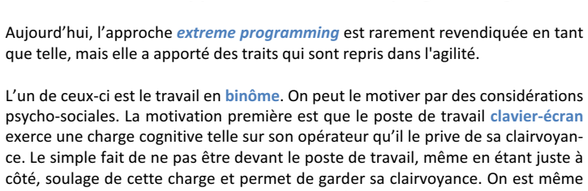
Le PO et le *scrum master* doivent aussi s'accorder sur la **validation** des *user stories*. À l'extrême, une *user story* peut être entièrement définie par des cas de test, dans le style ***test driven development*** (**TDD**). On a vu que le PO pouvait modifier le *backlog* à sa convenance, mais il ne peut pas intervenir sur la planification d'un sprint en cours, et c'est même une mission du *scrum master* que de l'en empêcher. Le client ne peut pas non plus « dévaliser » une *user story* qu'il aurait validée ; il peut y renoncer, en définir d'autres, mais ne peut pas revenir un travail fait et validé.

Pendant le déroulement d'un sprint, l'équipe note sa progression sur un tableau spécial, le ***scrum board***. Il peut prendre la forme emblématique d'un tableau mural couvert de Post It® mais il en existe aussi des versions numériques. Tous les jours, l'équipe tient un ***daily meeting*** durant lequel chacun explique où il en est, ses difficultés, et ses objectifs pour la journée. Chacun peut proposer son aide, on peut redistribuer les tâches, toujours dans le but de faire progresser le *sprint*. Ces pratiques se sont généralisées, même en cas de délocalisation lointaine.

À la fin du *sprint*, le PO valide ou non les *user stories* selon les critères convenus au départ du *sprint*. Noter que la fin d'un sprint n'est pas déterminée par l'équipe, mais par son objectif, mais simplement par l'équivalence de la durée prévue (***time boxing***). En cas d'inadéquation entre la durée du *sprint* et son objectif, on ne prolonge pas la durée du *sprint*, mais on reconnaît que l'objectif n'a pas été atteint. L'adéquation entre la *time box*, le planning et les capacités de l'équipe est un indicateur de qualité qu'il faut suivre.

Le cumul des points d'effort des *user stories* validées à la fin d'un *sprint* constitue la **vélocité** de l'équipe pendant ce *sprint*. Comme pour les points d'effort, il ne s'agit pas d'une mesure absolue qui permettrait de comparer des équipes, mais qu'estimer les coûts est une des choses les plus difficiles qui soit, on s'attend à ce qu'en procédant ainsi l'équipe fasse des estimations toujours plus précises.

La fin du *sprint* donne lieu à un cérémonial contenant entre autre une **rétrospective** (Aurait-on pu faire mieux ? Et éventuellement une **célébration** de fin de *sprint*, ex. un repas pris en commun. Ces rituels intègrent explicitement dans la démarche la dimension humaine des risques à anticiper, y compris via des détails comme de recommander que le *daily meeting* se tienne debout.



Le vocabulaire de l'approche XP – Extreme programming

Aujourd'hui, l'approche ***extreme programming*** est rarement revendiquée en tant que telle, mais elle a apporté des traits qui sont repris dans l'agilité.

L'un de ceux-ci est le travail en **binôme**. On peut le motiver par des considérations psycho-sociales. La motivation première est que le poste de travail **clavier-écran** exerce une charge cognitive telle sur son opérateur qu'il le prive de sa clairvoyance. Le simple fait de ne pas être devant le poste de travail, même en étant juste à côté, soulage de cette charge et permet de garder sa clairvoyance. On est même allé jusqu'à concevoir des environnements, mobilier et agencement des espaces de travail, supposés faciliter le travail en binôme.

Outre ce gain de clairvoyance, le travail en binôme permet aussi la diffusion des savoir-faire dans une équipe en formant des binômes expert-néophyte. L'expert peut avoir le clavier et alors le néophyte le questionne sur ce qu'il fait, ou bien le néophyte a le clavier sous la supervision de l'expert qui est à côté. C'est un autre exemple de prise en compte par l'agilité des aspects ressources humaines.

Un autre trait initié dans la méthode *extreme programming* et repris dans le Manifeste Agile est le soucis de la **durabilité** de l'effort de travail. Il s'agit encore de prendre en compte une dimension humaine. On pourra comparer avec l'attitude « marche ou crève » mise en avant dans certains milieux du « numérique ».

La gestion de la fin

En attribuant une mesure de coût à chaque fonctionnalité et en mesurant bien sa vélocité, l'équipe agile peut estimer le temps nécessaire à la réalisation d'un ensemble de fonctionnalités. Inversement, connaissant le temps qui reste, l'équipe agile peut estimer ce qu'il lui est possible de terminer. Le domaine ***fonctionnalité × temps*** joue donc un peu le rôle d'un ***espace × temps*** en délimitant un cône d'accessibilité dont il est impossible de sortir.

Le cône ouvert vers le futur, sommet sur le présent, montre le maximum qu'il est possible de faire en un temps donné. Le cône ouvert vers le passé, sommet sur une échéance future, montre le maximum qu'il est possible de finir avant cette échéance. L'équipe peut ainsi gérer sa fin de projet en ne commençant, ni même ne continuant, rien qui ne pourra pas être valorisé à la fin du projet.

L'agilité en enseignement – le cas du TP

L'agilité peut aussi être utile en situation d'enseignement. On ne parle pas ici du TP énigme à résoudre dans une séance de 2 heures, mais plutôt du TP problème ouvert s'étalant sur plusieurs séances. Dans cette situation, l'enseignant de TP est le *product owner*, et ce qui l'intéresse c'est l'usage fait par les étudiants de la technologie illustrée par le TP, via la réalisation d'une fonctionnalité qui sert de prétexte au déploiement de cette technologie.

Dans ce cadre, les risques sont principalement :

• Celui des **échéances**, et l'agilité permet de s'assurer de rendre quelque chose dans les délais, éventuellement incomplet, mais intéressant pour le *product owner*.

• Celui de la **technologie**, et l'agilité permet de prévoir des *sprints* purement technologiques pour l'apprivoiser, même en réalisant des fonctionnalités qui n'ont rien à voir avec le sujet apparent du TP ; ne pas s'en priver.

• Celui des **fonctionnalités**, et ici encore l'agilité commande de réaliser des applications partielles que le *product owner* enseignant pourra valider.

Souvent, la consigne s'analyse selon deux axes, ***fonctionnalités × couches technologiques***, ex. plusieurs fonctionnalités et des couches interfaces, services, et base de données. Ici, l'agilité commande de commencer par ne développer qu'une fonctionnalité embryonnaire, mais en traversant toutes les couches technologiques. Faire le contraire serait anti-agile, et expose à développer une grande quantité de code qui, si le TP n'arrive pas à bonne fin, ne servira à rien. C'est le syndrome du TP où rien ne marche malgré la quantité de travail. L'enseignant *product owner* ne peut que donner la note de 0, alors que les étudiants qui ont travaillé s'attendent à plus même sans résultat tangible.

DevOps

La notion de *product owner* risque de cacher la réalité parfois compliquée de l'exploitation d'un logiciel, avec évidemment ses **utilisateurs finaux**, parfois dans plusieurs rôles bien distincts comme dans les bases de données, mais aussi tous les **agents opérationnels** qui installent et exploitent le logiciel.

La démarche **DevOps** veut aller plus loin que ne le fait l'agilité quand elle intégre le client dans la boucle de développement. Elle propose **d'intégrer tous les agents opérationnels** qui auront à faire avec le logiciel. Dans cette démarche l'intégration continue n'est plus suffisante, car le déploiement doit lui aussi être continu.

Conclusion

Après une vingtaine d'années d'hésitation, l'industrie a largement intégré l'agilité. Cela ne veut pas forcément dire la mettre en œuvre littéralement mais plutôt l'intégrer dans le vocabulaire des solutions possibles. En la matière, tout commence par le client, et celui-ci n'est pas toujours agile. De son côté, l'enseignement a lui aussi traîné à intégrer l'agilité, en particulier en la mettant en concurrence avec une approche déductive où une excellente analyse garantirait l'excellence du résultat.

Par l'étendue des préoccupations qu'elle prend on compte, l'agilité montre une maturité qu'on ne voit pas dans d'autres approches qui se concentrent sur l'objet logiciel. En cela, l'agilité rejoint **CMM** (***capability maturity model***) qui n'est pas une méthode de gestion de projet, mais un cadre d'analyse de la maturité d'une entreprise. Une chose est notable avec CMM. Sur les 5 niveaux de maturité définis par ce cadre, seuls les 2 plus faibles sont essentiellement techniques ; les autres incluent des préoccupations de ressource humaine et de stratégie.

Cependant, l'agilité ne dit pas tout sur tout. En particulier, peu est dit sur la façon de constituer un *sprint* ou de valider des *user stories*. Il faut donc compléter l'agilité, vue comme un cadre stratégique qui répond à un **Pourquoi** complexe, par de solides outils/compétences techniques, vus comme des éléments de tactique qui réaliseront le **Comment**. Le plus grand risque réside dans une adoption ritualiste du vocabulaire et des procédures de l'agilité en oubliant à quels enjeux ils répondent.

Bibliographie

Certaines références sont délibérément choisies « datées » pour montrer l'ancienneté du sujet et l'antériorité de certains travaux.

• ***« The Psychology of Computer Programming »***, Weinberg (Van Nostrand Reinhold Company, 1970 multi-réédité).
• **« De la programmation considérée comme un des Beaux-Arts »**, Lévy (La Découverte, 1992).
• **« Précis de génie logiciel »**, Gaudel, Marre, Schlienger et Bernot (Masson, 1996).
• ***« The pragmatic programmer »***, Hunt et Thomas (The Pragmatic Bookshelf, 1999).
• ***« Extreme Programming »***, (O'Reilly, 2005).
• ***« Why Crunch Mode Doesn't Work: Six Lessons »***, Robinson (igda.org, 2005)
• ***« Scrum and XP from the trenches »***, Kniberg (InfoQ, 2007 réédité).
• ***« Integrating CMMI and Agile Development »***, McMahon (Pearson, 2010).

Approche agile du développement logiciel

Olivier Ridoux – 2020

Motivations

Proposée dans les années 90, l'approche agile du développement logiciel n'est rentrée dans le catalogue des pratiques communes que depuis quelques années.

À ce moment de bascule, le mot agilité est parfois utilisé comme un *buzzword*, et les pratiques agiles mises en œuvre comme des **rites**. Robert K. Merton définit le ritualisme comme **l'acceptation des moyens mais pas des objectifs**. Cela conduit à des comportements absurdes et contre-productifs qui se révèlent dans des assertions comme **« Nous avons vu le client régulièrement, donc nous avons suivi une approche agile »** suivi de **« Nous avons fini le développement, mais nous n'avons pas eu le temps de faire les tests »** (réellement observé).

Une autre difficulté réside dans **l'enseignement de l'agilité**. Ici, le risque est de ne considérer l'agilité que comme une pratique sociale et de la confiner aux enseignements de management ou de communication (réellement observé aussi). À nouveau, cela revient à réduire l'agilité à un rite en oubliant son rôle de guider des **décisions techniques** qu'on ne peut comprendre qu'en comprenant le **pourquoi** de l'agilité. Nous pensons qu'un bon point de vue sur ce pourquoi est dans **l'anticipation des risques**, et c'est ce que nous proposons ici.

Cependant, la mise en œuvre raisonnée de l'agilité est complexe. C'est pourquoi on doit profiter de tous les exercices de programmation qui s'offrent aux étudiants, même les plus simples, pour pratiquer l'agilité. Plus généralement, le **génie logiciel**, dont l'agilité, ne doit pas être enseigné uniquement quand la programmation devient difficile, mais comme le **seul exercice raisonnable de la programmation**.

Compléments de lecture : maturité, cycles de vie et risques.
Pliage : verso de cette page au-dessus, traits **gris** rentrants, traits **rouges** saillants. Découper selon le trait rouge entre les deux ●, puis achever le pliage.

① Cycles de vie de développement

En première approximation, le développement d'un logiciel demande une part :
• B : d'expression d'un **besoin** par un demandeur,
• A : **d'analyse** pour comprendre le problème posé et imaginer une réponse,
• C : de **codage** pour produire le logiciel proprement dit, éventuellement constitué de plusieurs parties,
• ID : **d'intégration/déploiement** pour assembler les différentes parties du logiciel, et l'installer dans son contexte d'utilisation,
• VV : et de **vérification/validation** pour s'assurer que le logiciel réalise bien les besoins du demandeur, et qu'il peut être utilisé dans le contexte où il doit l'être.

On appelle **cycle de vie de développement** la façon dont on enchaîne ces actions. Si on admet que vérification et validation peuvent échouer, le cycle de vie réel ne peut qu'itérer A, C, ID et VV. Si en plus, même l'expression des besoins peut évoluer, soit que les besoins évoluent, soit que leur expression se précise, il faudra aussi itérer sur B. Au cours de l'histoire du génie logiciel la façon de décrire le cycle de vie a beaucoup évolué, donnant lieu à de nombreux styles de modélisation.

Les modèles de cycles de vie présentés ici le sont d'une façon extrêmement simplifiée avec 2 objectifs : permettre d'utiliser une **symbolique homogène** et simple, et mettre en valeur les **traits saillants** de chaque modèle, au risque d'être accusé de caricature.

① Risques génie logiciel

Il est intéressant de considérer la gestion de projet, génie logiciel ou pas, non pas comme un ensemble de méthodes orientées vers le but à atteindre, mais plutôt comme un ensemble de méthodes qui aident à anticiper les risques. Anticiper n'est pas éviter, mais plutôt chercher à réduire l'impact du risque qui se réalise.

La plupart des risques qui menacent un projet de développement logiciel sont génériques et peuvent aussi menacer un projet d'une autre nature. Seuls les détails les distinguent. Dans tous les cas, la grande question est comment se rendre compte au plus tôt de la réalisation d'un risque, et quelle réponse adopter. Quels sont ces risques ?

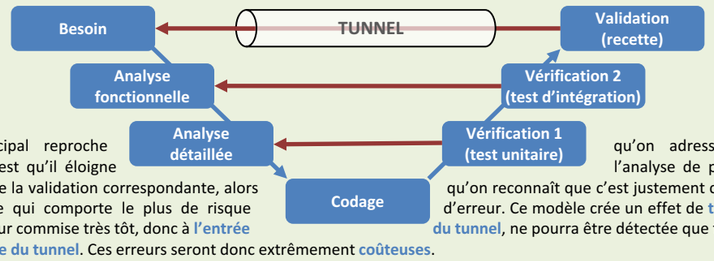
• Les **échéances** : ne pas pouvoir les tenir.
• **L'analyse** : ne pas comprendre la demande du client. Pire, penser qu'on l'a comprise, alors que ce n'est pas le cas.
• Le **codage** : même si l'analyse a été correcte, se tromper dans sa mise en œuvre. Pour un projet logiciel on distinguera souvent une facette fonctionnelle qui concerne plutôt les données et les résultats, et une facette non-fonctionnelle qui concerne plutôt les performances, ex. vitesse, capacité de traitement, ou sécurité.
• La **sous-traitance** : quand un sous-projet est confié à un tiers qui sera lui-même exposé aux mêmes risques.
• La **technologie** : ne pas savoir la maîtriser.
• Les **ressources humaines** : ne pas savoir gérer les différences d'aptitudes ou de motivations, les fluctuations dans les effectifs, les besoins de formation, se contenter de constater que c'est difficile, croire que c'est plus important de connaître les *threads* de Java.
• **L'environnement** : des événements de nature sociale, économique ou politique qui affectent le projet, voire son existence même. Changement de direction ou de réglementation, crise, etc.

Dans un cadre d'apprentissage, ces risques prennent un tour particulier. Les échéances sont beaucoup plus fermes qu'on ne l'imagine car **après le semestre ce n'est plus le semestre**. L'analyse, le codage, la technologie sont souvent le sujet d'étude et constituent un risque calculé. La sous-traitance et les effets de l'environnement sont rares, sauf à faire exprès de sensibiliser à ces risques. Et le risque dominant est le risque ressources humaines car il est constamment masqué par le **mode copain**. C'est pourquoi il faut savoir s'affranchir des affinités pour former des groupes de projet.

③ Cycle de vie en V

Plus avancé que le modèle en cascade, le modèle du **cycle de vie en V** modélise les étapes du développement de façon à mettre en correspondance les échecs de vérification/validation et leurs causes probables. Il est aligné sur un modèle général d'ingénierie des systèmes qui s'est développé dans les années 80.

Dans ce modèle, l'analyse est organisée en niveaux de détail croissants, et la vérification/validation l'est en niveaux d'intégration croissants. L'idée est de mettre en vis-à-vis les niveaux d'analyse les plus détaillés avec les niveaux de vérification/validation les moins intégrés. De façon duale, les niveaux d'analyse les moins détaillés seront mis en vis-à-vis des niveaux de vérification/validation les plus intégrés. De cette façon un échec à un niveau donné désigne le niveau d'analyse qui lui correspond comme cause probable.



Le principal reproche qu'on adresse à ce modèle est qu'il éloigne l'analyse de plus haut niveau de la validation correspondante, alors qu'on reconnaît que c'est justement ce niveau d'analyse qui comporte le plus de risque d'erreur. Ce modèle crée un effet de **tunnel** où une erreur commise très tôt, donc à **l'entrée** du tunnel, ne pourra être détectée que très tard, à la **sortie du tunnel**. Ces erreurs seront donc extrêmement **coûteuses**.

On doit reconnaître cependant que dans les premières présentations de ce modèle de nombreuses autres formes d'itération étaient prévues, ex. dans la branche de gauche du **V**.

①Le rôle occulte des documents logiciels

Le génie logiciel est une ingénierie très documentaire. Il ne faut pas en sous-estimer la richesse des documents manipulés. Concernant les documents de programmation et de test, et leur commentaires, on peut faire les observations suivantes :

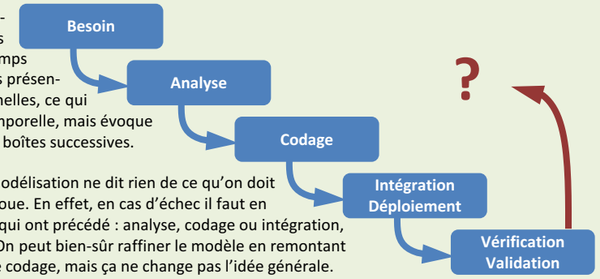
• **Programme :** permet de dire à un humain ce qu'on demande à une machine. Ne pas négliger ce rôle. Un programme doit être lisible par un humain.
• **Test :** permet de préciser formellement et de façon vérifiable des éléments de spécification. Ne pas hésiter à compléter une spécification par des cas de test. Toujours se demander à quel élément de spécification correspond un test.
• **Commentaire** : partout où il est permis d'en mettre, permet de prendre note de difficultés rencontrées et des réponses apportées, permet de lier des éléments logiciels entre eux. Sont de plus en plus formalisés pour entrer dans des traitements automatiques.

② Cycle de vie en cascade

Une des premières tentatives de modélisation du cycle de vie de développement est celle de la **cascade**.

Dans ce modèle, on imagine qu'un problème est décrit de façon plus ou moins formelle, que cette description est passée à des analystes qui vont tenter de la comprendre et proposer une solution logicielle, décrite elle aussi de façon plus ou moins formelle, que cette solution est passée à des programmeurs qui vont la coder sous forme de programmes, qui vont devoir être intégrés et déployés pour pouvoir exécuter le logiciel et le vérifier/valider.

La représentation traditionnelle de la cascade semble être bidimensionnelle, mais il n'en est rien. Elle représente juste le temps qui s'écoule le long des flèches. Certaines présentations montrent des flèches bidirectionnelles, ce qui éloigne encore un peu plus l'intuition temporelle, mais évoque plutôt la capacité de communiquer entre boîtes successives.



Cela a l'air très raisonnable, mais cette modélisation ne dit rien de ce qu'on doit faire quand la vérification/validation échoue. En effet, en cas d'échec il faut en chercher la cause dans toutes les étapes qui ont précédé : analyse, codage ou intégration, voire vérification/validation elle-même. On peut bien-sûr raffiner le modèle en remontant une partie de la vérification juste après le codage, mais ça ne change pas l'idée générale.

② La prise en compte des risques

Une fois les risques identifiés, et sans chercher à les éliminer, on peut réfléchir à comment en réduire l'impact.
• Les **échéances** : le chiffrage des coûts, et particulièrement du temps nécessaire à un développement logiciel, est une des problématiques la plus difficile qui soit. Il faut faire en sorte que même si on s'est trompé, un développement qui s'interrompt à l'échéance due ait produit quelque chose de pertinent du point de vue du client, ne serait-ce que pour être en meilleure position pour négocier une prolongation.
• **L'analyse** : ne jamais prétendre qu'on a compris la demande, ne même pas en faire un objectif, et soumettre sa compréhension régulièrement au demandeur, donc toujours présenter au demandeur une réalisation de bout en bout, même très incomplète. Rappel : le demandeur n'est pas forcément informaticien, et on doit lui parler dans son langage pour le convaincre.
• Le **codage** : toujours suspecter qu'un programme est faux et le vérifier/valider en continu, donc l'intégrer/déployer en continu. Ne pas croire que bien réfléchir a priori dispense de vérifier a posteriori.
• La **sous-traitance** : inverser les rôles du demandeur et du développeur.
• La **technologie** : commencer par ce qui est le moins maîtrisé, faire des maquettes purement techniques juste pour se former à la technologie.
• Les **ressources humaines** : se méfier du mode copain, qui rend aveugle, inclure la formation dans le cycle de vie, veiller à la durabilité de l'effort, ménager des moments de détente et de satisfaction.
• **L'environnement** : rester souple, ne pas se sentir propriétaire du projet, faire une veille technologique/réglementaire/économique/etc., s'assurer, ...

Les démarches agiles ont dès leur origine proposé une vision globale incluant les aspects techniques et les aspects psycho-sociaux. C'est une dimension que l'on perd quand on s'en tient à la lettre du protocole d'une démarche agile particulière sans en comprendre l'esprit.

② CMM ou évaluer la maturité d'une organisation

Dans les années 90, le ***Software Engineering Institute*** de l'Université de **Carnegie-Mellon** propose un modèle pour l'évaluation du niveau de maturité des organisations. Le premier modèle était spécialisé pour le développement logiciel, mais d'autres domaines ont été traités dans les années 2000 sous le nom générique de **CMMI**. Ces modèles s'appellent des ***Capability Maturity Models*** (**CMM**, **+ I** pour ***Integration***). Les modèles CMM distinguent 5 niveaux de maturité :

1. Initial ou **héroïque** : l'issue d'un projet ne dépend que des qualités individuelles des participants, et de la chance. Aucune maturité.
2. Discipliné ou **reproductible** : c'est le premier niveau qui témoigne d'un effort de gestion de projet. Un important apport de ce niveau est que le processus soit reproductible et accumule l'expérience. **Ce qui a marché remarchera**, ce qui n'a pas marché... Comparer avec la situation du TP qui marche un jour, mais ne marche plus le jour de le rendre.
3. Ajusté ou **défini** : les savoir-faire sont documentés, il est prévu de la formation pour les maintenir. Comparer avec la situation du TP qui marche pour un binôme, mais pas pour un autre.
4. Géré quantitativement ou **maîtrisé** : les objectifs sont documentés et la qualité des résultats évaluée par rapport aux objectifs. Comparer avec la situation du TP qui passe les tests des élèves, mais pas ceux du professeur.
5. **Optimisé** : un processus qualité vise à l'amélioration continue des performances en recherchant l'alignement des valeurs **techniques** et des valeurs **business**.

Les ambitions de chaque niveau peuvent sembler très modestes, mais en fait elles sont très difficiles à satisfaire car elles portent sur tous les aspects de l'organisation : ses processus techniques, mais aussi les ressources humaines, le management, etc. Très peu d'entreprises sont certifiées CMM à un niveau élevé, c-à-d. au-delà de 3. Ce qui est évalué est en fait **l'alignement** de tous ces processus.

Évidemment, n'être certifié qu'à un niveau n'interdit pas de mettre en œuvre partiellement les exigences d'un niveau supérieur. La variante **continue** de CMMI permet d'en rendre compte.

N'oubliez jamais !

Contrairement à une idée trop répandue, un programme n'est pas que le moyen pour un humain de **spécifier à une machine ce qu'elle doit faire**, mais c'est surtout le moyen **d'expliquer à un autre humain ce qu'on spécifie à une machine**. En effet, un logiciel (pas un programme jetable genre TP) se construit dans **l'espace**, c-à-d. les multiples intervenants qui collaborent à sa production, et dans le **temps**, c-à-d. les multiples étapes du développement de ce logiciel, possiblement prises en charge par des développeurs qui ne se connaissent pas. Même le cas extrême d'un développeur unique se ramène au cas général quand on considère la difficulté qu'éprouve un programmeur à se relire.

④ Cycle de vie en spirale

Afin de répondre au reproche fait au modèle en V, on propose un modèle en **spirale** où on itère les A, C, ID et VV pas seulement pour répondre aux échecs de vérification/validation, mais d'abord dans une prise en compte **progressive** des exigences de l'application visée. De cette façon, on enchaîne des « tours » de spirale assez rapides, ce qui évite l'effet de tunnel propre au modèle en V.

Le cycle de vie en spirale est la base de l'agilité quand on ajoute l'exigence supplémentaire que ce qui est produit à chaque tour ait du sens dans le monde du client. Ex. si un système logiciel comporte plusieurs couches, il n'est pas question de réaliser la 1ère couche dans une 1ère itération, puis la 2nde dans une 2nde itération, ..., car ces couches n'ont pas de sens pour le client. Elles sont juste le **Comment** imaginé par les développeurs pour la réalisation d'un **Pourquoi** seul compris par le client. Au contraire, il faut produire à chaque itération une réponse partielle au Pourquoi du client qui passe par des fragments de toutes les couches du système.