Conception et implémentation d'une infrastructure de déploiement continu pour le logiciel SPHER

Homada Boumedane

le cnam

CONSERVATOIRE NATIONAL DES ARTS ET METIERS

CENTRE EGIONAL ASSOCIE DE LYON

MEMOIRE

présenté en vue d'obtenir

le DIPLOME D'INGENIEUR CNAM

SPECIALITE : Informatique

OPTION : Architecture et ingénierie des systèmes et des logiciels (AISL)

par

Homada BOUMEDANE

Conception et implémentation d'une infrastructure de déploiement continu pour le logiciel SPHER

Soutenu le 11/12/2015

JURY

PRESIDENT :	**Mr. Christophe PICOULEAU**	**Professeur/Cnam Paris**
MEMBRES :	**Mr. Bertrand DAVID**	**Professeur/ECL Lyon**
	Mr. Claude GENIER	**Administrateur du Cern**
	Mr. Jean Claude JOUFFRE	**Ingénieur LISTIC**
	Mr. Ozan GUNALP	**Docteur informatique**

Claude Genier
Enseignant CNAM

Remerciements

Je remercie et exprime ma profonde gratitude à mon collègue, Ozan Gunlap, pour son soutien, son écoute, ses conseils, ses corrections et l'expérience dont il a su me faire profiter.

Je remercie mon tuteur Claude Genier pour ses corrections et ses conseils.

J'adresse aussi un remerciement à mon ami et ancien collègue Jean Claude Jouffre pour son aide tout au long de ce projet.

Ce projet n'aurait pas été possible sans le soutien permanent de ma famille. Je dédie ce mémoire à mes parents, ma femme et à mon fils.

B. H.

Sommaire

Liste des figures

1 Introduction

1.1 Contexte

Tout logiciel passe par plusieurs étapes pour être créé. Il s'agit des étapes de définition, de développement et de maintenance. On parle alors du cycle de vie d'un logiciel.

Depuis toujours, les projets informatiques sont gérés avec des méthode classiques qui se distinguent par des phases bien définies à savoir recueillir les besoins, faire la conception générale puis détaillée du produit, le développer et le tester avant de le mettre en production.

Il s'agit alors d'un enchaînement de phases successives où il faut terminer l'étape précédente pour passer à la suivante. Chaque étape possède obligatoirement une date de début et de fin, le tout est consigné dans un planning qui en plus contient les tâches à réaliser pour l'ensemble du projet.

Les méthodes classiques sont strictes et ne tolèrent pas la flexibilité face aux changements, donc il faut tout faire correctement dès le début du projet. En effet, ces méthodes se conforment à un plan strict et rigide qui incite uniquement au respect des délais établis pour chaque phases. En cas d'imprévu ou de demande de changement de dernière minute ses méthodes ne permettent pas de faire face.

Dans une gestion classique de projet, les risques sont détectés tardivement dans le processus de développement puisqu'il faut attendre la fin du développement pour procéder aux tests. Ce qui implique un retour en arrière difficile en cas d'anomalie, car le coût pour faire ce retour est très coûteux en terme de temps et d'argent.

Suite aux échecs subis par de nombreux projets dans les années 90, 17 d'experts en développement logiciel se sont réunis en 2001 afin de proposer des solutions aux problèmes posés par les méthodes classiques, ils ont mis au point les méthodes agiles.

Les méthodes agiles sont des procédures de développement de logiciel qui se veulent plus flexibles que les méthodes classiques. Ces méthodes amènent de la flexibilité à la gestion du projet en permettant une grande réactivité aux demandes de changement des clients. Elle permettent aussi d'impliquer d'avantage le client dans le processus de développement afin d'améliorer sa satisfaction.

Les méthodes agiles utilisent un cycle de développement itératif on découpant le projet en plusieurs cycles de développement appelés « itérations » afin de raffiner et réviser le logiciel. Chaque itération donne lieu à un produit fonctionnel qui contient les différentes fonctionnalités souhaitées par le client.

Ce procédé permet de répondre rapidement à des imprévus qui arrivent en cours de développement mais aussi de détecter les erreurs au plus tôt dans le processus de développement. Ainsi, le développement du logiciel devient rapide et la mise en production aussi. La phase obligatoire qui permet de mettre un logiciel en production est **le déploiement**.

Le déploiement, appelé aussi phase de livraison ou phase de mise en production, est la phase qui intervient en dernier dans le cycle de vie du logiciel. Il s'agit d'une étape qui regroupe les activités qui vont aboutir à la mise en marche du logiciel chez l'utilisateur final. Des activités qui incluent l'installation, la configuration, la mise à jour et pour finir la désinstallation du logiciel.

Grâce à l'évolution d'Internet ses dernières années, on est capable de déployer un logiciel chez le client à distance sans avoir besoin à se déplacer.

Malgré ce progrès, le déploiement reste une phase très critique qui peu rapidement devenir un cauchemar. En effet, lorsque on déploie le produit on peut rencontrer plusieurs problèmes parmi eux :

- des erreurs dans le code de l'application.
- l'application ne fonctionne pas comme prévu.
- une modification de dernière minute.

Tous ces problèmes peuvent engendrer, une perte d'argent et une perte de temps pour l'entreprise. En effet, si l'entreprise ne fait pas évoluer son produit constamment et qu'elle ne le met pas rapidement sur le marché, elle deviendra moins compétitive.

Il est primordial pour une entreprise de disposer d'une solution de déploiement automatisé. L'avantage de cette solution est justement d'éviter à l'humain d'exécuter des tâches répétitives et fastidieuses qui vont forcément générer des erreurs. Le coût de correction de ces dernières serait très élevé.

L'arrivée des méthodes agiles à révolutionner les méthodologies traditionnelles de gestion de projet mais aussi la phase de déploiement. Ainsi, il est possible d'introduire de l'agilité dans cette étape cruciale, on parle alors de déploiement continu.

Dans la section suivante, nous définirons la problématique et les objectifs de notre projet et nous verrons comment mettre en place une infrastructure de déploiement continu.

1.2 Problématique et objectifs

Comme nous l'avons abordé précédemment, un logiciel doit passer par une phase finale avant d'être utilisé par les utilisateurs. C'est une phase importante et critique à la fois dans le cycle de vie du logiciel. Elle permet enfin de livrer le logiciel aux utilisateurs et ainsi avoir des retours d'expériences d'utilisations. Enfin, elle est critique car généralement toutes les entreprises stressent avant le déploiement, le risque qu'il y ait des problèmes est toujours présent.

Afin de rendre cette phase plus maîtrisable, l'objectif dans ce dossier sera de décrire comment nous allons procéder pour implémenter un environnement de déploiement automatisé afin d'éviter les problèmes engendrés habituellement par cette phase.

En effet, pour commencer l'étude de la mise en place de cette infrastructure je vais tout d'abord définir les problématiques levées par ce projet. La première problématique est la suivante: comment réduire le temps de mise sur le marché. La deuxième problématique: comment repérer les erreurs très tôt dans le processus de développement afin d'augmenter la fiabilité de notre logiciel et la troisième problématique, comment mettre en place un processus de rétroaction qui permet d'avoir les retours d'expériences des utilisateurs afin de corriger et améliorer constamment le logiciel.

Afin de répondre à ces problématiques, nous allons étudier comment mettre en place une infrastructure de déploiement continu. Cette dernière permettra :

> **La réduction du temps de mise sur le marché** (Time To Market)
> le but est de réduire le temps de production de nos logiciels afin de les mettre rapidement sur le marché sinon le risque serait de perdre en compétitivité.

La rétroaction

Introduire une nouvelle fonctionnalité est toujours risqué, car on ne sait pas si elle sera beaucoup utilisée ou pas. Mais grâce au déploiement continu on peut tester la fonctionnalité sur un sous ensemble d'utilisateurs puis étudier leurs retours d'expérience.

La flexibilité face aux changements

En cas où le déploiement du logiciel se passe mal pour différentes raisons (erreurs dans le code, fonctionnalité non souhaitée par le client). Cette infrastructure doit nous permettre de revenir à une version antérieure sans encombre.

La diminution des risques

En effet, si on déploient des changements mineurs (ex:ajout d'une fonctionnalité), alors les risques serait bien calculés et maitrisés par rapport à un déploiement massif de plusieurs changements (ex: plusieurs fonctionnalités). En cas de problème, on peut passer un temps moindre à chercher l'origine du problème et la corriger. Cette formule nous montrent comment calculer un temps d'interruption :

La diminution du stress

Dans beaucoup d'entreprises le déploiement est source de stress pour les équipes qui l'effectue. Grâce à cette infrastructure le déploiement devient une étape banale. En effet, il n'est plus nécessaire d'intervenir manuellement pour effectuer le déploiement, car il y aura une procédure automatisée pour le faire.

1.3 Réalisation

Afin d'utiliser le déploiement continu, un ensemble de pré-requis sont nécessaires pour assurer le succès. Nous commencerons par la mise en place d'une usine de développement capable d'assurer la bonne compilation du code, le jeu des tests unitaires, le packaging, le déploiement et l'exécution des tests dans un environnement d'intégration. Ce processus s'appelle l'intégration continue.

Une fois l'intégration continue en place, nous continuerons la mise en place de l'infrastructure de déploiement continu. Notamment, par la mise en place des pratiques suivantes :

DevOps : pratique qui incite à repousser la frontière entre les développeurs du code et les opérateurs qui déploient le code. Cela afin d'accélérer, optimiser et fiabiliser le processus de mise en production.

Pipeline de déploiement : Le parcours que traversent les modifications apportées au code à partir du commit du développeur jusqu'à sa livraison en production.

Livraison continue : Consiste à respecter le premier principe agile : « Notre plus haute priorité est de satisfaire le client en livrant rapidement et régulièrement des fonctionnalités à grande valeur ajoutée »[1].

Dans ce document, l'ensemble de la démarche de mise en place du déploiement continu est décrite. La section suivante décrit son organisation.

1.4 Organisation du document

Ce dossier se compose d'un chapitre qui traite du contexte de projet. Ensuite, un chapitre qui revient sur les problématiques traitées par ce document suivi d'un chapitre sur l'état de l'art. Puis dans le chapitre conception et le chapitre réalisa-

[1] source : http://agilemanifesto.org/iso/fr/principles.html

tion nous détailleront les techniques, méthodologies et outils utilisés dans ce dossier. Le dossier s'achève par une conclusion.

Le chapitre 2 présente le contexte du projet en commençant par une présentation de l'entreprise et de ses objectifs. Ensuite, il donne une description du projet étudié dans ce document. C'est ici aussi que nous citerons le rôle occupé par l'auditeur dans le projet et pour clore ce chapitre nous parlerons des contraintes liées à ce projet.

Le chapitre 3 revient plus en détail sur les problématiques du projet.

Le chapitre 4 commence par un aperçu du processus traditionnel de développement du logiciel en mettant l'accent sur les problèmes engendrés par ce dernier. Ensuite, il présente les méthodes agiles et leur apport en matière de gestion de projet. Ce chapitre introduit aussi le concept de pipeline de déploiement continu.

Le chapitre 5 décrit les étapes nécessaires à la conception du projet telles que l'étude des besoins. Il donne aussi le schéma de l'infrastructure de déploiement continu tout en détaillant les différents composants, ensuite il met l'accent sur les techniques utilisées pour mettre en place cette infrastructure.

Le chapitre 6 est consacré à la réalisation du projet. Il dresse la liste des outils choisis pour réaliser cette infrastructure et le planning des livrables.

Enfin, le dossier se termine par une conclusion qui propose un bilan du projet et les perspectives.

2 Contexte du projet

2.1 Présentation de l'entreprise

Tactualities, localisé à Villaz (74), est un groupe international d'innovations technologiques. En 2014, le groupe rachète le concepteur et créateur de tables tactiles multi-utilisateurs Digitale Interactive, et devient très vite un acteur majeur du marché des tables tactiles. Résolument tourné vers l'avenir, **Tactualities** investit massivement dans la Recherche et le Développement.

Grâce à l'arrivée de nouveaux investisseurs, Tactualities devient le centre de recherche et développement **Nicolas Bourbaki**, un centre dédié exclusivement à la recherche dans les nouvelles technologies.

Se composant d'une dizaines d'ingénieurs et de quatre docteurs en informatique, le **CRD Nicolas Bourbaki** crée, développe et met en œuvre des solutions technologiques sophistiquées afin d'améliorer la capacité de l'individu à interagir avec l'environnement qui l'entoure.

Tous les produits résultats de la recherche du **CRD Nicolas Bourbaki** recherches sont commercialisés par la société, bien plus adaptée, **iSPHER**, basée en Suisse.

A ce jour **iSPHER**, sous l'impulsion des nouveaux investisseurs, a déjà en charge de proposer les tables tactiles, le logiciel **SPHER** et les plateformes de diffusion d'informations à destination, dans un premier temps, des tables tactiles.

LA figure 2.1 représente la structure de l'entreprise.

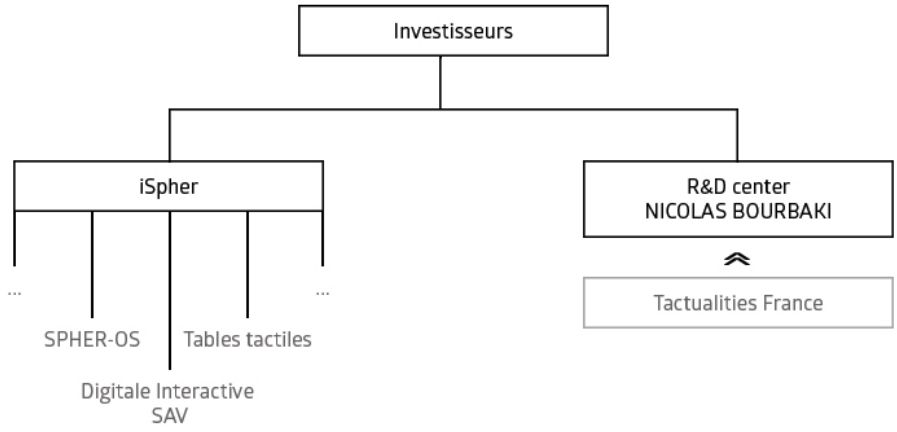

Figure 2.1 – Structure de l'entreprise

2.2 Description du projet

Le produit phare de notre entreprise est un logiciel adapté pour les surfaces tactiles, **SPHER-OS**. Il se compose de deux parties :

- une partie gestion des applications et des contenus : développée en Java et en utilisant la technologie OSGi. Cette partie gère la connexion de plusieurs utilisateurs.
- un moteur d'interfaces et de visualisation : développée en C++ avec le support du multi-touch. Cette partie est responsable de la gestion des fenêtres et des entrées utilisateurs (touches, clavier).

Figure 2.2 – Interface de connexion à SPHER

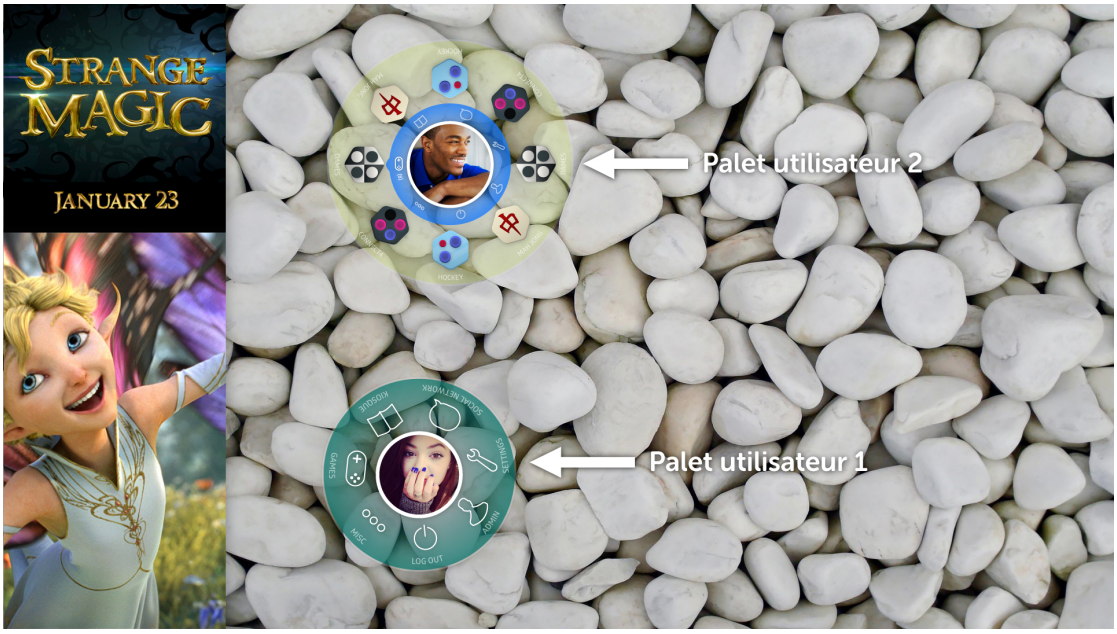

Figure 2.3 – Interface d'utilisation de SPHER

Les figures 2.2 et 2.3 montrent l'interface graphique de **SPHER**.

Actuellement, **SPHER** est déployé manuellement en procédant de la façon suivante:

1. un technicien télécharge un fichier compressé contenant le logiciel sur la table tactile cible.
2. le technicien arrête le logiciel puis désinstalle la version courante.
3. le technicien procède à l'installation de la nouvelle version et la mise en marche de celle ci.

Cette méthode présente plusieurs obstacles majeurs pour le développement futur de **SPHER**. En effet, on peut citer comme obstacles :

- le déploiement manuel sur les tables tactiles prend énormément de temps en plus d'être source d'erreur humaine. Selon les prévisions commerciales les ventes sont estimées à 100 000 licences par an. Ce qui implique l'automatisation du déploiement afin de passer à cette échelle.
- l'interruption du fonctionnement de l'application pendant le déploiement nous impose de négocier avec nos clients un créneau horaire afin de procéder à la mise à jour d'applications et/ou du système.

De ce constat est née l'idée de disposer d'une infrastructure de déploiement continu qui nous permettra de:

- déployer plus rapidement **SPHER** chez nos clients (réduire le Time To Market).
- diminuer les risques liés à l'intervention humaine.
- garantir la disponibilité de l'application pendant le déploiement grâce au Zero Downtime Deployment (ZDD).

2.3 Rôle de l'auditeur

Dans le cadre de mon travail au sein du **CRD Nicolas Bourbaki**, anciennement **Tactualities**, j'ai été amené à concevoir et mettre en place une infrastructure de déploiement continu de **SPHER** et des applications.

Mon apport s'est porté sur plusieurs points parmi lesquels, on peut citer :

- élaboration du schéma de l'infrastructure de déploiement continu.
- choix et mise en place des outils nécessaires au déploiement continu.
- participation dans le codage de certains modules de **SPHER**(mise à jour des applications...).
- la réalisation de tests d'intégration.
- la réalisation de tests du fonctionnement.
- la réalisation de modules de management et de monitoring.

Plusieurs personnes travaillent sur ce projet , dont monsieur **Thomas Leveque**, qui est le chef de ce projet.

2.4 Contraintes

Lors du lancement du projet, on m'a imposé de mettre en place une infrastructure de déploiement continu pour un projet moins critique que celui du **SPHER** pour commencer. Cette contrainte a pour objectif de maîtriser le processus du déploiement continu et de connaître tous ses rouages pour pouvoir le généraliser pour le projet **SPHER** et plus tard à l'ensemble des projets qui s'y prêtent.

Une autre contrainte a été de préférer les solutions open source pour la réalisation de cette infrastructure de déploiement continu. Cette dernière a pour but de réduire les risques liés aux bugs bloquants ou des fonctionnalités manquantes, car les solutions open source sont régulièrement corrigées et mises à jour dans la plupart du temps.

3 Problématiques

3.1 Introduction

La première partie de ce document présente le contexte de ce travail. Ce chapitre

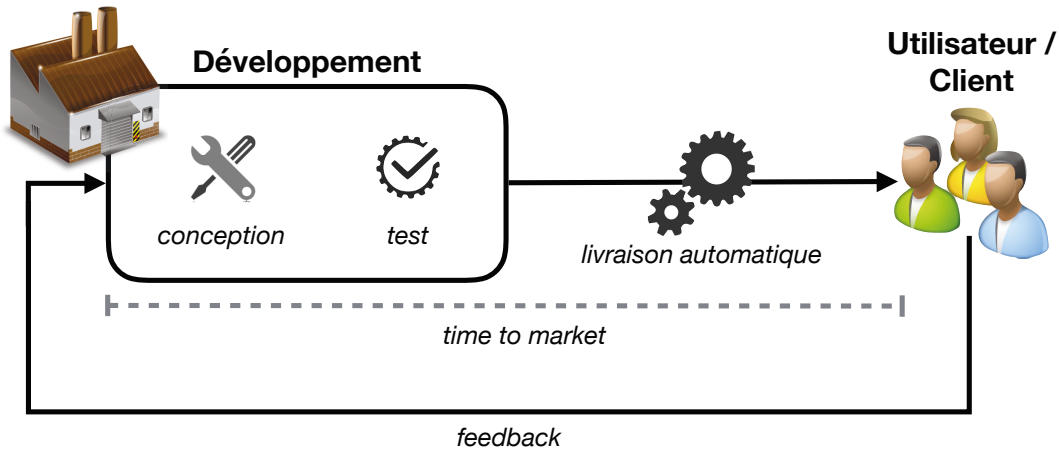

Figure 3.1 – Processus de développement et production

3.2 Réduire le temps de mise sur le marché

La définition du temps de mise sur le marché (TTM) peut varier d'une entreprise à l'autre mais aussi selon la complexité du projet. Dans ce dossier, nous allons définir le TTM par le délai nécessaire pour développer et mettre au point un logiciel avant qu'il puisse être mis sur le marché. Dans cet intervalle, il y a de nombreuses étapes complexes où un problème peut subvenir et compliquer la prédiction avec précision du temps de mise sur le marché.

A l'heure actuelle, les nouvelles technologies se reposent essentiellement sur le développement itératif. Ceci permet une évolution permanente ainsi qu'un raccourcissement du cycle de vie de ses produits. Elles se préoccupent également du nombre des concurrents sur le marché. Par conséquence, trouver des moyens d'optimiser le TTM est un facteur majeur dans la stratégie globale d'une entreprise. Par exemple, si le lancement d'un logiciel est retardé de plusieurs semaines, alors les concurrents auront ces semaines pour récupérer leur retard et ainsi gagner des parts de marché. Un retard aura également comme conséquence la diminution du chiffre d'affaire, pouvant impacter la commercialisation du logiciel, une fois celui-ci lancé sur le marché. En résumé, si l'entreprise contrôle mieux ses processus de développement logiciel, elle aura la maîtrise du TTM et pourra sortir son produit tant qu'il est nouveau sur le marché.

Les raisons qui poussent les entreprises soucieuses d'améliorer le temps de mise sur le marché de leurs produits sont :

- **Gestion efficace des ressources:** Avoir un planning fiable et détaillé permet de préparer à l'avance chaque étape et ainsi mieux prévoir leur coût humain et financier respectif. Construire des plannings basés sur des moments clés du projet et des plannings d'effectifs permet de mieux répartir les ressources dans les différentes phases du projet.
- **Maîtrise des dates de lancement:** Si l'entreprise arrive à prédire avec précision la date de mise sur le marché de son produit, alors elle pourra profiter des salons, des festivals, des périodes de fêtes et de plusieurs autres événements de marketing afin de commercialiser son produit.
- **Augmentation du chiffre d'affaires:** L'entreprise qui met son produit rapidement sur le marché, sans pour autant négliger la qualité ou la performance de ce dernier, pourra augmenter ses revenus, car son produit fera face à moins de concurrence.

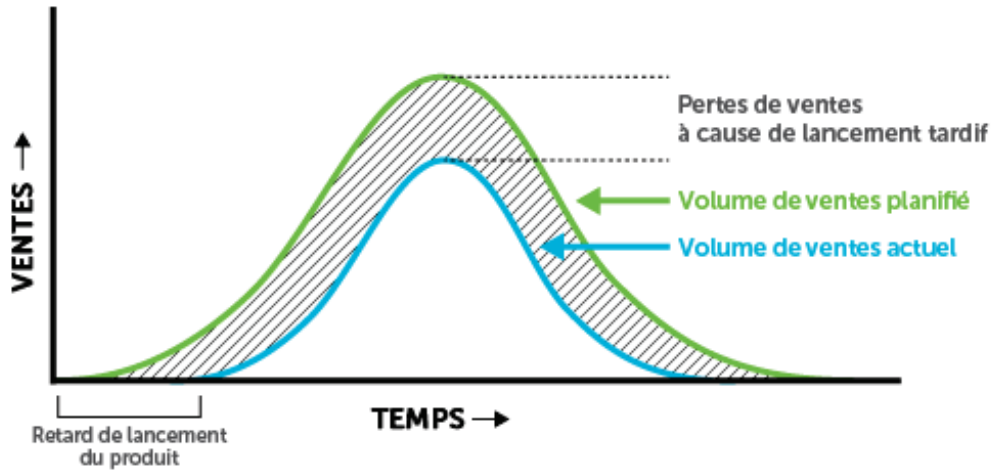

Figure 3.2 – Représentation du TTM

La figure 3.2 montre une représentation de l'évolution du TTM.

La notion du TTM joue un rôle important dans nombreux projets informatiques, il est souvent utilisé en combinaison avec les méthodes de développement agiles. Les grandes entreprises telles que Amazon, Facebook et Google ont adopté les principes de *Lean development* et de déploiement continu (*continuous deployment*) leur permettant ainsi de déployer une version de leurs logiciels plusieurs fois dans la journée.

Dans un rapport intitulé *"Agile Impact Report"*, le cabinet QSM a publié des résultats montrant que les équipes agiles étaient en moyennes 50% plus rapides en terme de TTM et 25% plus productif[1].

Nous pouvons affirmer que la combinaison de tous les outils fournis par les méthodes agiles entraine une amélioration conséquente du TTM. Ceci est dû en particulier au principe de la règle des 80/20, où on livre 20% du code de l'application qui contient 80% des fonctionnalités couramment utilisées par les utilisateurs.

Réduire le temps de mise sur le marché doit être un objectif présent dans chaque projet informatique. Néanmoins on ne doit pas viser uniquement cet objectif, car si on réussit à livrer des logiciels à temps et au bon moment, il n'empêche qu'il faut veiller sur la fiabilité des applications produites.

[1] Source: http://nyspin.org/QSMA-Rally%20Agile%20Impact%20Report.pdf

3.3 Augmenter la fiabilité

En génie logiciel, la fiabilité (*reliability*) désigne la capacité du logiciel à fournir les résultats attendus. Mais aussi, de garder un niveau de fonctionnement constant quelles que soient les conditions d'utilisations du logiciel.

Dans la majorité des cas de défaillance des systèmes informatiques, l'origine est logicielle. Le symptôme évident d'un manque de fiabilité est sans doute les bugs. On trouve deux type de bugs qui peuvent affecter les logiciels :

- le bug technique qui résulte d'une erreur dans le code.
- le bug qui résulte d'une mauvaise conception (non conformité d'une fonctionnalité au besoin).

Les exemples des défaillances logicielles qui ont coûté cher dans l'histoire sont nombreux. On peut citer comme exemple le bug qui a coûté la destruction de la fusée Ariane 5 après seulement 40 secondes de son décollage. A l'origine de cette catastrophe un bug informatique, en effet les ingénieurs informatiques ont utilisé la version du logiciel pour Ariane 4 dans la nouvelle fusée. Ce logiciel n'était pas adapté pour cette nouvelle fusée qui demande plus de puissance pour ces calculs. il faut savoir que cette erreur a coûté la bagatelle de 500 millions de dollars.

Malheureusement, tous ces cas d'erreurs ne sont pas spécifiques à certains types de projets, car on peut citer des milliers d'exemples similaires et pour des projets d'ampleurs différentes.

A l'origine de ses erreurs techniques il y a le développeur. Un développeur professionnel fait en moyenne entre 10 et 15 erreurs pour 1000 lignes de code [2].Or, la taille et la complexité des projets informatiques ne cesse d'augmenter au fil du temps. Par exemple :

[2]Livre «code complete» Sean Paul

- la NASA a besoin de 500 000 LOC pour le logiciel embarqué dans la navette spatiale et de plus de 3 millions de LOC au sol.
- Microsoft utilisa 3 millions de LOC pour Windows 3.1 et aujourd'hui elle a besoin de 40 millions de LOC pour Windows 7[3].

Sachant qu'un logiciel commercial fait on moyenne 350 000 LOC, on peut déduire que le nombre d'erreurs potentielles dans le code de ce dernier sera de 3500. Pour qu'il soit considéré comme un bon logiciel il ne doit pas dépasser 2 erreurs dans 1000 LOC.

Afin d'assurer la fiabilité du logiciel il faut mener une double action de vérification et validation. La vérification consiste à vérifier si le développement répond exactement aux besoins exprimés par le client. La validation quand à elle, valide l'étape précédente et confirme que le logiciel répond aux besoins du client et qu'il fonctionne correctement.

Parmi les méthodes de vérification et validation on trouve :

- Test statique : consiste à faire de la revue de code, des spécifications et des autres documents techniques.
- Test dynamique : consiste a exécuter le code avec des données différentes pour s'assurer de son bon fonctionnement.
- Vérification symbolique : consiste à la recherche d'erreur pendant l'exécution du logiciel (débordement de mémoire, erreur de cast, des exceptions...).
- Vérification formelle : consiste à faire la vérification sur le modèle formel du logiciel.

Actuellement, seuls les tests dynamiques sont répandus et utilisés dans les projets informatiques. Dans ce dossier nous aborderons uniquement les tests dynamiques.

[3] source : http://www.informationisbeautiful.net/visualizations/million-lines-of-code/

Tout d'abord c'est quoi un test dynamique ? selon G.J Myres : "Tester, c'est exécuter le programme dans l'intention d'y trouver des anomalies ou des défauts"[4]

Afin de nettement diminuer le risque de voir surgir des erreurs, les tests dynamiques sont la solution préconisée pour atténuer les dommages des bugs. Il faut savoir qu'il est plus facile et peu coûteux de détecter et corriger les bugs pendant le développement du logiciel, par contre lorsque le logiciel est prêt à être livré, les frais de correction des bugs sont élevés. Les meilleurs estimations estiment les frais de correction lors de la phase finale à 30 fois plus cher que pendant le processus de développement[5].

Les méthodes agiles prônent la réalisation des tests en parallèle au processus du développement du logiciel. Cette approche permet de garantir la qualité attendue du logiciel. Pour une meilleure efficacité des tests, il doit y avoir une stratégie de test qui définit le plan à suivre, les outils et les compétences nécessaires.

Le test est un élément central dans le processus de développement d'un logiciel. Les tests à eux seuls utilisent 40% à 60% du budget alloué à un projet informatique. Avec les bons outils de tests, la fiabilité du logiciel augmente grâce à l'efficacité et à la simplification du processus de vérification et de correction des fautes. Ainsi les fautes sont détectées dès le début du projet pour éviter que ses dernières causent des dégâts très coûteux.

A la démarche d'amélioration de la qualité logicielle qu'on vient de décrire, il lui manque un élément essentiel, la rétroaction[6]. En effet, sans rétroaction le logiciel ne sera pas amélioré de façon à ce qu'il réponde exactement aux attentes des utilisateurs.

[4] G. J. Myers (The Art of Software Testing, 1979)
[5] Les Hatton, professeur de l'université Kingston de Londres
[6] En Anglais : Feedback

3.4 Rétroaction

La rétroaction est une pratique essentielle dans les projets agiles, elle permet d'améliorer constamment et de façon automatisée le logiciel en prenant en compte les retours fournis. Afin que ce processus soit efficace il faut respecter certaines règles :

- Tout changement dans le code du logiciel doit déclencher le processus de rétroaction.
- La rétroaction doit être soumise rapidement.
- La rétroaction doit être traitée rapidement.

En général il y a quatre composants essentiels dans un logiciel: l'exécutable, les fichiers de configuration, l'environnement hôte et les données (ex: base de données). Le comportement du logiciel change si un des ses composants est modifié. Pour cela il faut garder ses composants sous surveillance et veiller à ce que le changement dans un de ces derniers soit vérifié et validé.

La modification du code source du logiciel entraine un changement de l'exécutable, qui doit être ensuite compilé et testé à nouveau. Ce processus répétitif est source de fautes s'il est exécuté par des humains, afin de le contrôler il doit être automatiser. Cette pratique qui consiste à compiler et à tester le logiciel à chaque contribution s'appelle "**Intégration continue**".

L'exécutable généré doit être ensuite déployé sur plusieurs environnements, que ce soit en production, développement ou test. Il faut **s'assurer que l'exécutable ne soit pas recompilé** pour ses différents environnements.

Toutes les modifications des fichiers de configuration de l'application doivent être capturées et traitées dans le but de tester cette nouvelle configuration. Le traitement

de ces modifications doit être automatisé. Il est question ici de la notion de gestion de configuration.

En cas **de changement dans l'environnement hôte de l'application**, alors elle doit être testée de nouveau en prenant en compte les modifications de l'environnement hôte. Ses changements concernent le changement du système d'exploitation, de la configuration réseau et de l'infrastructure. Il est question ici de la gestion de l'infrastructure et de l'environnement.

Dans la mesure du possible, le processus de rétroaction consiste à tester tout changement affectant l'application d'une façon entièrement automatique. Les tests devront vérifier au moins les points suivants :

- le code source de l'application doit être valide.
- les tests unitaires doivent être exécutés avec succès afin de prouver que l'application se comporte comme prévu.
- les tests d'acceptation fonctionnels doivent être exécutés avec succès afin de vérifier que l'application se conforme aux critères métiers. Ces critères sont définis en fonction des valeurs métiers poursuivies par les créateurs du logiciel.
- l'application doit répondre à des critères de qualité comme la couverture du code et d'autres mesures spécifiques.
- le logiciel doit passer des tests d'exploration et de démonstration auprès des utilisateurs. Ceci afin de permettre de trouver des fonctionnalités manquantes ou des bugs qui pourront ensuite être vérifiés par des tests afin d'éviter la régression.
- l'environnement d'exécution des tests doit se rapprocher au plus de l'environnement de production pour vérifier qu'un changement de l'environnement hôte n'affecte pas le bon fonctionnement de l'application.

L'automatisation du processus de rétroaction nous conduit vers la deuxième règle que ce processus doit respecter. Avec l'automatisation le seul obstacle reste la limitation du matériel, alors que si le processus est effectué manuellement il deviendra source d'erreurs, en effet les développeurs introduisent des erreurs dans le code et ne sont pas auditables. En plus, les processus manuels sont répétitifs et ennuyeux pour les développeurs, ses derniers sont des ressources précieuses qui doivent être utilisées pour créer des applications qui raviront leurs utilisateurs et laisser les tâches fastidieuses aux machines.

Le processus de rétroaction doit impliquer toutes les compétences qui se chargent habituellement du déploiement. Les développeurs, les testeurs, les administrateurs des bases de données, les architectes des infrastructures et les chefs de projets doivent collaborer ensemble afin d'améliorer le processus de rétroaction et de déploiement. Avoir un processus d'amélioration continu est essentiel pour construire un processus de déploiement rapide capable de livrer des logiciels de qualité.

Capturer et traiter une rétroaction rapidement signifie la diffusion de l'information. En effet, il est important d'avoir des tableaux de bord qui diffusent l'information au sein des équipes du projets. Ces tableaux de bord doivent utiliser le support électronique pour la diffusion et doivent être présents partout au moins un pour chaque équipe.

Finalement, la rétroaction n'est utile que si elle est prise en considération et pour ce faire il faut avoir une certaine discipline et le sens de planification. les équipes de développement doivent être capables d'arrêter leur travail en cours et de s'occuper de la rétroaction sinon l'objectif même du processus de rétroaction ne sera pas atteint.

Dans ce travail nous allons essayer d'apporter des solutions aux problématiques abordées dans ce chapitre.

4 État de l'art

4.1 Introduction

Avant de présenter le déploiement continu, il est important de reconnaître le contexte global dans lequel le déploiement est situé: le cycle de vie du logiciel (SDLC). Ce chapitre présente les concepts fondamentaux du génie logiciel nécessaires à la production du logiciel. Nous parlerons des nouvelles avancées dans le domaine du génie logiciel et nous introduirons le déploiement continu ainsi que les pratiques nécessaires à son utilisation.

4.2 Cycle de vie du logiciel

Les progrès du génie logiciel ont radicalement changé la façon dont le logiciel est développé. Auparavant, le logiciel était créé en deux phases, l'analyse et le développement. Actuellement, le processus de développement de logiciel est devenu plus méthodologique et contient plusieurs phases avec des caractéristiques distinctes.

Chaque étape nécessite des compétences spécifiques exercées par des acteurs spécialisés dans différentes activités du développement logiciel. Le résultat, le logiciel peut être considéré comme une entité vivante, il change et évolue au cours de sa durée de

vie grâce à une série d'activités. Le génie logiciel vise à définir les tâches, les activités et les processus nécessaires au développement et à la maintenance des logiciels.

La norme IEEE sur les processus du cycle de vie du logiciel (ISO/IEC 12207 à 2008), définit une liste exhaustive des processus appliqués au cours du cycle de vie du logiciel. Ces processus sont regroupés en différents groupes tels que les processus de l'accord, les processus organisationnels, les processus de gestion de projet et les processus techniques. Le domaine du génie logiciel s'intéresse surtout aux procédés techniques qu'on peut définir comme la suite:

Analyse des besoins : vise à définir les objectifs du projet. Elle identifie les parties prenantes qui sont impliquées dans le projet, ainsi que leurs besoins et désirs. Ensuite, les besoins et désirs exprimés sont analysés et réduits en un ensemble d'exigences qui expriment le fonctionnement attendu du système. Comme dans un projet non informatique, cette analyse peut être soutenue par une étude préalable du marché et de faisabilité, afin d'identifier les besoins des parties prenantes et si ils peuvent être satisfaits. Enfin les exigences sont transformées en un ensemble d'exigences techniques qui guideront la conception du logiciel.

Conception : processus axé sur la création du squelette du logiciel: sa conception globale. Le logiciel est divisé en plusieurs éléments et les exigences doivent être adressées par des éléments du système bien identifiés. Chaque élément est défini en termes d'opérations attendus, ainsi que ses relations avec les autres éléments du système. La phase de la conception d'architecture globale agit comme un plan et facilite la phase suivante, celle du développement. Elle améliore la prévisibilité sur projet en termes de coût et de temps.

Implémentation : consiste en la réalisation des éléments du système spécifiés lors de la phase de conception. Elle se compose principalement de l'activité de programmation. Les éléments du logiciel en résultant doivent être conformes aux spécifications désignées lors de la phase précédente. Des éléments individuels du logiciel peuvent

souvent être développés en parallèle et indépendamment.

Intégration : processus qui rassemble les éléments du système (y compris les logiciels, le matériel, autres systèmes tiers, etc.) afin de produire un système complet qui satisfera la conception du système et les exigences. Cette étape comprend habituellement le processus de build, qui est responsable de la construction d'un exécutable à partir du code source. Le processus de build applique des opérations telles que la compilation et la liaison en fonction de la technologie dans laquelle les éléments logiciels sont mis en œuvre. A la fin du processus d'intégration, le système est prêt à être testé dans son ensemble pour la vérification de la qualité.

Tests : ce sont des procédés réalisés transversalement aux autres phases de la SDLC pour vérifier et évaluer le logiciel. Il existe différents types de tests qui valident la conformité du système avec les spécifications et les exigences de la conception. Ils définissent généralement des critères pour évaluer le système pour la livraison. Les tests unitaires vérifient si chaque élément du système effectue se comporte comme prévu par la phase conception. Les tests d'intégration valident si l'assemblage des différents éléments est conforme aux attentes. Les tests d'acceptation de l'utilisateur vérifient que le logiciel résultant est conforme aux attentes des utilisateurs.

Installation : est l'ensemble des activités pour amener le logiciel à l'environnement cible en exécutant les configurations nécessaires pour l'exécution du logiciel sur les infrastructures existantes. Selon les environnements ciblés par le logiciel, (ordinateurs, les serveurs d'entreprise , etc.), ces activités peuvent être incluses dans le cycle de vie du logiciel (livré avec l'installation sur site), ou laissées à la volonté des utilisateurs. Dans les deux cas, l'installation doit faire en sorte que le système fonctionne comme prévu.

Maintenance : processus qui vise à garder le logiciel dans un état de fonctionnement optimal après son installation. Un logiciel opérationnel est soumis à des dysfonctionnements et des changements. Ceci peut être dû à une erreur dans la phase de

développement (par exemple un bug), ou d'un élément manquant. Une fois que le problème est signalé, une équipe de maintenance devra corriger le problème, proposer une solution et mettre à jour l'application.

Désinstallation : processus qui termine l'existence du logiciel. Il arrête le service de la maintenance, désactive et supprime le logiciel des environnements cibles. A la fin, il devrait laisser les environnements dans un état acceptable état, conformément aux exigences et aux accords prédéfinis.

Point important à noter est que l'ensemble du document standard IEEE ne définit ni ne se réfère à la notion de déploiement. Néanmoins, les processus d'installation, de maintenance et de désinstallation sont décrits comme se produisant dans l'environnement cible où le logiciel fonctionne. Ces processus techniques décrivent un développement linéaire pour un projet informatique, où chaque processus succède à un autre. Il est également reconnu que les éditeurs de logiciels sont libres de faire des personnalisations et des ajustements à ces processus et à la façon dont ils sont appliqués.

Un processus de développement logiciel décrit les activités réalisées à chaque étape d'un projet de développement logiciel. Ces modèles contiennent également des méthodes, des concepts et des bonnes pratiques pratiques pour la rationalisation du processus de développement. Au fil des années, divers modèles sont apparus, apportant des ajustements au processus de développement. La section suivante décrit les modèles de processus de développement les plus connus, en comparant les activités de déploiement dans chacun de ses modèles.

4.2.1 Cycle en cascade

Le modèle en cascade (en anglais : « Waterfall model ») est le plus fondamental et le plus ancien de tous les processus de développement qui formalisent les différentes étapes du cycle de vie du logiciel. Apparu en 1970 par W. W. Royce, Il implique l'application successive des phases de développement, d'analyse des besoins, de

conception, codage, test, installation et maintenance. Une phase commence une fois que la précédente est finie. Le principal intérêt du modèle en cascade est qu'il est facile à comprendre et convient aux projets où la qualité a plus d'importance que les coûts ou les délais. Il permet aussi aux développeurs inexpérimentés de travailler selon une structure rigide et bien définie.

Les besoins sont fixés au début du projet, de sorte qu'ils sont bien connus d'avance par les développeurs et les parties prenantes. Toutefois, le modèle en cascade est inapte pour la plupart des projets de logiciels en raison de sa structure inflexible. Il est difficile de répondre aux problèmes rencontrés en cours du projet et de répondre aux demandes de changement des besoins en utilisant ce modèle, car cela nécessiterait de dérouler toute la cascade depuis le début. En outre, passer du temps à analyser des besoins qui sont susceptibles de changer ralentit le développement du logiciel.

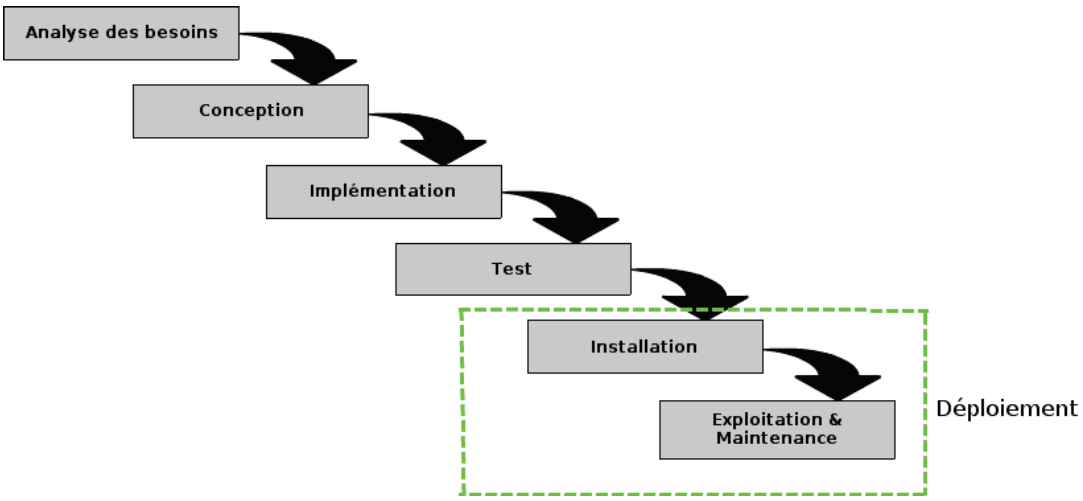

Figure 4.1 – Cycle en cascade

La figure 4.1 montre une représentation du cycle en cascade.

Dans le modèle en cascade, le processus de déploiement n'est pas décrit explicitement. Le logiciel est délivré une fois qu'il est entièrement développé et testé, ce qui signifie que l'étape de déploiement se produit à la fin du projet. Il correspond aux activités d'installation, d'exploitation et de maintenance.

4.2.2 Cycle Itératif

Le modèle de développement itératif vise à réviser et à améliorer le développement du logiciel en appliquant de multiples cycles de développement jusqu'à ce qu'il soit décidé que le logiciel réponde aux exigences. Chaque cycle comporte la même séquence d'étapes que le cycle en cascade. Le développement se fait de manière itérative jusqu'à ce que le logiciel soit jugé prêt à être déployé. Ce cycle, impose que les fonctions importantes avec un facteur risque élevé soient développés au début du projet, ensuite livrés aux clients afin de recevoir rapidement les retours pour les nouvelles itérations.

Parce que chaque cycle produit une version opérationnelle pour les clients, le délai de livraison se retrouve ainsi réduit. Avec des mises à jour fréquentes, l'équipe de développement peut réagir à l'évolution des besoins et ainsi ajuster le fonctionnement du logiciel en conséquence pour les prochaines versions.

La figure 4.2 montre une représentation du modèle itératif.

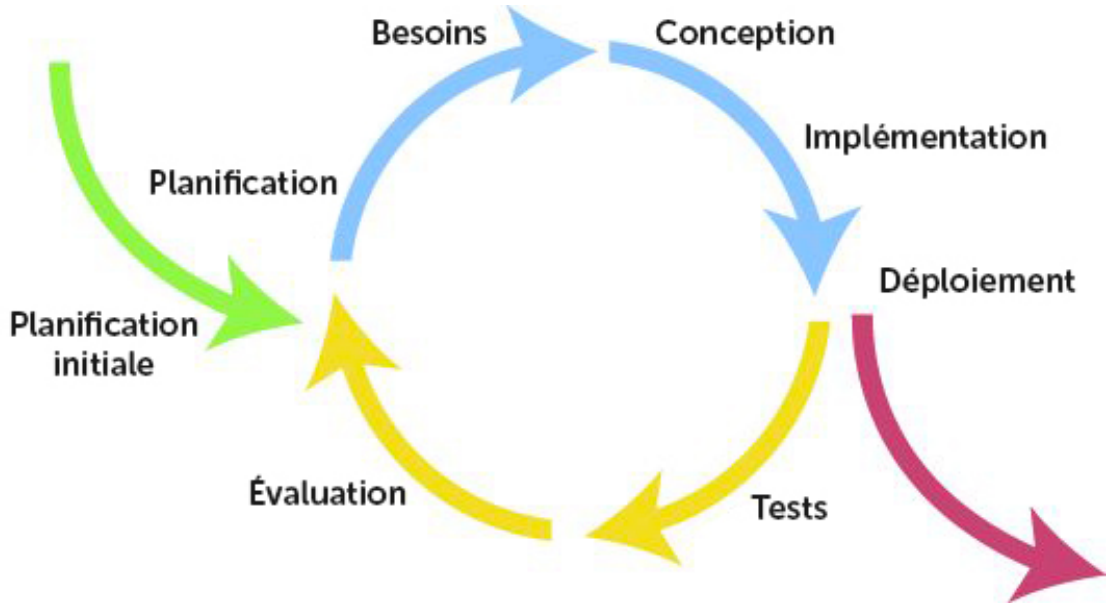

Figure 4.2 – Cycle itératif

Dans le modèle de développement itératif le déploiement se produit uniquement à la fin de certains cycles, lorsqu'une nouvelle version est décidée à être livrée au client. Par rapport au cycle en cascade, le logiciel est déployé beaucoup plus fréquemment. Même ainsi, les détails du processus de déploiement ne sont pas expliqués dans le modèle.

4.3 Les méthodes agiles

Les méthodes agiles sont un ensemble de méthodes qui sont basés sur le développement incrémental et itératif. Elles combinent le développement itératif avec le développement incrémental. Le modèle de développement incrémental divise le logiciel en incréments qui sont capables d'offrir les fonctions requises. Ensuite, les efforts des équipes de développement sont répartis sur l'ensemble des parties de haute priorité. Contrairement à l'approche monolithique où toutes les différentes parties sont assemblées à la livraison du logiciel, dans le modèle incrémental chaque incrément est constamment intégré à l'ensemble du système dès qu'il est terminé. La figure 4.3 illustre ce double fonctionnement.

Le manifeste agile[1] s'est initié sur la notion que la sur-planification et la sur-formalisation entrave le processus de développement, ce qui par la suite retarde la livraison des logiciels. Afin d'éviter cela, les méthodes agiles favorisent l'approche du développement itératif avec des temps d'itérations courts. Au lieu de passer du temps sur la façon de planifier les grandes décisions de conception, les méthodes agiles encouragent à travailler en étroite collaboration avec les clients pour comprendre leurs besoins et de réagir rapidement aux demandes des changements.

[1] Agile manifesto: http://agilemanifesto.org

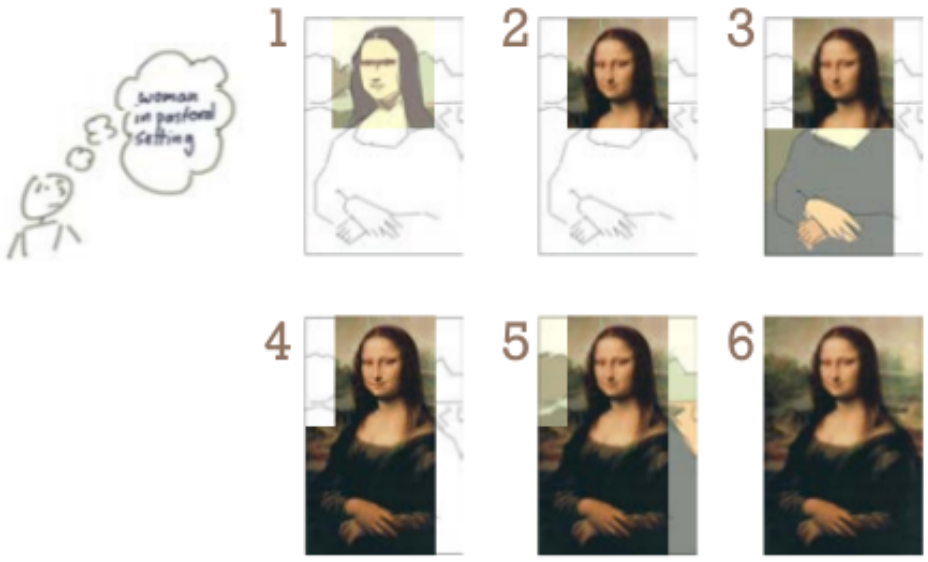

Figure 4.3 – Développement incrémental et itératif

Source: Photo originale créée par Jeff Patton: http://www.agileproductdesign.com

L'objectif principal de l'équipe de développement est de livrer les premières versions fonctionnelles du logiciel pour les clients et de garder le logiciel en état de fonctionnement. L'importance du déploiement augmente parce que chaque itération est susceptible de finir en livrant un logiciel opérationnel.

Avec les méthodes agiles le développement du logiciel est préféré sur la planification exhaustive et la documentation technique. Cela provoque beaucoup de critiques envers les méthodes agiles, en faisant valoir que ses dernières manquent de discipline pour le développement des logiciels importants. Boehm et Turner dans leur livre discutent de la division entre la discipline et l'agilité[2]. Ils démontrent que la discipline seule sans agilité conduit à la bureaucratie et la lenteur, alors que l'agilité seule sans discipline conduit à un enthousiasme incontrôlé et insignifiant. Ils présentent l'agilité comme un facteur qui améliore la discipline, grâce à son inventivité et sa flexibilité. En effet, comparée à des méthodologies plus structurées et axées sur la planification, le développement agile nécessite des équipes de développement expéri-

[2]Livre de Boehm et Turner «Balancing Agility and Discipline»

mentées capables de s'adapter aux changements, mais aussi capables d'avoir un sens de discipline.

Il existe plusieurs méthodes qui organisent les équipes de développement selon les principes agiles. les plus connues d'entre elles sont eXtreme Programming, Lean, Scrum et Kanban. Dans la partie suivante nous étudierons deux méthodes agiles parmi les plus connues et utilisées dans les projets informatiques.

4.3.1 Scrum

SCRUM est un mot d'origine anglaise qui veut dire: mêlée. **SCRUM** est aussi une des méthodes agiles les plus connues et utilisées dans les projets informatique. Conçue au départ pour les projets informatique, cette méthode peut être utilisée dans n'importe quel type de projet, du plus simple au plus complexe, et ce de manière très simple.

Les projets gérés par la méthode «**SCRUM**» sont divisés en plusieurs cycles appelés «**sprints**». Ces itérations durent généralement entre deux à quatre semaines d'effort basé sur le développement, les tests et le déploiement d'une fonctionnalité spécifique. Au début de chaque **sprint**, le client peut intervenir et redéfinir les priorités ou demander des changements. Cela permet aux équipes de développement de réajuster ou réorienter la direction prise par le projet sans remettre en cause le travail déjà développé. En effet, les équipes de développement se baseront sur ce qui est a été développé et validé par le **sprint** et non pas sur des prédictions.

Grâce aux retours positifs de la part des chefs de projets et des développeurs, **SCRUM** est devenue très utilisée dans les projets informatiques. En effet, la méthode **SCRUM** définit des principes qui ne changent jamais tels que les rôles, les responsabilités et les réunions, tout en gardant une gestion flexible des projets. Cela permet de rassurer l'équipe de développement lors de certaines phases critiques du projet.

Dans la pratique, le projet géré par la méthode **SCRUM** se constitue autour d'une

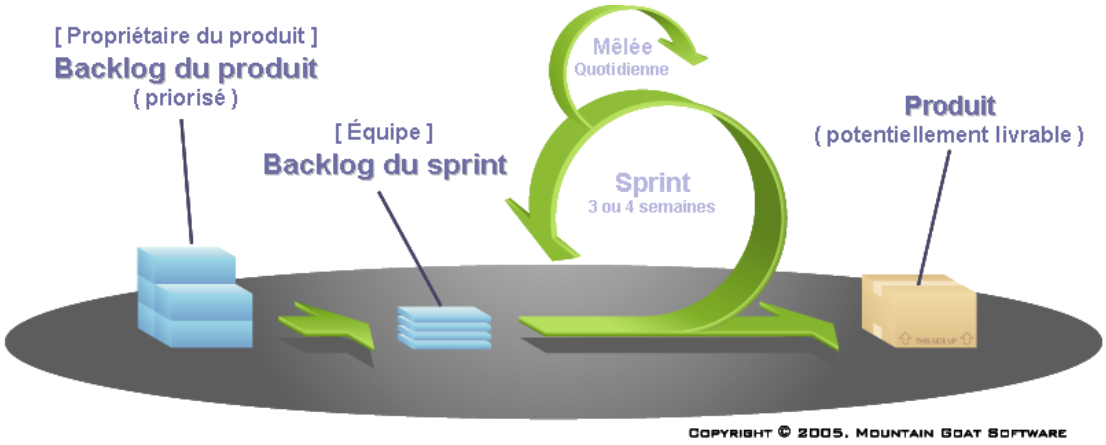

Figure 4.4 – Flux de travail Scrum

équipe autogérée et pluridisciplinaire. L'auto-gestion vient du fait qu'il n'est y a pas de notion de chef de projet qui définit le rôle de chacun et décide quelle direction doit prendre le projet, car c'est l'équipe dans son ensemble qui participe aux résolutions des problèmes rencontrés. La pluridisciplinarité découle du fait que l'équipe de développement réunit plusieurs compétences (développeurs, graphistes, testeurs, etc.) parmi ses membres.

La figure 4.4 illustre le flux de travail avec la méthode **SCRUM**.

On distingue trois importants rôles dans la méthode **SCRUM**: Le Scrum Master, le propriétaire du produit et l'équipe.

Le Scrum Master : Ce dernier joue l'intermédiaire entre le propriétaire du produit et l'équipe. Il ne gère pas l'équipe comme le fait un chef de projet traditionnellement, mais son rôle est d'identifier et d'éliminer les obstacles pour que l'équipe atteigne les objectifs fixés dans chaque sprint. Il doit veiller aussi à la protection de l'équipe des interférences externes. En résumé, grâce à son rôle la créativité et la productivité de l'équipe augmente significativement.

Le propriétaire du produit : « Product Owner » en Anglais, ce dernier représente à la fois les clients et les utilisateurs. Parmi ses responsabilités, définir les besoins des clients et les prioriser pour ensuite les communiquer à l'équipe de développement.

A ce sens, il est celui qui détient plus de responsabilités et d'autorité. En effet, si les choses ne se passent pas comme prévu il est le premier responsable.

L'équipe : Dans la méthode Scrum, l'équipe est chargée de la réalisation du sprint et du résultat final. Elle est composée généralement d'un nombre limité de personne avec des compétences différentes, afin d'améliorer la communication au sein de l'équipe. En effet, elle peut être composée de développeurs, de graphistes et de testeurs.

La méthode **SCRUM** implique que le projet avance à travers une série de cycles courts appelées « sprints ». Avant le démarrage d'un sprint, une réunion est organisée afin de créer une liste des tâches à exécuter, « sprint blacklog », au cours du sprint et de distribuer les tâches pour chaque membre de l'équipe.

Tous les jours du sprint, une réunion quotidienne, mêlée quotidienne, qui ne doit pas dépasser 15 minutes, doit se tenir entre l'équipe, le Scrum master et le PO. Dans cette réunion, chaque membre de l'équipe doit partager ce qu'il a accompli le jour précédent et aussi les obstacles qu'il a rencontrés pendant l'accomplissement de sa tâche. Ce partage doit servir pour tous les membres de l'équipe afin d'identifier les sources de problèmes qui peuvent entraver le bon déroulement du projet et que chacun des membres de l'équipe soit au courant de ce que font les autres membres de son équipe.

Le sprint se termine généralement par une présentation du résultat au PO au cours d'une réunion, qui doit permettre de valider le travail effectué ou de corriger les erreurs pour les futurs sprints.

Cependant, **SCRUM** ne décrit pas les contraintes ou les lignes directrices sur le déploiement du logiciel.

4.3.2 Kanban

Kanban est un terme japonais qui signifie « étiquette ». Ce terme est apparu dans les chantiers navals, qui dans les années 60, avaient besoin de livraison d'acier tous les 3 jours. Plus récemment cette méthode a été développée et propulsée dans les usines Toyota grâce à l'aide de Taiichi Ohno. Depuis 40 ans, Toyota continue d'utiliser et d'améliorer cette méthode.

Kanban est une méthode basée essentiellement sur une gestion de stock à zero et sur un flux de travail basé sur la circulation des étiquettes. Parmi les méthode agiles, Kanban et SCRUM sont les plus utilisées en ce moment pour la réalisation des logiciels. Tous les deux sont des méthodes agiles mais chacune à ses propres processus. Toutefois, il faut avouer que Kanban offre plus de flexibilité dans sa manière de gestion par rapport à SCRUM qui est plus rigide, car ne tolère pas de modifications de ses règles.

Kanban est une méthodologie agile qui vise à établir un flux de travail (Workflow) pour l'amélioration continue du produit. Pour cela, elle se base sur un tableau de bord sur lequel une représentation visuelle est affichée. Cela permet un suivi sur l'avancement des tâches spécifiques et sur l'ensemble du projet.

Sur le tableau de bord, des étiquettes représentant des tâches sont mise en place dans des colonnes qui représentent leurs états courants. Lorsque une tâche évolue elle change de colonne jusqu'à ce qu'elle soit terminée. Chaque colonne du tableau peut contenir un nombre prédéfini de tâches, ce nombre est défini selon la capacité de l'équipe à traiter des tâches simultanées. Ainsi, on maitrise le WIP (Work In Progress). Une chose importante à ce stade, la mesure du **lead-time**, qui représente le temps moyen pour effectuer une tâche. Au fur et à mesure que le projet avance, ce temps ce réduira et deviendra prévisible.

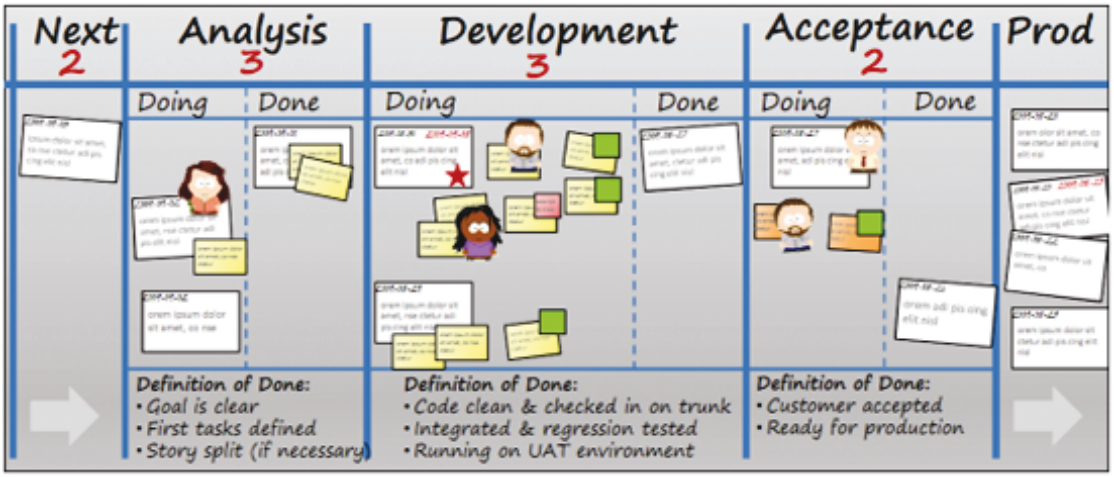

Figure 4.5 – tableau de bord - Kanban

Source: http://www.crisp.se/

La figure 4.5 nous montre une illustration d'un tableau de bord utilisé par la méthode Kanban.

Les avantages qui poussent à l'utilisation de Kanban comme méthode de gestion de projet sont principalement :

- La transition vers les méthodes agiles se fait plus facilement et progressivement comparer à SCRUM qui est plus rigide.
- La facilité du suivi de l'état d'avancement des tâches et du projet, ce qui offre une meilleure communication.
- La maîtrise des risques devient facile. En effet, les problèmes apparaissent très tôt dans le projet ce qui permet une grande réactivité pour les résoudre.
- La notion d'itération n'existe pas, ce qui permet d'éviter une contrainte de temps et permet une réactivité rapide face aux changements ou face aux problèmes.
- Une meilleure qualité grâce à une définition claire de l'ensemble de critères permettant de classer une tâche comme finie.

4.4 L'intégration continue

Une caractéristique commune dans nombreux projets informatique est que pendant des années et durant le processus de développement, le logiciel ne se trouve pas dans un état de fonctionnement. En fait, la plupart des logiciels mis au point par de grandes équipes passe la majorité de son temps de développement dans un état inutilisable. La raison pour cela est facile à comprendre, personne n'est intéressé à essayer d'exécuter l'application jusqu'à ce qu'elle soit terminée. Les développeurs continuent à modifier le code et pourraient même exécuter des tests unitaires, mais personne n'essaie d'exécuter l'application dans un environnement semblable à celui de la production.

Ceci est doublement vrai dans des projets qui reportent les tests d'acceptation jusqu'à la fin. Beaucoup de ces projets programment les phases d'intégration à la fin du développement pour permettre à l'équipe de développement de fusionner leurs modifications de sorte que ses dernières passent les tests d'acceptation. Pire encore, certains projets se rendent compte que quand ils arrivent à la phase d'intégration, le logiciel produit ne fonctionne pas comme prévu. Ce qui implique des phases d'intégration relativement longues et le pire c'est que le temps passé à réaliser cette phase n'est pas quantifiable.

Certains projets informatiques échappent à cette situation, en fait leurs logiciels restent dans un état de non-fonctionnement que quelques minutes suite à des modifications dans le code source et ensuite passent à un état de fonctionnement. Ceci est dû à l'utilisation de l'intégration continue. L'intégration continue exige que chaque fois qu'un développeur *commit* un changement dans le code, un build de l'application entière doit être exécuté et un ensemble complet de tests automatisés doit être lancé. Ensuite, si le processus de **build** ou de test échoue, l'équipe de développement arrête tout ce qu'elle fait immédiatement et s'occupe de résoudre les problèmes. L'objectif de l'intégration continue est que le logiciel reste dans un état de bon fonctionnement

tout le temps.

L'apparition de l'intégration continue, IC, pour la première fois est due au livre de Kent Beck, *Extreme Programming Explained*, publié en 1999. L'idée derrière l'intégration continue est que, si l'intégration habituelle de votre code source se passe correctement, pourquoi ne pas répéter cette phase tout le temps? Pour la IC, tout le temps signifie à chaque commit du code source fait par un développeur.

L'intégration continue s'impose comme une solution incontournable. En effet, sans IC, le logiciel développé est considéré comme non fonctionnel jusqu'à ce que quelqu'un l'exécute sans problème, habituellement dans une phase de test ou d'intégration. Avec l'IC, le logiciel est certifié 100% fonctionnel malgré les nouvelles modifications. Ces dernières peuvent causer des bugs, que l'équipe de développement sait quand elles peuvent surgir et comment les résoudre immédiatement. Les logiciels développés en utilisant l'intégration continue sont connus pour être livrés plus rapidement et avec moins de bugs, par rapport à des logiciels qui ne sont développés avec l'IC. En effet, les bugs sont découverts très tôt dans le processus de développement afin d'éviter qu'ils soient chers à réparer. Ce qui fournit d'importantes économies en coûts et en temps. Par conséquent, on peut considérer l'IC comme une pratique incontournable pour les projets professionnels.

La figure 4.6 qui suit, une représentation des étapes et des acteurs de l'intégration continue.

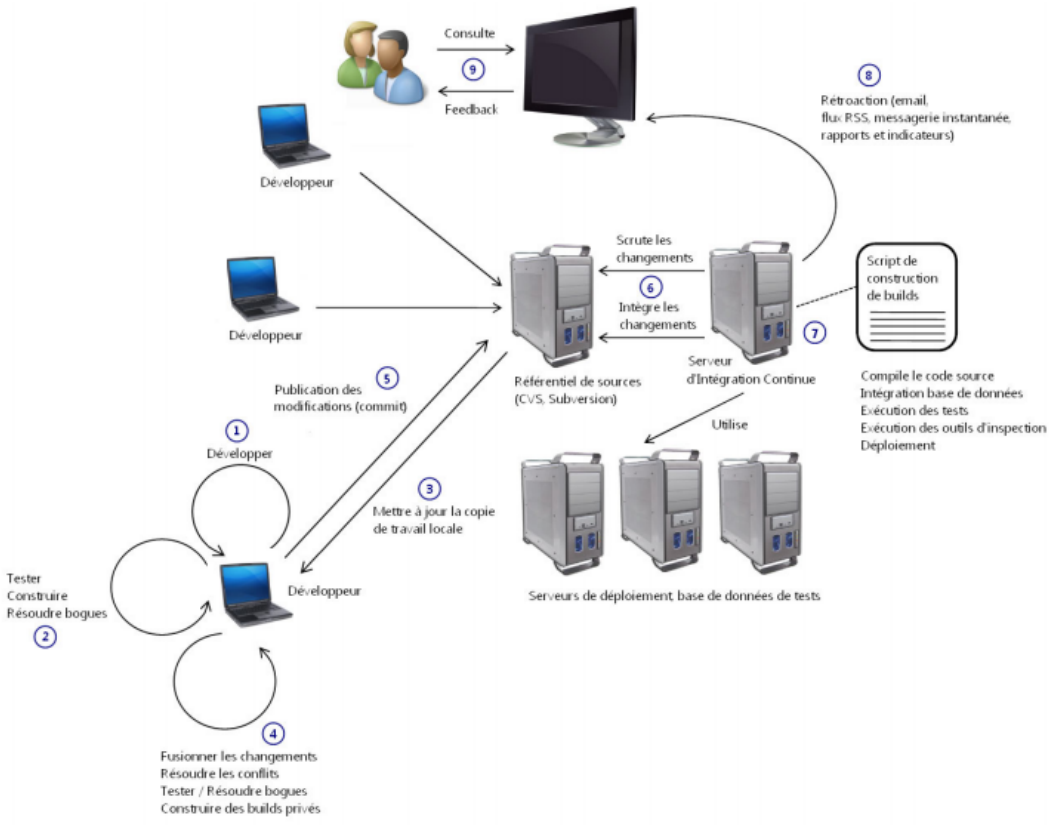

Figure 4.6 – schéma et acteurs de l'IC

Source: http://www-igm.univ-mlv.fr/ dr/XPOSE2011/IntegrationContinue/fonctionnement.html

Il est important de donner une définition du mot "build" avant de continuer d'expliquer le fonctionnement de l'intégration continue. Le build est un terme anglophone qui désigne l'activité principale qui conduit à la construction de l'application. Il comprends plusieurs tâches comme :

- Mise à jour du code source de l'application depuis le gestionnaire de version.
- Compilation du code source.
- Lancement des tests unitaires et d'intégration.

- Revue du code pour vérifier la qualité.
- Génération de la documentation.
- Génération de l'exécutable.

Afin d'implémenter l'intégration continue, certains pré-requis sont indispensables. Nous aborderons ces pré-requis et certaines bonnes pratiques que les équipes doivent mettre en place afin d'assurer un bon fonctionnement de l'intégration continue.

Le premier important pré-requis pour implémenter l'IC est sans doute **un logiciel de gestion de version**. En effet, le développement logiciel implique généralement des développeurs qui chacun travaillent sur une copie du code source et ensuite les copies sont rassemblées pour constituer une seule. Afin de permettre cela, un gestionnaire de version est nécessaire qui en outre garde l'historique des modifications de chacun afin d'assurer un suivi, ce dernier permettra de facilement d'identifier et de résoudre les problèmes potentiels. Il est évident, que le gestionnaire de version doit être utilisé pour archiver le code source, les tests, les scripts de base de données, les scripts du build et les scripts de déploiement.

Le deuxième pré-requis est **l'automatisation du processus de build**. Il faut être en mesure de démarrer ce processus par ligne de commande ou par un ensemble complexe de scripts. Quel que soit le mécanisme utilisé, une personne ou un programme doit être capable de lancer le build, les tests et le processus de déploiement de manière automatisée.

Le dernier pré-requis est **le consentement de l'équipe de développement**. L'intégration continue étant une pratique et non un outil, elle nécessite que chacun des membres de l'équipe de développement consente à respecter ses règles. Notamment, la modification et la publication du code source par petit lot et l'acceptation de donner la priorité absolue à la correction des bugs. Si les membres de l'équipe de développement n'adoptent pas l'esprit et la discipline nécessaire pour l'intégration continue,

toutes tentatives de l'implémenter sera un échec.

L'objectif de l'intégration continue est d'assurer que le logiciel fonctionne tout le temps de son développement. Afin de veiller à ce que ce soit le cas, nous allons présenter les bonnes pratiques qu'il faut mettre en place des équipes de développement. Ces conseils vont aider à changer les habitudes des développeurs pour qu'ils adoptent l'intégration continue.

Publier plus souvent : les développeurs sont réticents à publier leurs modifications dans le code source pour différentes raisons. Parmi elles, la peur de casser le build de l'application et la peur que leurs modifications impactent le travail des autres. Sauf que ces réticences peuvent coûter cher en cas d'apparition de problèmes, car le coût de correction de ces dernières sera élevé. Il est absolument nécessaire de publier le plus souvent possible.

Ne pas publier un code non testé : Un développeur ne doit jamais publier un code qu'il n'a pas passé par les tests ou qu'il ne compile pas. Pour s'assurer que le code est bon, le développeur peut lancer un build privé (un build local sur son poste).

Corriger les builds cassés le plus rapidement possible : Les causes d'un build raté peuvent varier, ça va d'une erreur dans le code source, un test qui échoue, la revue de la qualité du code ne passe pas ou un déploiement qui a échoué. Dans ce cas, l'intégration continue préconise la résolution immédiate des problèmes et ce en faisant participer les développeurs qui ont apporté les dernières modifications.

Écrire des tests automatisés : L'intégration continue ne servira à rien si les tests ne sont pas automatisés. D'autre part, un build doit échouer si les tests échouent.

Les tests et la revue de la qualité du code ne doivent pas échouer : commenter un test parce qu'il a échoué est une mauvaise pratique, car elle participe à la détérioration du taux de couverture de test du code.

Faire des builds privés : Avant de lancer des builds intégrés qui peuvent échouer, les développeurs vont devoir simuler des builds d'intégration sur leurs postes. Ces builds privés ont un double intérêt, le premier est que si le build échoue alors le développeur ne va pas publier un code qui cassera le build. Le deuxieme intérêt, si le build réussit alors le développeur pourra publier son code avec l'assurance d'aboutir à un build d'intégration réussi.

Éviter de récupérer du mauvais code : Quand un build échoue sur le serveur d'intégration, il faut absolument que les développeurs ne récupèrent pas la dernière version du code source. Car, ils perdront leur temps à essayer de corriger des erreurs qui ne sont pas les leurs. En attendant, ils peuvent participer avec les développeurs responsables de l'échec à la correction ou attendre la nouvelle version corrigée du code source. On voit qu'il est très important d'avoir un suivi régulier de l'état de chaque build afin d'informer les développeurs. Pour cela il faut que le serveur d'intégration soit équipé avec un mécanisme d'envoi de courriel.

4.5 Le mouvement DevOps

DevOps est un terme qui regroupe plusieurs concepts, bien que tous non nouveaux, qui se sont catalysés en un mouvement émergent et se répandant rapidement dans la communauté informatique. Lancé par Patrick Dubois en 2009 au cours d'une conférence en Belgique, ce terme est la contraction de deux mots anglais «*development*» et «*operations*». Les deux mots qui représentent les deux fonctions principales dans le monde informatique, les développeurs et les administrateurs systèmes (sysadmins).

La figure 4.8 nous montre une illustration du mur de la confusion qui existe entre les équipes d'un projet informatique.

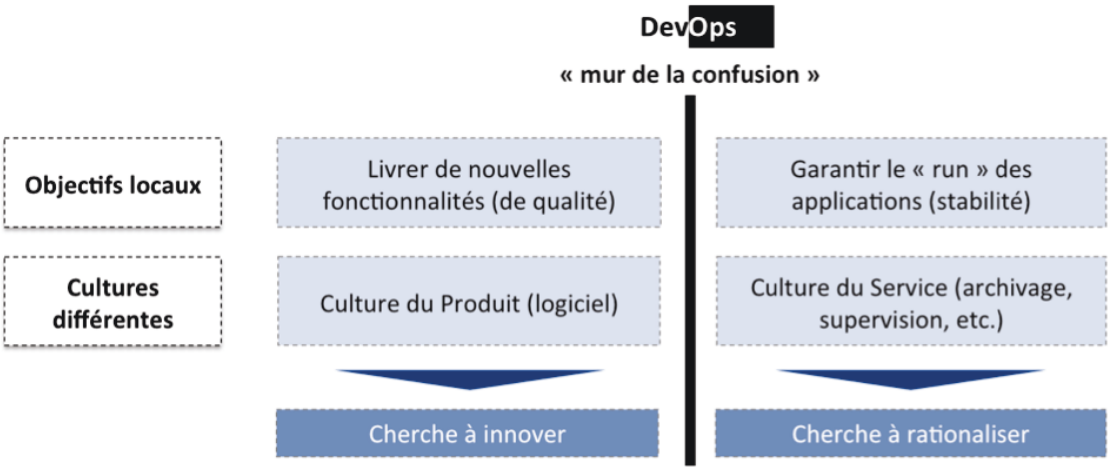

Figure 4.7 – DevOps - Mur de la confusion

Source: http://blog.octo.com/devops/

Ce mouvement a été initié à la base par des administrateurs systèmes mécontent que les méthodes agiles ne s'intéressaient qu'à la partie développement et négligeaient la partie exploitation. Pour remédier à cela, le **DevOps** est apparu afin de supprimer les frontières entre les développeurs et les administrateurs, et ainsi fluidifier le processus de travail entre les deux fonctions dans le but de constituer un seul ensemble capable de livrer rapidement des logiciels de qualité meilleure.

En effet, **DevOps** ne fait pas de distinction entre les différents sous-domaines du sysadmin, "Ops" est un terme générique pour désigner les ingénieurs systèmes, les administrateurs système, les ingénieurs du déploiement, les administrateurs de base de données, les ingénieurs réseau, les professionnels de la sécurité et autres sous-disciplines. "Dev" désigne les développeurs en particulier, mais dans la pratique il est plus large et désigne toutes les personnes impliquées dans le développement du logiciel, comme les QA, les architectes et d'autres disciplines.

DevOps a de fortes affinités avec les méthodes agiles. Les méthodes agiles regardent le côté "Dev" comme étant les décideurs et le côte "Ops" comme étant les personnes qui utilisent le logiciel après son développement. Cette vision a causé du tort à l'industrie informatique car elle sépare ces deux fonctions. A partir de ce constat,

DevOps peut être considéré comme un prolongement des principes des méthodes agiles. En effet, les méthodes agiles exigent une étroite collaboration entre les clients et les développeurs pour développer des logiciels de meilleure qualité. Alors que **DevOps** dit « oui, mais la prestation de services et la façon dont l'application et les systèmes interagissent est une partie essentielle pour la commercialisation du produit ». La figure 4.8 illustre ce principe.

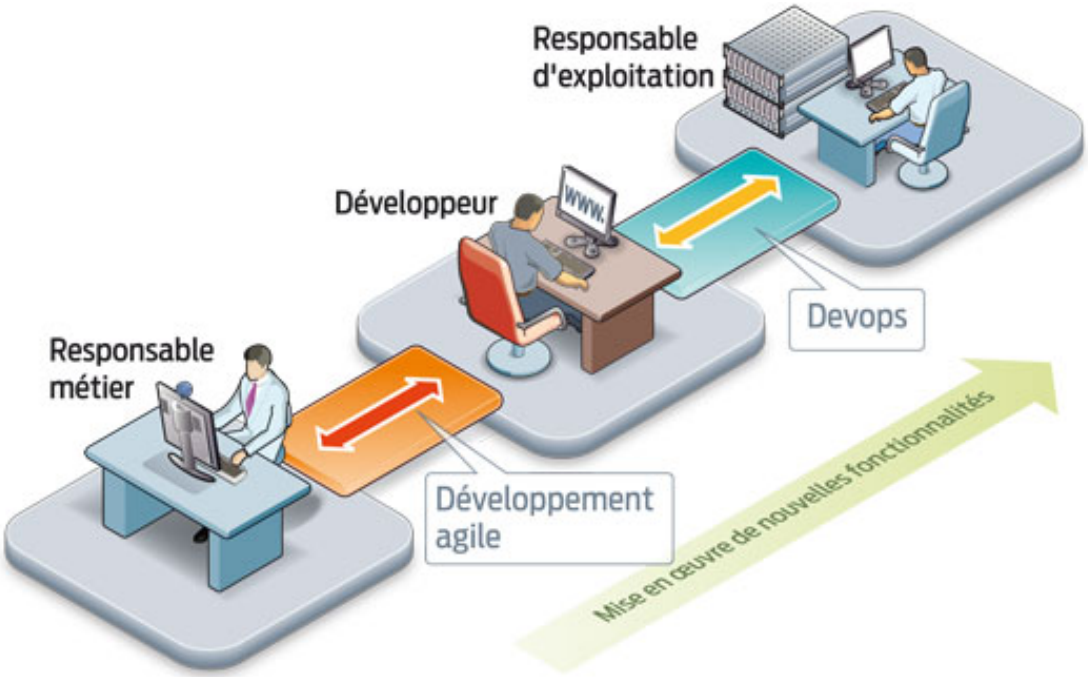

Figure 4.8 – Schéma DevOps

Source: le site 01net.com

En fin de compte, ce que DevOps espère apporter aux méthodes agiles est la compréhension et la pratique que **le logiciel ne se fait pas jusqu'à ce qu'il soit livré avec succès à un utilisateur et qu'il répond à ses attentes en matière de disponibilité, de performance et de rythme de mise à jour.**

DevOps n'est pas un outil ni une méthode de gestion de projet et certainement pas un titre d'emploi. Par contre, on peut dire que c'est une stratégie dans la pratique qui a pour but l'amélioration de la communication entre le développement et l'exploitation afin de réduire le temps de mise sur le marché.

DevOps se base sur trois principaux processus qui sont, **l'intégration continue**, **la livraison continue** et **le déploiement continu**. Ce qui nous amène à la présentation du **déploiement continu** dans la section suivante.

4.6 Le déploiement continu

L'objectif de déploiement est de construire un logiciel en utilisant les artefacts créés dans la phase de développement. Cette section se concentre sur une tendance récente, qui change radicalement la façon dont le logiciel est déployé: le déploiement continu.

Avant d'entrer dans les détails, nous allons définir premièrement la livraison continue et le déploiement continu. **La livraison continue** (*Continuous delivery Cd*) est un ensemble de pratiques qui transforme le cycle de vie du logiciel. Elle peut être grossièrement résumée par l'expression suivante « tous les commits doivent créer une nouvelle version ». Ainsi, chaque modification apportée par un développeur est intégrée dans une nouvelle version du logiciel. **Le déploiement continu** (*Continuous Deployment CD*) prolonge ce principe au déploiement effectif en production de la nouvelle version créé. En résumé, chaque commit est directement poussé à la production.

La figure 4.9 illustre la différence entre livraison continue et déploiement continu.

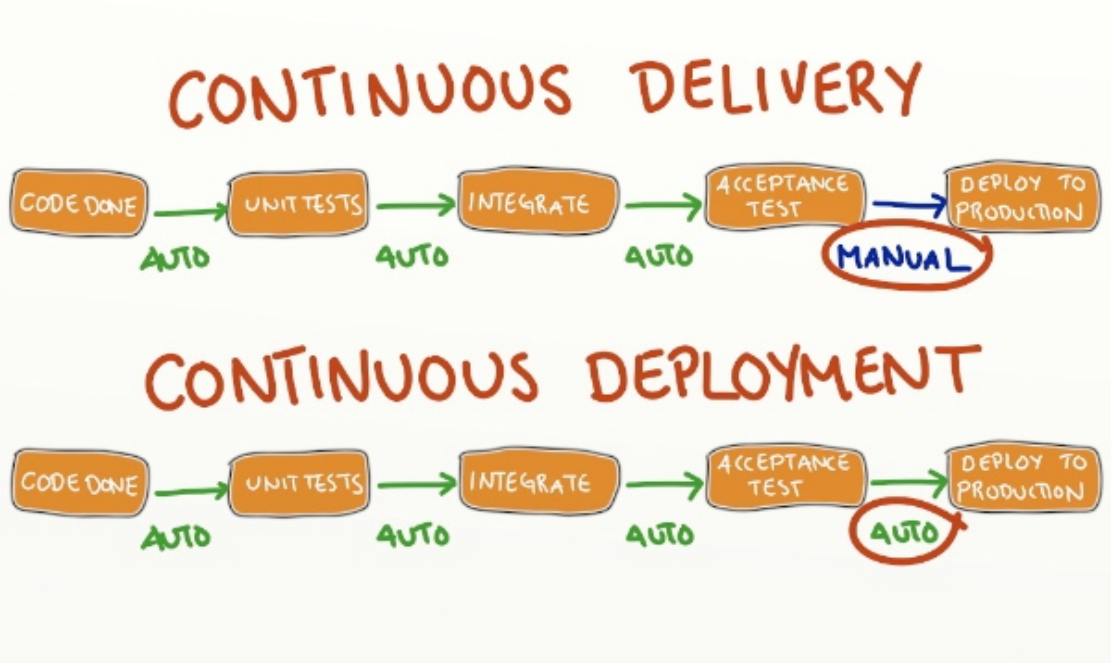

Figure 4.9 – Comparaison entre Cd et CD

Source: Source from Yassal Sundman blog – En Continuous Delivery, le déclenchement de la mise en production est manuel

Évidemment, les deux processus nécessitent des méthodes rigoureuses et des outils sophistiqués, qui tous reposent sur le principe du **pipeline de déploiement**. Ce pipeline représente le voyage de l'application depuis le développement jusqu'au gestionnaire des versions ou l'environnement de production.

Nous allons introduire l'idée générale derrière **le déploiement continu**. Pour cela, on commence par présenter les concepts du *lean development* et du **pipeline de déploiement**. Ensuite, nous examinerons les pratiques actuelles qui sont utilisées pour la mise en œuvre des pipelines de déploiement ainsi que les exigences pour la mise en œuvre de déploiement continu.

Le manifeste agile affirme que la réactivité face au changement est plus importante que de respecter un cadre strict de gestion de projet. Les processus de développement en accord avec cette vision reconnaissent que des changements sont inévitables dans tout le projet et que l'investissement dans la conception des systèmes immuables

est contre-productif. Toutefois, cela ne signifie pas que les producteurs de logiciels doivent faire des compromis sur la qualité des logiciels et la rigueur du processus qui produit ces derniers. Au contraire, les différentes étapes du cycle de vie du logiciel nécessitent des optimisations plus que jamais, pour être en mesure de faire face à ce changement et encore fournir des logiciels de qualité.

Lean Software Development (LSD) est l'application des principes du **Lean Manufacturing** aux processus de cycle de vie du logiciel. **Lean** vise à créer de la valeur avec moins de travail. Le concept de **valeur** est définie comme toute action ou processus qui apporte une valeur ajoutée au produit ou au service. **Lean manufacturing** est basée sur l'optimisation des flux de création de cette valeur dans le but d'accroître l'efficacité et de réduire les gaspillages. Le but de LSD est donc, de réduire le temps et les efforts gaspillés pour produire et déployer les logiciels. Ceci grâce à la mise en place des pratiques et des procédés rigoureux qui reflètent en continu la valeur produite sur le logiciel. **La livraison continue** applique ce principe en transformant chaque valeur créée par les développeurs en une version du logiciel, tout en garantissant la qualité des logiciels développés.

Comme nous l'avons vu précédemment, la méthode **Kanban**, est une méthode agile qui permet de garder la trace sur l'avancement du travail en cours et de la gestion de son flux. Elle fournit un bon cadre pour les organisations qui souhaitent appliquer les principes du **Lean**. Le tableau de bord de **Kanban** représente le flux de valeurs qui sont créés. Le flux de valeur (**value stream**) représente l'avancement des valeurs qui passent par différents états. La clé de réussite en utilisant **Kanban** est que, dans toute activité à valeur ajoutée la quantité totale de travail est limitée. Par conséquent, le flux de valeur contient une quantité limitée de travail. Cela incite les équipes à optimiser et adapter la chaine des activités du flux de valeur afin d'éviter les goulots d'étranglement.

Quand on parle de **valeur**, il existe plusieurs formes sous laquelle **la valeur** produite dans le cycle de vie du logiciel peut prendre forme. La forme la plus connue des valeurs

est sans doute **le code source**, qui est intégré dans l'exécutable. Dans la plupart du temps, les exécutables seuls ne suffisent pas à lancer le logiciel. L'exécution correcte d'un logiciel nécessite d'autres ressources. Parmi lesquelles un autre type de valeur qui est l'ensemble **des fichiers de configurations**, qui sont généralement nécessaires pour exécuter le logiciel. Les logiciels s'exécutent sur des infrastructures logicielles et matérielles qui sont minutieusement déterminées afin de garantir la bonne exécution du logiciel. Ces **environnements** représentent un autre type de valeur que les équipes d'exploitation mettent en place. Enfin, les applications exigent généralement **des données**. Par exemple, les schémas de base de données sont des valeurs produites au cours du développement. Au final il y a quatre types de valeur.

Le processus **lean** encourage les équipes de développement à travailler selon une approche pragmatique, dans laquelle ils peuvent tester des nouvelles idées et obtenir rapidement des retours grâce aux tests automatisés ou les clients. La livraison continue et le déploiement continu, sont des processus qui évaluent continuellement la fiabilité du logiciel à chaque modification du code, de telle sorte que la dernière version fiable du logiciel est toujours disponible. Ceci est possible grâce au pipeline de déploiement qui implémente ce processus.

Le pipeline de déploiement propose une solution pour le problème de **la livraison continue**. Il offre une approche de bout en bout pour la livraison du logiciel en automatisant tous les processus depuis la gestion de version jusqu'à l'exécution. Dans ce pipeline tout changement dans le logiciel doit passer par un processus complexe jusqu'à qu'il soit livré. Ce processus consiste à construire le logiciel (build), suivi de multiples étapes de test, le déploiement dans des environnements différents et enfin la sortie de la nouvelle version.

Un pipeline de déploiement est un processus global qui automatise certaines activités du cycle de vie du logiciel, tels que le build, les tests, le déploiement et la création de nouvelles versions, ce qui permet le suivi de chaque valeur à partir de sa conception jusqu'à sa création dans le système.

Le pipeline de déploiement améliore la visibilité sur la production des logiciels par l'observation et le contrôle de l'état d'avancement de chaque changement à travers différentes activités. Tel que montré précédemment, le cycle de vie du logiciel nécessite l'implication et la collaboration de nombreux acteurs, tels que les développeurs, les testeurs et les équipes d'exploitation. Avoir un pipeline de déploiement global améliore la façon dont de nombreuses personnes de différentes équipes collaborent efficacement.

Le pipeline de déploiement est un système basé sur la production tirée (Pull). Plutôt de pousser les changements vers différents acteurs, les changements produits par les développeurs sont construits et stockées dans des dépôts d'artefacts. De cette façon, les changements sont utilisés pour générer une nouvelle version. Seulement alors les différents acteurs tels que les équipes d'exploitations et les testeurs peuvent tirer ces versions. Le pipeline de déploiement est un processus reproductible, fiable et automatisé qui produit des résultats déterministes.

La figure 4.10 illustre les différentes étapes du pipeline de déploiement.

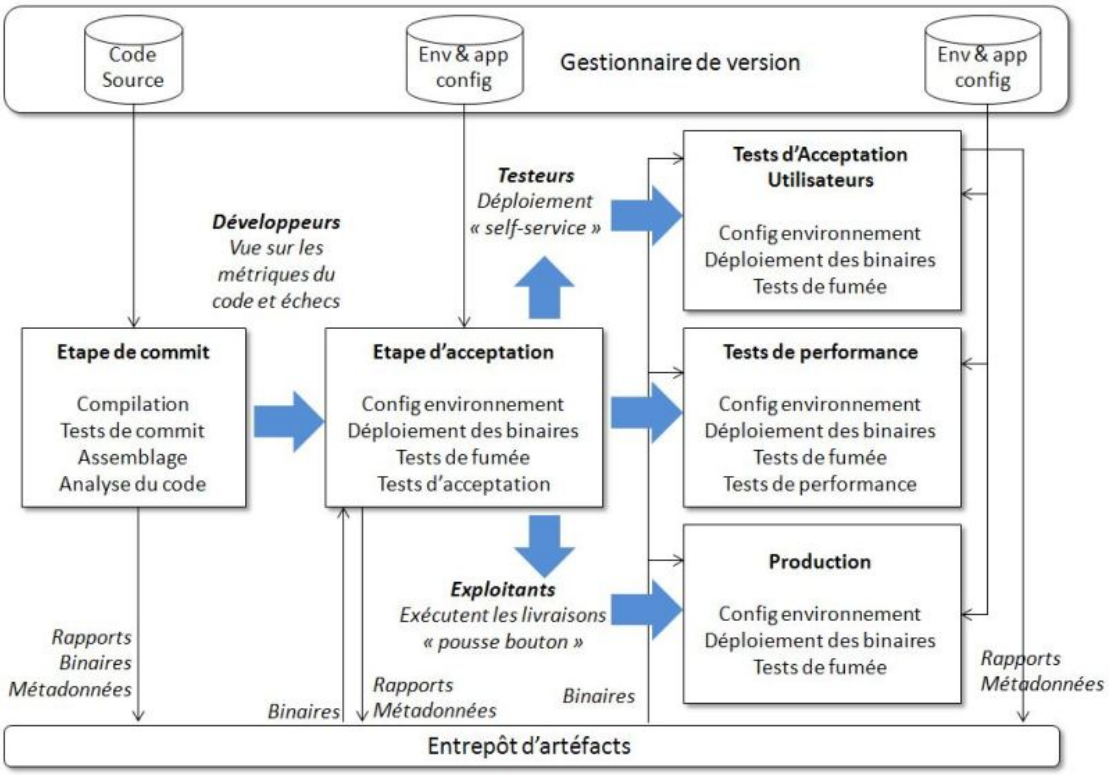

Figure 4.10 – Schéma du pipeline de déploiement

Source: Livre « Continuous Delivery » de Jez Humble et David Farley.

Étant donné que le processus de déploiement est automatisé, le logiciel peut être exécuté et testé régulièrement. En effet pour chaque changement (dans le code source ou la configuration) il peut y avoir un déploiement sur un environnement de test. En conséquence, les équipes de test peuvent obtenir une rétroaction rapide sur le code et le processus de déploiement. L'idée de transférer régulièrement les informations du processus de déploiement vers l'équipe de développement a donné naissance au mouvement **DevOps**. Comme nous l'avons présenté plus haut, l'idée derrière le mouvement **DevOps** est d'améliorer la communication entre les développeurs, les testeurs et les sysadmins. La rétroaction continue fournie par le pipeline de déploiement permet de rassembler les développeurs, qui sont responsables de l'analyse des besoins, conception et développement avec les équipes d'exploitation, qui supervisent le déploiement, l'exécution et la maintenance.

Le déploiement continu, est construit sur un ensemble de bonnes pratiques autour de la notion centrale du pipeline de déploiement. Le pipeline de déploiement est rendu possible par un ensemble d'outils et des pratiques. Parmi lesquelles, la gestion du code source, le build automatisé, l'intégration continue, la gestion d'artefacts et le déploiement automatisé.

En conclusion, on peut citer parmi les avantages du **déploiement continu** :

- **La réduction du risque lié au déploiement :** Puisqu'on déploie des changements mineurs le risque diminue.
- **La mesure réelle de l'avancement du projet :** On peut vérifier réellement si les tâches exécutées par les développeurs sont terminées et cela sur la base de leurs mises en production.
- **Le retour des utilisateurs :** Grâce aux retours fournis par les utilisateurs les développeurs connaissent réellement ce que les utilisateurs attendent du logiciel final.

5 Proposition

5.1 Introduction

La première partie de ce document présente le contexte de ce travail, les problématiques et l'état de l'art sur les méthodes de développement logiciel en général. Dans le chapitre état de l'art nous avons présenté le déploiement continu et les étapes préalables à sa mise en place. Ce chapitre est consacré à l'approche globale adoptée pour mettre en place le déploiement continu au sein du CRD Nicolas Bourbaki.

Le **CRD Nicolas Bourbaki** développe plusieurs projets informatiques dans différentes branches. Le projet phare de l'entreprise reste le logiciel pour surfaces tactiles **SPHER**. Ce projet à des enjeux stratégiques forts et conséquents en taille, ce qui a motivé le responsable du projet d'attendre l'implémentation du déploiement continu sur des projets de taille plus petite avant d'appliquer les pratiques du déploiement continu au projet **SPHER**.

Pour commencer l'implémentation du déploiement continu, j'ai choisi, avec l'accord de mon responsable, de commencer l'implémentation du déploiement continu sur un projet moins conséquent et de taille moyenne. Ce projet consiste en un site web dynamique qui permet de gérer les utilisateurs du **SPHER**.

5.2 Méthodologie de développement

En fait, tous les projets lancés par le **CRD Nicolas Bourbaki** sont gérés selon les principes des méthodes agiles et du développement ***lean***. Cela va nous faciliter le passage des projets en cours où le déploiement se fait manuellement vers le déploiement continu.

Dans notre démarche agile, l'équipe en charge du projet est constituée principalement d'un développeur, d'un testeur, d'un responsable qualité (QA) et du propriétaire du produit (PO) qui dirige le développement. Le développeur joue le rôle du testeur en plus du sien et le testeur joue le rôle de l'opérationnel en plus du sien.

La figure 5.1 montre le schéma des acteurs du projet.

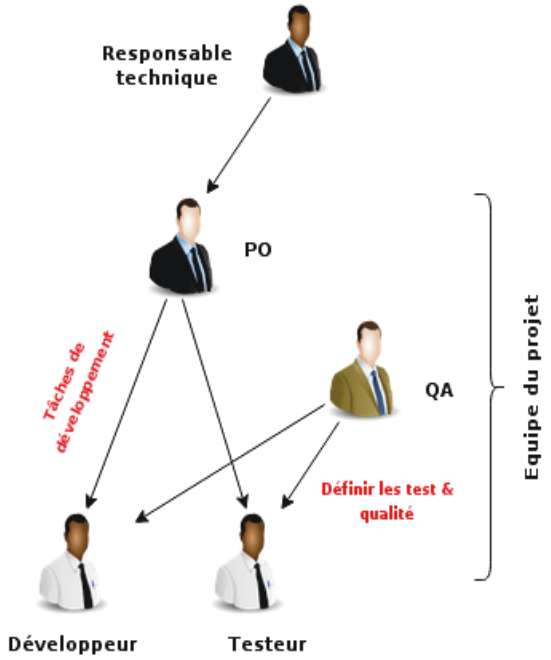

Figure 5.1 – Acteurs de l'équipe du projet

Les besoins des clients sont remontés par un représentant, qui est le responsable technique pour notre projet. En effet, c'est lui qui décide de la direction que doit prendre le projet et qui définit les besoins des utilisateurs.

Lorsqu'on veut développer une nouvelle fonctionnalité, l'équipe de développement se réunit avec le QA et le PO afin d'établir une **Epic**[1]. Au cours de cette réunion, la valeur ajoutée de cette nouvelle fonctionnalité est examinée minutieusement. Ensuite, le PO définit et détermine les priorités des tâches à réaliser. Il dispatche ensuite les tâches pour les membres de l'équipe. Chaque développeur crée une branche locale par fonctionnalité depuis le gestionnaire de version afin de commencer l'implémentation de la nouvelle fonctionnalité. Une fois que le développeur a exécuté sa tâche, il peut alors lancer le processus du build, de test et de revue de code. Enfin, il doit notifier (*pull request*) le QA pour qu'il fusionne ses modifications.

La figure 5.2 illustre les étapes du développement d'une nouvelle fonctionnalité.

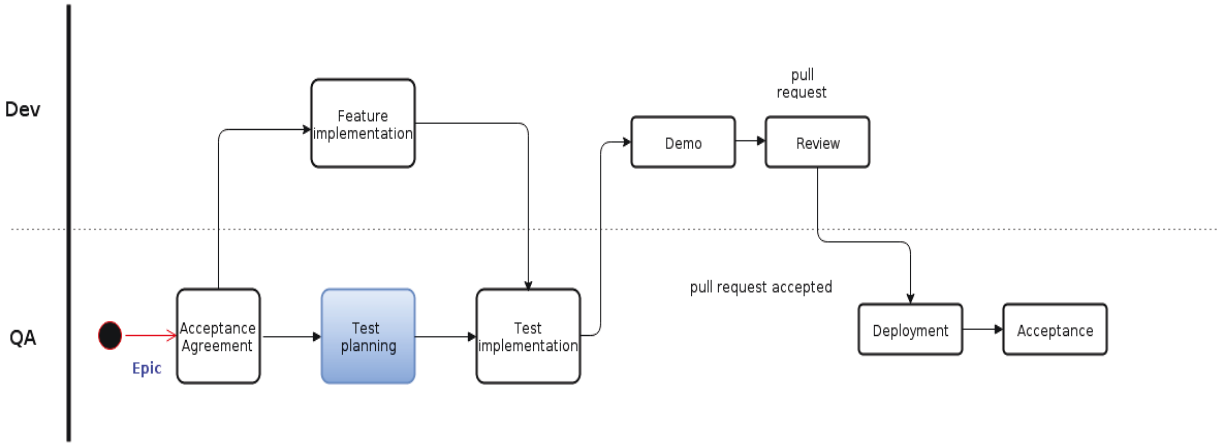

Figure 5.2 – Développement d'une nouvelle fonctionnalité

[1] un ensemble de user story à compléter afin que l'utilisateur soit en mesure d'utiliser une fonctionnalité

En parallèle du développement de cette nouvelle fonctionnalité, on planifie les tests. C'est ici qu'intervient le QA afin de définir les scénarios des tests qui vont être exécutés. Les tests doivent au moins couvrir tous les scénarios démontrés lors de la planification de cette fonctionnalité.

Une fois que le build est terminé avec succès, le testeur déploie l'application dans des environnements de test et vérifie le bon fonctionnement de l'application suite aux changements apportés par l'implémentation de la nouvelle fonctionnalité.

Une fois les tests réussis, la nouvelle branche est fusionnée avec la branche principale sur le gestionnaire des versions. Ensuite, l'application est déployée sur un environnement semblable à celui de la production pour subir des tests fonctionnels avant d'être déployé en production.

Ma responsabilité au sein de cette organisation était d'établir une approche adaptée afin de faciliter l'implémentation du déploiement continu. Pour rappel, le but d'utilisation du déploiement continu est d'automatiser le processus de build, test et de déploiement et d'avoir des retours d'expériences rapides. Afin de réaliser cette implémentation, j'ai établi un plan qui consiste à procéder par **incrémentation**. Concrètement nous allons commencer avec l'implémentation de l'**intégration continue**, ensuite nous implémenterons **la livraison continue** et finir avec **le déploiement continu**. En effet, l'ordre de ces actions a de l'importance car pour faire du déploiement continu il faut d'abord faire de la livraison continue et pour la livraison continue nécessite l'intégration continue. Pour arriver au déploiement continu, nous allons mettre en place **un pipeline de déploiement.**

5.3 Pipeline de déploiement

Le pipeline de déploiement représente l'automatisation du processus de fabrication du logiciel depuis le contrôle de version jusqu'à arriver aux utilisateurs. Chaque modification du code source doit passer par un processus complexe. Ce processus consiste à construire le logiciel (*build*), exécuter plusieurs étapes de tests et déployer le logiciel en final. Ceci, nécessite une collaboration entre de nombreuses personnes et peut-être plusieurs équipes.

Le pipeline de déploiement conceptualise ce processus par cinq éléments principaux : l'intégration continue, le gestionnaire de code, le serveur des artefacts, l'automatisation de déploiement et les outils de test.

Le cœur du pipeline de déploiement est l'intégration continue. Pour cette raison, nous commencerons avec la mise en place de cette dernière.

5.3.1 Architecture de l'intégration continue

Martin Fowler définit l'intégration continue par « *une pratique de développement logiciel ou les membres d'une équipe intègrent leur travail fréquemment, habituellement chacun au moins une fois par jour entraine plusieurs par jour. Chaque intégration est validée par un* ***build automatique*** *(ce qui inclut les tests) pour les erreurs d'intégration aussi vite que possible. Cette approche permet de réduire significativement les problèmes d'intégration et permet à une équipe de développer des logiciels de qualité plus rapidement* ».[2]

Cette définition nous décrit le processus de l'IC par deux choses. La première est que l'IC permet le développement par incrémentation et la deuxième consiste dans l'automatisation des tâches de build et de test.

[2]Source : http://www.martinfowler.com/articles/continuousIntegration.html

Pour faire de l'IC, nous avons besoin de trois choses essentielles:

1. **Un logiciel de gestion de version (VCS [3]) :** Dans un environnement où chaque développeur garde une copie du code pour travailler sur son poste et ensuite fusionner son travail avec les autres. Le VCS nous servira pour garder l'historique de toutes les modifications effectuées et les auteurs de ses modifications. Cela nous permettra d'avoir un suivi rigoureux et aussi d'identifier rapidement les sources d'éventuels problèmes. Le VCS ne doit pas servir qu'à sauvegarder le code source de l'application, mais il doit aussi servir à sauvegarder les tests, les scripts de base de données, les fichiers de configurations, les scripts de build et les scripts de déploiement.

2. **Un processus de build automatisé :** Nous devons avoir un processus qui permet le build complet plus le lancement des tests de manière automatique.

3. **Le consentement de l'équipe de développement :** L'équipe de développement doit respecter les règles imposées par l'IC sinon l'utilisation de l'IC n'aura aucun effet sur la qualité du logiciel final.

Nous avons présenter à la figure 4.6 le fonctionnement de l'IC en règle générale. On peut en déduire un fonctionnement semblable, que nous allons mettre en place pour nos projets. Ce fonctionnement on peut le résumer par les étapes suivantes :

1. Le développeur publie ses modifications de code plus les tests correspondant sur le serveur VCS (commit) après avoir lancé un build en local sur sa machine et corriger d'éventuels bugs.

2. Le serveur d'intégration qui scrute en continu les changements sur le VCS, réalise un build d'intégration en utilisant les scripts ou outils à sa disposition.

[3] Version Control System

3. Après chaque build, un courriel de notification est envoyé à tous les membres de l'équipe afin de les informer sur le statut du dernier build. Les développeurs peuvent aussi suivre l'état du build en direct en se connectant sur le serveur d'intégration.

Voici quelques pratiques que nous allons instaurer avant de commencer à faire l'IC. En effet, l'IC ne représente pas la solution magique pour notre processus de build. Elle peut même devenir problématique si on commence à l'utiliser en plein milieu du projet.

1. **Gestion de l'environnement de développement :** Il est important que l'environnement du développement soit bien organisé afin que le développeur se sente prêt à développer des applications de meilleure qualité. La première chose à mettre en place est le poste de travail, le développeur doit être capable de lancer un build local, exécuter les tests et déployer l'application dans sa propre machine. Lors de l'exécution de l'application le développeur doit suivre les mêmes processus automatisés utilisés par l'intégration continue.

2. **Publier le code régulièrement :** C'est la pratique la plus importante pour le bon fonctionnement de l'IC. La règle est de publier du code au moins deux fois par jour. Cette mesure fait que nos modifications restent limitées et risquent moins de casser un build. Elle nous offre la possibilité de revenir en arrière en cas de problèmes en utilisant la dernière bonne version. Elle nous permet aussi d'éviter les modifications qui altèrent beaucoup de fichiers, qui sont souvent la source des conflits avec le travail des autres personnes.

3. **Créer une série de tests automatisés :** Il est essentiel d'avoir une série de tests automatisés pour s'assurer que notre application fonctionne réellement. En effet, le build signifie seulement que notre application peut être compilée et assemblée. Nous nous intéressons à la combinaison de trois types de test automatisé: les tests unitaires, les tests d'intégrations et les tests d'acceptation.

Les tests unitaires sont écrits pour vérifier le bon fonctionnement d'une partie ou d'un morceau de l'application (par exemple une fonction). Il n'est pas nécessaire de lancer l'application pour les exécuter, car elle n'utilise pas de connexions vers une base de données et n'accède pas au système de fichier. En général, le temps d'exécution des tests unitaires ne dépasse pas les 10 minutes.

Les tests d'intégration sont écrits pour vérifier le comportement des composants de l'application. Contrairement aux tests unitaires, ils peuvent lancer toute l'application car ils accèdent à la base de données, au système de fichier et au réseau. Les tests d'intégration sont gourmands en ressources et prennent plus de temps pour s'exécuter.

Les tests d'acceptation vérifient si l'application satisfait aux critères d'acceptation décidées par nos utilisateurs. Ses critères concernent à la fois les fonctionnalités de l'application, la disponibilité et la sécurité. Les tests d'acceptation sont exécutées dans un environnement semblable à celui de la production et nécessite le lancement de toute l'application.

4. **Maîtriser le temps d'exécution du build et du test :** Il est important que le temps d'exécution du build et des tests ne soit pas très long. Cela pour éviter d'engendrer des problèmes dont :

   (a) Les développeurs vont arrêter de faire un build complet avec l'exécution des tests avant qu'ils publient.

   (b) Le processus de l'IC prendra plus de temps alors que des publications de code attendent leurs tours. Ce qui implique une accumulation de publications de telle sorte qu'on déterminera plus qui sera responsable d'un éventuel échec du build.

   (c) Les développeurs vont moins publier leurs modifications, car ils vont s'asseoir et attendre que le build se termine.

5.3.2 Architecture du pipeline de déploiement

Nous avons défini le pipeline de déploiement par l'automatisation du processus de développement depuis le contrôle de version jusqu'au déploiement de la nouvelle version dans un environnement de production. Chaque modification doit passer par un processus de build, de test et de déploiement. La figure 5.3 montre la sequence complète de ce processus.

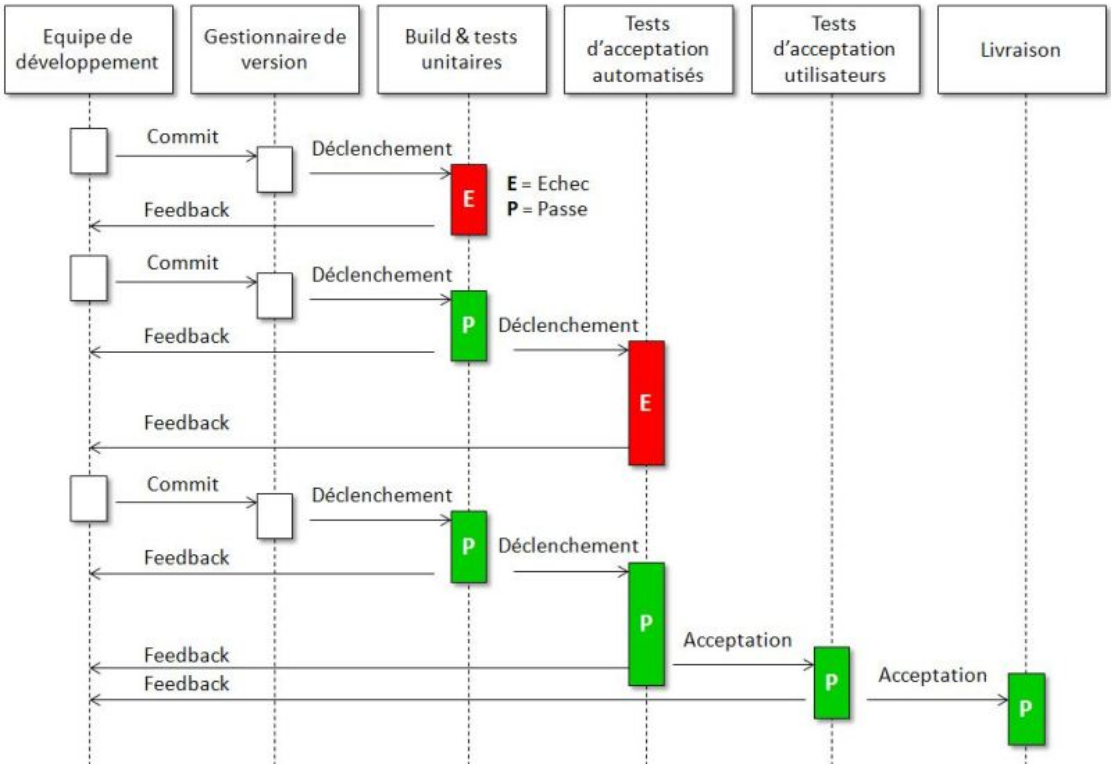

Figure 5.3 – Mouvement des changements au sein du pipeline de déploiement

Source: http://www.agiliste.fr/livraison-continue/

Si des modifications sont publiées successivement avant que le processus de l'IC a eu le temps de finir cela peut poser problème. En effet, une fois que l'IC est terminée elle prendra la dernière version du code et recomencera son processus , mais en cas d'échec l'identification du commit coupable sera difficile. Dans ce cas une intervention manuelle est nécessaire pour demander à l'IC de traiter les commits un par un pour identifier le commit coupable.

Le processus de pipeline de déploiement se devise en plusieurs étapes comme nous montre la figure 5.4 suivante.

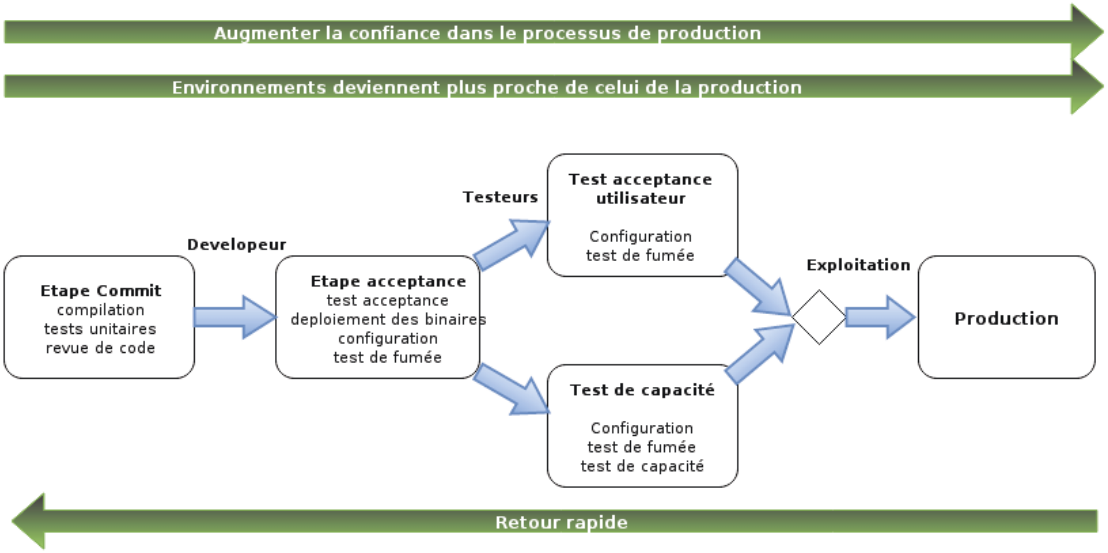

Figure 5.4 – Etapes du pipeline de déploiement

Source: adaptation du livre «Continuous Delivery» Jez Humble.

L'étape commit est la première étape, pendant cette étape l'application est compilée et testée avec des tests de commit (ensemble de tests unitaires qui s'exécutent rapidement) afin de vérifier qu'elle fonctionne au niveau technique (au niveau du code source).

L'étape test d'acceptation vérifie si l'application fonctionne au niveau fonctionnel et non fonctionnel, en vérifiant si l'application répond aux besoins et aux spécifications des utilisateurs et du client.

L'étape test d'acceptation utilisateur contrôle si l'application fonctionne correctement et selon les exigences des utilisateurs. Elle vérifie aussi si les dernières modifications apportent vraiment de la **valeur** aux utilisateurs.

L'étape de test de capacité teste les performances de l'application dans un environnement semblable à celui de la production.

L'étape production livre l'application pour les utilisateurs, que ce soit en tant que logiciel à installer ou en la déployant dans un environnement de production (un environnement identique à l'environnement réel).

5.3.3 Le build

Le **build** est un processus constitué de plusieurs phases successives qui transforme les artefacts de développement (code source, configurations, etc.) en produits livrables (exécutables binaires, distributions, etc.). Dans notre projet, ce processus exécute les étapes de calcul des dépendances, de compilation, d'exécution des tests et de packaging (type de l'exécutable : jar, zip, exe, etc.).

Pour la cohérence du pipeline de déploiement, il faut que le processus de **build soit automatisé** et capable de produire des produits livrables. Il est important aussi que le processus de build exécute des tests afin de vérifier la qualité des livrables produits et détecter d'éventuels bugs très tôt. Ces tests sont généralement constitués de tests unitaires qui vérifient que le code fonctionne correctement et de tests d'acceptation.

Il existe plusieurs outils de build qui permettent d'automatiser ce processus, nous présenterons les plus connus et utilisés dans le chapitre suivant.

5.3.4 Stratégie de test

Les tests sont une part importante dans le pipeline de déploiement, cette activité doit impliquer toute l'équipe. Les tests doivent être exécutés de façon continue dans toutes les étapes du pipeline de déploiement.

Il y a quatre types de tests dans un pipeline de déploiement comme le montre la figure 5.5.

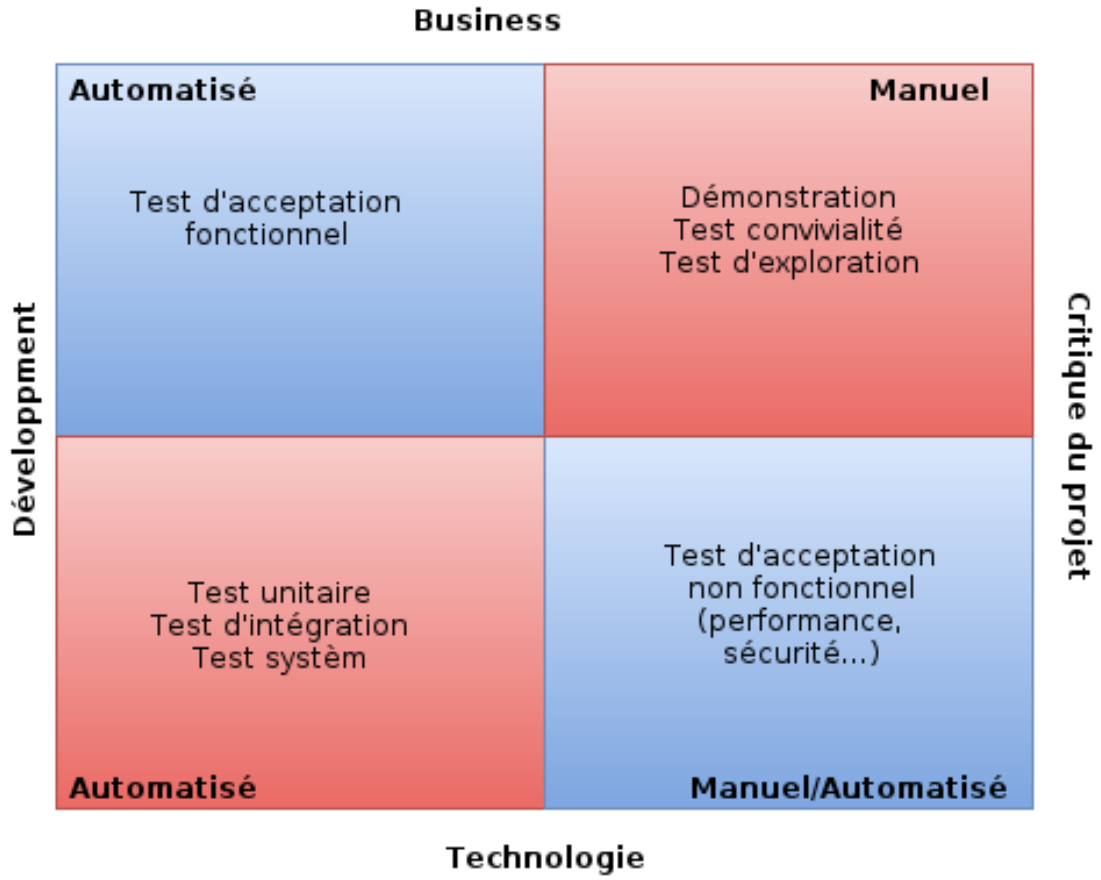

Figure 5.5 – Types de test

Source: adaptation du site http://continuousdelivery.com

Pour notre projet nous allons mettre en place des tests unitaires, des tests d'intégration et des tests d'acceptation fonctionnels et non fonctionnels.

5.3.5 Gestion des artefacts

« Un artefact en UML est n'importe quel produit du travail : code, graphique, schémas de base de données, documents texte, diagrammes, modèles, etc. C'est la spécification d'une information physique utilisée ou produite par un procédé de développement logiciel ou par déploiement et opération d'un système. » [4]

[4] https://fr.wikipedia.org

Les artefacts sont un ensemble de morceaux qui constitue le logiciel livrable. L'exécutable, les fichiers de configuration, les bases de données et les images de machines virtuelles sont des exemples d'artefacts. Chaque artefact est identifié par une version et un nom unique.

Afin de gérer une multitude d'artefacts, on a besoin d'un moyen pour les référencer et les archiver. Ce moyen est un référentiel d'artefacts. Le référentiel d'artefacts archive et organise les artefacts par version. Le référentiel permet aux membres de l'équipe de partager les artefacts entre eux.

Il existe plusieurs référentiels sur le marché, nous présenterons les plus utilisés dans le chapitre suivant.

5.3.6 Déploiement automatisé

Le pipeline de déploiement exige d'avoir un moyen de déploiement automatisé. En effet, tout le long du pipeline de déploiement le processus de déploiement est invoqué dans les différentes étapes afin de déployer l'application dans des environnements de test et de production.

Pour automatiser le déploiement nous avons besoin de deux outils:

Le premier, est un outil qui permet de créer des environnements à la demande (machines virtuelles) et qui permet aussi de les configurer sans intervention manuelle (installation de l'OS, installation d'artefacts, lancement de service...). Ses outils permettent de provisionner les machines virtuelles.

Le deuxième, est un outil qui, à partir de ces environnements provisionnés, peut récupérer les binaires de l'application depuis le référentiel d'artefacts et la configuration de l'application.

Le processus d'approvisionnement et de déploiement doit être automatisé et facile

à exécuter. Quel que ce soit le moyen utilisé pour invoquer ce processus, le résultat produit à la fin doit être toujours le même peu importe l'environnement dans lequel il est utilisé. Cela est une obligation afin de garantir que l'application qui passe dans le pipeline sera la même que celle déployée en production.

5.3.7 Bonnes pratiques

Afin d'obtenir les avantages de l'utilisation du pipeline de déploiement, il y a quelques pratiques que nous allons instaurer dans notre projet.

Lancer le build une seule fois

Si à chaque étape du pipeline de déploiement on «build» l'application alors on risque d'introduire involontairement des erreurs (une différente version de compilateur peut introduire des bugs). Cela est contraire à ce que nous cherchons à faire avec le pipeline de déploiement, car premièrement le processus perd de l'efficacité on fournissant des retours erronés et deuxièmement il faut toujours déployer la même version de l'application dans les différentes étapes du pipeline. C'est pour cette raison qu'il faut utiliser les binaires générés dans l'étape commit dans toutes les autres étapes. Les binaires générés seront stockés dans ce qu'on appelle **dépôt d'artéfacts**.

Déployer de la même façon dans chaque étape

Nous utiliserons les mêmes procédures de déploiement pour les environnements de développement, de test et de production. Cela afin que le processus de déploiement soit testé plusieurs fois et ainsi fiabilisé. Ainsi on aura un moyen d'atténuer le risque de déploiement de nouvelles versions en production.

Lancer des tests de fumée sur le déploiement

Lorsque l'on voudra déployer l'application sur n'importe quel environnement, il est nécessaire de lancer un test de fumée pour vérifier simplement que l'application

fonctionne et que tous les services dont elle dépend sont lancés (base de données, réseau et autre).

Déployer sur un environnement semblable à celui de la production

Nous allons essayer de déployer notre application dans des environnement de développement et de test qui ressemblent à l'environnement de production. Cela nous donnera l'assurance que lorsqu'on livrera notre application ou lorsqu'on déploiera en production on n'aura pas de problème, ce processus d'habitude risqué deviendra facile. Afin d'uniformiser ces environnements nous allons recourir à la virtualisation et des outils pour gérer les configurations.

En cas d'échec sur une section du pipeline, on s'arrête

Si le déploiement dans un environnement échoue, alors toute l'équipe ou au moins une personne doit cesser ce qu'elle est en train de faire et s'occuper de résoudre le problème. Cela rentre parfaitement dans ce que suggère le pipeline de déploiement, c'est-à-dire être réactif face aux erreurs.

6 Implémentation

6.1 Introduction

Le but de ce chapitre est de présenter comment les propositions du chapitre précédent sont mises en place. Plus précisément, il présente les outils et les techniques utilisées pour mettre en place une infrastructure de déploiement continu. En ce qui concerne les outils, nous donnerons une liste pour chaque domaine et nous justifierons notre choix de l'outil utilisé dans notre projet.

Le projet en lui-même consiste en une application web écrite en Java côté serveur et Javascript, HTML côté client. L'application sert comme un portail d'authentification pour les utilisateurs de **SPHER** à travers une **API REST**. De plus, chaque utilisateur peut se connecter et s'authentifier directement sur le site web pour consulter son profil, télécharger une photo, consulter les scores de jeux, sélectionner les applications qui seront chargées lors de sa connexion sur le système **SPHER**.

La partie serveur de cette application est basée sur un outil nommé **Wisdom Framework**[1]. Wisdom Framework est une pile web basée sur Java qui permet d'avoir la modularité et le dynamisme. Contrairement à d'autres pile web, Wisdom permet de construire des applications web en assemblant des modules différents et de façon

[1] http://wisdom-framework.org/

dynamique. Wisdom fournit un mode appelé *watch mode*, qui permet de rendre le développement très efficace et rapide.

Wisdom est basé sur le **NIO**[2] et le framework asynchrone **VertX**[3]. Il est entièrement construite sur OSGi, pour offrir la modularité et **Apache Felix iPOJO**[4] pour faciliter le dynamisme.

La figure 6.1 illustre la modularité sous Wisdom framework.

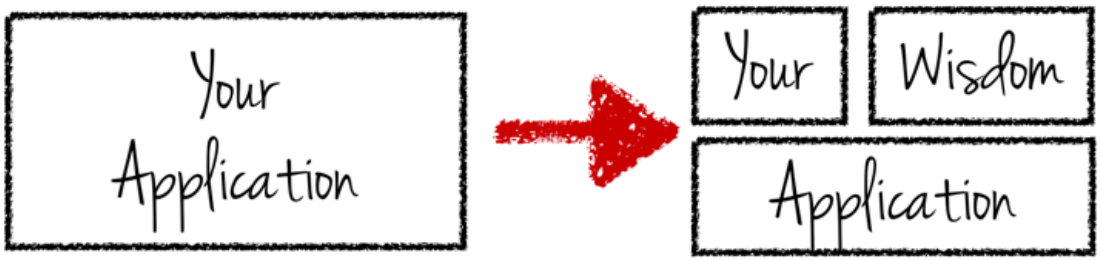

Figure 6.1 – modularité sous Wisdom

Source: http://wisdom-framework.org

6.2 Gestion de projet

Le projet est géré selon les principes des méthodes agiles, pour cette raison nous avons besoin d'un outil de gestion de projet qui respecte ces principes. Cet outil doit permettre d'organiser les tâches, suivre l'avancement du projet et le partage d'information entre les collaborateurs.

Pour notre projet, nous utiliserons **Jira**[5]. Développé par **Atlassian**, ce produit offre en plus des fonctionnalités de gestion de projet, la possibilité de déclarer et le suivre les bugs et la gestion des accidents.

Jira peut s'utiliser de deux façons, soit directement dans le *cloud* ou sur un serveur

[2]NIO : https://en.wikipedia.org/wiki/Non-blocking_I/O_(Java)
[3]VertX : http://vertx.io
[4]iPOJO : http://felix.apache.org/
[5]Jira : https://fr.atlassian.com

local. Dans notre cas Jira est installé localement sur un de nos serveurs.

Les figures 6.2 et 6.3 nous montre les interfaces web de l'outil **Jira**.

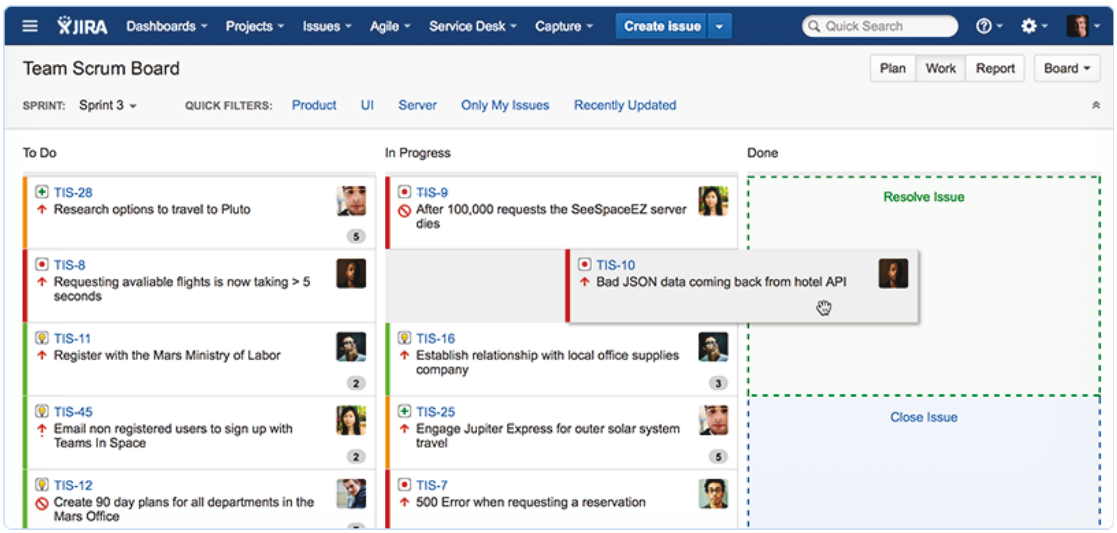

Figure 6.2 – Tableau de bord de Jira - scrum

Source: https://es.atlassian.com/software/jira/agile

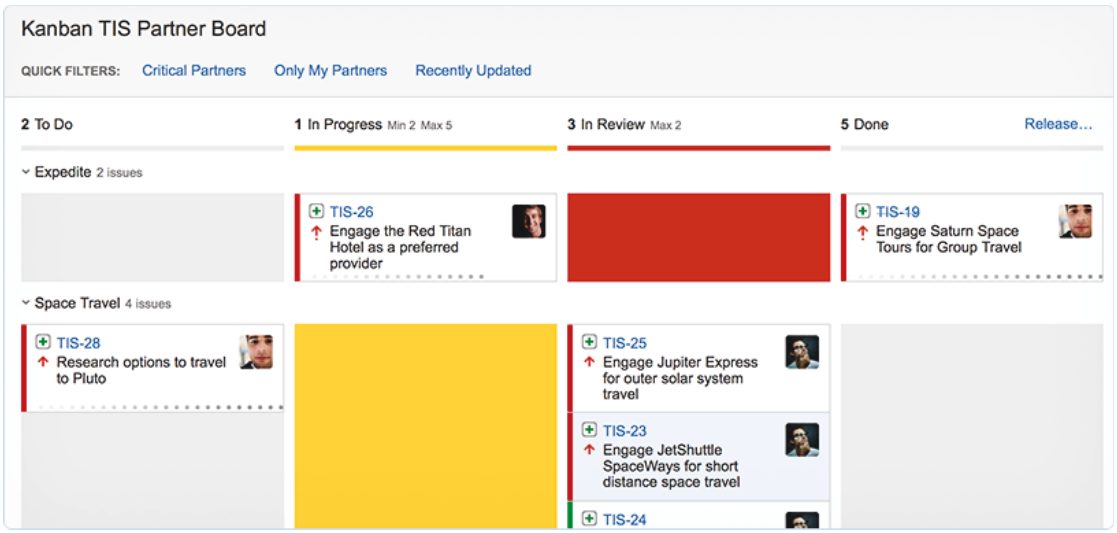

Figure 6.3 – Tableau de bord de Jira - kanban

Source: https://es.atlassian.com/software/jira/agile

6.3 Intégration continue

Il existe beaucoup d'outils sur le marché pour faire de l'intégration continue. Nous fournirons une liste des outils les plus utilisés sur le marché qui peuvent être utilisés pour faire de l'intégration continue. Ensuite, nous sélectionnerons celui qui convient pour notre projet tout en justifiant ce choix dans la mesure du possible.

Gestionnaires de versions

Le gestionnaire contient le code source, les scripts de base de données, les fichiers de configuration, les scripts de test et les scripts de déploiement. Il garde l'historique des modifications sur ses fichiers. Voici une liste de gestionnaires les plus utilisés :

- SVN (subversion)[6].
- Git[7]
- Mercurial[8]
- Bazaar[9]
- Perforce[10]
- BitKeeper[11]
- SourceSafe[12]

Pour notre projet nous avons opter pour l'outil **Git** et **Gitlab**[13] pour l'interface de gestion. Git est un logiciel distribué créé par Linus Torvalds, qui est à l'origine du système d'exploitation **Linux**. Notre choix pour cet outil est motivé par les arguments suivants :

[6]SVN: https://subversion.apache.org/
[7]Git: https://git-scm.com/
[8]Mercurial: https://www.mercurial-scm.org/
[9]Bazaar: http://bazaar.canonical.com/en/
[10]Perforce: http://www.perforce.com/
[11]BitKeeper: http://www.bitkeeper.com/
[12]SourceSafe: https://msdn.microsoft.com
[13]https://about.gitlab.com/

- Outil puissant et très rapide.
- Gestion plus facile des branches.
- Communauté d'utilisateurs très importante.
- Logiciel OpenSource et Multiplateforme (disponible pour plusieurs OS).
- Multiple interface de gestion (Github, Bitbucket, Ungit, Gitlab).
- Facile à prendre en main par les développeurs même les novices.

Malgré le fait que **Git** soit un logiciel distribué, pour notre projet nous allons adopter une architecture centralisée. Nous allons mettre on place un dépôt central qui sera le point d'entrée de toutes les modifications.

La figure 6.4 nous montre le flux de travail sous Git.

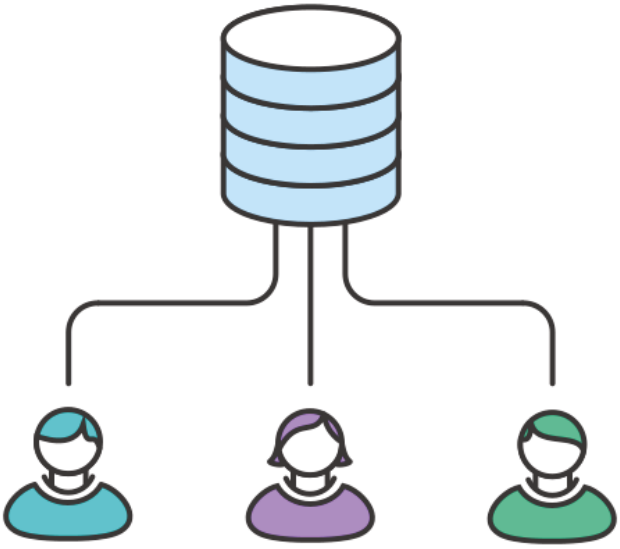

Figure 6.4 – Flux de travail centralisé

Source: https://www.atlassian.com/git/tutorials/comparing-workflows/

Pour organiser le flux de travail sous **Git**, nous allons recourir aux branches comme le montre la figure 6.5.

Figure 6.5 – Modèle de flux de travail sous Git

Source: https://www.atlassian.com/pt/git/workflows#workflow-gitflow

D'après la figure 6.5 nous allons créer plusieurs branches :

- La branche ***Master*** est la branche principale, elle ne doit contenir que du code stable qui servira pour la production.
- La branche ***Release*** est créée à partir de ***Develop***, elle est utilisée lorsqu'on veut passer un ensemble de fonctionnalités terminées en production. Dans cette branche nous pourrons mettre à jour les informations de version (fichiers README, numéro de version, ...).
- la branche ***Hotfix*** sert à corriger les bugs urgents, elle est créée à partir de ***Master***.

- la branche ***Develop*** correspond à la dernière version stable du code qui sera prochainement fusionnée avec la branche ***Master*** pour la production.
- la branche ***Feature*** est créée à partir de ***Develop***, elle est utilisée lorsqu'on veut développer une nouvelle fonctionnalité. Généralement elle porte le nom de la nouvelle fonctionnalité.

6.3.1 Serveurs IC

Le serveur de l'IC est responsable du déclenchement du processus de l'intégration continue. Voici une liste des serveurs les plus connus et utilisés :

- Travis[14]
- Bamboo[15]
- Jenkins[16]
- TeamCity[17]

Nous avons choisi d'utiliser **Jenkins** pour les raison suivantes :

- Facile à l'utilisation et à l'administration.
- Une visibilité rapide des métriques et des résultats de build grâce à son interface.
- Gestion des utilisateurs.
- La richesse de son système de plugin.
- S'intègre facilement avec les outils de build, revue de code et de déploiement.

[14]Travis : https://travis-ci.org/
[15]Bamboo : https://www.atlassian.com/software/bamboo/
[16]Jenkins : http://jenkins-ci.org/
[17]TeamCity : https://www.jetbrains.com/teamcity/

- Utilisé par de grandes sociétés du web tel que Google, Github, Facebook et d'autres.

Les figures 6.6 et 6.7 nous montre l'interface web de gestion du serveur **Jenkins**.

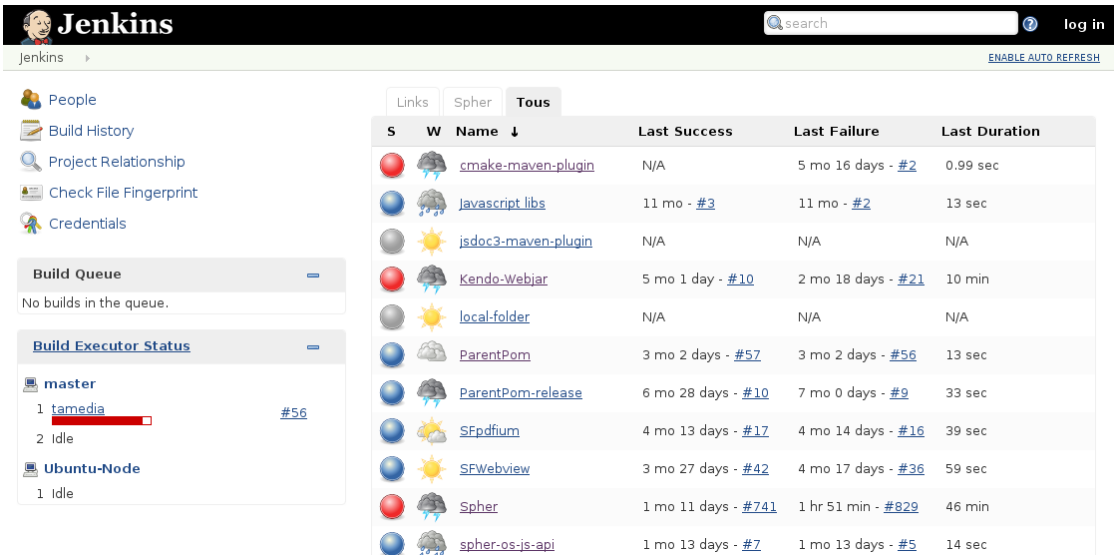

Figure 6.6 – Interface web montrant les tâches Jenkins

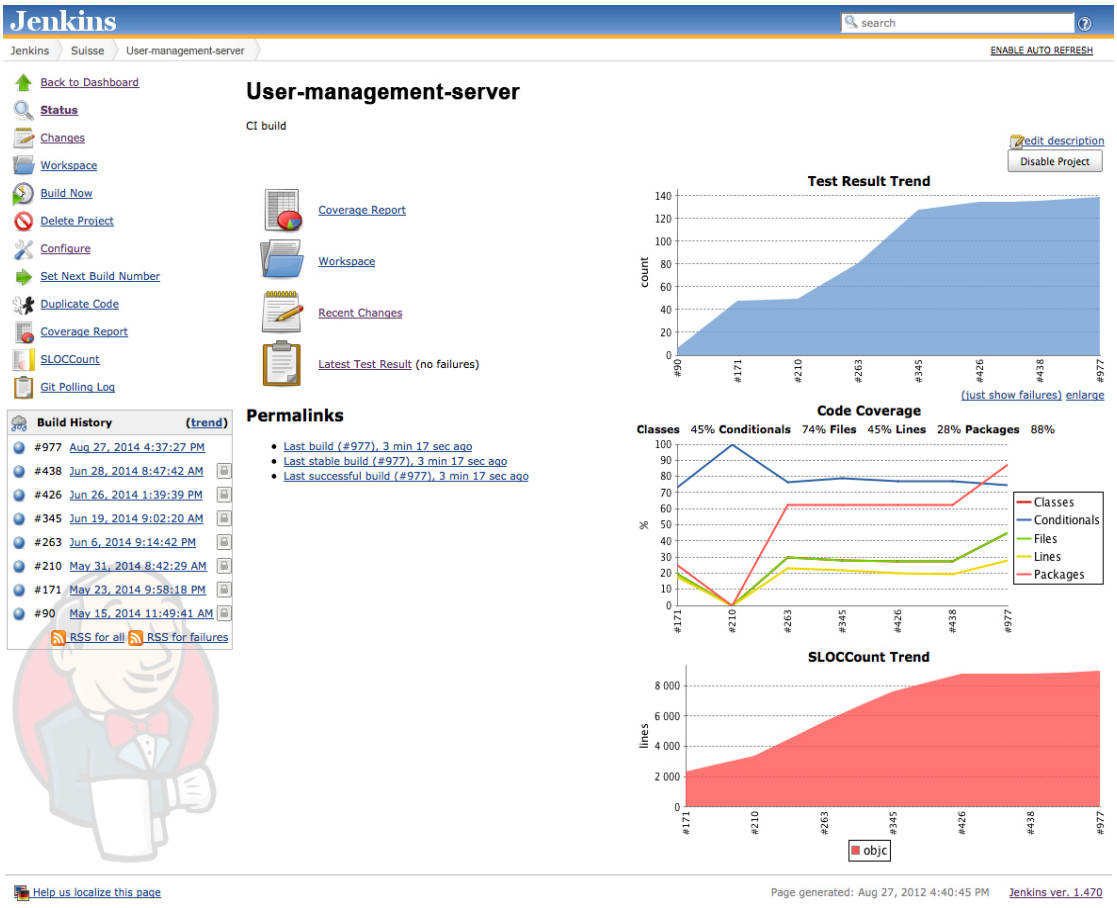

Figure 6.7 – Interface web Jenkins montrant des métriques

6.3.2 Build automatisé et de gestion des dépendances

Les outils disponibles pour cette partie sont :

- Apache Ivy[18].
- Gradle[19]
- Apache Maven[20].

Comme la plupart de nos projets utilise **Java**, alors notre choix s'est porté vers **Apache Maven**. En effet, **Apache Maven** est un outil qui permet d'automatiser le build et de gérer les dépendances en même temps.

Exemple de création d'un projet Wisdom framework avec Apache Maven :

```
mvn org.wisdom-framework:wisdom-maven-plugin:0.9.1:create \
 -DgroupId=YOUR_GROUPID \
 -DartifactId=YOUR_ARTIFACTID \
 -Dversion=1.0-SNAPSHOT
```

Pour lancer un build (exécution des tests comprise) il suffit de lancer :

```
mvn clean install
```

Cette commande compile, lance les tests et génère les artefacts qui seront ensuite déployés directement dans le dépôt des artefacts.

[18]Apache Ivy : http://ant.apache.org/ivy/
[19]Gradle : http://gradle.org/
[20]Apache Maven : https://maven.apache.org/

6.3.3 Outils de tests

Nous utilisons un ensemble d'outils de test qui sont :

- Junit[21].
- Mockito[22].
- Hamcrest[23]
- Selenium[24].
- Wisdom Framework.

Exemple d'un test unitaire :

```
public class TestSimple {


    @Test
    public void additionAvecDeuxNombres() {
        final long lAddition = Operations.additionner(10, 20);
        Assert.assertEquals(30, lAddition);
    }


    @Test
    public void additionAvecCinqNombres() {
        final long lAddition = Operations.additionner(256, 512, 1024,
          2048, 4096);
        Assert.assertEquals(7936, lAddition);
    }
}
```

[21]JUnit: http://junit.org/
[22]Mockito: http://mockito.org/
[23]Hamcrest: http://hamcrest.org/
[24]Selenium: http://www.seleniumhq.org/

Exemple d'un test d'intégration tiré de notre projet:

```
public class ApiUserControllerIT extends WisdomBlackBoxTest {


@Test
public void testAuthentification() throws Exception
{
  ObjectMapper mapper = new ObjectMapper();
  JsonNode rootNode = mapper.createObjectNode();
  ((ObjectNode) rootNode).put("password", "Andrew");
  ((ObjectNode) rootNode).put("email", "Spiderman@marvel.com");
  HttpResponse<String> response =
    post("/api/user/login?token=XXXXXXXXXXXXXXXXXXXXXXXX").
  header("Content-Type", "application/json").body(rootNode).asString();
  assertThat(response.code()).isEqualTo(OK);


}
}
```

Dans cet exemple de test d'intégration nous utilisons **Wisdom Framework**, qui nous offre la possibilité de faire du **Black-box testing**[25]. Dans l'exemple ci-dessus nous testons si l'authentification à travers notre API REST fonctionne comme prévu.

[25]https://en.wikipedia.org/wiki/Black-box_testing

6.3.4 Outils de qualimétrie

Pour notre projet nous utiliserons **SonarQube** pour mesurer la qualité du code. SonarQube supporte plusieurs langages de programmation et peut facilement être intégré avec Jenkins grâce à des plugins. La figure 6.8 montre l'interface web de SonarQube avec un exemple de rapport suite à une analyse du code de notre application.

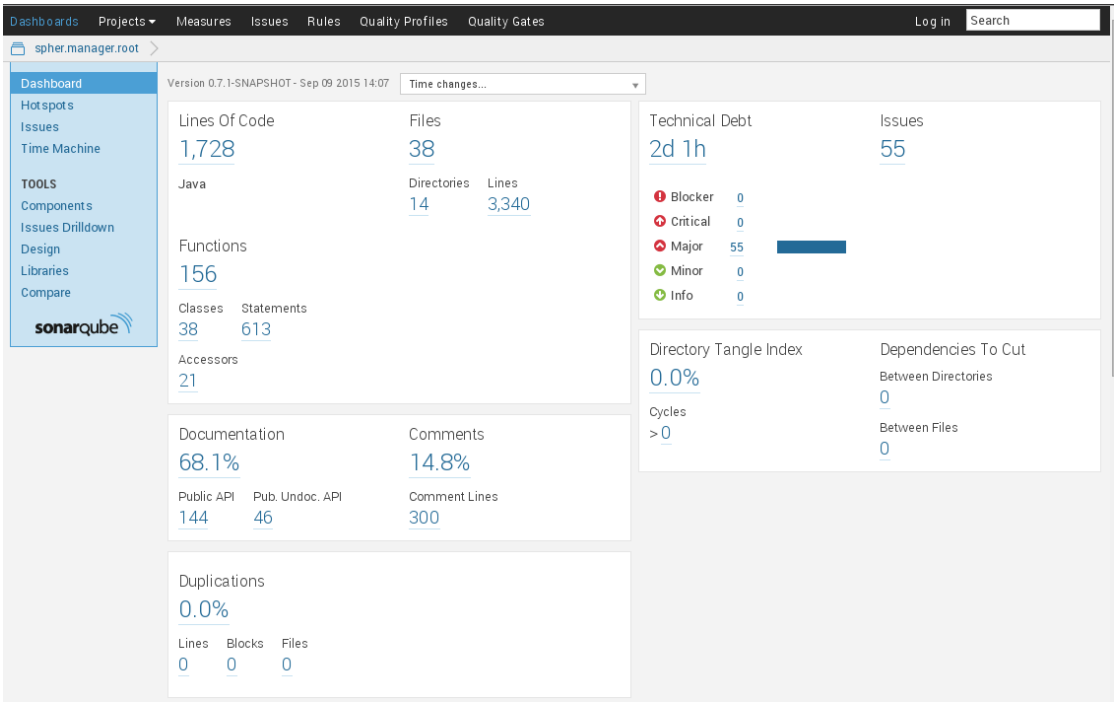

Figure 6.8 – Tableau de bord SonarQube

SonarQube assure les fonctionnalités suivantes :

- La vérification de la duplication du code.
- La vérification du respect des règles établies pour le développement.
- La vérification de la complexité du code.
- La vérification de la documentation.

6.4 Pipeline de déploiement

Dans cette partie nous allons parler des outils utilisés pour mettre en place le pipeline de déploiement. Nous donnerons des exemples de code ou de configuration pour certains outils.

6.4.1 Référentiel d'artéfacts

Le référentiel d'artefacts est l'équivalent à ce que c'est le VCS au code source mais pour les dépendances et les artefacts générés par le build. En effet, il permet de gérer et d'organiser plusieurs versions d'artefacts dont dépend notre application et ainsi éviter que chaque développeur télécharge inutilement de son côté les dépendances nécessaires au build de l'application. Cela implique, une accélération du temps de build et une amélioration de la collaboration entre les développeurs par le partage des artefacts.

On peut citer comme référentiels d'artéfacts sur le marché :

- Archiva[26]
- Artifactory[27]
- Nexus [28]

Pour notre projet nous utiliserons une version professionnelle de **Nexus**. Ce choix était justifié par le fait que Nexus est à la base un outil de gestion des dépôts pour **Apache Maven**. Comme nous utilisons ce dernier comme outil de build et de gestion de dépendances le choix de **Nexus** s'imposait.

[26]Archiva: https://archiva.apache.org
[27]Artifactory: https://www.jfrog.com/artifactory/
[28]Nexus: http://www.sonatype.com/

En plus, Nexus offre la possibilité d'avoir un contrôle strict des versions d'artéfacts utilisées dans le projet, comme par exemple empêcher la mise à jour d'une dépendance pour éviter de casser le build.

Nexus est maintenu par les mêmes personnes qui développent **Apache Maven** ce qui garantit une stabilité et une documentation abondante. La figure 6.9 nous montre l'interface web de gestion de Nexus.

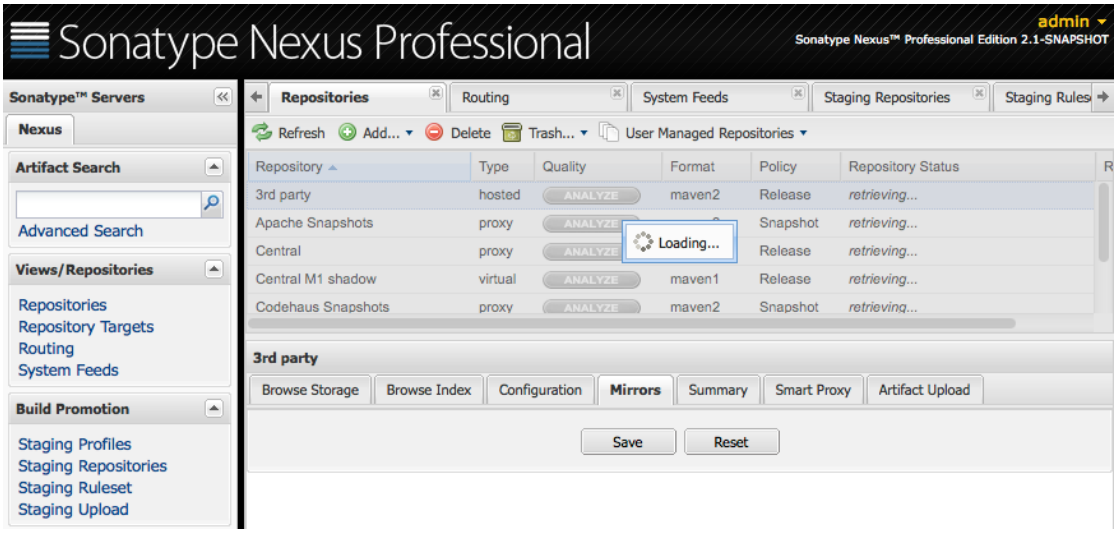

Figure 6.9 – Interface web de Nexus

On peut configurer Apache Maven pour lui ordonner de résoudre les dépendances à partir de notre référentiel Nexus de la manière suivante :

```
<repositories>
    <repository>
        <snapshots>
            <enabled>false</enabled>
        </snapshots>
        <id>digitale-release</id>
        <name>digitale-release</name>
        <url>
    http://binaries.nicolas-bourbaki.com/nexus/content/repositories/releases
        </url>
```

```
        </repository>
        <repository>
            <snapshots/>
            <id>digitale-snapshot</id>
            <name>digitale-snapshot</name>
            <url>
       http://binaries.nicolas-bourbaki.com/nexus/content/repositories/snapshots
            </url>
        </repository>
    </repositories>
```

6.4.2 Gestion de l'infrastructure

Comme nous l'avons expliqué dans le chapitre précédent, il y a deux étapes à faire avant de déployer l'application:

- Création et gestion de l'environnement dans lequel l'application sera exécutée.
- Installation et configuration de l'application.

Les environnements comprennent les serveurs, les systèmes d'exploitation et les middlewares nécessaires au bon fonctionnement de l'application. La gestion de ces différents environnements doivent être automatisés et les environnements doivent être identiques pour assurer que la livraison en production ne sera pas un échec.

Pour faire face à cette problématique, des outils de gestion de configuration ont commencé à voir le jour. Ces outils permettent d'automatiser la gestion des environnements à partir d'un fichier script, dans ce fichier on décrit le système d'exploitation à installer, les services à démarrer, configuration, etc. Ce même fichier est utilisé dans les autres environnements.

Parmi ces outils on trouve :

- Ansible[29]
- Chef[30]
- Puppet[31]

Pour notre projet nous utiliserons **Ansible**. En effet, Ansible offre d'excellentes performances et a l'avantage d'être plus facile à utiliser. Ansible utilise des fichier YAML pour la création de configuration des environnement. Pour faire fonctionner Ansible il suffit de l'installer sur une machine.

Dans notre projet nous allons combiner Ansible avec Packer[32], qui est un outil qui permet de créer de machines virtuelles de manière automatisée. Grâce à ces deux outils on est capable de créer une machine virtuelle et de la configurer par un simple script qui sera exécuté automatiquement.

Voici un exemple de fichier **Packer** qui permet de créer une VM et de la configurer :

```
{
  "builders": [
    {
      "type": "xenserver-iso",
      "remote_host": "your-server.example.com",
      "remote_username": "root",
      "remote_password": "password",

      "boot_command": [
```

[29] Ansible : http://www.ansible.com/
[30] Chef : http://www.getchef.com/
[31] Puppet : http://puppetlabs.com/
[32] Packer:https://www.packer.io/

```
      "<tab> text ks=http://{{ .HTTPIP }}:{{ .HTTPPort
        }}/centos6-ks.cfg<enter><wait>"
    ],
    "boot_wait": "10s",
    "disk_size": 40960,
    "http_directory": "http",
    "iso_checksum": "4ed6c56d365bd3ab12cd88b8a480f4a62e7c66d2",
    "iso_checksum_type": "sha1",
    "iso_url": "{{user
      `mirror`}}/6.6/isos/x86_64/CentOS-6.6-x86_64-minimal.iso",
    "output_directory": "packer-centos-6.6-x86_64-xenserver",
    "shutdown_command": "/sbin/halt",
    "ssh_username": "root",
    "ssh_password": "vmpassword",
    "ssh_wait_timeout": "10000s",
    "vm_name": "packer-centos-6.6-x86_64",
    "vm_description": "Build time: {{isotime}}"
  }
 ],

 "variables": {
   "mirror": "http://www.mirrorservice.org/sites/mirror.centos.org"
 }

 "provisioners": {
  "type": "ansible-local",
  "playbook_file": "local.yml"
 }
}
```

Dans ce script nous créons une machine virtuelle compatible avec **XenServer** avec une configuration de base, ensuite **Ansible** entre en action pour configurer notre VM à partir du fichier **local.yml**.

Voici un exemple d'un fichier de configuration **Ansible** :

```
---
- name: Download WordPress
  get_url: url=http://wordpress.org/wordpress-{{ wp_version }}.tar.gz
    dest=/srv/wordpress.tar.gz

- name: Extract archive
  command: chdir=/srv/ /bin/tar xvf wordpress-{{ wp_version }}.tar.gz
    creates=/srv/wordpress

- name: Add group "wordpress"
  group: name=wordpress

- name: Add user "wordpress"
  user: name=wordpress group=wordpress home=/srv/wordpress/

- name: Fetch random salts for WordPress config
  local_action: command curl
    https://api.wordpress.org/secret-key/1.1/salt/
  register: "wp_salt"
  sudo: no

- name: Create WordPress database
  mysql_db: name={{ wp_db_name }} state=present

- name: Create WordPress database user
  mysql_user: name={{ wp_db_user }} password={{ wp_db_password }} priv={{
    wp_db_name }}.*:ALL host='localhost' state=present

- name: Start php-fpm Service
  service: name=php-fpm state=started enabled=yes
```

6.4.3 Gestion des données

L'application évolue et croît dans le temps ainsi que ses données. Dès lors, il devient nécessaire de gérer l'évolution des données utilisées par l'application de façon complètement automatisée. En effet, il est pratiquement difficile d'utiliser par exemple un seul script SQL pour la création de la base de données et ensuite d'y ajouter progressivement les nouvelles modifications. Cela devient encore plus difficile dans un projet agile où la base de données subit des évolutions constantes par plusieurs personnes qui ajoutent des modifications chacune de leur côté sans partager ses modifications avec le reste de l'équipe.

Pour résoudre ce problème, des outils de gestion de version pour base de données sont apparus. Parmi ces outils il y a :

- Flyway[33]
- Liquibase[34]

Pour notre projet nous utiliserons **Flyway**, car il s'intègre facilement avec l'outil de build Apache Maven grâce à un plugin, mais qu'on peut aussi utiliser facilement en ligne de commande.

Flyway s'installe avec Apache Maven de la manière suivante :

```
<plugin>
    <groupId>org.flywaydb</groupId>
    <artifactId>flyway-maven-plugin</artifactId>
    <version>3.2.1</version>
</plugin>
```

[33] Flyway: http://flywaydb.org
[34] Liquibase: http://www.liquibase.org/

Ensuite, il suffit d'écrire un script pour chaque migration. Ce script doit se trouver absolument dans un répertoire qui s'appelle « db.migration » . **Flyway** utilise une convention de nommage pour différencier les scripts. En effet, le script de migration doit commencer par la lettre « V », suivie du numéro de la version, séparée par des «_» ou des points et la description du script pour plus de lisibilité.

La figure 6.10 nous montre la structure d'un projet qu'utilise le plugin **flyway**.

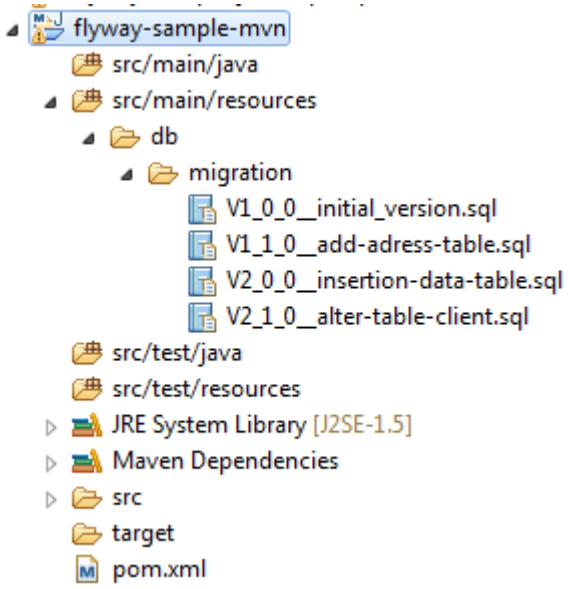

Figure 6.10 – structure projet maven avec flyway

Pour exécuter une migration il suffit de lancer cette commande :

```
mvn flyway:migrate

[INFO] [flyway:migrate {execution: default-cli}]
[INFO] Current schema version: 0
[INFO] Migrating to version 1.0.0
[INFO] Migrating to version 1.1.0
[INFO] Migrating to version 2.0.0
[INFO] Migrating to version 2.1.0
[INFO] Successfully applied 4 migrations (execution time 00:00.133s).
[INFO]
  ------------------------------------------------------------------------
```

```
[INFO] BUILD SUCCESSFUL
```

A titre de démonstration, voilà le schéma 6.11 de l'architecture finale :

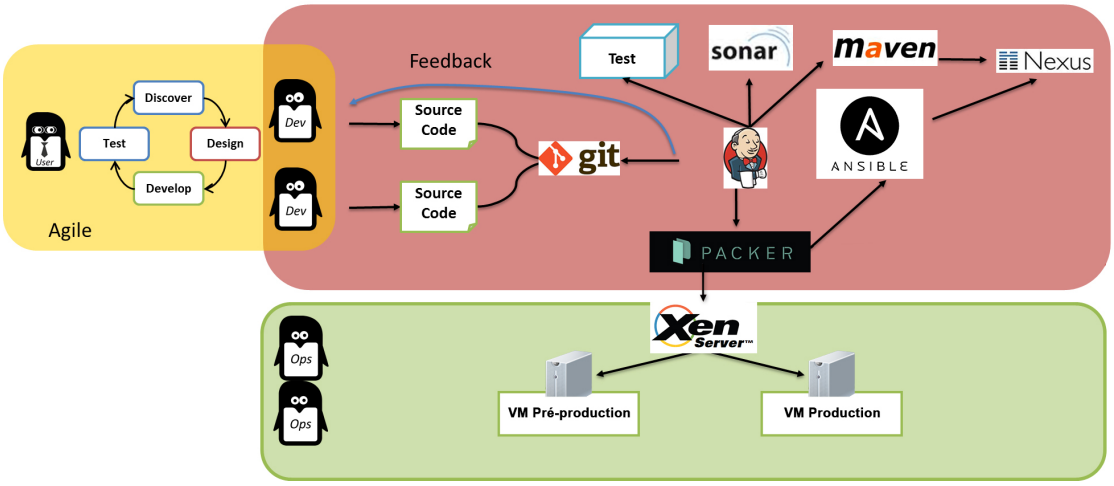

Figure 6.11 – structure projet maven avec flyway

Source: Adaptation tirée du site : http://blog.d2-si.fr/2015/03/03/automatisation-dev-deploiement/

Avec cette approche, on a des images de machines complètes qui contiennent notre application avec la configuration nécessaire à son exécution. Ses VM sont déployées sur un serveur Xen pour être utilisées. Cette technique nous permet d'avoir exactement les mêmes environnements hôtes pour notre application.

7 Conclusion

Le développement d'applications modernes représente un défi d'ingénierie logicielle. Il impose de fournir aux développeurs les bons outils et les processus cohérents pour assurer le bon développement et la livraison rapide du logiciel.

La mise en place d'une infrastructure de déploiement continu n'est pas chose facile. Ou cours de ce travail j'ai rencontré plusieurs problèmes : humain, organisationnel, technique et budgétaire.

Dans l'ordre de l'énumération, ci-dessus, sur le plan humain il est difficile d'imposer une nouvelle méthodologie qui s'écarte des sentiers battus, cette condition implique un déploiement d'effort important de la part des développeurs en terme d'adaptabilité et de flexibilité. Par ailleurs, concernant l'organisation des équipes, cette dernière à fait l'objet d'une refonte majeure; passer d'une équipe à compétence homogène à une équipe constituée par une synergie de compétences est délicat à mettre en place. Il faut savoir associer les compétences de manière à avoir une équipe pluridisciplinaire capable de livrer un produit de qualité dans des délais toujours aussi courts.

De surcroît, il a fallu investir dans la formation pour former les développeurs et les opérationnels à l'utilisation de nouveaux outils introduit par la nouvelle infrastructure de déploiement continu. Cette étape permet aux développeurs de s'aguerrir à cette nouvelle méthode de travail.

Enfin, le coût important de l'installation (serveurs, licences logiciels payants, etc.) est à prévoir dans une démarche d'implémentation du déploiement continu. Au vus des ambitions de l'entreprise qui cherche une expansion en rapide, nous avons aidé à estimer quel serait le budget nécessaire pour une telle infrastructure. Malgré tous, l'entreprise fait des économies en déployant rapidement ses produits.

Ce travail a aidé l'entreprise à avoir une vision sur la meilleure façon de mettre en place le déploiement continu pour améliorer la livraison de ses produits. Avant que je commence ce projet, il y avait uniquement l'intégration continue en place sans la partie test. Suite à ce travail nous avons tout d'abord continuer à mettre en place le processus de l'intégration continue vu que c'est le cœur d'une infrastructure de déploiement continu.

D'un point de vue personnel, l'écriture de ce mémoire m'a permis de connaître un nouveau concept qui m'était encore inconnu et qui me permettra à l'avenir de mieux appréhender les problèmes liés au déploiement continu. Les différents outils m'ont permis également de percevoir au mieux les différences entre le développement et l'opérationnel, ceci afin comprendre comment les exploiter au mieux.

Références

[1] BERNAILLE, L. automatisation du cycle de vie des applications : du développement au déploiement, mars 2015. [En ligne]. Disponible sur : http://blog.d2-si.fr/2015/03/03/automatisation-dev-deploiement/. (consulté le 13/10/2015).

[2] DUVALL, P. M. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison Wesley, California, 2007.

[3] ET JOANNE MOLESKY, J. H. Why enterprises must adopt devops to enable continuous delivery. *Cutter IT Journal 24* (Aout 2011), 6–12.

[4] ET MICHAEL A CUSUMANO, M. P. Lean software development: A tutorial. *Software, IEEE 29* (2012), 26–32.

[5] ET R TURNER, B. B. *Balancing Agility and Discipline: A Guide for the Perplexed*. Pearson Education, 2003.

[6] ETMAJER, M. Automated deployments with ansible. [En ligne]. Disponible sur : http://fr.slideshare.net/MartinEtmajer/automated-deployments-with-ansible?related=2. (consulté le 13/10/2015).

[7] FALGUIERE, C. Deploiement continu breizh camp, 2011. [En ligne]. Disponible sur : http://fr.slideshare.net/claude.falguiere/deploiement -continu-breizh-camp?related=7. (consulté le 13/10/2015).

[8] FOWLER, M. Continuous integration, 2006. [En ligne]. Disponible sur : http://martinfowler.com/articles/ continuousIntegration.html. (consulté le 21/09/2015).

[9] FOWLER, M. Continuous delivery, 2013. [En ligne]. Disponible sur : http://martinfowler.com/bliki/ContinuousDelivery.html. (consulté le 23/09/2015).

[10] HASSEN, R. Continuous delivery un peu de théorie, 2014. [En ligne]. Disponible sur : http://blog.valtech.fr/2014/05/14/continuous-delivery-un-peu-de-theorie/. (consulté le 19/09/2015).

[11] JEZ HUMBLE, D. F. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, California, 2010.

[12] MARTIN, R. C. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, California, 2008.

[13] MERLE, N. *Architecture pour les systemes de déploiement logiciel à grande échelle*. PhD thesis, Université Joseph Fourier, Grenoble, 2005.

[14] PHAM, H. *Software reliability*. Springer, 2000.

[15] RFELDEN. Devops : mission [im]possible ?, 2015. [En ligne]. Disponible sur : http://fr.slideshare.net/rfelden/devops-mission-impossible?related=10. (consulté le 15/10/2015).

Glossaire

Automatisé On parle d'automatisation lorsque le processus est complètement automatisé et ne requiert pas d'intervention d'un utilisateur. En anglais, on parle d'un processus d'un processus « headless » ou « hands-off ». cf. [Duvall : Continuous Integration, Improving Software Quality and Reducing Risk 2007, p. 27]

Build Il s'agit d'un ensemble d'activités dont l'objectif est de générer, tester, inspecter et déployer le logiciel. C'est donc bien plus qu'une simple compilation: on met le code- source en commun et on vérifie qu'il réagit bien comme une unité cohésive. cf. [Duvall : Continuous Integration, Improving Software Quality and Reducing Risk 2007, p. 27]

Commit Le terme anglais commit désigne une validation12 de transaction qui fait référence à la commande synonyme Commit présente dans la plupart des systèmes de gestion de base de données et des logiciels de gestion de versions. cf. [wikipedia]

Ecran tactile Un écran tactile est un périphérique informatique qui combine les fonctionnalités d'affichage d'un écran (moniteur) et celles d'un dispositif de pointage, ou comme la souris ou le pavé tactile mais aussi avec un stylet optique. cf. [wikipedia]

Environnement de développement Il s'agit de l'environnement dans lequel le pro-

gramme est écrit. Le plus souvent on parle d'un IDE (Integrated Development Environment) comme Eclipse ou Visual Studio, mais les librairies, les fichiers de configuration, les serveurs ou les « builds scripts » en font aussi partie. cf. [Duvall : Continuous Integration, Improving Software Quality and Reducing Risk 2007, p. 28]

Hyperviseur En informatique, un hyperviseur est une plate-forme de virtualisation qui permet à plusieurs systèmes d'exploitation de travailler sur une même machine physique en même temps. cf. [wikipedia]

Inspection Il s'agit de l'analyse du code-source pour les attributs de la qualité interne. On considérera les aspects automatiques (analyses statiques et du temps d'exécution) comme une inspection de logiciel. cf. [Duvall : Continuous Integration, Improving Software Quality and Reducing Risk 2007, p. 28]

Intégration C'est l'acte de combiner des parties de codes-source séparées, pour déterminer comment elles fonctionnent comme un tout. cf. [Duvall : Continuous Integration, Improving Software Quality and Reducing Risk2007, p. 28]

Java Le langage Java est un langage de programmation informatique orienté objet créé par James Gosling et Patrick Naughton, employés de Sun Microsystems, avec le soutien de Bill Joy (cofondateur de Sun Microsystems en 1982), présenté officiellement le 23 mai 1995 au SunWorld. cf. [wikipedia]

Machine virtuelle En informatique, une machine virtuelle (anglais virtual machine, abr. VM) est une illusion d'un appareil informatique créée par un logiciel d'émulation. Le logiciel d'émulation simule la présence de ressources matérielles et logicielles telles que la mémoire, le processeur, le disque dur, voire le système d'exploitation et les pilotes, permettant d'exécuter des programmes dans les mêmes conditions que celles de la machine simulée. cf. [wikipedia]

OSGI L'OSGi Alliance (précédemment connue en tant qu'Open Services Gateway initiative) est une organisation qui spécifie une plate-forme de services fondée sur le langage Java qui peut être gérée de manière distante. Le cœur de cette spécification est un framework (canevas) qui définit un modèle de gestion de cycle de vie d'une application, un répertoire (registry) de services, un environnement d'exécution et des modules. cf. [wikipedia]

Package Ensemble de marchandises ou de services, proposés groupés à la clientèle sans qu'il soit possible de les dissocier. cf. [Larousse]

Qualité La qualité est une valeur donnée aux différentes spécificités du produit. Ce terme est très subjectif et peut varier en fonction des priorités de chacun et de la pondération qu'on attribue à ces spécificités. Parmi elles on trouve : l'entretien, la sécurité, les performances, l'extensibilité, les tests opérés durant la conception du programme, etc. cf. [Duvall : Continuous Integration, Improving Software Quality and Reducing Risk2007, p. 28]

REST REST (representational state transfer) est un style d'architecture pour les systèmes hypermédia distribués, créé par Roy Fielding en 2000 dans le chapitre 5 de sa thèse de doctorat. Il trouve notamment des applications dans le World Wide Web. cf. [wikipedia]

Risque Le risque est la potentialité qu'une erreur survienne. On observera que la plupart des programmateurs les classent en fonction d'un rapport entre leurs typologies (degrés de « dangerosité ») et d'une estimation de leur probabilité d'apparition. On peut ainsi réduire les risques en suivant un schéma basique : d'analyse du risque, de réduction/prévention et de suivi. cf. [CNRS]

Test de fumée test de fumée, en anglais «Smoke-tests», sont un jeu réduit de tests pour valider un build avant de passer à la validation.

Version Control System Un logiciel de gestion de versions est un logiciel qui permet de stocker un ensemble de fichiers en conservant la chronologie de toutes les modifications qui ont été effectuées dessus. Il permet notamment de retrouver les différentes versions d'un lot de fichiers connexes.

YAML YAML, acronyme récursif de YAML Ain't Markup Language, est un format de représentation de données par sérialisation Unicode. Il reprend des concepts d'autres langages comme XML, ou encore du format de message électronique tel que documenté par RFC 2822. YAML a été proposé par Clark Evans en 20011, et implémenté par ses soins ainsi que par Brian Ingerson et Oren Ben-Kiki. cf. [wikipedia]

Liste des abréviations

API Application Programming Interface

CD Continous Deployment

CI Continous Integration

LOC Line Of Code

NIO Non-blocking Input Output

OSGI Open Services Gateway Initiative

PO Product Owner

QA Quality Analyst

REST REpresentational State Transfer

SDLC Systems Development Life Cycle

TTM Time To Market

UML Unified Modeling Language

VCS Version Control System

VM Virtual Machine

YAML YAML Ain't Markup Language

ZDD Zero Downtime Deployment

Résumé

Durant son cycle de vie, le logiciel passe par plusieurs étapes de développement et évolution. Le déploiement est une phase cruciale parmi d'autres, qui transforme le code source en programme en exécution. Traditionnellement, cette étape est exécutée manuellement ce qui la rend particulièrement critique et source de problèmes pour l'entreprise. Grâce aux méthodes agiles le déploiement a connu des bouleversements majeurs avec l'apparition de nouvelles techniques tels que l'intégration continue, la livraison continue et le déploiement continu.

Ce mémoire traite de la manière dont le déploiement continu a été mis en place, de la conception à l'implémentation tous en décrivant les étapes de ce projet.

Mots clés: Déploiement continu, livraison continu, pipeline de déploiement, DevOps, intégration continue.

During its life cycle, the software goes through several stages of development and evolution. The deployment is a crucial stage among others, which transforms the source code into running program. Traditionally, this step is performed manually which makes it particularly critical and cause many problems for the company. With Agile deployment has undergone major changes with the emergence of new techniques such as continuous integration, continuous delivery and continuous deployment.

This report discusses how the continuous deployment has been established, from design to implementation by describing all the stages of this project.

Keywords: Continuous Deployment, Continous Delivery, Deployment Pipeline, DevOps, Continous Integration.
