

# Les enjeux de l'intégration continue

Cédric TESNIERE & Maxime HORCHOLLE

Lundi 13 Janvier 2014

---

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>L'intégration continue</b>	<b>4</b>
2.1	Principe . . . . .	4
2.2	Les motivations des entreprises . . . . .	4
2.3	Les motivation au niveau projet . . . . .	5
2.4	Qu'est-ce que c'est et pourquoi l'utiliser . . . . .	5
<b>3</b>	<b>Les outils les plus utilisés du marché</b>	<b>5</b>
3.1	Tests . . . . .	5
3.1.1	Les tests unitaires . . . . .	6
3.1.2	Les tests fonctionnels . . . . .	6
3.1.3	Les tests d'intégrations . . . . .	6
3.2	Gestionnaire de versions . . . . .	6
3.3	Détecteur de copier coller . . . . .	7
3.4	Revue de code . . . . .	7
3.5	Analyseur de code . . . . .	7
3.6	Logiciel de suivi de problemes . . . . .	8
3.7	Test de couverture . . . . .	8
3.8	Coding style checker . . . . .	9
3.9	Serveur d'integration continue . . . . .	10
3.10	Test GUI . . . . .	10
3.11	Conclusion . . . . .	10
<b>4</b>	<b>Les méthodes qui supportent l'intégration continue</b>	<b>10</b>
4.1	Methodse agiles . . . . .	11
4.1.1	SCRUMS: . . . . .	11
4.1.2	Une Autres: . . . . .	11
4.1.3	Une Autres: . . . . .	11
4.2	Le développement piloté par les tests (TDD) . . . . .	11
4.2.1	TDD et les Design Patterns . . . . .	12
4.3	Behavior Driven Development (BDD) . . . . .	12

<b>5</b>	<b>Ça va bientôt arriver dans le cloud !</b>	<b>12</b>
5.1	Intégration continue . . . . .	12
5.2	La livraison continue . . . . .	12
<b>6</b>	<b>Conclusion</b>	<b>12</b>
<b>7</b>	<b>Les sources:</b>	<b>12</b>
7.1	CheckStyle . . . . .	12
7.2	Bug Trackers . . . . .	13

## 1 Introduction

On entend de plus en plus parler d'intégration continue dans les médias spécialisés, de plus en plus d'entreprises se spécialisent dans ce domaine, mais le public sait-il réellement de quoi il s'agit. Sans doute, sait-il que cette technique sert à améliorer la qualité du code, mais en sait-il réellement plus.

C'est pour cela que nous avons décidé dans ce document de faire un panorama très exhaustif de ce qu'est l'intégration continue ainsi que de répondre à la question du pourquoi devrions nous choisir cette méthode et comment une équipe de développeurs en arrive à l'utiliser.

Pour répondre à notre sujet ce document sera découpé en trois grandes parties. Une première expliquera en quoi consiste globalement l'intégration continue et ce qu'elle peut apporter à un projet. Dans la seconde nous verrons les types d'outils les plus utilisés et pourquoi les mettre en place. Et enfin, dans notre dernière partie nous aborderont les outils encore peu connus et très peu utilisés par les entreprises qui pourraient bien révolutionner ce qui existe à l'heure actuelle en matière d'intégration continue et pour cela nous irons regarder ce qui se passe du côté du monde de l'open-source.

## 2 L'intégration continue

### 2.1 Principe

### 2.2 Les motivations des entreprises

Il faut en général déterminer avant de commencer si le projet utiliserait l'intégration continue ou non, ce choix induirait des conséquences (positives ou négatives). Donc pourquoi de nos jours les entreprises optent de plus en plus pour l'intégration continue ? La question est simple mais reste complexe dans le cas où certaine entreprise resterait perplexe à cette pratique car ils ne l'ont généralement jamais testé.

D'un point de vue du marketing, l'utilisation de l'intégration continue permet d'avoir des demandes de démonstrations non planifiées. Le projet étant constamment compilé et envoyé sur le serveur de dev, cela permet au client de visualiser le rendu du projet à chacun build.

Budgets

- Démontrer rapidement l'avancement d'un projet
- Projets gérés par tranches, par lots conditionnels : focus sur le fonctionnel important !

Ressources, équipes

- Coordination d'équipes distribuées : le reporting projet ne suffit pas !

- Il faut partager les mêmes éléments d'évaluation de l'état d'avancement d'un projet
- Des changements dans l'organisation : fusion-acquisition, restructuration, ...

Besoins : les besoins varient continuellement en fonction

- Des produits de concurrents éventuels
- Des changements légaux, réglementaires (contraintes d'importation, de confidentialité, etc).

Besoin d'intégrer les évolutions d'un projet en continu

## **2.3 Les motivations au niveau projet**

Nécessité d'améliorer :

- La qualité des livrables
- Réduire la complexité (meilleure maintenabilité)
- Adéquation
- La traçabilité
- des changements
- des déploiements
- La productivité
- Se focaliser sur le métier, pas sur la technique

Principes « agiles »

- Fabriquer souvent
- Tester souvent (tests unitaires)
- Tester les performances souvent
- Intégrer souvent dans le SI

## **2.4 Qu'est-ce que c'est et pourquoi l'utiliser**

# **3 Les outils les plus utilisés du marché**

## **3.1 Tests**

On ne sait pas faire de logiciel sans défaut et le coût des corrections de bug peut coûter cher et prendre beaucoup de temps sur certains projets.

### 3.1.1 Les tests unitaires

### 3.1.2 Les tests fonctionnels

### 3.1.3 Les tests d'intégrations

## 3.2 Gestionnaire de versions

Un gestionnaire de versions sert à stocker, versionner et partager son code. On utilise surtout des gestionnaires de versions quand le besoin de partager le code source entre les différents membres d'une équipe se fait ressentir. En effet le plus souvent stocker son code sur un gestionnaire de source alors que l'on est le seul utilisateur à peu d'intérêt, à part de celui de ne jamais perdre ses sources. Les plus gros avantages se font ressentir sur les projets où il y a une équipe en effet voici une liste des gros avantages d'avoir un gestionnaire de sources quand on est en équipe :

- Tous les membres de l'équipe ont toujours une version du code à jour. Donc un gain de temps énorme lors de la mises à jour.
- Garder un historique de toutes les modifications et de qui les à effectuer. Ce qui permet un retour rapide en cas d'introduction de bug et de plus avoir peur de modifier le code source.
- Géré finement vos versions. Par exemple figer une version stable qui ne bougera plus et qui contient un nombre limité de bugs (une version "stable").

Voilà les avantages de tous les gestionnaires de sources il faut savoir qu'il existe deux types de gestionnaire de sources les centralisés qui existe depuis des années et les décentralisés qui inondent le marché depuis quelques années car il possède toutes les qualités des outils centralisés mais ajoute encore d'autre fonctionnalité voici les avantages des gestionnaires décentralisés face au centralisés.

- Possibilité de faire des commits atomique sur sa machine et les pousser ensemble en une seule fois sur le serveur maître pour résoudre un bug ou créer une fonctionnalité. Il est aussi partie de travailler hors ligne ce qui est totalement impossible en centralisé
- Un aspect communautaire renforcé :
  - Une jolie interface graphique où chacun peut voir et annotez-le commit de l'autre.
  - Chacun peut proposer des idées postées des bugs, etc...
- Plus de sécurité :

- Si les droits d'accès sont bien fait le développeur doivent passer par la case "pull request" c'est-à-dire qu'il faut que son commit soit accepté par l'administrateur pour être mis en ligne sur le serveur maître.

Au vu des fonctionnalités proposé par les serveurs de sources décentralisé il est conseillé d'opter pour ceux-ci, en particulier pour Git.

### 3.3 Détecteur de copier coller

Rien qu'au titre on voit tout de suite l'intérêt de cet outil en effet il permet de vérifier que le code n'est pas remplis de copier-coller plus ou moins justifié. En effet la multiplication des copié-coller rend le code de moins en moins maintenable car si un morceau de code copier-coller un peu partout dans le programme est modifié à un endroit car il entraîne des bugs il faudra mettre aussi à jour tous les autres morceaux où le code a été copié-coller. Ce qui rend donc le code très vite inmaintenable.

L'outil lui va aider les équipes à détecter les morceaux de code recopie ce qui permettra donc aux développeurs d'identifier les parties du code qu'il faut factoriser au plus vite pour rendre le code plus maintenable et éviter les modifications en chaîne de code, les bugs et régressions.

Exemple de détecteur de copier/coller:

Mess Detector [http://fr.wikipedia.org/wiki/PMD\\_\(logiciel\)](http://fr.wikipedia.org/wiki/PMD_(logiciel))

### 3.4 Revue de code

La revue de code consiste à faire relire le code par une personne qui n'a pas écrit ce code en presence ou non de la personne qui à codé. On voit tout de suite l'effet positif principal de cette technique: si une personne extérieur vérifie le code de l'autre il peut évaluer la qualité du code qui a été écrit et faire des retour à la personne qui l'a écrite. Il peut être aussi interessant de faire participer la personne qui a codé à la relecture du code afin d'expliquer ce qu'il a fait à l'autre, cette méthode permet en plus de vérifier la qualité du code, de partager l'expérience et les connaissances de chacun pour éviter qu'une seule personne soit au courant d'un morceau de code spécial (work around, etc...)

### 3.5 Analyseur de code

Une analyse du code permet de générer des données d'analyse sur la mise en conformité par rapport aux standards du marché car la qualité d'une application est directement liée à la qualité du code et à la productivité.

De nombreux outils permettent de contrôler quelques aspects de cette qualité du code, principalement sur l'exécution de tests unitaires, l'analyse de la couverture du code par ces tests, la

vérification du respect des règles de codage, etc. Ainsi ces outils permettent d'avoir une confiance accrue en son application ! Un contrôle fréquent de la qualité du code va donc pousser l'équipe de développement à adopter et à respecter certains standards de développement. Un code qui respecte ces standards est un code plus sûr car cela permet de trouver immédiatement les erreurs.

Lorsque l'on sait que le coût de la correction d'une erreur augmente considérablement avec le temps, un outil de surveillance permet la détection précoce de ces éventuels problèmes et l'on comprend très vite l'importance de la détection rapide des erreurs ...

### 3.6 Logiciel de suivi de problemes

Aussi communément appelé Bug trackers se sont tous les outils qui servent à suivre l'évolution du projet. Dedans est consigné les évolutions et les bugs de l'application. Ce qui est très utile pour garder un historique de tout ce qui a été réalisé jusqu'alors. C'est aussi un outil indispensable pour distribuer les tâches.

Avant la création de ce genre d'outils la gestion de projets était bien plus difficile et l'équipe était beaucoup moins réactive puisque l'information était souvent soit non mises à jour où pire complètement indisponible. Avec un bug tracker efficace c'est un jeu d'enfant de rentrer des tâches, des bugs, de voir l'état d'avancement de ses collègues. Les bugs tracker moderne proposent aussi de nombreuses autres fonctionnalités comme des forums, des diagrammes de Gantt, etc.

La majorité des bugs tracker peuvent/doivent être installés sur un de ses serveurs dans ce genre de solutions on trouve Redmine (gratuit) ou Jira (payant pour les projets non open-source) qui est de très bons bugs trackers pour les gros projets . Mais récemment un nouveau type de bugs trackers est apparu : les bugs tracker sur le cloud, ils sont souvent fait pour être utilisé avec les méthodes agiles et sont surtout là pour les projets de petite envergure (équipe restreinte, taille du projet limité). Ces solutions sont souvent gratuite de base et propose un plan payant pour les utilisateurs professionnels. Dans cette catégorie on trouve des sites comme Trello ou Assana.

### 3.7 Test de couverture

Après tout ce que nous avons pu voir j'espère que vous avez compris que le test est une chose essentielle en intégration continue. Mais une question se pose comment estime-t-on qu'un logiciel est suffisamment testé ? Une question bien difficile à répondre sauf dans un cas : celui des tests unitaires, en effet il existe des solutions dans tous les langages qui calculent ce que l'on appelle le taux de couverture. Il s'agit du pourcentage de ligne qui est testé via les tests unitaires sur le nombre total de ligne du projet, donc plus on se rapproche de 100% plus les tests unitaires couvrent tous les cas possibles. Quelques solutions vont même plus loin dans le test de couverture en proposant par exemple de consulter le code du projet et de voir visuellement les lignes tester sont celles qui ne le sont pas encore. Cette fonctionnalité est aussi la bienvenue



pour voir les parties des fonctions qui n'ont pas été testé, par exemple il est rare de voir les développeurs testés tous les cas d'erreur possible d'une fonction.

En général lorsqu'un logiciel est testé à moins de 50% on doit commencer à se poser des questions. Et recadrer au plus vite le projet de faire remonter le taux de couverture afin d'être sûr que le projet n'a pas subi de régressions. On estime un projet bien testé unitairement lorsqu'il a un taux de couverture supérieur à 70%. Et qu'il est très bien testé en 90% et 100%, le taux de 100% reste assez utopique pour les gros projets.

Pour finir sur le taux de couverture il faut justes signaler qu'il ne teste en aucun cas la qualité des tests unitaires donc si les tests ne sont pas pertinents (pas d'assertions, etc...) le logiciel ne le détectera pas...

### 3.8 Coding style checker

Cet outil fait partie des outils importants pour faciliter la relecture et la maintenabilité du code. En effet Il permet de tester si les normes de codage fixé sont bien respecté par les programmeurs. Ces règles doivent être respectées car si elles ne sont pas suivies à la lettre on se retrouve en général avec un code non uniforme ce qui ne simplifie pas sa lecture.

Voici quelques règles assez communes :

- Utiliser des quatre espaces au lieu des tabulations (ce qui permet d'avoir le même code sur n'importe quelle machine).
- Une ligne ne doit pas faire plus d'un certain nombre de caractères. Le nombre de caractères est en général situé entre 80 et 120. Si une ligne fait plus en général il vaut mieux la découper sur plusieurs lignes ce qui facilitera sa lecture.
- La façon d'écrire du code. Par exemple doit on sauter une ligne après la fin d'une méthode, doit on mettre un espace entre une `if` et la parenthèse.

Il existe des règles de codage que l'on considère comme des références dans chaque langage qui sont en général décidées avec des membres importants de la communauté. Pour illustrer prenons la norme PSR-2 de PHP<sup>1</sup>. Il existe aussi les conventions pour le langage Java<sup>2</sup>.

Maintenant pour en revenir à l'outil qui gère ces conventions, il a tout son intérêt car si un développeur ne les respecte pas les erreurs seront tout de suite mises en évidence et il pourra les corriger rapidement ce qui évitera que le code ne devienne illisible dans un futur proche, car un code homogène est un réel plus pour sa compréhension et sa maintenabilité.

Quelques exemples d'outils, les outils en général sont nommés CheckStyle dans chaque langage:

Checkstyle pour Java: <https://github.com/checkstyle/checkstyle>

PHPCheckStyle pour PHP: <https://code.google.com/p/phpcheckstyle/>

<sup>1</sup><https://github.com/php-fig/fig-standards/blob/master/accepted/fr/PSR-2-coding-style-guide.md>

<sup>2</sup><http://www.oracle.com/technetwork/java/javase/documentation/codeconventions-139411.html>

### **3.9 Serveur d'intégration continue**

### **3.10 Test GUI**

### **3.11 Conclusion**

Dans cette conclusion, je souhaiterais donner un ordre d'importance à tous les outils car en effet, ils n'ont pas tous le même intérêt ni la même importance. Je vais donc cité ici les outils vraiment indispensables pour avoir une bonne base pour son intégration continue.

Dans un premier temps il faut absolument versionner son code si cela n'est pas fait même pas la peine de penser à l'intégration continue, l'utilisation d'un gestionnaire de sources décentralisé peut être un plus. Utiliser les outils de tests unitaires, avant d'écrire toutes sortes de tests plus ou moins utile (fonctionnel, GUI, etc...) il faut absolument écrire des tests unitaires qui valideront à chaque modifications que le code est toujours aussi stable qu'avant. Enfin il vous faudra bien évidemment le serveur d'intégration continue qui sera chargé de faire le lien entre les différents outils cité précédemment.

Si les outils que j'ai cités ci-dessus sont les plus importants cela ne signifient pas que les autres ne sont pas utiles à l'intégration continue et qu'ils ne doivent pas être installés. Cela signifie qu'ils ne sont pas nécessaires pour les petites équipes ou les petits projets, ou qu'ils peuvent être facilement intégrés par la suite dans le workflow de son intégration continue si on veut augmenter la qualité du code produit. En effet les outils que j'ai cités comme nécessaire le suis surtout car, sans eux il serait impossible de parler d'intégration continue. Il peut être aussi intéressant de garder uniquement très peu de technologie pour éviter de dérouter les développeurs en les assommant de nouvelles contraintes, puis à mesure que le projet grossit et que les développeurs s'habituent à incorporer de nouveaux outils, en bref rendre cela ludique et leur montrer que l'intégration continue n'est pas juste quelques choses qui sanctionnent, mais aussi un support d'amélioration pour les développeurs. Qu'y a-t-il de plus gratifiant que de voir la courbe de qualité de son projet monté !

## **4 Les méthodes qui supportent l'intégration continue**

En général dès que l'ont veut mettre en place tout les outils permettant d'écrire du code de qualité. Les chefs de projet ce rendent compte très vite lors de leur premier projet utilisant l'intégration continue que les résultats escompté ne sont pas là. Pourquoi? Car bien souvent les équipes qui n'utilise pas l'intégration continue n'ont jamais connu d'autre méthode de développement que l'antique cycle en V. Mais pour qu'un projet utilise tout le potentiel de l'intégration continue il faudra changer les habitudes de l'équipe et utiliser des méthodes de développement bien plus moderne. C'est pour cela que dans cette partie nous verrons les méthodes de développement qui se marie le mieux avec le concept d'intégration continue. Il existe une catégorie de méthode de développement qui correspond exactement à la philosophie

de l'intégration continue, il s'agit des méthodes Agiles, c'est pour cela que toutes les méthodes présentées ci-dessous seront des méthodes agiles. Dans un premier temps nous parlerons des méthodes Agiles les plus communes et utilisées comme SCRUM et Extreme Programming qui sont des méthodes qui sont compatibles. Puis dans les deux parties suivantes deux autres méthodes qui utilisent pleinement l'intégration continue car sans elle ces deux méthodes n'auraient aucun sens.

## 4.1 Méthodes agiles

Dans cette partie nous allons dans un premier temps expliquer ce qu'est une méthode agile ensuite nous verrons les méthodes agiles les plus utilisées.

Les méthodes agiles se veulent en rupture avec la gestion de projet des débuts (cycle en V, etc...). Le plus gros écueil de ces méthodes était ce que l'on appelle l'effet "tunnel", c'est à dire que le client ne voit plus rien des développements une fois qu'il avait validé les spécifications de son projet. Ce n'est qu'à la fin qu'il découvre ce qui a été fait, deux cas arrivent fréquemment, soit le projet subit de gros retard car tout n'a pas été finalisé à temps dans quel cas le client n'a rien et est complètement dépendant de l'équipe technique. Soit au final les besoins du client ont changé entre la validation des spécifications par le client et la livraison, dans ce cas tout le développement est alors jeté pour être recommencé. C'est donc en se basant sur ce postulat que les méthodes agiles ont été créées, donc le point commun de toutes les méthodes agiles est de faire disparaître l'effet "tunnel". Pour cela on se base sur plusieurs principes:

- Transparence: Le client verra l'avancement au fur et à mesure de l'avancée du projet
- Moins de documentation: Écrire du code plus clair plus lisible mais moins bien documenté car écrire de la documentation n'est en général jamais lu (ou très peu).
- Flexibilité: Le client doit être très impliqué dans le projet il se doit de faire des retours (positifs et/ou négatifs). En contrepartie il peut faire évoluer ses besoins (ajout de fonctionnalité, etc...) en cours de développement.

Maintenant voici quelques méthodes agiles les plus connues et utilisées:

### 4.1.1 SCRUMS:

### 4.1.2 Une Autre:

### 4.1.3 Une Autre:

## 4.2 Le développement piloté par les tests (TDD)

Quand on est en phase de développement, on n'a pas toujours le temps de créer les tests adéquats pour son bon fonctionnement et on oublie souvent de les faire faute de temps. Sur

certain projet on prend des gens qui ne coûtent pas cher et qui ne sont pas développeurs afin de tester le cas fonctionnel des nouvelles fonctionnalités, dans certains projets c'est le chef de projet ou le scrum master qui teste la fonctionnalité rapidement. Mais ceci est une mauvaise pratique car d'ici quelque semaine voir quelque mois, quand une fonctionnalité importante ne fonctionne plus, on se demandera depuis combien de temps elle est comme cela et surtout comment !

En utilisant des tests de régression, le problème serait résolu, mais il faut cependant les écrire et cela représente un budget supplémentaire lors de l'écriture de ces tests, une autre solution existe qui consiste à utiliser la méthode TDD ("Test Driven Development" ou en français "Le développement piloté par les tests") permet d'écrire le test avant le code. Le développeur conçoit un ensemble de tests pour la première "user Story" du cahier des charges, certes ces tests échouent mais c'est le but car on devra faire en sorte qu'il fonctionne en écrivant le minimum de code possible. Généralement un développeur aura tendance à écrire plus de code que le nécessaire cependant avec le processus de TDD, le codeur écrira le minimum de code et si les tests passent alors ce code sont considérés comme solides.

Ainsi le TDD permet de concevoir du code en état de marche à n'importe quel moment du développement du projet en plus d'améliorer la qualité du code et nous assure que le besoin du client est respecté.

#### 4.2.1 TDD et les Design Patterns

### 4.3 Behavior Driven Development (BDD)

## 5 Ça va bientôt arriver dans le cloud !

### 5.1 Intégration continue

### 5.2 La livraison continue

## 6 Conclusion

## 7 Les sources:

### 7.1 CheckStyle

Conventions de codage PHP: <https://github.com/php-fig/fig-standards/blob/master/accepted/fr/PSR-2-coding-style-guide.md>

Conventions de codage Java: <http://www.oracle.com/technetwork/java/javase/documentation/codeconventions-139411.html>

## 7.2 Bug Trackers

Wikipédia: [http://fr.wikipedia.org/wiki/Logiciel\\_de\\_suivi\\_de\\_probl%C3%A8mes](http://fr.wikipedia.org/wiki/Logiciel_de_suivi_de_probl%C3%A8mes)

Redmine: <http://www.redmine.org/>

Jira: <https://www.atlassian.com/fr/software/jira>

Trello: <https://trello.com/>

Asana: <https://asana.com>