Chapitre 1 - Gestion d'un projet informatique et cycle de vie

BUT 2 - R3.03 Analyse

Thomas Dufaud
thomas.dufaud@uvsq.fr

Septembre 2022

Nous présentons le cadre considéré pour traiter de la gestion de projet informatique en précisant les contraintes visées, ainsi que le cadre théorique du génie logiciel au travers de sa définition et des activités du génie logiciel. Dans une troisième section nous présentons les cycles de vie du logiciel et les transformations qui leurs sont applicables pour gagner en efficacité. Ce chapitre a donc pour objectif de pouvoir prendre du recul sur l'organisation d'un projet informatique, il est complémentaire des éléments de formation se concentrant sur la gestion et tente d'apporter des réponses quant à l'application des techniques de gestions de projet en informatique. Les chapitres suivant porteront sur les activités d'analyse des besoins et de spécification.

1 Gestion de projet informatique

1.1 Développer des systèmes complexes

Dans ce cours, nous nous intéressons à la gestion de projet informatique pour la production de solutions informatiques sous fortes contraintes de qualités. Par exemple intéressons nous au développement logiciel dans le cadre des développements des systèmes d'information des organisations. La définition que nous considérons ici est la définition 1.1 proposée dans [Reix *et al.* 2016]. Cette définition de type organique précise la nature hétérogène du système et les tâches qu'il accomplit. Bien que de nos jours nous puissions manipuler de l'information numérique via des systèmes informatiques, il est possible également de manipuler de l'information d'une autre nature et avec d'autres outils que l'outil informatique. Il est par exemple possible de collecter une information en l'écrivant sur un support papier, de conserver les feuilles manuscrites, et de les diffuser en les photocopiant. L'outil informatique est donc une possibilité et non une nécessité, et parmi les ressources, le logiciel est une ressource possible. Un système informatique fait donc partie d'un système d'information mais ne constitue pas nécessairement un système d'information.

Définition 1.1 (Système d'information (SI) source : [Reix *et al.* 2016])
Un système d'information est un ensemble organisé de ressources (matériels, logiciels, personnel, données et procédures) qui permet de :

- *acquérir l'information (sur différents supports)*
- *traiter l'information (effectuer des opérations)*
- *stocker l'information (conserver l'information)*

— *communiquer l'information (transmettre : éditer, imprimer, etc.)*

En conclusion, les outils de l'informatique permettent de réaliser les spécifications d'un SI. Plusieurs systèmes informatiques peuvent interagir pour réaliser ces spécifications, ou remplacer d'autres outils. Nous nous intéressons donc ici aux méthodes nécessaires au développement des applications utiles aux systèmes d'information, et s'intégrant de fait dans un environnement complexe avec différents acteurs.

Afin de pouvoir proposer une méthodologie pour la gestion de projet informatique dans le contexte des systèmes d'information, il nous faut définir le cadre théorique spécifique à l'informatique et la manière dont on peut découper les activités d'un développeur. Dès lors il est possible de proposer différentes recettes, c'est à dire différents enchainements d'activités, pour réaliser le projet. Cet enchaînement définit ce que l'on appelle le cycle de vie logiciel. Il n'y a pas de cycle unique. Pour chaque projet, et chaque équipe, il est possible d'adapter son cycle de vie logiciel et définir un processus métier particulier pour le développeur.

1.2 Le Génie logiciel

Le génie logiciel nous permet de définir une méthodologie pour la réalisation d'un logiciel. Nous considérons la définition 1.2.

Définition 1.2 (Définition du Génie Logiciel (GL))
Le Génie Logiciel est la discipline liée à tous les aspects de la production du logiciel complexe et avec d'importantes contraintes de qualité. Elle est liée à l'application de théories, de méthodes et à l'utilisation d'outils pour le développement logiciel d'une façon professionnelle. Elle favorise et permet le travail en équipe.

Les objectifs du génie logiciel sont :

— d'adopter une approche systématique et organisée.
— d'utiliser les techniques et outils appropriés selon la nature du problème, les contraintes de développement et les ressources.

On considère ici le développement de systèmes logiciels comme un processus industriel.

Afin de définir une méthodologie, le GL pose les principes suivants :

1. rigueur et formalisme
2. abstraction
3. modularité (décomposition en sous-système)
4. anticipation
5. généralisation
6. croissance incrémentale.

La **rigueur et le formalisme** permettent de communiquer et discuter en équipe et avec le client. La capacité d'**abstraction** permet d'identifier et définir les concepts du projet ainsi que les éléments qui le compose. À partir de l'**abstraction** on peut réaliser un modèle du système. Ce modèle peut, en appliquant le principe de **modularité**, proposer une décomposition en sous-système. Dès lors chaque sous-système peut constituer un sous projet, on peut définir les dépendances entre

ces sous projets et réaliser le projet par étape en réalisant les dépendances entre sous-systèmes dans l'ordre. Le principe d'**anticipation** implique de prendre en compte les multiples contraintes d'un projets tant techniques qu'organisationnelles, et de penser différentes possibilités d'organisation du projet en fonction des difficultés rencontrées au cours du projet. L'**anticipation** est facilité par l'application des principes précédents.

Puis vient la **généralisation**, qui permet d'envisager une utilisation plus générale des composants d'un système ou des procédures de réalisation.

Enfin, la **croissance incrémentale** est un principe à viser si l'on veut pouvoir faire évoluer le système par étape.

1.3 Les activités du Génie Logiciel

Pour mener à bien un projet informatique, le développeur ou l'équipe de développement doit réaliser un ensemble d'activité allant de l'analyse des besoins du projet à la maintenance du logiciel produit en passant par son implémentation et les tests. Il faut pouvoir organiser ses activités et les enchaîner de manière à produire un logiciel respectant certains critères de qualité.

Nous proposons ici une description de ces différentes activités. Elles sont décrites de façon indépendantes en indiquant les données, le rôle et les résultats. Elles utilisent et produisent des *artefacts*. Selon le modèle, une activité peut jouer un rôle prépondérant ou pourrait ne pas exister.

Les activités du GL interagissent selon certains **modèle** (cycle de vie). Elles sont développées en appliquant une **méthode**. La méthode est basée sur des **principes**. L'application de la méthode peut être aidée par des **outils**.

On distingue les activités suivantes :

a) Analyse des besoins
b) Spécification
c) Conception
d) Programmation
e) Validation et vérification (Tests)
f) Intégration et gestion des configurations

a) Analyse des besoins (ADB)

- **Rôle** :
  - Identifier le problème
  - Documenter les exigences
  - Impliquer les utilisateurs et les experts dans le domaine de l'application
- **Entrées** :
  - Cahier des charges (rédigé par le client)
  - Données fournies par les experts du domaine et par les utilisateurs potentiels
- **Résultats** :
  - Document d'ADB destiné aux utilisateurs et utile pour les développeurs.
  - Recueil des besoins
  - Cahier des charges (rédigé de manière technique avec le client)

Remarque 1.3 *Attention, de nombreux produits ne correspondent pas aux besoins du client. On peut identifier plusieurs causes, comme par exemple :*

- *Le client ne sait pas ce qu'il veut*
- *Il existe des problèmes de communication entre le client et l'équipe de développement.*
- *L'équipe de développement ne comprend pas la politique d'organisation du client*
- *On observe un changement des exigences au cours du projet*
- *Les délais ne sont pas raisonnables*
- *etc.*

D'où l'importance cruciale de cette étape.

b) Spécification

- **Rôle** :
  - Décrire de façon précise le système à construire, **le QUOI** et non le comment.

- **Entrées** :
  - Document d'ADB destiné aux utilisateurs et utile pour les développeurs.
  - Recueil des besoins
  - Cahier des charges (rédigé de manière technique avec le client)

- **Résultats** :
  - Documentation textuelle avec description de scenarii (cas d'utilisation) possiblement appuyée par des diagrammes (ex : std UML)

Remarque 1.4 — ***dans certains modèles elle peut remplacer la conception***
- *dans certains cas elle peut être réalisée dans l'ADB*
- *elle peut être nécessaire dans plusieurs étapes du processus (différent niveau d'abstraction)*
- ***elle peut être rédigée selon plusieurs standard** : en effet comme elles doivent être comprise par le client ou les utilisateurs, il faut tenir compte de leurs métiers et des standards qu'ils maîtrisent.*

c) Conception

- **Rôle** :
  - Enrichir la spécification du logiciel en l'orientant vers la réalisation, le **COMMENT** et non le quoi.

- **Entrées** :
  - Spécifications
  - Détail de la plateforme de développements (dans le Recueil des besoins)

- **Résultats** :
  - **Modèle** défini dans une documentation textuelle appuyée par des diagrammes (ex : std UML)

On peut distinguer deux niveaux d'abstraction dans la conception et donc deux activités :

1. **conception architecturale** → spécifie le système en terme de composant bien défini
2. **conception détaillée** → spécifie chaque composant

Remarque 1.5 *Au S3, la conception architecturale est abordée notamment au travers de la notion de patron de conception (design pattern) dans le module Qualité de développement. Nous étudions aussi la conception détaillée.*

d) Programmation

- **Rôle** :
  - Spécifier le système en utilisant un langage de programmation.
- **Entrées** :
  - Documentation de la conception détaillée
  - Spécification (si pas de conception détaillée)
- **Résultats** :
  - Code qui implémente les fonctionnalités
  - Documentation technique du code
  - dépôt, historique de gestion de version

e) Validation (e1) et Vérification (e2)

- **Rôle** :
  (e1) Répond à la question "Le logiciel satisfait-il les attentes de l'utilisateur ? "
  (e2) Répond à la question "Le logiciel satisfait-il les spécifications ? "
- **Entrées** :
  - Documentation de la conception détaillée pour les tests unitaires
  - Documentation de la conception architecturale pour les tests d'intégration
  - Documentation d'ADB pour les tests d'acceptation
- **Résultats** :
  - Cas de tests
  - Analyse des résultats d'exécution des cas de tests

De même que l'activité de conception peut être découpée en sous-activités, l'activité de Validation (e1) et Vérification (e2) ou aussi appelé plus généralement l'**activité de test**, peut être divisée en trois activités distinctes :

- Test d'Acceptation qui consiste à valider les besoins fonctionnelles (*cf.* R4.02 Qualité de développement)
- Test d'Intégration qui consiste à vérifier l'intégration des composants et leurs interactions (*cf.* R4.02 Qualité de développement)
- Test Unitaire qui consiste à vérifier une unité (*cf.* R2.03 Qualité de développement)

De part la validation ou vérification que ces sous-activités ciblent, elles peuvent être conçues dès que les artefacts nécessaires ont été produit. On peut donc concevoir les test d'acceptation une fois l'ADB réalisée, concevoir les test d'intégration suite à la conception architecturale et concevoir les tests unitaire suite à la conception détaillée.

Remarque 1.6 *Ce découpage est par exemple utilisé pour la création des cycles en V.*

f) Intégration et gestion des configurations

- **Rôle** :
  - Décrire le système et ses évolutions.
- **Entrées** :
  - Version du logiciel
- **Résultats** :
  - Dossier de maintenance

2 Cycle de vie du logiciel

L'équipe projet suit un processus de développement : ensemble d'activités et de résultat qui conduisent à la création d'un produit logiciel.

Le processus de développement indique la forme dans laquelle les activités sont connectées entre elles. L'ordre dans lequel s'enchaînent les activités s'appelle le cycle de vie du produit logiciel.

Au semestre 2 du DUT Informatique dans la ressource R2.03 Qualité de développement nous avons étudié les tests unitaires et analysé comment l'on pouvait obtenir des développements dirigés par les tests. Nous avons donc étudié deux cycles de vie dit *linéaires* : **Cascade** et en **V**. Puis nous avons proposé un cycle en V itératif en proposant de développer chaque unité (fonction) l'une après l'autre en faisant à chaque fois la conception du test puis la programmation de la fonction puis l'exécution du tests. Maintenant que nous avons défini les activités du GL, nous illustrer ces différents arrangements.

La figure 1 montre l'organisation des activités selon un cycle de vie linéaire Cascade. Cette figure est un diagramme d'activité suivant le standard UML. Il est utilisé ici pour montrer l'enchaînement des tâches effectuées par les développeurs, ou autrement dit leur processus métier. Chaque rectangle à bord arrondi représente une activité. Lorsqu'une activité est terminée on passe à l'activité suivante de manière instantanée en suivant la flèche (la transition). Les données ne sont pas représentées. Pour comprendre la dépendance entre les tâches, il faut connaître les données produites et nécessaires en entrée, d'où la description de chaque tâche précédemment. Nous verrons en détail comment réaliser un tel diagramme lorsque nous analyserons les processus métiers au moment de l'ADB. Dans la figure 1 on observe que les activités s'enchaînent naturellement en cascade, sans retour possible en arrière. Si ce processus de développement est suivi il implique donc que chaque tâche soit réalisée complètement et parfaitement. Les tests seront réalisés et produit uniquement une fois que la programmation sera réalisée. Par conséquent il est probable que la phase de test soit longue et difficile à définir.

Figure 2 montre un cycle linéaire en V. Ici nous avons fractionné les activités de conception

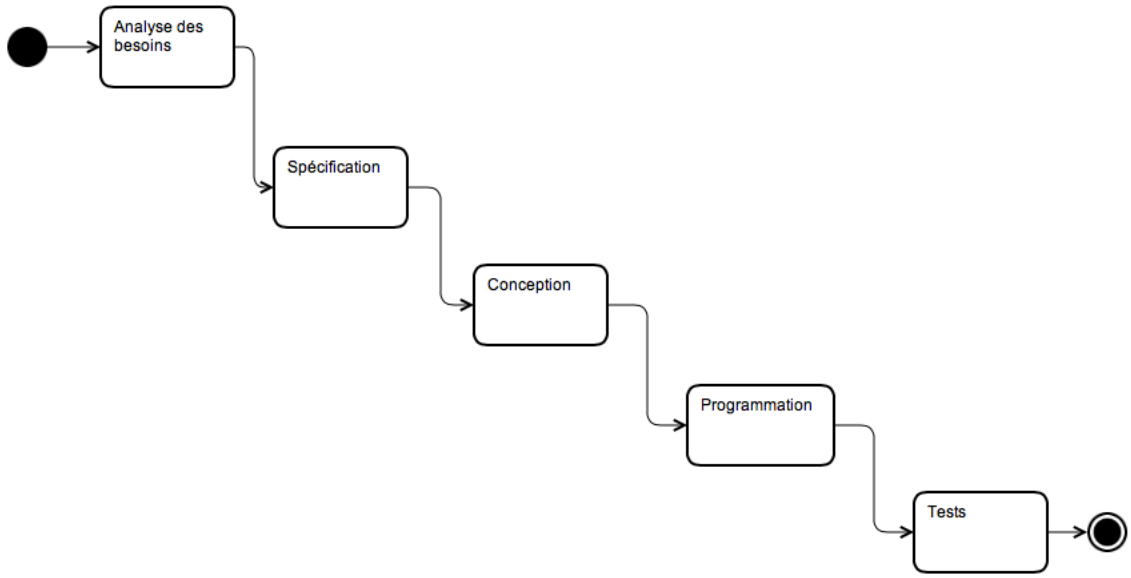

FIGURE 1 – Cycle de vie linéaire Cascade

et de test afin de pouvoir obtenir un contrôle de chaque étape. Ainsi, plutôt que de réaliser l'enchaînement en cascade on crée dès que c'est possible les tests. Ainsi lors du développement à chaque étape on réfléchit aux erreurs que l'on cherche. Une fois la programmation effectuée, on exécute les tests en commençant par les unités, puis l'intégration des composants et enfin la validation des fonctionnalités. A chaque étape de test on se donne la possibilité d'itérer en revenant à l'activité à l'origine du test. On se trouve donc dans un cycle linéaire qui permet l'itération en cas d'erreur.

A partir de ces modèles ont peut imaginer tout de même des solutions permettant plus d'**anticipation**. Par exemple on peut créer un modèle dit incrémental, c'est à dire, reposant sur des itérations prévues à l'avance et reposant soit sur les spécifications soit sur l'architecture.

Par exemple, prenons une équipe de développeurs utilisant l'approche orientée objet et notamment au niveau de l'architecture une approche par composant. On peut réaliser l'analyse des besoins, la spécification et la conception architectural en séquence. À l'issue de cette séquence, on produit une architecture avec n composants. En identifiant les dépendances entre composant ont peut définir un ordre de développement des composants. Ceci défini nos n itérations. On peut alors pour chaque composant réaliser une itérations qui consiste à faire la conception détaillée, la programmation et les tests. Une fois les tests validés, on passe au composant suivant.

▸ **Exercice 1.** *Proposez un cycle en V itératif pour un développement dont l'architecture suis le patron Modèle-Vue-Controleur.*

Dans tous les cas ces cycles de vie linéaire couvre l'intégralité du projet. Plus il y a de fonctionnalité plus on passe de temps dans les premières étapes sans avoir l'occasion de mettre à l'épreuve le modèle. Pour un projet avec un nombre conséquent de fonctionnalité et une complexité importante,

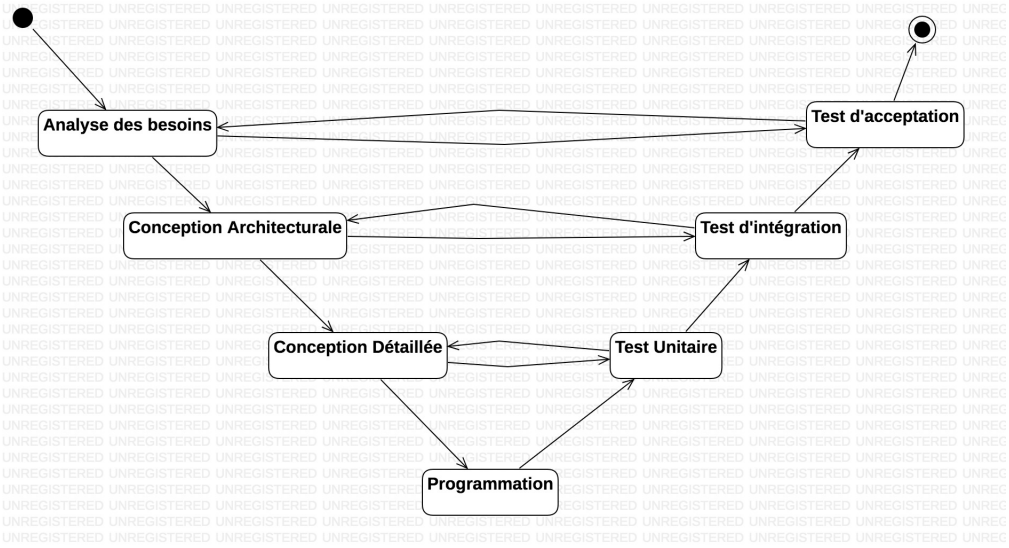

FIGURE 2 – Cycle de vie linéaire V

ces deux cycles peuvent amener à l'"effet tunnel". C'est à dire qu'ils emprisonnent le développeurs dans un tunnel à sens unique dont ils ne verront le bout qu'une fois tous les problèmes résolus, sans avoir pu anticiper les difficultés.

Pour conclure, on remarque donc que les modèles linéaires et linéaires itératifs, reposent sur une analyse des besoins censée spécifier de manière correcte des besoins stables pour l'ensemble du projet. Malheureusement, un logiciel peut être amené à évoluer !

Nous nous proposons d'étudier d'autres types de méthodes plus flexibles, s'adaptant à une évolution des besoins au cours du projet, et permettant de réduire les coûts et favoriser la réussite d'un projet.

3 Vers d'autres méthodes de gestion

Chaque organisation dont le rôle est de développer des produits logiciels se doit de définir en fonction des projets un cycle de vie. Ce cycle de vie comme nous l'avons vu dans la section précédente permet d'organiser les activités du génie logiciel afin de réaliser le projet considéré. Ceci constitue donc un enchainement de tâche par les développeur pour fournir une valeur ajouter : c'est le processus métier du développeur logiciel. Dans le cas des cycles de vie linéaire, on traite le projet dans son intégralité dès la lecture du cahier des charges qui constitue un contrat relativement rigide. Par conséquent les activités s'enchainent sans retour (théorique) possible à moins de recommencer le projet. Nous allons voir quelles sont les alternatives possibles à ces cycles de vie linéaire en partant du constat fait par des études sur la gestion de projet informatique puis nous étudierons l'application des méthodes agiles. Enfin nous évoquerons une méthode à plusieurs niveaux reposant sur l'approche objet.

Remarque 3.1 *Cette section s'appuye sur les études CHAOS du Standish Group. Ces études sont reprises et analysées par un certains nombre de blog d'entreprise ou de mspécialiste en managment. Les figures et résultats du standish group sont tirées de ces différentes sources que l'on retrouvera dans la bibliographie du présent document [Zucker 2016, StandishGroup 2015, Mersino 2018,*

McSweeney 2019]

3.1 Constat

Une étude de *Standish Group* [StandishGroup 2015], dont les résultats sont présentés dans la figure 3 issu du livre [McSweeney 2019], montre que depuis 1994 on observe une croissance de la part des projets réussi mais que globalement on n'est pas bien meilleur qu'il y a 20 ans. Dans cette étude on entend par projet réussi (Succeed), un projet qui satisfait toute les exigences fonctionnelles et qui est délivré dans les temps en suivant le budget initial. A contrario, un échec (Failed) correspond à un projet qui ne satisfait pas les exigences du client ou qui a été annulé. Un succès mitigé (Challenged) correspond à un projet dont une part des exigences fonctionnelles sont réalisées, les contraintes de temps et de budget ne sont pas respectées.

On observe tout de même qu'en 1994 la réussite des projets était nettement inférieure aux années 2000.

Les principales causes d'échec observées dans cette étude, concernent le manque d'implication des utilisateurs et le fait que les exigences ainsi que les spécifications soient incomplète et qu'elles puissent changer. En conclusion, l'étude montre que de petites fenêtres de temps combinées à la production de livrables tôt dans le projet et souvent augmenteront le taux de succès.

De plus petites période de travail conduisent à un processus itératif incluant la conception, la réalisation, les tests et déploiements de petits éléments. Chacune de ces itérations est un incrément du projet et doit aboutir à un livrable.

Il existe un type de méthode qui permet de favoriser la discussion avec le client et implique la production de livrable fréquent : les méthodes agiles. Le Standish Group conclue son étude en recommandant les méthodes agiles.

3.2 Les méthodes agiles

Les méthodes agiles reposent sur des principes, également appelés les valeurs de l'agilité.

Parmi ces valeurs ont pourra retenir les quatre suivantes :

— l'interaction avec les personnes plutôt que les processus et les outils.
— une production opérationnelle plutôt qu'une documentation pléthorique
— la négociation avec un client plutôt que le respect d'un contrat
— la collaboration au changement plutôt que le suivi du plan.

On retiendra de ces principes qu'il s'agit d'une part de favoriser la discussion avec le client mais également au sein de l'équipe et d'autre part de permettre au projet de s'adapter aux nouvelles exigences en permettant de la flexibilité sur le cahier des charges et des productions opérationnelles.

Les objectifs sont de :

— réduire le cycle de vie logiciel
— contrer l'effet tunnel des modèles classiques en s'adaptant aux besoins des clients tout au long du projet

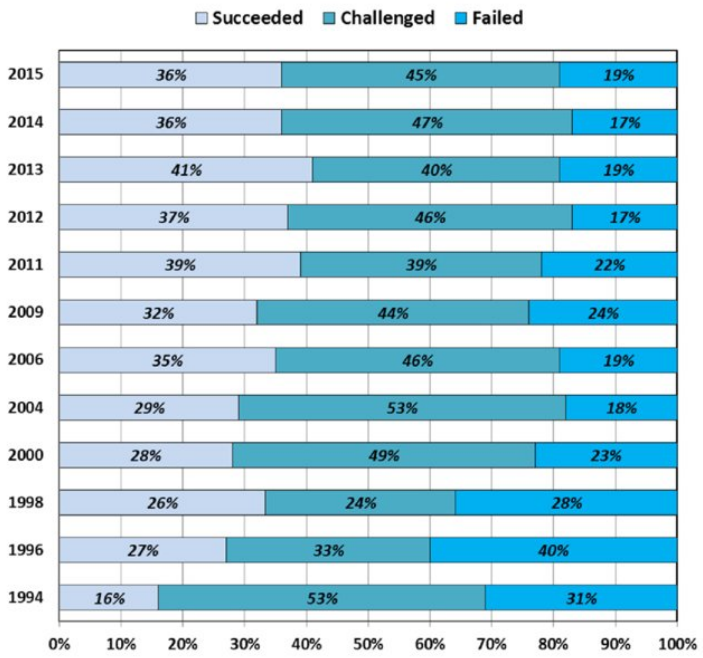

FIGURE 3 – Réussite des projets informatiques, étude Standish Group, CHAOS, [McSweeney 2019]

Afin de satisfaire ces objectifs en appliquant les principes de l'agilité on cherchera à développer une version minimale puis on intégrera les fonctionnalités par un processus itératif en lien avec le client avec des tests tout au long du cycle de développement.

Il existe plusieurs méthodes dont deux qui sont souvent utiliser en développement logiciel : eXtreme Programmaing (XP) et SCRUM (mêlée). Voici quelques points qui caractérise chacune de ces méthodes.

eXtreme Programming (XP)
cyle de développement court (1 à 6 semaines)
cycle piloté par le client qui fournit des scenarii
auto-organisation de l'équipe de développement
programmation à tour de rôle (en binôme par exemple)
le logiciel est livré à l'issue d'un cycle quand les tests fonctionnels sont OK

SCRUM
cycle itératif (Sprint) (2 à 6 semaines)
auto-organisation de l'équipe de développement
équipe encadré par un SCRUM master
nouvelle version livrée à l'issue d'un sprint

Le Standish Group a publié les résultats de ses études concernant la comparaison entre un cycle cascade et une méthode Agile. Figure 4, extraite du site [Mersino 2018] montre que les projets réalisés avec une méthode Cascade échoue à 8% contre 21%, soit un taux presque trois fois plus élevé d'échec.

Il faut retenir qu'appliquer une méthode Agile nécessite de préparer les tests d'acceptaion en

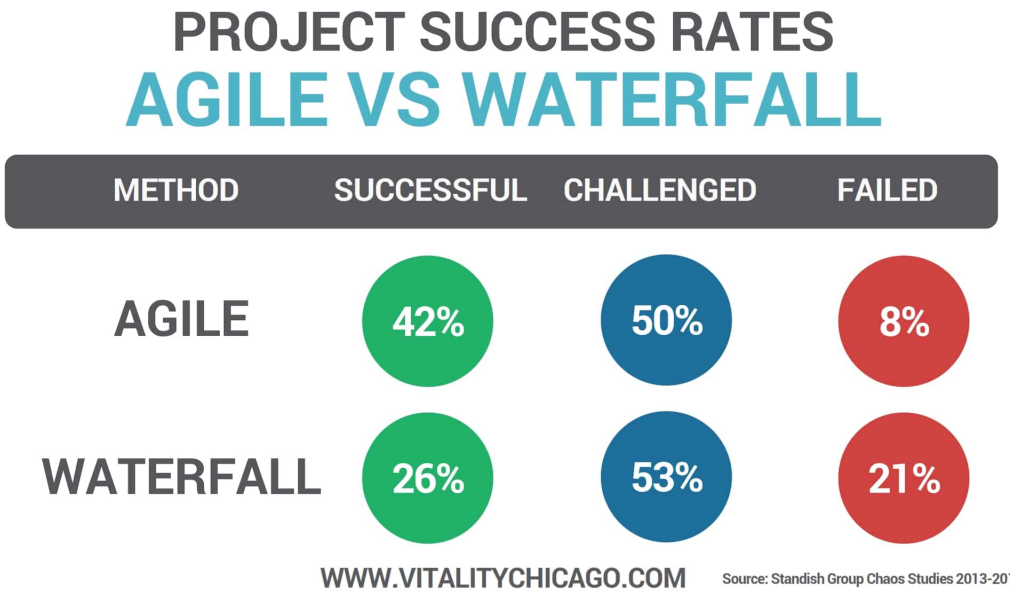

FIGURE 4 – Gain observé entre un choix Agile ou Cascade, étude Standish Group, CHAOS

début de cycle (sprint) et de passer ces tests à la fin du cycle. On effectue une revue du code à chaque début de cycle.

3.3 Cycles de vie multiniveaux, l'exemple du processus unifié

Pour organiser les activités lors des cycles on peut travailler avec un processus de développement pour le développement Orienté Objet : le Processus Unifié (Unified Process) (UP).

Dans ce processus on répète un certain nombre de fois une série de cycles se concluant tous par la livraison d'une version du produit au client. Le livrable est donc le produit logiciel à la fin d'un cycle.

Définition 3.2 (Livrable)
le livrable d'un produit logiciel comporte :

- ***un corps de code source réparti sur plusieurs composants pouvant être compilés et exécutés***
- ***un manuel*** *(et/ou une documentation technique)*
- ***les produits associés (ou dépendances)***

Il doit prendre en compte les besoins des utilisateurs, et de tous les intervenants (tous ceux amenés à l'exploiter) Il comprend donc :

- *un recueil des besoins*
- *les cas d'utilisations (besoins fonctionnels)*
- *les spécifications non fonctionnelles*
- *les cas de test*

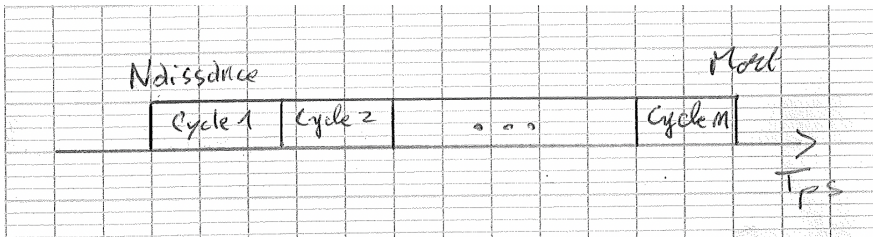

FIGURE 5 – Le Processus Unifié est un enchainement de cycles

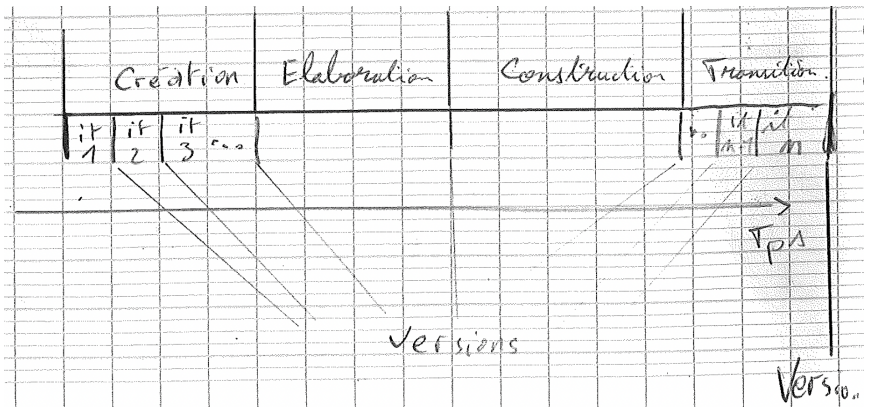

FIGURE 6 – Un cycle := Phases composées d'itérations, chaque itérations est composées d'activités

Il englobe tous les éléments permettant de concevoir, programmer tester et utiliser le système (produits des activités du GL) comme :

- *l'architecture*
- *la définitions des comportements du système*
- *les modèles visuels*
- *etc.*

Principes des cycles :

- À chaque cycle correspond une nouvelle version du logiciel (voir Figure 5)
- Un cycle est composé de 4 phases (voir Figure 6) :
  1. Étude préliminaire / création
  2. Élaboration
  3. Construction
  4. Transition

- chaque phase se divise en itérations (voir Figure 6)

Il apparait que le processus unifié est un cycle de vie multiniveaux. Il est piloté par les cas d'utilisations et organisés selon l'architecture. Il est possible de faire correspondre un cycle à un ou plusieurs cas d'utilisation, et d'itérer sur les composants logiciels.

Il est possible de s'inspirer de cette méthode pour composer sa méthode agile.

Références

[McSweeney 2019] A. McSweeney. Introduction to Solution Architecture. Independently Published, 2019.

[Mersino 2018] Anthony Mersino. *Agile Project Success Rates are 2X Higher than Traditional Projects (2019).* `https://vitalitychicago.com/blog/agile-projects-are-more-successful-traditional-projects/`, 2018. Accessed : 2020-10-07.

[Reix *et al.* 2016] Robert Reix, Bernard Fallery, Michel Kalika and Frantz Rowe. Systèmes d'Information et Management des Organisations - 7ème édition . Vuibert Gestion, August 2016.

[StandishGroup 2015] StandishGroup. *CHAOS Report 2015.* `https://www.standishgroup.com/sample_research_files/CHAOSReport2015-Final.pdf`, 2015. Accessed : 2020-10-07.

[Zucker 2016] Alan Zucker. *Successful Projects, What We Really Know.* `http://www.planningplanet.com/blog/successful-projects-what-we-really-know`, 2016. Accessed : 2020-10-07.