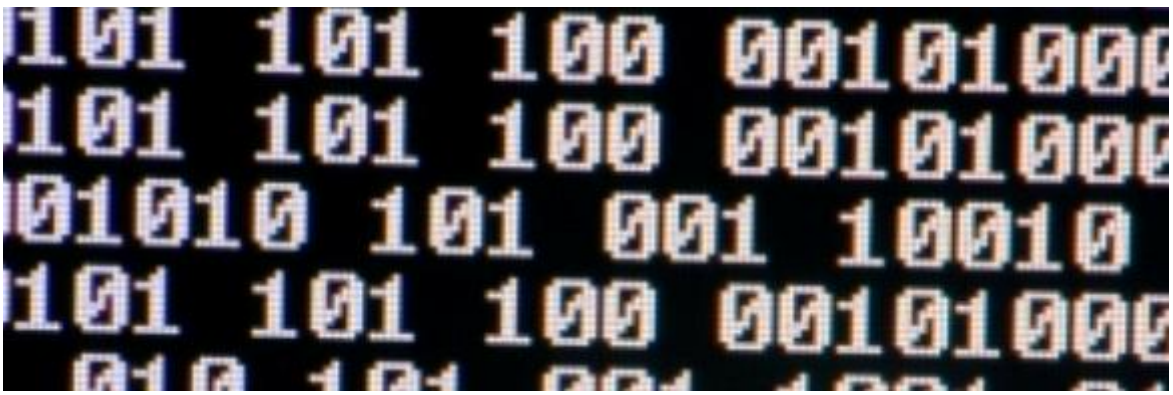
Génie logiciel

Version beta
F. Lemainque
Septembre 2016



Sommaire

Introduction

Est- il donc nécessaire d'être un génie pour créer des logiciels ? Pas exactement : nous parlons ici d'ingénierie logicielle, ou comme vous l'entendez plus souvent de développement logiciel.

De toute évidence, un logiciel n'est pas construit en navigant à vue : nous sommes déjà depuis quelques années dans l'ère de l'industrialisation du logiciel.

Pour trouver sa place, le développeur doit comprendre le processus de développement adopté. Le développement n'est qu'une étape du cycle de vie d'un logiciel (le cycle de vie d'un logiciel, c'est l'ensemble des phases ou étapes par lesquels il passe entre le moment où on a l'idée de le construire et le moment où on le retire de l'exploitation).

Et le codage n'est qu'une des activités du développement d'un logiciel. Développer, c'est intuitivement (ou non) concevoir, programmer, mettre au point, vérifier et valider, documenter. On pourrait toutefois aller bien plus loin :

▪ Analyse des besoins ▪ Étude de faisabilité ▪ Spécification des besoins	Tâches de pré-conception
▪ Conception du système (conception architecturale détaillée) ▪ Planification du projet et contrôle du déroulement ▪ Validation du développement	Conception : le développement au sens strict
▪ Formation des utilisateurs ▪ Maintenance et/ou évolution etc.	Tâches de post-conception

Un développeur logiciel peut donc être amené à exercer de nombreuses tâches différentes, pas uniquement à coder sur son ordinateur…

1 L'étape de pré-conception

De toute évidence, le développement d'une application impliquant une trentaine de personnes sur deux ans ou celui d'un site Web ultra simple exigeant une quinzaine de jours pour un concepteur Web n'ont rien à voir. Pourtant, les étapes, les prérequis et les exigences sont exactement similaires.

Voici en quoi se décompose cette phase indispensable de préconception :

- **Identification des besoins** – la première étape consiste bien évidemment à recueillir les désirs et les besoins du client, ce qui implique le plus souvent d'interroger non seulement les dirigeants mais également les futurs utilisateurs finaux.
- **Reformulation et validation des besoins** - tout commercial apprend vite (à ses dépends) qu'il existe souvent une énorme différence entre les besoins énoncés par un client et ses besoins réels. Le fait qu'en réalité il ne s'agisse pas des besoins énoncés mais des besoins compris par l'interlocuteur y joue un grand rôle : il est capital de reformuler ces besoins pour s'assurer d'une bonne compréhension mutuelle, et de les faire valider.
- **Détermination des moyens à mettre en œuvre** – les besoins une fois définis, il devient possible de chercher à sélectionner les outils les plus adaptés pour répondre à la demande, afin de pouvoir établir une proposition concrète et matérielle, avec détermination d'un délai et d'un coût financier.
- **Opportunité de la réalisation** – cette étape consiste à comparer les avantages espérés de la réalisation et son coût. Parfois (souvent ?) cela aboutit à une redéfinition des besoins et/ou des moyens à mettre en œuvre.

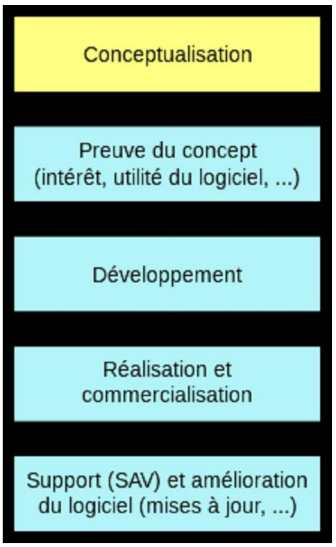
Ce n'est qu'une fois ces prérequis (très souvent sont rassemblés sous le simple titre « détermination des besoins ») satisfaits que le développement peut réellement prendre place.

2 Les différents modèles de développement logiciel

Au fil du temps plusieurs méthodes de développement sont apparues, chacune possédant ses avantages et inconvénients et/ou correspondant parfois à une mode, ce qui ne veut pas dire que chacune ne possède pas ses farouches défenseurs… On peut les répartir en quelques grandes familles.

Modèle en cascade

Hérité de l'industrie du BTP, il part du principe qu'on ne peut pas construire une toiture avant les fondations et que les conséquences d'une modification en amont du cycle ont un impact majeur sur les coûts en aval.

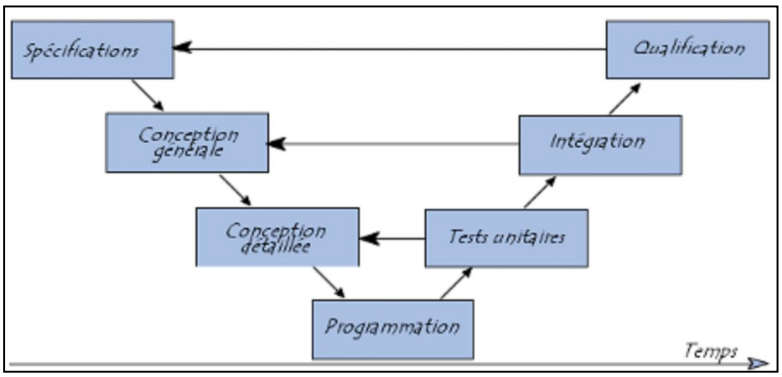


Le développement s'effectue par phases successives, avec un retour sur les précédentes, voire au tout début du cycle. Ces phases possèdent les caractéristiques suivantes :

- la production est définie au préalable ;
- la date de fin est précise ;
- la fin n'intervient que lorsque la livraison est jugée satisfaisante lors d'une étape de validation-vérification.

Modèle en V

Le modèle en V répond au manque de réactivité du modèle en cascade. Il permet en cas d'anomalie de limiter un retour aux étapes précédentes. Les phases de la partie montante doivent renvoyer de l'information sur les phases en vis-à-vis lorsque des défauts sont détectés afin d'améliorer le logiciel.



Ce modèle met en outre en évidence la nécessité d'anticiper et de préparer dans les étapes descendantes les « attendus » des futures étapes montantes : ainsi les attendus des tests de validation sont définis lors des spécifications, les attendus des tests unitaires sont définis lors de la conception, etc. Il est devenu un standard de l'industrie du développement logiciel depuis les années 1980.

Ces deux modèles sont souvent qualifiés de « modèles lourds » (sans que cela ne soit forcément péjoratif). Voici leurs avantages et leurs inconvénients :

Avantages	Inconvénients
▪ les efforts de réflexion investis dans les phases en amont font gagner du temps dans les phases en aval ▪ logiciel = code source + documentation : les deux ont la même importance ▪ processus structurés, donc « simple » à organiser, expliquer, suivre,prédire	▪ impose de produire un design parfait du premier coup ▪ le logiciel apparaît tard (effet tunnel) ▪ pas adaptatif (les retours en arrière sont très coûteux) ▪ orienté projet et outils, ne tient pas compte de l'équipe

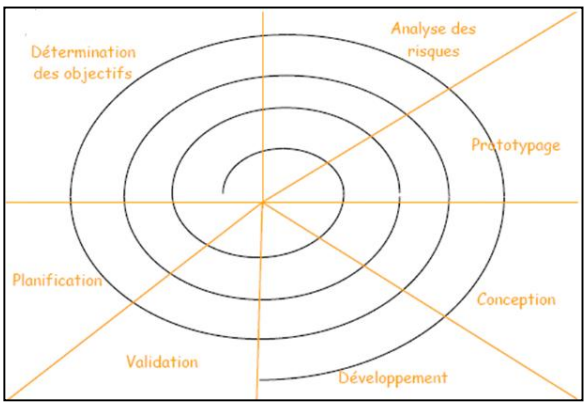
Si en théorie ce sont de bons modèles, ils se révèlent difficiles à utiliser en pratique. La plupart du temps, deux principaux reproches sont formulés à l'encontre de ces méthodes :

- Le temps passé à rédiger la documentation. Cette rédaction nécessite l'intervention de nombreuses parties à des moments clefs mais pas à chaque étape du projet.
- Conséquence de la remarque précédente, les documents produits sont réputés immuables pendant toute la durée du projet, d'où l'effet tunnel, préjudiciable au projet lui-même. En effet, selon la durée du projet, il est plus que probable qu'entre la rédaction du Cahier des Charges et la livraison, quelques changements interviennent dans l'environnement cible de l'application. Ou que de nouvelles contraintes techniques (internes ou externes) apparaissent, contraignant l'équipe de développement à revenir sur les fonctions escomptées.

L'utilisation de documentation étant intensive, le processus de développement en devient assez rigide. Les membres des différentes équipes auront tendance à s'appuyer exclusivement sur les spécifications publiées, comme d'un rempart à la critique. Cette pratique les amène à se "désimpliquer" du but ultime du projet : répondre aux besoins des utilisateurs finaux, y compris en termes de performances et, surtout, d'évolutivité.

Modèle en spirale

Le développement reprend les différentes étapes du modèle en V. La mise en œuvre de versions successives permet d'aboutir à un produit de plus en plus complet et robuste.

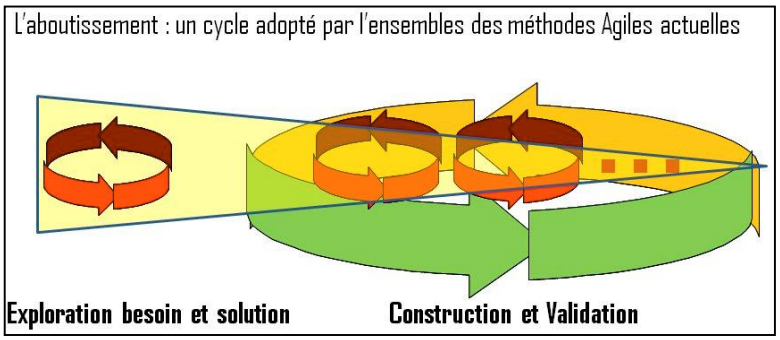


Le cycle en spirale insiste toutefois plus sur la gestion des risques que le cycle en V, ceux-ci étant analysés au début de chaque itération. Cela est nécessaire car un développement cyclique présente plus de risques de défaire, au cours de l'itération, ce qui a été fait au cours de l'itération précédente.

Modèle semi-itératif

La méthode Agile en est un exemple. Ce cycle de développement fut formalisé en 1991 dans le livre RAD (Développement rapide d'applications). À partir de 1994, des travaux complémentaires menés en France (RAD) et en Angleterre (DSDM) apportèrent des spécialisations aux techniques basiques initiales. L'adoption par RUP (*Rational Unified Process*) d'IBM consacra l'apogée de ce cycle toujours en vigueur dans les projets importants.

Dans le cycle semi-itératif les deux premières phases classiques (*top down*, par la structure) consistent en l'expression des besoins et la conception de la solution. C'est lors de la troisième et dernière grande phase, la construction du produit (*bottom up*, par le besoin) que la notion d'itérations courtes intervient. C'est vers 2001 avec l'apparition de plusieurs méthodes dont ASD, FDD, Crystal, Scrum ou l'*extreme programming* et la vision uniformisée de leurs auteurs dans le cadre du Manifeste Agile (Agile Manifesto) et de l'Agile Alliance, que le cycle semi-itératif fut généralisé. Nous reviendrons sur ces méthodes.

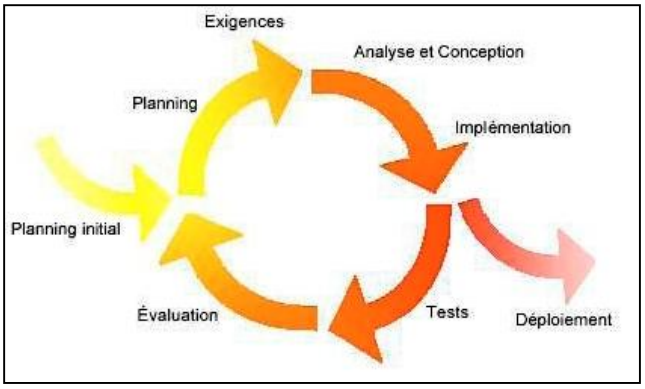


Modèle itératif

On sépare les activités des produits issus de celles-ci. Ainsi, on applique un cycle sur la production d'une documentation, d'un composant, d'un test, etc.

Rapportée à une activité de type gestion de projet, la première phase sera celle de :

- la faisabilité : l'acceptation d'un nouveau besoin
- l'élaboration : on imagine comment on va le réaliser
- la fabrication : construction
- la transition : tout est mis en œuvre pour livrer au client



Il n'y a pas correspondance stricte avec le modèle en V. L'idée est de livrer au plus tôt quelque chose qui puisse être testé par le client. On peut en effet réaliser plusieurs itérations sur une documentation telle que l'architecture. De la même manière, si un document n'est qu'un artefact parmi d'autres, il ne faut pas obtenir un document complet. On préférera utiliser la loi de Pareto : ne pas passer 80 % de l'effort sur les 20 % restants.

3 Les méthodes Agiles

On s'est aperçu depuis la mise en place du cycle en V que les équipes de développeurs sont faillibles. Elles peinent notamment à tenir un rythme. On voulait industrialiser la production de logiciels et, pour cela, créer des chaînes de montage (voir le Taylorisme…). Hélas, l'humain est inconstant. Et les groupes d'humains le sont encore plus.

Quelques acteurs de l'industrie logicielle ont donc décidé de se réunir dans les années 90 pour réfléchir à leurs conditions de travail. Cette réflexion a donné naissance à un Manifeste Agile (http://agilemanifesto.org) qui a permis d'imaginer une autre manière de travailler en s'appuyant sur les personnes plutôt que sur la documentation, les contrats et la planification à long terme.

Au cœur de ces concepts se trouvent le Cycle Itératif et Incrémental ainsi que l'Intégration Continue. Ses grandes orientations sont les suivantes :

- Orientation vers l'adaptabilité (réactivité aux changements dans les besoins du client pendant le déroulement du projet) plutôt que vers la prédictibilité
- Centrage sur les développeurs plutôt que sur le processus
- Viser en premier lieu à produire du logiciel fonctionnel plutôt qu'une documentation complète.
- Intégration du client au développement (distribution fréquente, négociation avec lui sur la conduite du projet)

Le Manifeste Agile décrit notamment les principes suivants :

- acceptation des changements dans l'expression des besoins
- itérations de 2 semaines à 2 mois, avec livraison d'une brique logicielle fonctionnelle
- collaboration journalière entre développeurs et maîtrise d'ouvrage
- disposer de développeurs motivés, bien équipés, et adopter un rythme de travail soutenable
- communication en face-à-face (et pas par documents)
- mesurer de l'avancement par la portion du logiciel en état de fonctionner
- viser l'excellence technique et une conception de qualité
- toujours favoriser la simplicité

Méthode SCRUM

C'est une des méthodes Agile les plus populaires. Elle définit plusieurs rôles au sein de l'équipe projet :

- Le **Product Owner** (PO) : garant de la vision du projet, il administre la liste des fonctions à délivrer (*product backlog*), sous une forme synthétique que l'on appelle des **Stories**
- Le **Scrum Master** (SM) est l'animateur de l'équipe qui s'assure de la distribution des tâches et du rythme de développement, par exemple en fournissant un support d'architecture et de programmation
- Le **Developer** est réputé polyvalent ; certains développeurs expérimentés sont appelés des Champions (c'est toujours bon pour l'égo)
- Le **StackHolder** est un acteur externe au développement mais qui suit l'avancement du projet ; il peut s'agir d'utilisateurs finaux, ou de responsables hiérarchiques par exemple

Des échanges journaliers (la plupart du temps matinaux et COURTS) appelées **scrums** assurent que l'information est bien partagée entre les membres de l'équipe et permettent de détecter les problèmes éventuels.

Une cérémonie de lancement d'un **Sprint** (quelques semaines de développement) permet d'évaluer la charge de travail nécessaire pour réaliser les fonctions (*user stories*) jugées prioritaires et préciser la vision du **Product Owner** à l'instant présent. En fonction de l'estimation donnée par les développeurs eux-mêmes, le PO peut décider de revoir l'ordre de ses priorités.

A la fin d'un **Sprint**, l'équipe organise une **Sprint Review** (démonstration) des fonctions implémentées à l'intention de toutes les parties prenantes ; ceci afin de valider la direction prise par le projet.

Enfin, des Rétrospectives permettent de faire un point régulier sur les problèmes rencontrés, qu'ils soient techniques ou méthodologiques, et d'ajuster le fonctionnement de l'équipe pour en améliorer le rendement.

C'est grâce aux Sprints (courts et intenses) que l'adaptation au changement est rendu possible dans les méthodes agiles. On supprime ainsi au maximum l'effet tunnel décrié dans le Cycle en V et l'on minimise le coût éventuel d'un retour en arrière. La documentation pléthorique et contractuelle est remplacée par le dialogue et la négociation. Les développeurs sont amenés à confronter leur travail à une vision élevée de l'application et à la réutilisation de leur propre code au sein de l'équipe.

L'eXtreme Programming

C'est un processus de développement Agile formalisé par K. Beck vers le milieu des années 90. Centré sur les développeurs , il permet d'avoir une vision « extrême » de leur travail.

L'eXtreme Programming (ou XP) est fondé sur quatre valeurs fondamentales :

- **Communication** : s'assurer que tout le monde (développeurs, MOE, MOA) est sur la même longueur d'onde
- **Simplicité** : favoriser la solution la plus simple (ne pas résoudre les problèmes qui ne se posent pas, ou encore : l'art de maximiser la non réalisation de ce qui n'est pas nécessaire)
- **Feedback** : retour permanent sur le travail des développeurs par le système (tests unitaires), par le client (validation/qualification/choix des priorités), par le reste de l'équipe (code collectif, revues de code, etc.)
- **Courage** : savoir défaire/jeter ce qui n'est plus adapté

Ces quatre valeurs se déclinent en bonnes pratiques :

- **retour permanent sur le travail** (*fine scale feedback*) : *pair programming*, *planning game*, développement guidé par les tests, interaction du client à l'équipe
- **processus continu** : intégration continue, *refactoring* continu, livraison par petits morceaux
- **partage des connaissances** : standard de codage, propriété collective du code, conception simple, utilisation de métaphores
- **bien-être des programmeurs** : garder un rythme de travail « supportable »

Bien évidemment, les méthodes Agiles, si elles présentent des avantages certains ne sont pas dénuées d'inconvénients :

Avantages	Inconvénients
▪ Production d'un logiciel simple, sain, adapté aux besoins et évolutif ▪ Capacité à réagir aux changements dans l'expression des besoins	▪ Pas ou peu structuré : difficile à prédire, pas adapté aux développeurs peu expérimentés ou aux grandes équipes ▪ Implication nécessaire du client : pas toujours applicable (problème de compétence, de disponibilité, d'habitude) ▪ Construit pour l'adaptabilité : pas optimal pour les projets cadrés

4 Pratique du développement logiciel

Lorsqu'une entreprise a besoin d'un logiciel sur-mesure pour son activité, elle contacte un prestataire de services informatiques (SSII). Le travail de celui-ci peut se subdiviser en deux phases (et ce quel que soit le modèle de développement qu'il suit) :

Conception du logiciel

1. Cahier des charges

C'est la première étape du développement spécifique. Il faut concevoir le cahier des charges en fonction des besoins/désirs du client.

2. Conception générale

A partir du cahier des charges il est défini les langages et éléments à utiliser, ainsi que la durée de développement estimée. Cela aboutit à une proposition chiffrée présentée au client pour validation du projet.

3. Conception détaillée

Le projet entre en phase de pré-réalisation. Son architecture, ses fonctions et ses composants sont précisément détaillés.

Création du logiciel

4. Codage des ensembles

C'est là que débute le développement réel. Selon la méthodologie retenue, plusieurs équipes peuvent travailler en collaboration. Dans le cadre des Méthodes Agiles, une équipe unique est toujours préférable afin de favoriser la communication indispensable.

5. Tests avant intégration

A chaque étape de la production des tests doivent être réalisés pour garantir l'efficacité du résultat. Les tests individuels de chaque module sont indispensables. L'efficacité de la solution complète pourrait dépendre d'un unique composant imparfait.

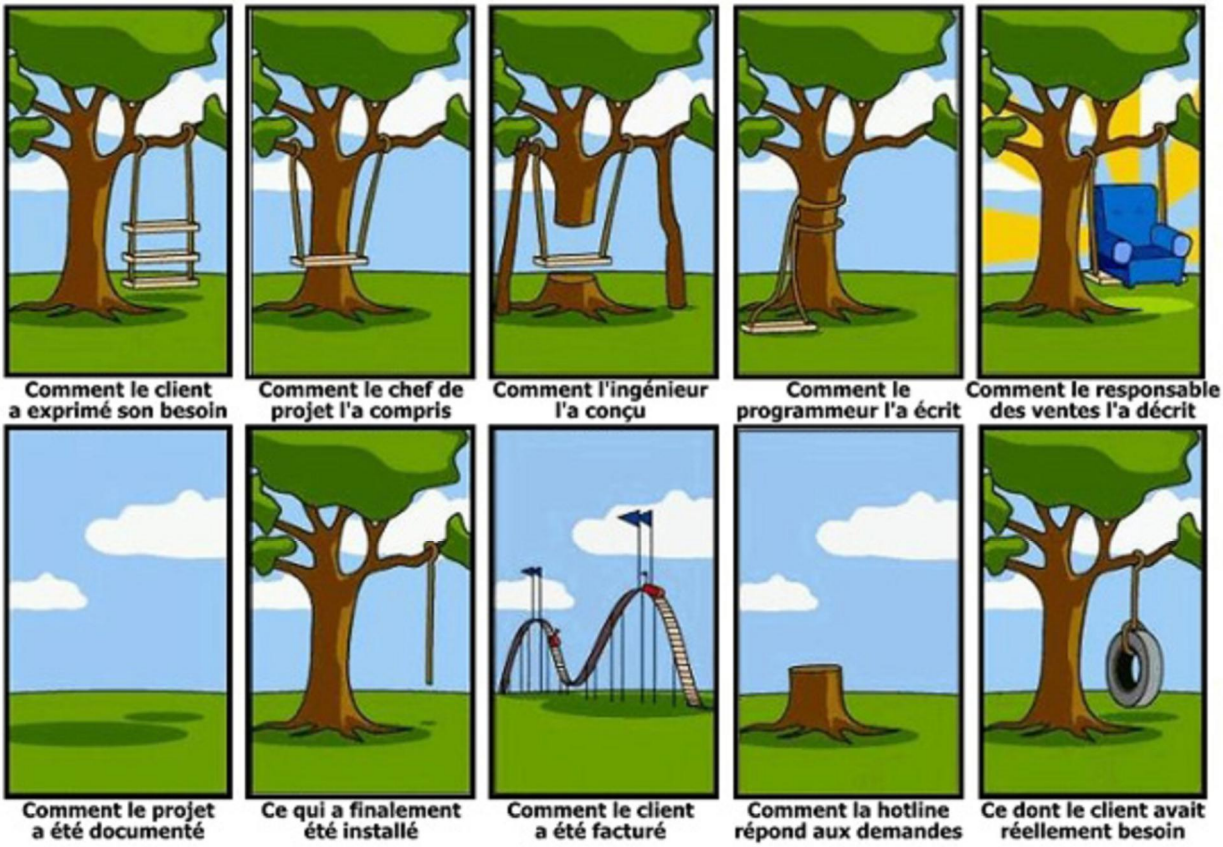
6. Intégration, tests complets et qualification

Les différents modules sont assemblés et les interfaces créées. Des tests sont effectués lors de l'intégration de chaque composant afin de garantir la compatibilité des modules. Lors des tests finaux, on vérifie la conformité de l'application avec le cahier des charges.

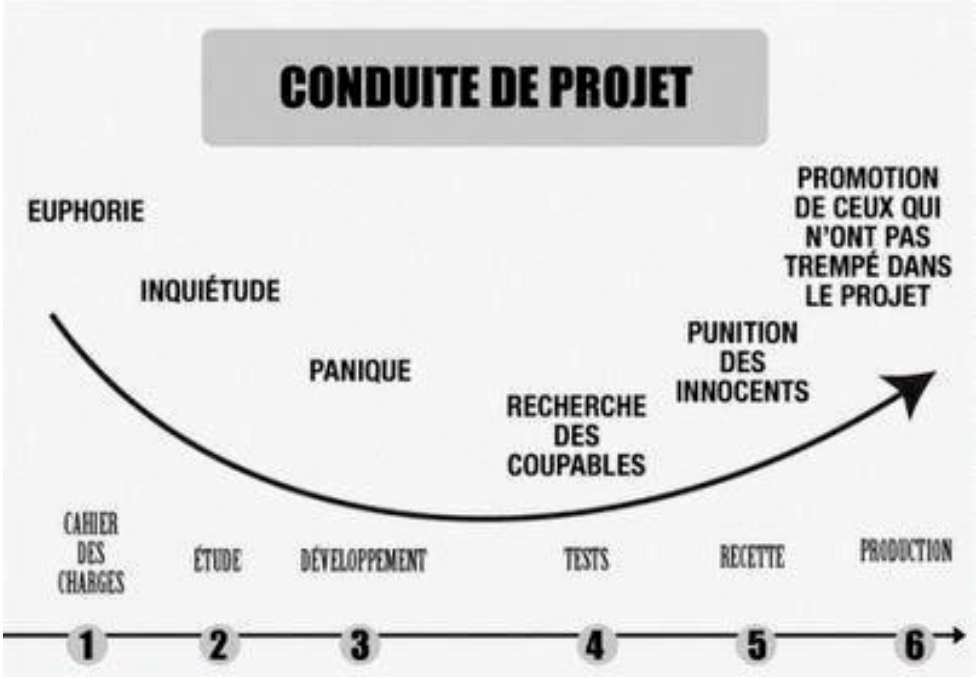
7. Implémentation et documentation

Si les tests sont satisfaisants, on passe aux dernières étapes du développement spécifique : l'implémentation et la documentation. Puis enfin le nouveau produit est installé chez le client.

Le besoin client et la balançoire



Les phases d'un projet, une affaire de sentiments



5 Développement d'une application mobile

De nos jours, il semble presque illusoire de penser pouvoir seul développer une application susceptible de connaître un avenir commercial radieux. Toutefois, la croissance exponentielle des smartsphones, phablettes et autres tablettes montre l'intérêt des applications mobiles, dont le développement reste encore pour le moment à la portée d'une équipe même restreinte. Encore faut-il s'y prendre avec méthode…

Le développement d'applications mobiles devient une industrie à part entière. La tendance récente est très claire : les utilisateurs plébiscitent les applications mobiles au détriment des sites web. Toutes les entreprises sont maintenant touchées par ce besoin : avoir son application mobile.

Ce mouvement de fond est très similaire, en apparence, au mouvement d'il y a une dizaine d'années, où chaque entreprise se devait d'avoir son site internet. On pourrait alors considérer le développement d'applications mobiles comme similaire à la création de sites internet. Or plusieurs différences fondamentales rendent un projet d'application mobile plus complexe, donc potentiellement plus coûteux.

1. Définition de la stratégie

La première étape consiste bien sûr à définir votre stratégie, à cerner le besoin. Pour cela, commencez par vous poser les questions suivantes :

- Quel est l'objectif de mon application ?
- Quel est son public ciblé ?
- Quel service veut-on fournir ?
- Si l'application doit compléter un site web, que va-t-elle apporter de plus ?
- Quelle est la plus value de mon application par rapport à ce qui existe sur le marché ? De nombreuses applications sont déjà disponibles sur les stores, il est possible –voire probable- que votre idée ait déjà été développée. Cela se révèle également être une aide pour se démarquer.

2. Choix de la technologie

Vous devez clairement établir sur quel type de matériel l'application doit (pouvoir) fonctionner : smartphone, phablette, tablette, ordinateur ? Cela conditionne en partie le choix de la technologie : Web (conception adaptative), Hybride ou Natif ? Si votre projet insiste sur l'expérience utilisateur, sa présence sur les stores (App Store, Google Play, Windows Store), l'utilisation de fonctions telles que les SMS, le téléphone, l'appareil photo ou le GPS, qu'elle requiert un mode hors ligne ou nécessite de stocker des données, le choix d'une technologie hybride ou native s'impose. Si votre application nécessite une bonne réactivité et de bonnes performances, avec une expérience utilisateur poussée, il sera

préférable de partir sur une technologie native. Il en va de même si vous voulez ne cibler que les deux principaux OS du marché (iOS et Android).

Le minimum obligatoire est d'éditer une application mobile pour les deux principaux systèmes. Ceux-ci diffèrent toutefois profondément différents tant pour l'interface utilisateur que pour le langage de programmation : Objective C pour iOS, Java pour Android.

Cela oblige donc soit à développer chaque version en « natif », ce qui équivaut à pratiquement doubler le temps (et le budget) de développement, ou à employer un environnement « multi-OS » comme PhoneGap ou Appcelerator Titanium qui promet de développer un « code source » unique et de générer instantanément plusieurs applications pour tous les systèmes mobiles.

Cette promesse se heurte aujourd'hui à plusieurs risques : d'abord la performance pour les animations et graphiques, ensuite les problèmes induits automatiquement par l'ajout d'une couche logicielle « au dessus » de l'OS mobile : bugs supplémentaires et délai d'adaptation quand une nouvelle version d'iOS ou d'Android est publiée. D'ailleurs, Facebook et LinkedIn ont d'abord développé « multi-OS » avant de revenir à une stratégie « native ».

3. Approfondissement du concept

C'est le moment de décrire en détail votre concept d'application. Comment y parvenir ? Grâce à un atelier créatif (*brainstorming*) pour dégager les idées et à un travail de conception pour dégager les fonctionnalités de l'application. C'est aussi le moment où vous effectuez des choix capitaux en matière économique : faut-il privilégier la visibilité, le retour sur investissement ? Comment monétiser l'application ?

4. L'expérience utilisateur d'abord

Envisagez la façon dont les utilisateurs vont utiliser l'application et définissez leur parcours au sein de l'application en privilégiant un parcours simple et rapide.

- Détailler les cas d'utilisation et les flux de l'expérience utilisateur
- Concevoir les maquettes visuelles. Dans le cadre d'une application mobile, l'expérience (comment l'utilisateur utilise l'application) et l'interface utilisateur (la conception) revêtent une importance capitale. Recueillir des retours d'utilisateurs potentiels et chercher à améliorer ces maquettes en fonction de ces retours est souhaitable.
- N'oubliez pas de prévoir les différentes tailles d'écran : de l'ordinateur de bureau au smartphone en passant par la phablette et la tablette, pour prendre tous les extrêmes. Aujourd'hui, toute conception se doit d'être « adaptative ».
- Quel budget prévoir ? En fonction des fonctionnalités, du parcours utilisateur, de la complexité du concept et du nombre de plateformes, un budget pourra être établi. Ceci permettra de définir un plan de route d'évolution des versions ultérieures.

5. Conception

Vous devez à ce stade connaître votre ou vos plateformes cibles. Les fournisseurs comme Google, Apple ou Microsoft proposent des règles (*guidelines*) : ce sont les grandes lignes de chaque fournisseur en termes de design (icônes, couleurs, polices), d'expérience utilisateur (emplacement des menus, icônes, etc.) et de bonnes pratiques de développement (structure du code, nommage, etc.). Chaque OS aborde différemment les problématiques UI/UX, il est important de suivre ces règles pour espérer conquérir le plus grand nombre d'utilisateurs. Cette partie est capitale : c'est elle qui rend l'application plus visuelle et constitue l'interface des utilisateurs.

Pensez au comportement de l'application en mode « déconnecté », lorsque le mobile n'a pas accès à Internet. Il devient inacceptable pour un utilisateur de ne pas pouvoir utiliser son application, même quand il n'est pas connecté. Or accéder à des données, les modifier ou même en créer de nouvelles hors connexion rend nécessaire le développement de logiques de « cache mémoire » et de synchronisation, relativement complexes.

6. Mesure des performances

Mettre en place les balises de mesure de performance de l'application et des statistiques qu'il faut faire remonter. Ce tableau de bord permet de savoir où se trouvent les points faibles de l'application.

7. Le développement au sens strict

Tout est prêt pour le développement. En natif, hybride ou en web, l'application est maintenant en phase de développement. Il existe même des frameworks de développement qui permet de décliner plus facilement les applications entre plateformes.

8. Tester, Tester, Tester !

Au cours du développement (et pas uniquement à la fin, lorsque l'application est développée), selon les technologies utilisées, vous devez utiliser des outils de tests pour vos applications (tests unitaires, tests de performance (par exemple Monkey pour Android), etc.). Une batterie complète de tests est nécessaire afin d'éviter tout problème susceptible de l'être (évité). Testez votre application sur plusieurs OS et versions d'OS, plusieurs matériels, plusieurs tailles d'écran, … Et n'hésitez pas à la faire tester par des utilisateurs au fil du projet.

9. Lancer l'application sur les stores

L'application a été testée et est prête à être placée sur les stores. Préparez cette mise en ligne en choisissant un nom sympathique, une icône attractive, un descriptif impactant, des visuels saisissants… Ensuite seulement vous pouvez la déployer sur les stores, mais cela à un coût : vous devez obtenir une licence pour chaque fournisseur. Au moment de la rédaction de ce document, la licence revient à 25 $ pour Google Play (paiement unique, valide à vie), 99 $/ an pour Apple, 19 $/ an (particuliers) ou 99 $/ an (professionnels) pour la licence Microsoft.

10. Et ensuite ?

Dans le meilleur des cas votre application décolle et vous emmagasinez les téléchargements – avec les revenus qui les accompagnent. Il faut toutefois penser à faire évoluer votre application en y apportant les modifications nécessaires, réparer les bugs et surtout écouter les ressentis des utilisateurs.

Pour une entreprise, ou un projet personnel, ces éléments doivent être pris en compte pour estimer au mieux les coûts, le retour sur investissements, ainsi que s'assurer de la bonne réussite du projet en termes d'utilisation.