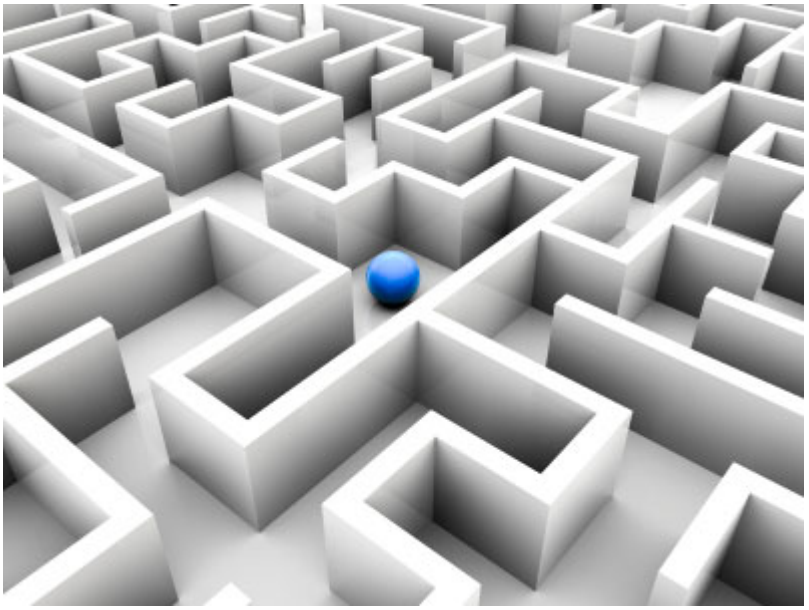
Fabian Piau

INTEGRATION CONTINUE



Juillet 2009

Améliorer la qualité des logiciels et réduire les risques

… A software development practice where team members integrate frequently, leading to multiple integrations per day. Each integration is verified by an automated build and test to detect integration errors as quickly as possible…
Martin Folwer

TABLE DES MATIÈRES

1. INTRODUCTION 3
1.1. Les méthodes agiles 3
1.2. Organisation du rapport 3
1.3. Une histoire de déjà-vu ? 4
1.4. L'intégration continue, la solution ? 6
2. LE GESTIONNAIRE DE VERSIONS, PRE REQUIS INDISPENSABLE 9
3. FONCTIONNEMENT DE L'INTEGRATION CONTINUE 10
3.1. L'intégration continue en un schéma 10
3.2. Le build 11
3.3. L'intégration continue en trois étapes 11
3.4. Le développeur au centre de l'intégration continue 13
3.4.1. L'intégration locale 13
3.4.2. Les 8 bonnes pratiques du développeur 14
3.5. Les différents types de build 17
3.5.1. Le build privé 18
3.5.2. Le build d'intégration de jour 18
3.5.3. Le build d'intégration de nuit 18
3.5.4. Une bonne pratique dans la construction des builds 19
3.6. Le suivi du statut des builds 19
3.6.1. Blogs 19
3.6.2. Autres 20
3.7. Les bonnes pratiques générales 22
4. LES OUTILS 23
4.1. Liste globale (non exhaustive) 23
4.2. Outils de qualimétrie 24
5. L'INTEGRATION CONTINUE PAR L'EXEMPLE 25
5.1. Une journée ordinaire sur le "mode IC" 26
5.2. Bilan de l'exemple 28
6. LES AVANTAGES 29
6.1. Réduction du temps de correction des bogues 29
6.2. Meilleure visibilité du projet 30
6.3. Amélioration du travail collaboratif 30
6.4. Autres avantages 31

7. DIFFICULTES ET INTERROGATIONS 32
7.1. Méthode de mise en place 32
7.2. Détection du besoin d'intégration 34
7.3. L'intégration continue vue comme une perte de temps 34
7.4. La peur de l'intégration continue 35
7.5. Le piège de la compilation continue 38
7.6. Les bonnes questions à se poser 38
8. EVOLUTION DE L'INTEGRATION CONTINUE 40
9. CONCLUSION 45
9.1. Communication et Stratégie 45
9.2. L'intégration continue comme un jeu 45
9.3. Ce qu'il faut retenir ! 46

Intégration continue

AMELIORER LA QUALITE DES LOGICIELS ET REDUIRE LES RISQUES

1. INTRODUCTION

1.1. Les méthodes agiles

Les méthodes Agiles sont des procédures de conception de logiciel qui se veulent plus pragmatiques que les méthodes traditionnelles. En impliquant au maximum le demandeur (client), ces méthodes permettent une grande réactivité à ses demandes, et améliorent sa satisfaction. La notion de méthode agile a été officialisée en 2001 par un document Manifeste Agile (*Agile Manifesto*) signé par 17 personnalités impliquées dans l'évolution du génie logiciel, et généralement auteurs de leur propre méthode.*

Les méthodes de développement agiles deviennent de plus en plus communes et de plus en plus matures. Elles sont appuyées par des bonnes pratiques dans le développement des logiciels qui ont su faire leurs preuves au fil du temps. Il faut savoir que, sans ces bonnes pratiques, les tentatives d'amélioration de la qualité du produit grâce aux méthodes agiles peuvent échouer.

1.2. Organisation du rapport

Ce dossier présente l'intégration continue (IC), l'une des pratiques agiles les plus populaires. Il est à la fois une synthèse et un recueil des bonnes pratiques sur l'IC. Ce document répondra également aux questions qui peuvent être soulevées par une telle pratique. La partie concernant le retour d'expérience sera amenée à évoluer, l'intégration continue est un processus long. Pour l'instant, l'objectif principal de ce document est de vous

* Extrait de la définition "Méthode Agile" de Wikipedia

présenter le concept. Ainsi, vous trouverez une description de cette méthode, les avantages qui en découlent, mais aussi les difficultés qui peuvent survenir et comment les surmonter. Vous y trouverez un exemple parlant sous la forme d'un scénario réaliste, nous parlerons aussi des évolutions possibles de l'intégration continue. Autrement dit, ce dossier contient tout ce qu'il faut savoir avant de se lancer dans la mise en place de l'IC au sein de son entreprise.

Nous nous permettrons l'usage de nombreux anglicismes tels que "commiter" (opération de validation des modifications réalisées dans le répertoire de travail local et de propagation dans le gestionnaire de sources d'où il est issu), "build" (ensemble des étapes nécessaires à la création des livrables d'un projet, compilation, tests, packaging…), "release" (une version finale de test avant sa livraison au client), et bien d'autres…

1.3. Une histoire de déjà-vu ?

Entrons rapidement dans le vif du sujet avec des histoires (pas très drôles !) de projets sans Intégration Continue. Tout d'abord, voici un projet comme on en vit tous les jours :

Le syndrome du "Je comprends pas, ça marche sur mon poste !".
Les symptômes :

- Commit partiels ;
- Fichiers de configuration dépendants du poste de travail.

Résultat : équipe régulièrement bloquée une demi-journée.

Maintenant, énonçons un cas un peu plus difficile :

Considérons un projet en .NET avec 25 personnes.
Sur au moins 3 releases, il a été impossible de faire le build pour l'équipe de test. A chaque fois, on a une perte d'1 journée avec 6 personnes en attente et 2 développeurs pour reprendre les commits un par un.
Sur au moins 2 releases, le build a été construit sans problème particulier. Livraison à l'équipe de test. Après 2 jours de tests par l'équipe (de 6 personnes), découverte d'un bogue bloquant. Livraison en urgence d'un patch et d'un commit.

Release après une semaine de tests, mais le build ne passe plus. 2 jours de reprise du patch par deux développeurs (qui représentent 2 jours où l'équipe de test est au chômage technique).
Finalement, l'ensemble des tests de la semaine est à repasser, car la confiance dans la constitution de la version est trop faible.

Résultat : 42 jour-hommes perdus.

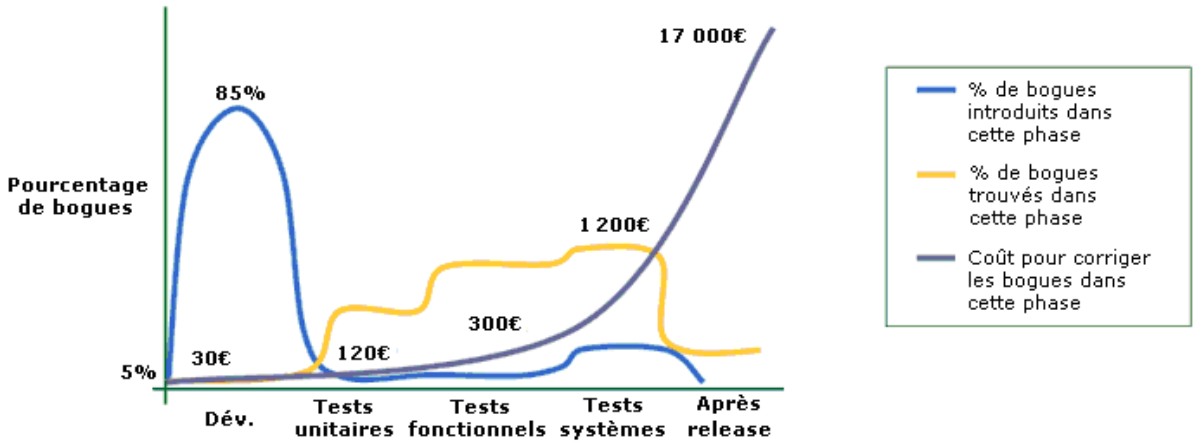
Un dernier exemple, encore pire :

Projet Java, 60 développeurs.
Après 6 mois de développement, l'équipe décide de créer une release pour pouvoir tester.
Après 3 tentatives pour faire la release se soldant par des échecs, il est décidé de constituer une équipe de 6 personnes pour effectuer cette tâche.

Résultat : 2 ans de retard !

Le constat indéniable qu'il faut retenir de ces expériences est, plus une erreur est détectée tard et plus le coût de la correction sera élevé !

Voici un graphique qui représente bien la situation.

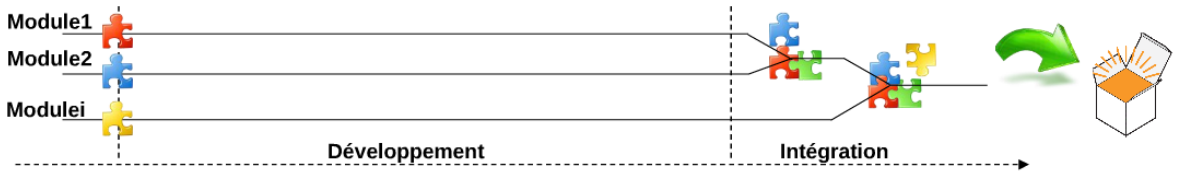


Les 5% de bogues découverts après release représentent 95% des coûts de correction.

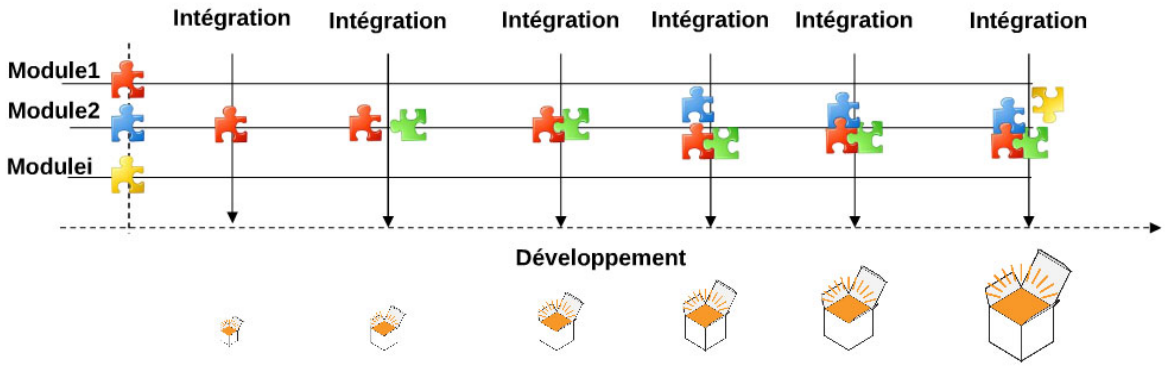
1.4. L'intégration continue, la solution ?

Voici une série de schémas qui comparent "l'avant IC" et "l'après IC" sur un projet quelconque. La phase d'intégration n'est plus une phase finale, mais est incorporée tout au long de la phase de développement.

Avant la mise en place de l'intégration continue :

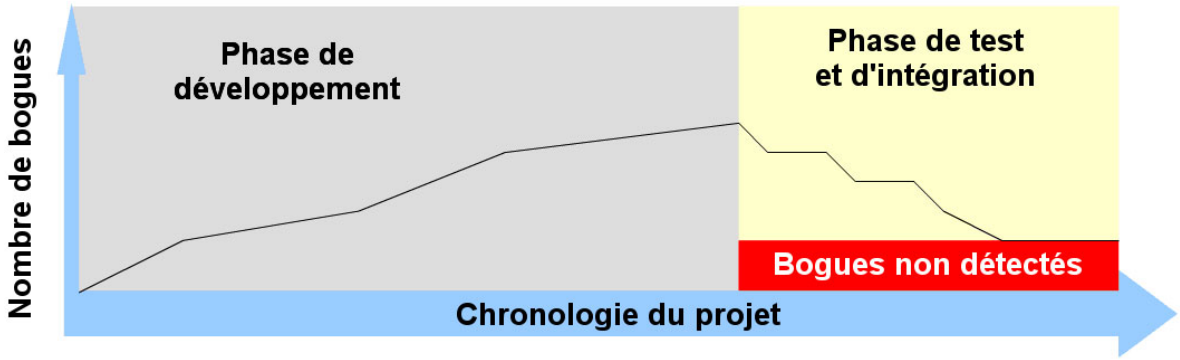


Après la mise en place de l'intégration continue :

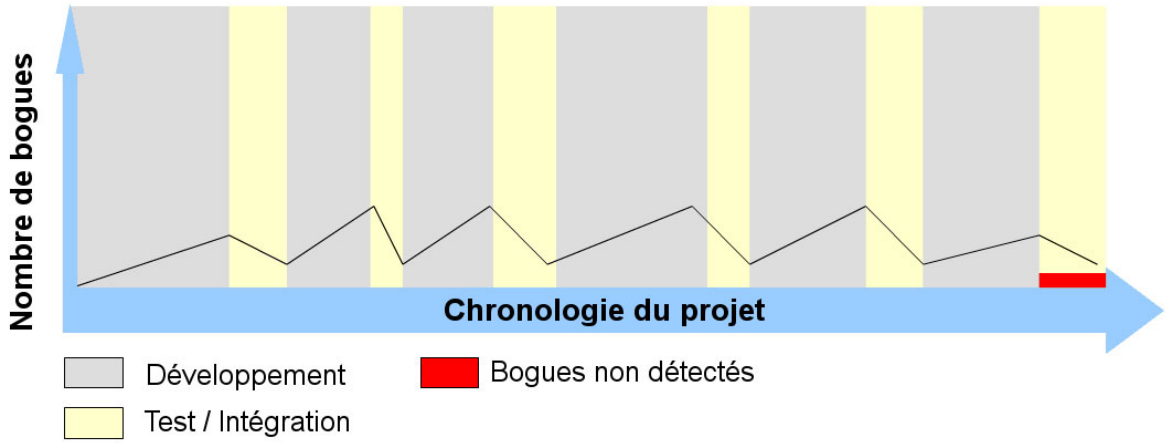


Les deux diagrammes suivants reprennent le principe des précédents en abordant le nombre de bogues dans la comparaison.

Avant la mise en place de l'intégration continue :



Après la mise en place de l'intégration continue :



Martin Fowler décrit l'Intégration Continue comme :

> *… Une pratique de developpement logiciel ou les membres d'une equipe integrent leur travail frequemment, habituellement chacun au moins une fois par jour – ce qui entraine plusieurs integrations par jour. Chaque integration est validee par un 'build' automatique (ce qui inclut les tests) pour detecter les erreurs d'integration aussi vite que possible.*
> *Cette approche permet de reduire significativement les problemes d'integration et permet a une equipe de developper des logiciels de qualite plus rapidement… [*]*

[*] Extrait (traduit en français) du site *http://www.martinfowler.com/articles/continuousIntegration.html*

Comme nous l'avons énoncé, l'Intégration Continue est une méthode agile. Plus précisément, elle fait partie des 12 méthodes d'*eXtreme Programming*. Elle se base sur les principes agiles suivants :

- Fabriquer souvent ("build") ;
- Tester souvent ("test") ;
- Intégrer souvent ("integrate").

Les bonnes pratiques reprises, de l'article de M. Fowler, sont les suivantes. Nous y reviendrons en détail tout au long de ce dossier:

- Maintenir un dépôt unique de code source versionné (SVN) ;
- Automatiser les compilations (Maven) ;
- Rendre les compilations auto-testantes (Tests XUnit) ;
- Tout le monde commite tous les jours ;
- Tout commit doit amener à une compilation du tronc sur une machine d'intégration ;
- Maintenir une construction de build courte ;
- Tester dans un environnement de production cloné ;
- Rendre disponible facilement le dernier exécutable ;
- Tout le monde doit voir ce qui se passe ;
- Automatiser le déploiement.

L'intégration continue industrialise :

- La génération des packages ;
- Les tests unitaires ;
- La qualimétrie ;
- Le reporting.

L'objectif de l'IC est de vérifier, de façon automatique et à chaque modification de code source, que le résultat des modifications ne produit pas de régression de l'application en cours de développement.

2. LE GESTIONNAIRE DE VERSIONS, PRE REQUIS INDISPENSABLE

Le développement des logiciels implique généralement des développements parallèles, avec des développeurs qui intègrent leurs travaux sur un produit unique. Il est très rare, dans les entreprises, de voir des projets où il n'y a qu'un seul développeur. Tous les changements des développeurs ont besoin d'être suivis pour pouvoir identifier et résoudre les problèmes potentiels qui pourraient apparaitre suite à l'intégration de ces changements.

Pour cette raison, il est important d'utiliser un logiciel de gestion de configuration (Software Configuration Management ou SCM), il versionnera tous les changements du code source. En plus de versionner le code, tout ce qui est nécessaire pour compiler le projet devrait être placé sous un contrôle de version, c'est-à-dire :

- Les librairies tierces ;
- Les fichiers de propriétés ;
- Les schémas de base de données ;
- Les scripts de test ;
- Les scripts d'installation.

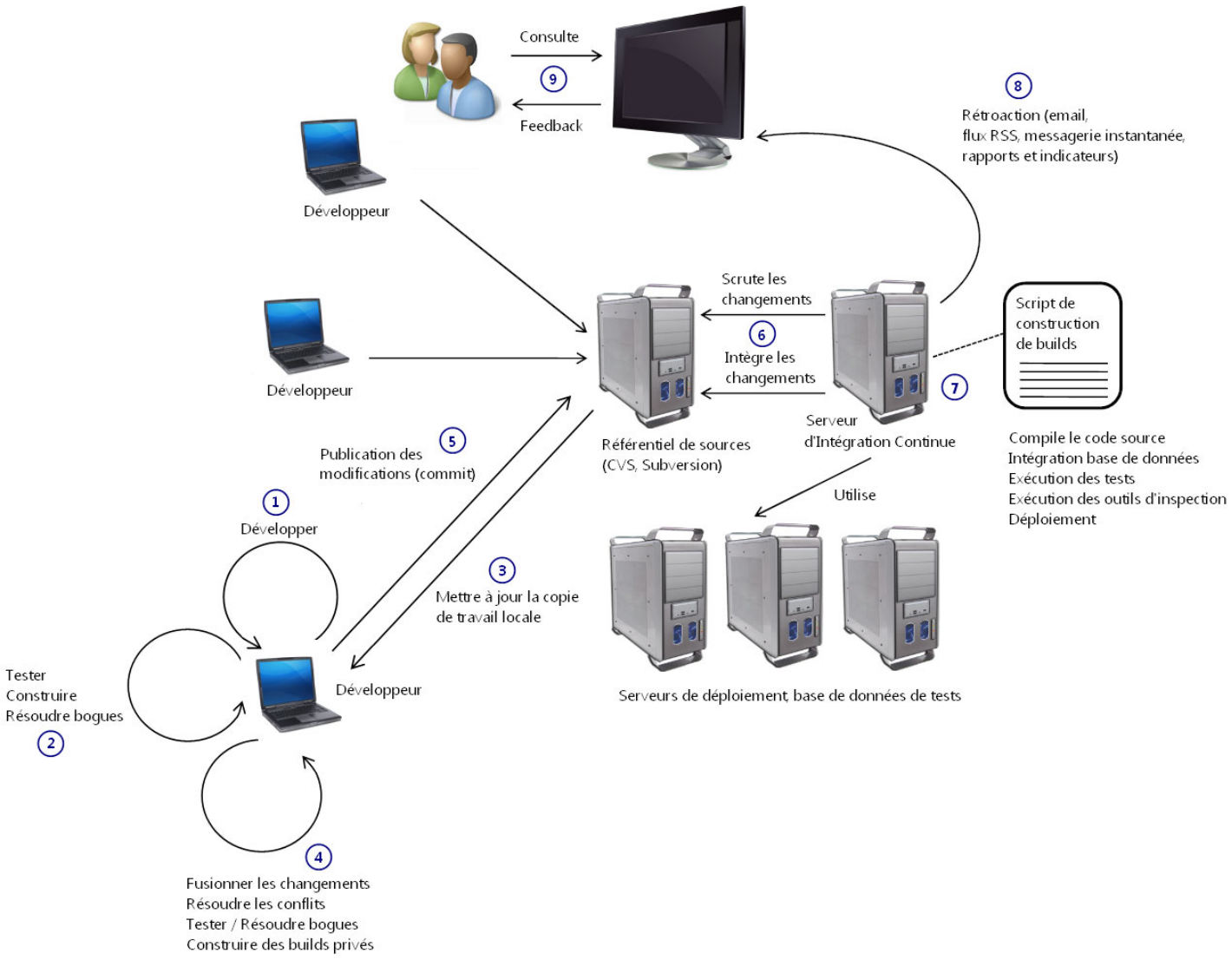
Tous les développeurs auront, au moins, un accès en lecture à tous les fichiers nécessaires pour construire un "build". Cette procédure assure que tous les développeurs puissent travailler avec la dernière version en date du code source.

Pour la rédaction de ce dossier, nous avons choisi Subversion (SVN) comme logiciel de gestion de configuration, mais il aurait pu s'agir de CVS, SourceSafe ou d'un autre logiciel équivalent.

3. FONCTIONNEMENT DE L'INTEGRATION CONTINUE

3.1. L'intégration continue en un schéma

Voici un schéma présentant les étapes (numérotées de 1 à 9) et les acteurs de l'intégration continue.



3.2. Le build

Cela fait plusieurs fois que l'on cite le mot "build". Ce terme anglophone est au cœur des concepts de l'IC, tâchons d'en donner une définition. Un build n'est pas qu'une simple compilation, c'est beaucoup plus. En réalité, il n'y a pas de définition précise. Toutes ne sont pas obligatoires, mais on trouve en général les tâches suivantes dans un build :

a. Chargement de la dernière version du projet depuis le SVN ;
b. Compilation ;
c. Exécution des tests unitaires ;
d. Inspection du code (en vue de générer les métriques de qualité) ;
e. Construction des releases (JAR, WAR, EAR…) ;
f. Déploiement de l'application sur l'environnement de développement (copie des librairies, configuration et redémarrage + sur le remote repository dans le cadre de Maven) ;
g. Exécution des tests d'intégration ;
h. Génération de la documentation, des rapports, des notes de release (par exemple la Javadoc ou les rapports Maven).

Un build, une fois construit, est en fait un package de livrables pour le client qui ont été pleinement testés. Il contient tous les fichiers nécessaires (exécutables, documentation…).

Il est très important que la construction des builds se fasse en une seule commande (la même pour tous les développeurs). Les développeurs ne doivent pas perdre du temps à compiler le projet, exécuter les tests, déployer l'application… faire toutes ces étapes une par une manuellement. De plus, les builds doivent être rapides. En effet, le nombre de builds construits dans la journée augmente exponentiellement avec le nombre de développeurs et le nombre de projets, mais également avec la complexité du projet. Enfin, il existe plusieurs types de builds (voir 3.5. Les différents types de build).

Si le schéma de la page précédente comporte 9 étapes, nous pouvons en agréger certaines, et expliquer l'intégration continue en trois étapes principales.

3.3. L'intégration continue en trois étapes

1. Les développeurs publient leurs modifications de code sur le serveur SVN (commit). Ils auront pris soin d'écrire des tests sur leurs nouveaux codes et de lancer des builds privés (= locaux) sur leurs postes. C'est l'étape la plus importante du processus, elle est détaillée dans une prochaine partie de ce chapitre (3.4.1. L'intégration locale).

2. Le serveur d'intégration scrute les changements sur le SVN. A chaque fois qu'il analyse un changement, il réalise un build d'intégration. C'est une méthode d'automatisation "on stimulus".

Il est important de noter que le serveur d'intégration continue utilise des scripts et des environnements de build déjà existant. Par exemple, si 'make' est utilisé pour compiler et lier les composants écrits en C, alors le serveur appellera simplement le 'makefile' pour initialiser la construction du build.

Parce que le serveur d'intégration continue utilise un système de build déjà existant, il est important pour l'équipe de développement de consacrer du temps et des efforts à :

- Diminuer, optimiser au maximum le temps de création d'un build ;
- Ecrire des tests ;
- Inclure ces tests dans la construction du build.

Prendre du temps sur ces éléments, même si cela peut demander des modifications en profondeur sur le projet, est indispensable pour être en accord avec l'intégration continue. Non seulement le processus de build sera amélioré, mais c'est la qualité globale du projet qui le sera également.

Nous avons parlé d'une automatisation "on stimulus", mais il est possible d'avoir une construction automatique de build régulière, par exemple toutes les 30 minutes. Cette méthode d'automatisation "on the clock" permet d'avoir une charge moins importante sur le serveur, mais peut se révéler moins efficace, dans la résolution des erreurs, qu'une construction de build à chaque commit (notamment si les développeurs ont commité beaucoup de codes en l'espace de ces 30 minutes). Gardez en mémoire que, plus il y a de builds construits, plus il sera facile de diagnostiquer l'origine d'un problème de compilation ou de comprendre pourquoi certains tests sont en échec. En effet, il y aura moins de changements entre les deux builds.

3. A chaque fois qu'un build est réalisé, un courriel de notification est envoyé à tous les membres de l'équipe. Ce courriel contient notamment le statut du dernier build. En cas d'échec du build, les développeurs pourront travailler directement sur sa correction.
En plus des courriels automatiques, les développeurs consultent régulièrement le portail du serveur d'intégration depuis leurs postes ou sur un écran dédié (encore mieux !). Il contient des indicateurs parlants (exemple de la météo sous Hudson) et des statistiques précises (exemple de Sonar). En cas d'échec d'intégration du projet, on

peut combiner les méthodes, par exemple, envoyer un courriel aux développeurs et publier un message d'erreur sur une page web de synthèse. Les moyens disponibles varient selon les outils d'intégration continue. Nous verrons davantage d'exemples un peu plus loin (3.6. Le suivi du statut des builds).

3.4. Le développeur au centre de l'intégration continue

Cette section apporte plus de détails sur la première étape qui est l'une des plus importantes dans la réussite des builds.

3.4.1. L'intégration locale

Les développeurs modifient le code, lancent un build privé puis commitent en cas de build privé réussi (sinon corrige les erreurs). Voici le processus complet :

a. Le développeur récupère une copie du dernier code source en date à partir du SVN.

b. Il fait toutes les modifications au code dont il a besoin pour accomplir sa tâche.

c. Une fois qu'il a terminé, le développeur lance un build privé sur sa machine.

d. Si le build réussi, le développeur peut commencer à songer à commiter.

e. Le problème est que d'autres développeurs auront peut-être commité leur codes avant lui (c'est généralement le cas).

f. Pour le vérifier, le développeur doit donc mettre à jour sa copie locale pour récupérer les dernières modifications.

g. Ensuite, il doit refaire un build privé et…

h. …Si le build privé rate, le développeur doit résoudre le problème.

i. Une fois que le build réussit avec le code synchronisé, le développeur pourra enfin commiter ses changements.

j. Une fois le code commité, le serveur d'intégration lance un build d'intégration (dans le cas d'une automatisation "on stimulus")

k. Le développeur reçoit un courriel quelques secondes plus tard lui indiquant le statut du build. Ou, il peut consulter, par lui-même, le portail du serveur ou un blog dédié ou un moniteur dédié.

On se rend compte que, si tous les développeurs suivent ces étapes à la lettre, la possibilité d'obtenir des builds d'intégration qui échouent sur le serveur d'intégration, est très faible.

3.4.2. Les 8 bonnes pratiques du développeur

Le développeur est le principal acteur de l'intégration continue. Sans son implication, l'intégration continue est, tout simplement, impossible.

Le développeur devra changer ses habitudes et devra "jouer le jeu". Il aura 8 bonnes pratiques à mettre (constamment) en œuvre :

1. **Commiter le code fréquemment** *["Commit early, commit often"]*

   Souvent les développeurs ne veulent pas commiter leurs changements dans le code, soit ils ne veulent pas affecter le travail des autres, soit ils ne veulent pas être accusés d'avoir cassé le dernier build. Malheureusement, cette stratégie s'avère lourde de conséquences sur l'intégration continue. En effet, il faudra débugger des portions de code plus large, car les changements seront plus importants (représentant le travail de plusieurs jours).
   Attendre plus d'un jour pour commiter son code mettra en péril l'intégration continue, à cause d'une perte de temps trop importante pour résoudre les builds qui seront, à coup sûr, des échecs. Le but est de faire des petits changements sur son code et de commiter après chaque tâche. Dès que cela est possible, le développeur doit commiter son code. Il faut commiter, au minimum, une fois par jour !

2. **Commiter du code bon** *["Never commit broken code"]*

   Une hypothèse dangereuse sur un projet est : tous les développeurs ne commitent jamais du code qui ne fonctionne pas. Les développeurs ne devront jamais commiter un code qui ne passe pas un test ou qui ne compile pas avec un autre code. Ils peuvent le vérifier avec la construction de builds privés.

3. **Résoudre les builds ratés rapidement** *["Fix build failures immediately"]*

   La cause d'un build qui a raté peut être n'importe quoi, cela peut être une erreur de compilation, un test qui ne passe pas, une métrique en dessous d'un certain seuil, un déploiement qui a échoué.
   Dans un contexte d'intégration continue, chaque erreur découverte doit être résolue immédiatement, elle a de grandes chances d'être petite. Tous les développeurs qui ont

commité précédemment et qui ont amené à un build raté doivent être impliqués dans la correction.
Une culture doit être adoptée par tous les membres de l'équipe. Elle se résumerait ainsi : la correction d'un build qui échoue est la "top" priorité de toute l'équipe.

4. Ecrire des tests automatisés *["Write automated developer tests"]*

Les tests sont aussi importants que la compilation. Si un seul test échoue, c'est le build lui-même qui échoue.

5. Tous les tests et les rapports d'inspection doivent passer *["All tests and inspections must pass"]*

Un développeur peu scrupuleux peut avoir l'idée de commenter le test qui ne passe pas. Non seulement, il sera identifié rapidement avec le SVN, mais il participera à la baisse du taux de couverture de test du code.

6. Créer des builds privés *["Run private builds"]*

Pour éviter des builds intégrés qui échouent, les développeurs vont devoir émuler des builds d'intégration sur leurs postes. Ces builds privés vont permettre l'intégration de ses modifications avec celles des autres développeurs.
Le développeur récupère la dernière version du code depuis le SVN puis exécute le processus de création du build d'intégration localement.
Dans le cas d'un build privé réussi, qui inclut les récents changements des développeurs, le développeur pourra alors commiter ses changements qui auront alors très peu de chance d'aboutir à un build d'intégration raté.

7. Garder les builds dans le vert tout au long de la journée *["Keep builds in the green"]*

A ce sujet, deux mesures sont très significatives : le nombre de commit par jour et le statut des builds de la journée.
Rappelons que chaque développeur devra faire, au minimum, un commit par jour. Le nombre de commit démontre généralement la taille des changements apportés au code. Il est évident que, plus il y a de commits sur le code, plus la taille de ces changements sera petite et plus les problèmes d'intégration seront rapidement identifiés et résolus.
Le statut du build devra être au vert durant un grand pourcentage de la journée. Obtenir un build de statut rouge arrivera forcément tôt ou tard, et il est très important

de toujours revenir à un statut vert. Il ne faut jamais habituer l'équipe à attendre dans un statut rouge le temps qu'une autre tâche du projet soit réalisée. Le développeur devra annoncer à l'équipe qu'il a cassé le build et qu'il s'en occupe. L'intégration continue n'est pas utilisée pour stigmatiser, mais pour responsabiliser.

S'il n'y avait qu'une phrase à retenir et qui résume bien cette septième bonne pratique, ce serait : il est normal de casser le build de temps en temps, mais il n'est pas normal de le laisser cassé.

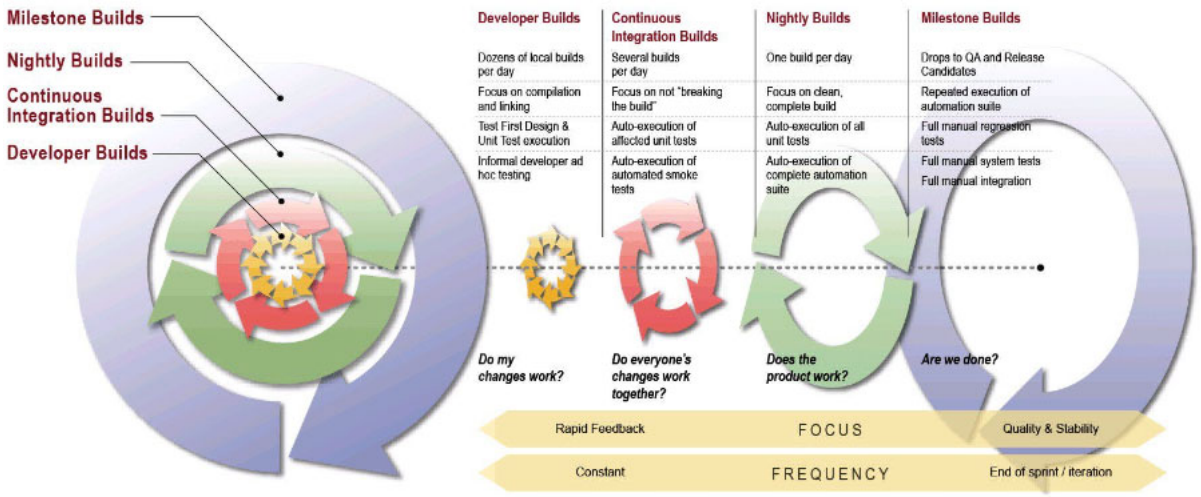
8. Eviter de récupérer du mauvais code *["Avoid getting broken code"]*

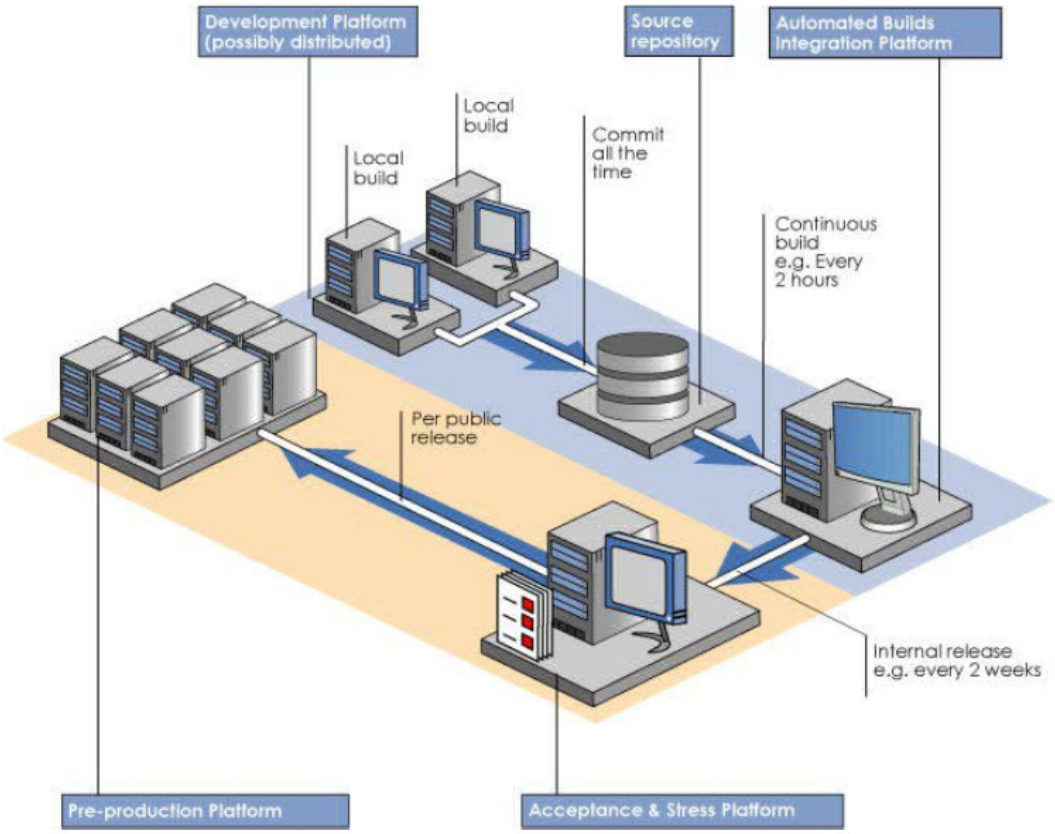
Quand le build est dans le rouge sur le serveur d'intégration, les développeurs ne doivent pas récupérer la dernière version du code. Autrement, ils vont perdre du temps à corriger des erreurs qui ne sont pas les leurs. D'autant plus que, les développeurs responsables du build raté doivent travailler sur la résolution de leur code et vont le commiter de nouveau dans le SVN. Deux choix s'offrent à vous : soit vous attendez que le changement soit commité, soit vous allez aider les développeurs qui travaillent sur le bogue.
Ainsi, on voit qu'il est très important, qu'à tout moment les développeurs doivent connaitre l'état du dernier build. L'envoi de courriel par le serveur d'intégration continue est donc très important. La communication orale sur les statuts de build l'est tout autant dans le cas où les courriels ne seraient pas tous lus (amélioration de la cohésion de l'équipe).

Le challenge permanent de l'intégration continue est de toujours mettre en œuvre ces bonnes pratiques.

3.5. Les différents types de build

Comme vous avez pu le constater, il y a plusieurs types de build. Les diagrammes qui suivent illustrent les différents types de build et leurs fréquences (en anglais). La prochaine page contient toutes les explications en français.





3.5.1. Le build privé

Le build de développeur ou build privé ou build local ("private build") par le développeur: de l'ordre d'une douzaine par jour par développeur (lancement manuel).

Il répond à la question : Est-ce que mes changements marchent ?

Pourquoi faire des builds privés ?

- Faire son build de manière autonome. S'il rate, on est le seul à le savoir et on sait que le problème vient de nous, donc on met tout en œuvre pour résoudre ses erreurs ;
- On peut conserver les builds privés sur son poste et pouvoir revenir dessus pour s'aider (ce sont des sortes de "checkpoint") ;
- Les builds privés permettent de partager du code mature avec les autres développeurs. Autrement dit, ce que l'on commite est du code qui a été testé grâce au build privé.

3.5.2. Le build d'intégration de jour

Le build d'intégration ("integration build") par le serveur d'intégration continue: plusieurs par jour (lancement automatique).

Il répond à la question : Est-ce que les changements de tout le monde marchent ?

Tous les jours, à chaque commit des développeurs ou à intervalles de temps réguliers, les projets sont compilés, testés unitairement, déployés et les résultats sont générés sur un site internet, blog, envoyés par courriel, etc. (voir 3.6. Le suivi du statut des builds)

3.5.3. Le build d'intégration de nuit

Le build de nuit ("nightly build") par le serveur d'intégration continue: une fois par jour (lancement automatique).

Il répond à la question : Est-ce que le produit marche?

Toutes les nuits à une heure bien définie, un build d'intégration plus complet est réalisé sur chacun des projets. Tous les tests sont exécutés. Par exemple, les tests de performance qui sont ignorés lors des builds de jour.

Les builds d'intégration de jour (qualifiés de partiel) sont plus "light" que les builds de nuit (qualifiés de complet) pour éviter un temps d'attente trop long aux développeurs.

La raison est simple : imaginez-vous attendre une heure après chaque commit pour connaitre l'état du build.

3.5.4. Une bonne pratique dans la construction des builds

Build de jour : compilation + tests unitaires.
La durée de ce type de build ne doit pas excéder 10-15 minutes.

Build de nuit : compilation + toutes les catégories de tests + rapports de qualité + packaging + déploiements.
La durée de ce type de build importe moins. Eviter cependant de dépasser une heure.

3.6. Le suivi du statut des builds

Cette section apporte plus de détails sur la troisième étape : la notification de l'utilisateur après la création d'un build par le serveur d'intégration continue. Plusieurs possibilités, souvent complémentaires, sont envisageables pour notifier l'équipe le plus rapidement.

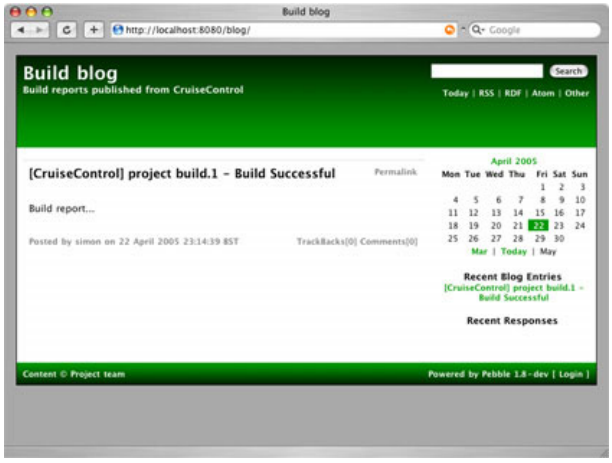
3.6.1. Blogs

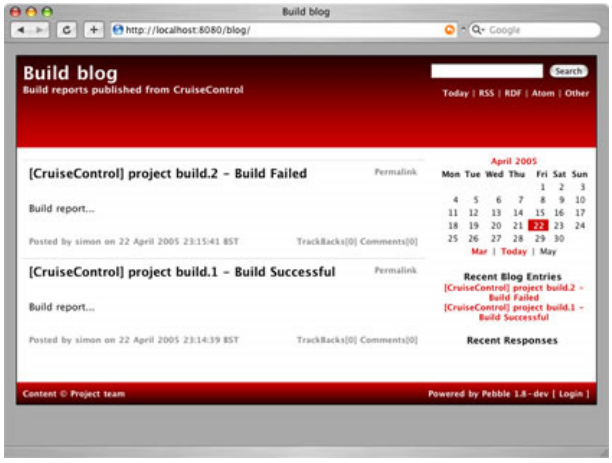
En plus de l'envoi automatique de courriels par le serveur, il peut être intéressant d'ouvrir un "build blog" et/ou un "project blog" que les développeurs pourront consulter régulièrement.

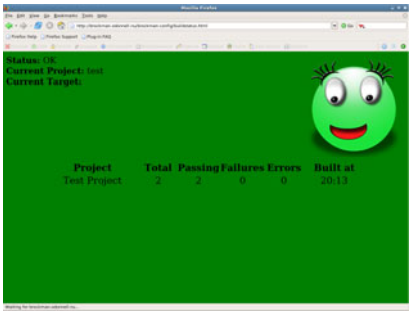
Le "project blog" pourra contenir des billets sur les objectifs de la journée, partager des informations entre les développeurs, etc. Les développeurs auront le droit de créer des billets et de poster des commentaires.

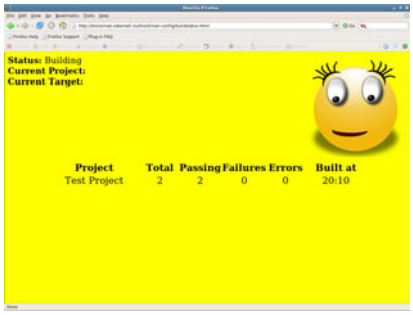
Le "build blog" contient le résultat des builds et peut également contenir les métriques les plus importantes du projet. Seul le serveur d'intégration peut créer des billets sur ce blog, les développeurs ont seulement le droit de poster des commentaires.

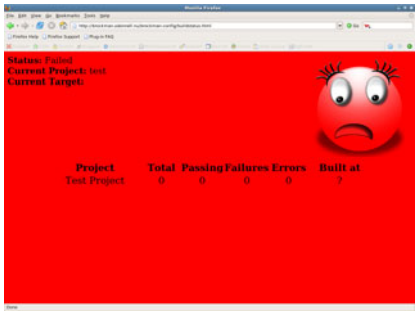
Une bonne idée serait d'adapter la couleur du "build blog" (verte ou rouge) en fonction des résultats du build (comme ci-dessous).











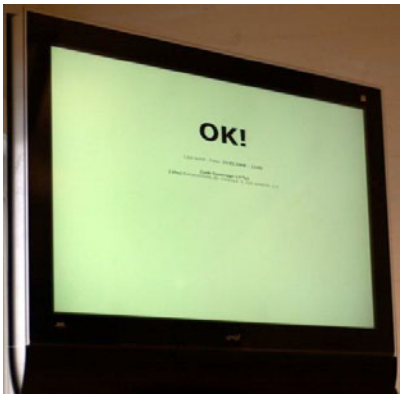
Ces techniques de blogs peuvent faire sourire, mais elles sont novatrices, elles apportent un feedback évident au développeur et sont donc de plus en plus utilisées. Contrairement aux courriels qui peuvent se perdre, le blog formera une sorte d'historique du projet et sera un parfait complément au serveur d'intégration.

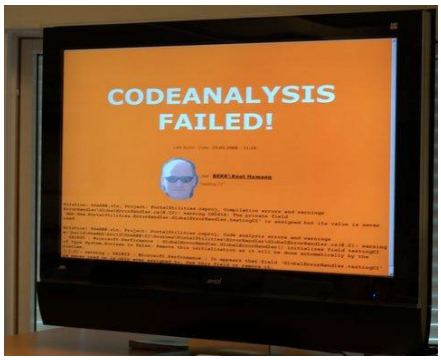
Pour en savoir plus, renseignez-vous sur "Projects blog", "Team blog", "Build blog", "Summary page" et "Brockman".

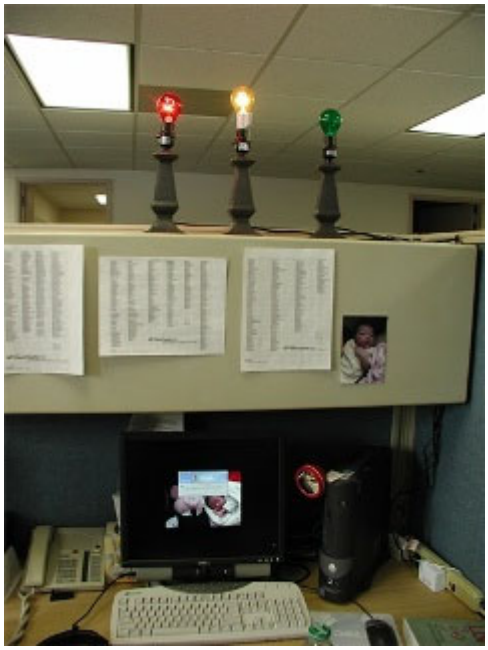
3.6.2. Autres

Des techniques plus simples que la mise en place d'un blog existent, par exemple l'installation d'un écran dédié, visible par tous les membres de l'équipe, qui affiche le portail d'intégration continue.

Ci-dessous, des exemples un peu plus “geek”… Un programme personnalisé relié au serveur d’intégration et affiché sur un écran de télévision, ou encore plus farfelu, des jeux de lumière.







3.7. Les bonnes pratiques générales

Nous avons vu les bonnes pratiques qu'un développeur doit constamment mettre en œuvre (3.4.2. Les 8 bonnes pratiques du développeur). Voici maintenant des bonnes pratiques générales, concernant l'ensemble des membres de l'équipe. Elles sont à mettre en œuvre et à garder à l'esprit pour maximiser les chances de réussite de l'intégration continue.

- Lors de la construction des builds, opter pour une compilation complète ("clean build") plutôt qu'une compilation incrémentale, c'est-à-dire en supprimant toutes les classes déjà compilées, même si elles n'ont pas été modifiées depuis la dernière intégration. Cela vous évitera pas mal de problèmes.

- Conserver une fréquence élevée des intégrations, toutes les nuits est le strict minimum. Le coût de correction d'un problème est proportionnel à son délai de détection, même au niveau de la journée… Simple exemple : il est des fois plus rapide de corriger un bogue le jour même plutôt que d'attendre le lendemain matin. Se replonger dans le code n'est jamais instantané !

- S'assurer que les commits sont fréquents dans le SVN tout en évitant les commits massifs en fin de journée. Un SVN n'est pas qu'un outil d'archivage. Les données doivent toujours être cohérentes ! Une bonne pratique serait un commit après chaque évolution mineure. Faire un commit pour l'ajout d'un commentaire serait absurde. A l'inverse, archiver une journée de travail en une fois peut vite tourner à la catastrophe (conflit avec d'autres commits, modifications de l'architecture trop importantes, etc.).

- Etre réactif aux échecs d'intégration. Tout l'intérêt est perdu si personne ne se sent concerné. Le but est de corriger les problèmes tant qu'ils sont récents et donc (plus) faciles à résoudre.

4. LES OUTILS

Voici la liste des outils que vous pouvez utiliser pour faire de l'intégration continue. Le but de ce document étant de présenter l'intégration continue et non les différents outils du marché, le lecteur sera libre de se renseigner et de choisir suivant ses critères (gratuité, langages supportés, taille de la communauté, interopérabilité, etc.). Nous listons ici les plus utilisés.

4.1. Liste globale (non exhaustive)

Gestionnaires de versions :

- SVN (subversion) ;
- CVS ;
- ClearCase ;
- SourceSafe, etc.

Serveurs d'intégration continue :

- Cruise Control ;
- Banboo ;
- Hudson ;
- Continuum, etc.

Gestionnaires de dépendances :

- Maven ;
- Ivy, etc.

Assemblage et building :

- Maven ;
- Ant ;
- Buildr, etc.

Outils de test :

- XUnit ;
- Mock object ;
- Selenium, etc.

Outils de qualimétrie (génération de métriques) :

- Sonar ;
- PMD ;
- CheckStyle, etc.

4.2. Outils de qualimétrie

Je vais m'attarder un peu sur les outils de qualimétrie et, en particulier sur *CheckStyle* et *PMD* qui sont des produits très répandus dans le monde Java.

Ces deux outils sont gratuits et disponibles sous la forme de plug-in pour les principaux environnements de développement (Integrated Developement Environement ou IDE) du marché. Ils vérifient les règles de programmation et la complexité du code source. En étant intégrés à l'IDE, ces outils permettent au développeur de visualiser directement les remarques concernant le code qu'il vient d'écrire. Notez qu'il est possible de les intégrer à Maven. Les résultats d'analyse apparaissent alors dans les tableaux de bord avec des accès au code concerné.

Voici les principaux éléments qui sont vérifiés et détectés :

- Taille du code trop grande ;
- Lisibilité du code (cohérence du style de nommage, organisation des sources : rendre une copie propre) ;
- Documentation (présence de la Javadoc) ;
- Respect des règles de paquetages, de modularité, de dépendances et d'architecture en couches ;
- Test et couverture de test ;
- Les copier-coller ;
- Respect des règles de programmation (nombreux sous-domaines) ;
  - Variables locales non utilisées ;
  - Blocs "catch" vides ;
  - Paramètres inutilisés ;
  - Blocs "if" vides ;
  - Imports dupliqués et inutilisés ;
  - Méthodes privées inutilisées ;
  - Classes à transformer en Singleton ;
  - Noms de variables et de méthodes trop longs ;
  - Etc.

Avec ces outils, on cherche à améliorer le code, et plus particulièrement sa :

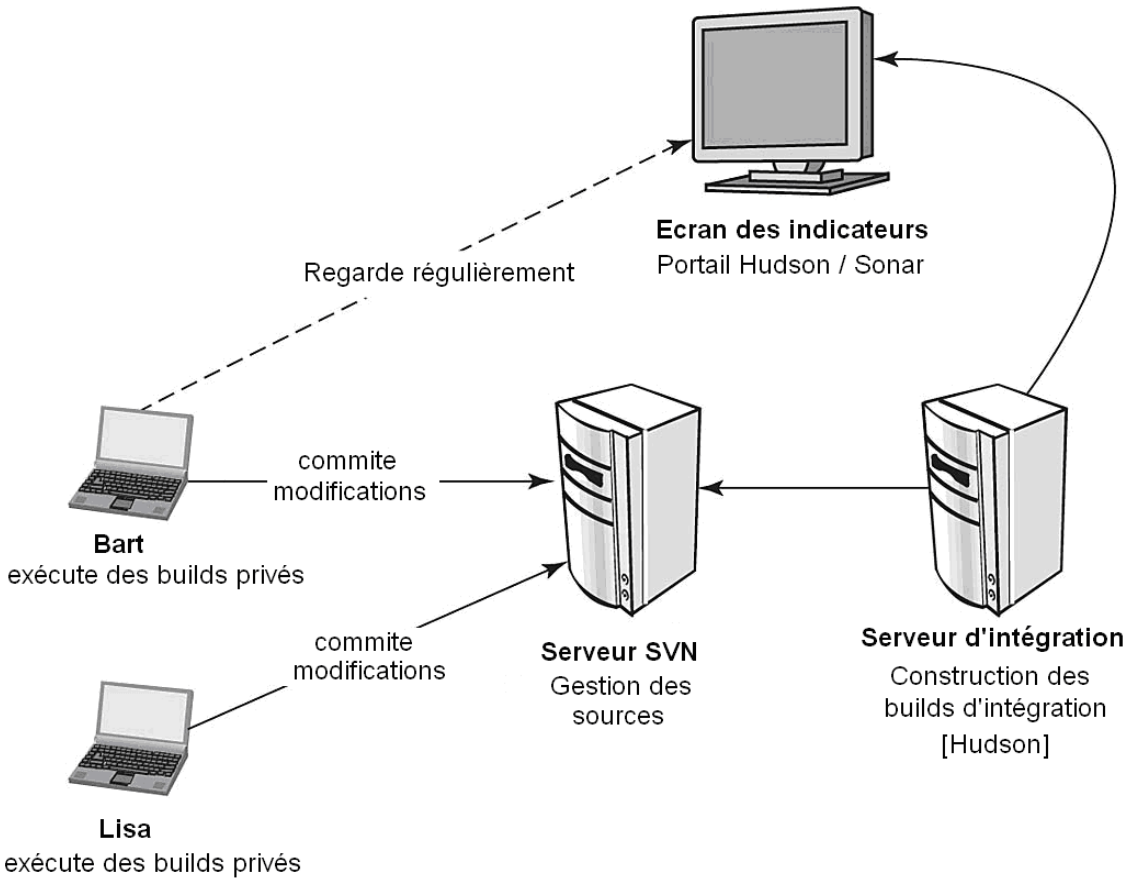
- Usabilité / Lisibilité ;
- Maintenabilité ;
- Efficacité / Performance ;
- Portabilité ;
- Fiabilité ;
- Testabilité.

5. L'INTEGRATION CONTINUE PAR L'EXEMPLE

A ce niveau de lecture, vous devriez avoir une assez bonne compréhension de l'intégration continue. Pour en être sûr et tester vos acquis, voici un exemple concret. L'exemple qui suit (encore fictif…) décrit une situation quotidienne entre deux développeurs à Capgemini, entreprise pratiquant l'intégration continue.

Bart ouvre la porte de son bureau, il observe le large écran qui lui montre des informations en temps réel sur son projet. L'écran lui indique que le dernier build d'intégration, datant de quelques minutes, a été un succès sur le serveur d'intégration. L'écran lui montre également une liste de métriques sur la qualité du projet (qualité du code, duplication de code, etc.). Bart est l'un des 15 développeurs sur le projet Java de création d'un système de réservation pour la SNCF.

Ci-dessous, le schéma montre un exemple d'activité de l'équipe. A titre d'exemple, les logiciels utilisés sont Hudson (serveur d'intégration continue), Sonar (générateur de métriques) et SVN (gestionnaire de versions).



5.1. Une journée ordinaire sur le “mode IC”

Voici le déroulement d’une journée plutôt “classique” dans le contexte de l’intégration continue.

Consultation des métriques

Bart a un objectif à accomplir dans sa journée. Il doit faire du “refactoring” de code sur certaines classes Java ayant une duplication de code trop élevée. Ces observations sont basées sur le dernier rapport généré par Hudson, le serveur d’intégration.

Modifications dans le code

En fin de matinée, Bart a fini de faire les modifications sur son poste. Il a enlevé tout le code dupliqué et effectué un bon nettoyage !

Exécution du build privé

Avant de commiter tous ses changements sur le serveur SVN, Bart exécute d’abord un build privé sur son poste, qui va compiler le nouveau code et exécuter tous les tests sur ce nouveau code. Bart a pris soin de prendre la dernière version du code depuis le SVN.

Commit du nouveau code

Après un build privé réussi (pas d’erreur de compilation, tous les tests exécutés avec succès), Bart commite ses modifications sur le serveur SVN.

Build d’intégration automatique

Après un petit moment, le serveur d’intégration Hudson qui surveille à intervalle régulier le serveur SVN, est mis au courant des modifications de Bart et démarre alors un build d’intégration. Ce build d’intégration démarre plusieurs outils d’inspection automatisés (dont Sonar). L’objectif est de vérifier que le code respecte toutes les conventions de codage qui ont été définies.

Vérification du statut du build d'intégration

Quelques secondes plus tard, Bart reçoit un courriel du serveur d'intégration lui signalant que certains standards n'ont pas été respectés par son nouveau code. Les indicateurs sont à l'orange (présence de nuage sur Hudson). Bart se rend compte qu'il a involontairement négligé les rapports locaux CheckStyle, PMD et compagnie.

Action (si besoin)

Bart fait donc les modifications sur son code, vérifie ses rapports locaux, exécute un build privé et commite ses changements dans le SVN. Le serveur d'intégration exécute un nouveau build, cette fois tous les indicateurs sont au vert (soleil sur Hudson). En regardant le rapport généré par le serveur d'intégration, Bart s'aperçoit que son "refactoring" a été un succès et qu'il a pu réduire de manière significative le taux de duplication de code. Mission accomplie !

Exemple de difficulté entre développeurs

Un peu plus tard dans la journée, Lisa, un autre développeur travaillant sur le même projet, arrive dans le bureau de Bart.

Lisa: *Je pense que les changements que tu as faits un peu plus tôt aujourd'hui ont cassé le dernier build !*

Bart: *Hmm, pourtant, j'ai exécuté tous les tests avec succès.*

Lisa: *Oh, je viens de comprendre, je n'ai pas eu le temps d'écrire les tests pour mon code.*

Bart: *Est-ce que tu as bien suivi la métrique de couverture des tests qui a été générée par Sonar?*

Après cette discussion, ils décidèrent qu'un build ne serait pas déclaré réussi si la couverture des tests est inférieure à 85%.

Lisa a ajouté les tests qu'elle avait oubliés. Comme ils ne passaient pas tous à cause du bogue découvert par le serveur d'intégration, elle a corrigé le problème. Le build d'intégration continue à être "dans le vert".

5.2. Bilan de l'exemple

Cet exemple a pour objectif d'être très pragmatique pour comprendre l'intégration continue au quotidien. Il démontre également, qu'il faut que tout le monde soit impliqué pour que l'intégration continue fonctionne et ne cesse de s'améliorer.

Les développeurs, les chefs de projets… en fait, toutes les personnes participantes au projet doivent consulter régulièrement le portail du serveur d'IC (Hudson/Sonar) pour voir où en sont les builds.

Le mot "continue" prend tout son sens. On génère des builds de manière continue tout comme on regarde les rapports générés de manière régulière.

6. LES AVANTAGES

Ce chapitre fait une synthèse de tous les avantages qui peuvent découler de l'intégration continue. Certains sont évidents, d'autres le sont peut-être un peu moins…

6.1. Réduction du temps de correction des bogues

L'un des principaux avantages de l'intégration continue est qu'elle permet d'éviter que la "chasse aux bogues" ne se limite à rechercher à quel endroit s'est chevauché le travail de deux ou plusieurs membres de l'équipe. Ces bogues sont très difficiles à localiser, car le problème ne vient pas d'une personne en particulier, mais de l'interaction du travail de deux personnes différentes. Et cela ne s'arrange pas au fil du temps. Les bogues peuvent souvent avoir été intégrés des semaines, voire des mois, avant d'être détectés et localisés ; ils en deviennent d'autant plus difficiles à retracer. De plus, la correction d'un bogue coûte moins cher pendant la phase de développement qu'en recette client. L'IC permet donc de réduire les coûts liés à la résolution des bogues.

Grâce à l'intégration continue, l'équipe est prévenue rapidement en cas de code incompatible ou manquant (meilleure réactivité). Une grande majorité des bogues dus à des problèmes d'interaction sont ainsi résolus le jour même, en amont. En outre, on repère immédiatement, dans la moitié des cas, où l'interaction a eu lieu. Cela réduit considérablement le champ de recherche du bogue.

Néanmoins, l'intégration continue ne garantit pas que tous les bogues soient identifiés. Le processus repose sur le test, et comme chacun le sait, tester ne signifie pas pour autant une absence totale d'erreurs. Finalement, il faut surtout s'assurer que l'intégration continue corrige suffisamment de bogues pour justifier sa mise en place et son coût.

6.2. Meilleure visibilité du projet

Grâce aux états des builds successifs, la santé du logiciel est mesurable en temps réel. Le responsable de projet, et même toute l'équipe, a une connaissance en temps réel de l'avancement du projet. Par l'intermédiaire des rapports, des indicateurs et métriques générés grâce aux outils de reporting, le chef de projet connait l'état d'avancement du développement très précisément, il a ainsi une meilleure appréciation, tendance et budgétisation du projet. De plus, cette information est disponible en temps réel, ce qui permet de prendre des décisions efficaces à temps.

Nous savons où est-ce que nous en sommes, ce qui marche, ce qui ne marche pas encore, quels sont les bogues à corriger ?… Au sein de l'équipe, les questions et affirmations suivantes auront tendance à disparaitre : "Qu'est ce que tu voulais dire par les tests ne réussissent pas ?", "Quel sont les nouveautés dans la nouvelle version 1.2.3 ?", "Quel est le taux de couverture du code ?".

6.3. Amélioration du travail collaboratif

Les membres d'une équipe de développement intégrant leur travail quotidiennement, il y a une amélioration de l'esprit d'équipe et une meilleure cohésion entre les développeurs. Au sein de l'équipe, les questions et affirmations suivantes auront tendance à disparaitre : "Les modifications que tu viens de faire sont incompatibles avec les miennes ! Comment je vais faire le merge moi maintenant ?", "T'as archivé tes modifications ? ", "Quand est-ce qu'on va décider d'utiliser la nouvelle version 2.6 de la librairie blabla ?", "Je pensais que le problème avait été résolu il y a 2 mois !".

L'IC développe aussi un lien de confiance entre les développeurs et leur logiciel. Avec tous les builds, l'équipe sait que les tests sont toujours exécutés (ils sont donc utiles) pour vérifier le comportement de l'application. L'équipe sait également que le code et le design de l'application répondent aux exigences des standards et que, le résultat du build est un produit complètement fonctionnel et testable.

6.4. Autres avantages

Voici une liste non exhaustive des autres avantages de l'intégration continue:

- Déployer le logiciel n'importe quand, n'importe où. D'un certain point de vue, c'est l'avantage le plus évident de l'intégration continue. Au sein de l'équipe, les questions et affirmations suivantes auront tendance à disparaitre : "J'ai besoin d'un nouveau build dans la journée, tu peux me faire ça rapidement ?", "Pourtant ça marche bien sur ma machine, tu n'as pas oublié un truc ?". De cette façon, une version est toujours disponible pour un test, une démonstration ou une distribution. Cela peut représenter un avantage marketing en cas de demande de démonstration non planifiée de la part du client.

- Les problèmes d'intégration sont détectés et réparés de façon continue, évitant les problèmes de dernière minute. Il y a un gain de temps au niveau du déploiement et du développement.

- Chaque développeur bénéficie régulièrement des évolutions des autres développements de son équipe. La qualité du code est améliorée.

- Il y a un test immédiat des unités modifiées. L'intégration continue encourage de bonnes habitudes de test.

- Le processus est clairement défini et a été approuvé dans de nombreuses entreprises. Tout le processus fonctionne de la même façon tout le temps. Le processus est ordonné. D'abord, en local sur son poste, on regarde l'analyse statique des outils d'inspection avant de lancer les tests. On intègre, puis on regarde ensuite les rapports du serveur d'intégration.

7. DIFFICULTES ET INTERROGATIONS

Il n'y a pas vraiment d'inconvénients à mettre en place l'intégration continue. En revanche, il est très facile de la remettre en cause à la suite d'une mauvaise mise en place, qui, à long terme ne répondra pas aux besoins. Il y a également beaucoup d'interrogations qui peuvent survenir, voire même une "peur" de se lancer… L'objectif de cette partie est de tenter de répondre à ces interrogations et de vous conseiller pour une intégration continue efficace et "sans accrocs".

7.1. Méthode de mise en place

→ Combien de temps prendra la mise en place de l'intégration continue ?

Il est difficile de répondre à cette question. Néanmoins, ce qui est sûr, c'est que mettre en place l'intégration continue sur un nouveau projet prendra moins de temps que sur un projet en cours.

L'implémentation d'une plateforme d'intégration continue sur un nouveau projet prendra quelques heures tout au plus, plus si vous n'avez pas de script de build existant. Au fur et à mesure que la connaissance de l'IC augmente, il augmentera avec le nombre d'outils d'inspection, les déploiements qui seront plus complexes. Toutes ces fonctions additionnelles prendront un peu plus de temps.

L'implémentation d'une plateforme d'intégration continue sur un projet existant peut prendre des jours, voire des mois. En général, beaucoup de tâches doivent être mises en place.

Les gens risquent d'être résistants à ces nouvelles pratiques, une conduite au changement est donc capitale. Il est important d'apporter les changements petit à petit.

Avant même de commencer la mise en place du serveur d'intégration, il est important d' :

- Habituer les développeurs à commiter fréquemment.
- Habituer les développeurs à créer des tests. Demandez à votre équipe de développeurs "Qui écrit des tests unitaires au quotidien durant le travail ?", il y a des chances que la réponse vous surprenne.
- Habituer les développeurs à respecter les rapports locaux (CheckStyle, PMD) déjà installés…

Il faut adopter une approche itérative, augmentez le nombre de règles progressivement, devenez de plus en plus strict sur les règles passées. Motiver les équipes plutôt que réprimander ("Médailles vs. Statistiques Excel").

Il n'y a donc pas une seule approche pour mettre en place l'intégration continue. Dans tous les cas, il faut garder en mémoire que l'intégration continue n'est pas qu'une implémentation technique, c'est une implémentation culturelle et organisationnelle. Les gens résistent au changement naturellement, la meilleure approche est donc d'ajouter ces mécanismes d'automatisation pièce par pièce.

Dans le cas où vous souhaitez démarrer un build à chaque changement dans le SVN, vous pouvez commencer par démarrer un build par jour, de façon à sensibiliser les développeurs à cette nouvelle méthode de build.

Dans un premier temps, la construction du build peut n'inclure que la compilation des sources et la construction des livrables sans exécuter les tests. Cette méthode peut être efficace dans le cas où les développeurs ne sont pas très familiers avec les outils d'automatisation de tests. Une fois qu'ils ont bien compris ce concept et qu'ils ont écrit les tests, vous pouvez exécuter ces tests lors de la construction des builds.

De la même façon, ne démarrez pas l'intégration continue sur l'ensemble de vos projets, mais commencez par un ou deux qui seront les projets pilotes (choisissez des projets qui auront le plus de chance de réussir).

Concernant les indicateurs, mettez des seuils "lâches" au départ puis, resserrez-les progressivement par la suite. Un taux de couverture de tests de 10% sur tout le projet alors qu'il faut atteindre les 80% en affolera et démotivera plus d'un !

Vous l'aurez bien compris, il n'y a pas de recette miracle. La route vers l'adoption d'une plateforme complète d'intégration continue peut être assez longue et sinueuse, mais cela en vaudra la peine.

7.2. Détection du besoin d'intégration

→ A quelle fréquence dois-je lancer des builds ?

Plusieurs possibilités :

- Soit à intervalles de temps réguliers (toutes les 10 minutes, toutes les nuits…), c'est le déclenchement "on the clock" ;
- Soit dès la détection d'une modification dans le SVN (ajout/modification/suppression de fichiers), c'est le déclenchement "on stimulus".

Il est possible que le développeur puisse oublier de commiter toutes ses modifications ou qu'il puisse mettre plusieurs minutes avant d'archiver toutes ses modifications. Durant ce temps, l'état du projet sera tel qu'il amènerait à un build raté. Ces cas sont gérés en temporisant le début de l'intégration. Après chaque commit, on laisse par exemple 5 minutes au développeur pour archiver un second fichier ou ses oublis, et cela jusqu'à ce qu'aucune modification n'ait eu lieu dans les 5 dernières minutes.

7.3. L'intégration continue vue comme une perte de temps

A la lecture de ce document, peut-être vous êtes-vous demandé si un développeur allait devenir un intégrateur, l'intégration continue lui prenant du temps sur son temps de travail.

Et bien non ! Bien au contraire ! Il est vrai que la phase d'intégration est une étape longue, fastidieuse et pas forcément intéressante. Mais, en pratique, plus l'intégration est fréquente, moins elle est longue. L'intégration continue donne la possibilité au développeur, de modifier son code et de savoir s'il a "cassé" le logiciel, car il recevra immédiatement un retour. Ce feedback quasi instantané lui donnera du temps pour qu'il ajuste son code plus rapidement. L'intégration continue permet donc de passer moins de temps sur l'intégration, donc plus de temps sur le développement en lui-même, l'intérêt même du développeur.

L'intégration n'est pas l'activité la plus "glamour" dans le développement, mais elle n'en ait pas moins une des plus importantes, encore plus dans les projets complexes. Les utilisateurs du logiciel ne vous diront jamais "Waouh, j'ai bien aimé votre façon d'intégrer le logiciel dans la dernière release". Et comme cela n'arrive jamais, il ne faut pas croire, pour autant, que l'intégration est une tâche inutile et que tous les efforts seront vains.

On peut voir l'intégration continue comme l'action de mettre les composants d'un logiciel ensemble. Mais la bonne vision serait de la voir comme un pilier central dans le développement du logiciel. Elle assure la santé du programme grâce à des builds successifs réalisés au fil des changements.

7.4. La peur de l'intégration continue

L'intégration continue présente aussi des désavantages. Elle fait peur, car elle peut être à l'origine de bouleversements qui viennent changer les habitudes des développeurs.

Néanmoins, une bonne compréhension de ces inconvénients est indispensable pour les minimiser et obtenir une intégration continue "propre" et efficace.

Les principales questions qui font "peur" :

- Comment va-t-on gérer plus de 200 builds par jour ?
- Comment va-t-on diminuer le temps de construction d'un build à 10 minutes ? Il nous faut plus d'une heure actuellement !!!
- Et si le serveur d'intégration continue tombe en panne ?
- Et si les développeurs ignorent complètement l'intégration continue ?

Voyons quelques éléments de réponse :

- Le serveur d'intégration doit être puissant.
- Il faut se renseigner sur la construction parallélisée de builds ou builds distribués, le clustering, la répartition de charge. Hudson supporte ces technologies.
- Si besoin, utiliser des builds "on the clock" pour diminuer la charge serveur. Il faut trouver le bon compromis.
- Se renseigner sur les outils qui pourraient accélérer un build. Instaurer des nouvelles règles. Par exemple, un build complet (de nuit) ne doit pas dépasser 60 minutes.
- Pensez à vous tourner vers des outils payants dans le cas où les builds sont toujours trop lents ou sont souvent ratés.

- Pensez à installer un écran dédié dans chaque bureau qui montre à l'équipe les états des derniers builds (première page du portail Hudson rafraichie automatiquement par exemple).
- L'intégration continue est une culture ! C'est un peu bête à dire, mais Hudson (ou tout autre logiciel pour le serveur d'intégration continue) doit être vu comme le dernier membre de l'équipe du projet.

Continuons un peu dans le jeu des questions/réponses avec une sorte de FAQ de l'intégration continue. Elle vous aidera à avancer en cas de doute ou d'interrogation.

- **Les développeurs prennent plus de temps pour intégrer leurs codes qu'ils ne le produisent.**

  Cela peut-être vrai au début et c'est une bonne nouvelle pour vous. Les développeurs ont compris qu'il est important de commiter régulièrement, de créer des builds privés pour s'assurer du bon fonctionnement, de regarder les métriques générées localement et celles du serveur d'intégration, de corriger le code en conséquence… Ce travail est loin d'être inutile et permet le développement d'un logiciel de meilleure qualité (et, rappelons-le, une satisfaction du client qui augmente).

  A l'inverse, il faut faire attention à ne pas trop impacter la productivité. Limitez le nombre d'outils, la granularité des rapports, etc. Il peut être judicieux de nommer une personne qui s'occupe de la qualité sur le projet (pas nécessairement le chef de projet, mais un développeur expérimenté). Il sera responsable de regarder les états des builds, les statistiques et d'informer les développeurs en fonction. Il faut également rester à l'écoute des développeurs, assouplir certaines règles des outils. Par exemple, dans le code, est-il vraiment nécessaire de limiter les lignes à 180 caractères ?

- **Au contraire, les développeurs n'ont pas l'air de faire attention à l'intégration continue.**

  Cela peut arriver, il faut prendre soin d'identifier les freins, de réexpliquer les démarches, "récompenser les bons élèves", quitte à réduire vos attentes (réduction ou assouplissement des règles, réduction du nombre d'outils, etc.)

  On ne le dit jamais assez, le processus d'intégration continue sur un projet existant doit être itératif.

- **Il y a trop de changements à faire, ce qui est mis en place actuellement marche bien, on ne va pas s'embêter…**

  Bien sûr qu'il sera impossible de mettre en place une plateforme d'intégration continue du jour au lendemain.

  Premièrement, l'intégration en vaut vraiment la peine à la lecture de tous les bénéfices dégagés. Deuxièmement, une mise en place progressive et une sensibilisation efficace des acteurs ne seront pas si difficiles que cela à mettre en place.

- **Beaucoup de travaux à mener en parallèle et à coordonner (build, intégration continue, analyse du code, gestion de configuration, gestion des tests…), comment va-t-on s'y prendre ?**

  Cette question rejoint la précédente. Ayez une approche méthodique et itérative. Une intégration continue sur un mode "jour au lendemain" sera vouée à l'échec.

- **Tous les builds que nous avons réalisés aujourd'hui ont raté, que se passe-t-il ?**

  Cela arrive typiquement quand les développeurs ne créent pas de builds privés avant de commiter leur code sur le serveur SVN. D'autres raisons peuvent intervenir, soit car ils ont oublié de vérifier les tests (malgré une compilation réussie), soit ils ont oublié de les écrire, soit ils ont oublié de commiter tous les fichiers modifiés. Quoi qu'il en soit, la réponse de la part des développeurs après un build raté ne doit jamais se faire attendre. Il faut toujours corriger les bogues, trouver l'erreur afin de "revenir dans le vert".

- **Il faut acheter un serveur, des nouveaux outils, former des gens… cela coûte trop cher.**

  Réaction compréhensible, mais sans fondement. Il faut effectivement un serveur dédié puissant qui va s'occuper de créer les builds et fournir le portail de métriques, mais cette dépense est véritablement insignifiante quand on voit les bénéfices. Enfin, les outils utilisés pour l'intégration continue sont pour la plupart open-source et gratuits. Pour n'en citer que quelques-uns : Subversion, Archiva, Maven, Hudson, Sonar…

7.5. Le piège de la compilation continue

Il ne faut pas croire faire de l'intégration continue quand on fait de la "compilation continue". Les entreprises peuvent "tomber dans le panneau" très rapidement.

En effet, il n'est pas rare de rencontrer des entreprises qui clament haut et fort qu'elles font de l'intégration continue. En leur posant quelques questions telles que : Quel est le taux de couverture de vos tests ? Combien de temps prend la création d'un build ? Quel est la complexité moyenne de vote code ? Avez-vous beaucoup de duplication de code ? Où stockez-vous vos logiciels déployés ? Vous découvrez rapidement, qu'en réalité, l'entreprise fait une sorte de "compilation continue". Elle a mis en place un outil tel que Hudson qui leur permet de créer des builds dès qu'une modification survient dans le serveur SVN, et en cas d'échec de build, envoie un courriel… Et le processus s'arrête là !

Bien sûr, cette compilation automatique sur une machine dédiée est mieux que rien du tout, mais entreprendre une telle pratique ne donne pas tous les bénéfices d'une véritable plateforme d'intégration continue.

7.6. Les bonnes questions à se poser

Comme on l'a vu, la pratique de l'intégration continue ne se limite pas à une simple installation et configuration d'outils. En permanence, il y a beaucoup de bonnes questions à se poser, mais aussi des mauvaises réponses à éviter…

- Vos développeurs connaissent-ils les 8 bonnes pratiques de l'intégration continue ?
  Connaissent-ils les outils installés sur le serveur d'intégration continue ?
  Les utilisent-ils régulièrement pour consulter les métriques ?
- En moyenne, est-ce que tous les développeurs de l'équipe commitent leurs codes au moins une fois par jour ?
  Employez-vous des techniques particulières pour avoir des commits plus nombreux / réguliers ?
- Quel est le pourcentage de builds réussis dans la journée ?
  Ce pourcentage s'améliore (se dégrade) t-il au fil des mois ?

- Est-ce que les développeurs exécutent des builds d'intégration privés avant de commiter leurs changements, réduisant ainsi les erreurs d'intégration ?
- Est-ce qu'un build d'intégration qui a échoué est une priorité dans le projet ?
  Le build raté est-il résolu dans l'heure qui suit ? Au cours de la journée ? Dès le lendemain ?
- Est-ce que les développeurs évitent de récupérer les dernières sources du SVN lorsque le dernier build d'intégration a échoué ?
- Avez-vous envisagé d'ajouter de nouveaux outils, de processus automatisés à votre plateforme d'intégration continue ?

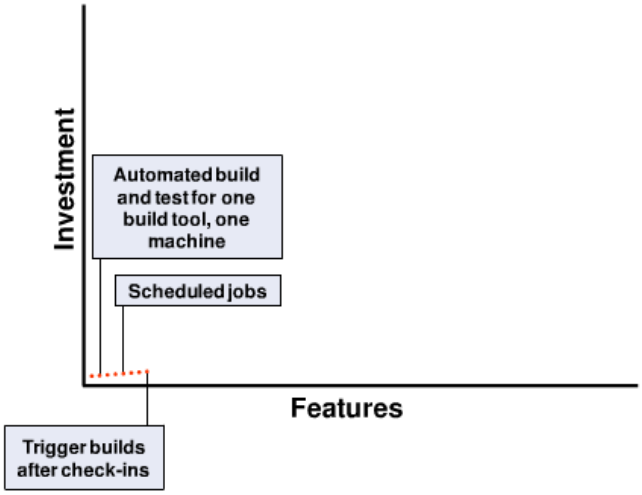
8. EVOLUTION DE L'INTÉGRATION CONTINUE

L'intégration continue n'est pas quelque chose de figé. Il y a toujours des améliorations à mettre en place.

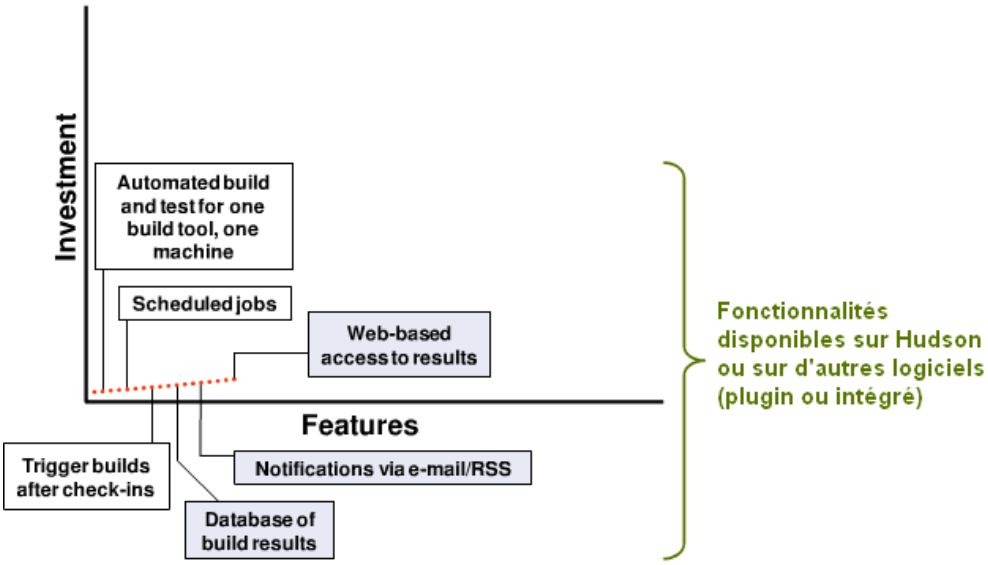
Pour vous en convaincre, voici des schémas, étape après étape. Ils sont en anglais, mais n'en demeurent pas moins explicites.

Source : http://www.scribd.com/doc/6466697/Enterprise-Continuous-Integration-Presentation

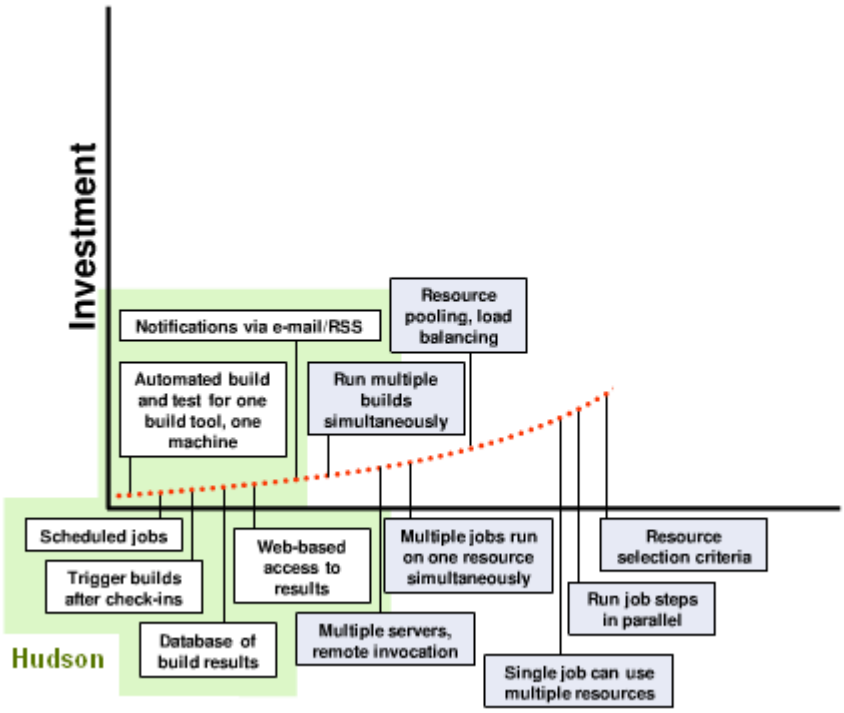
1. Cela semble plutôt simple…



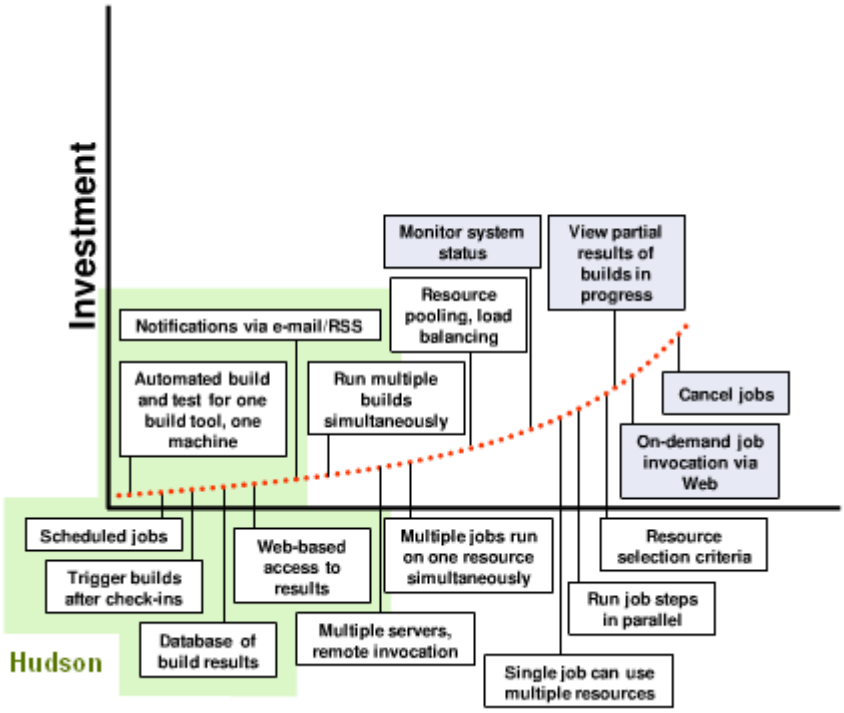
2. Quelques touches de finition…



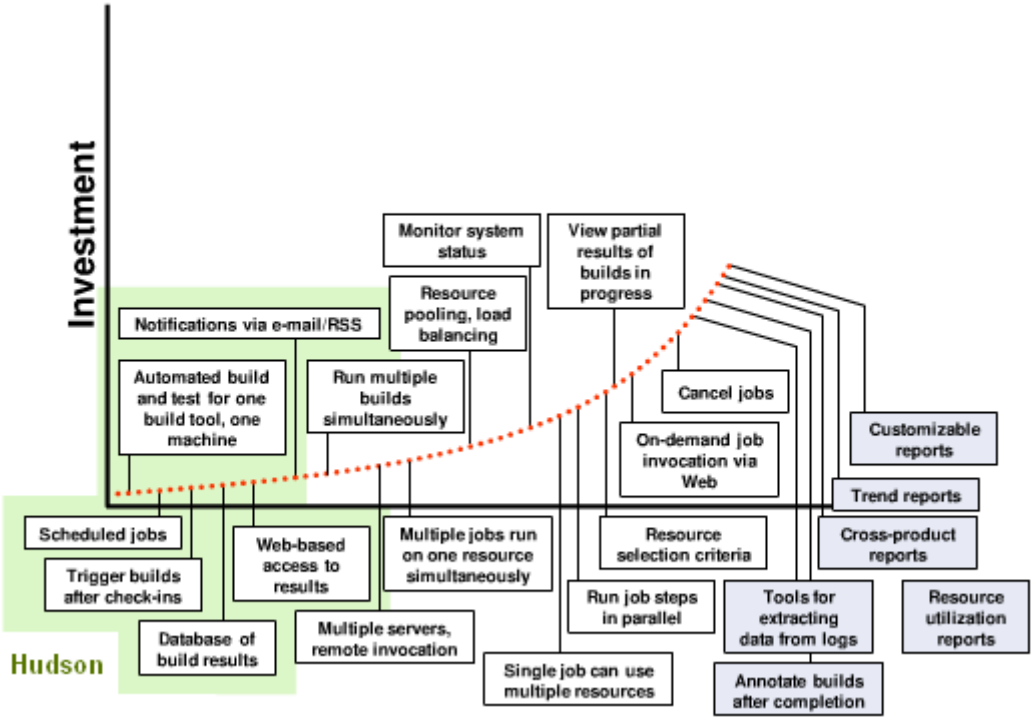
3. Plusieurs serveurs, parallélisation…



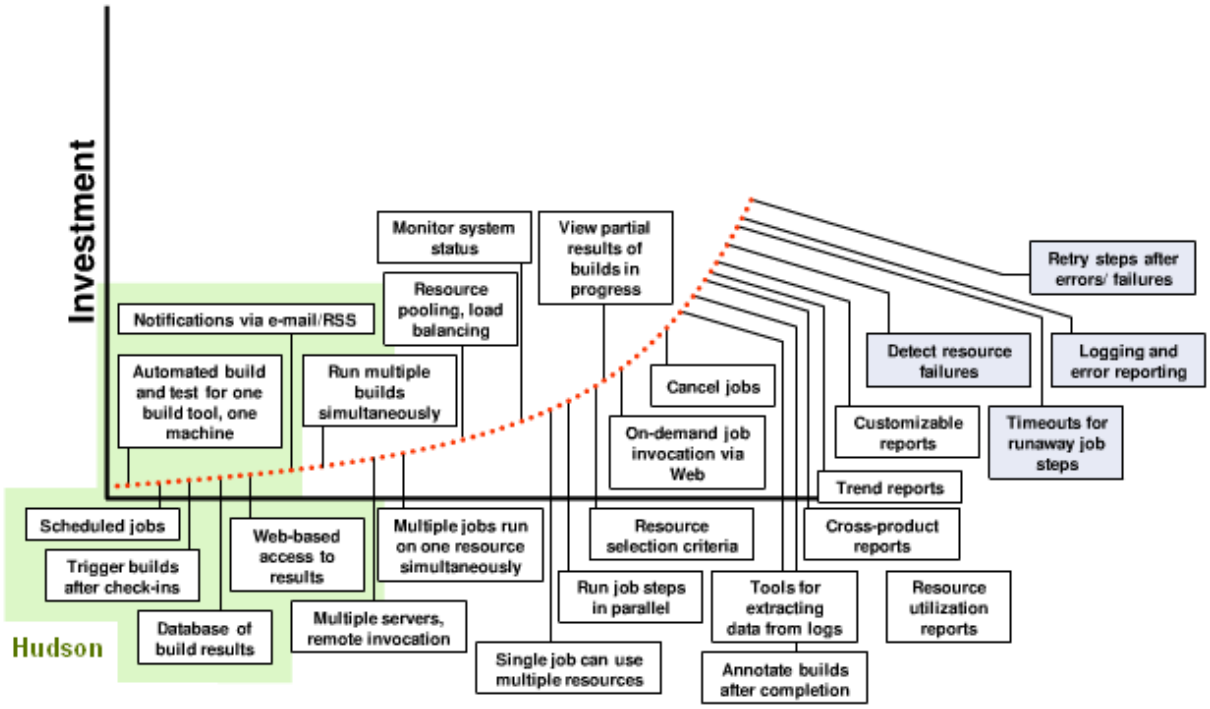
4. Un meilleur monitoring et contrôle…



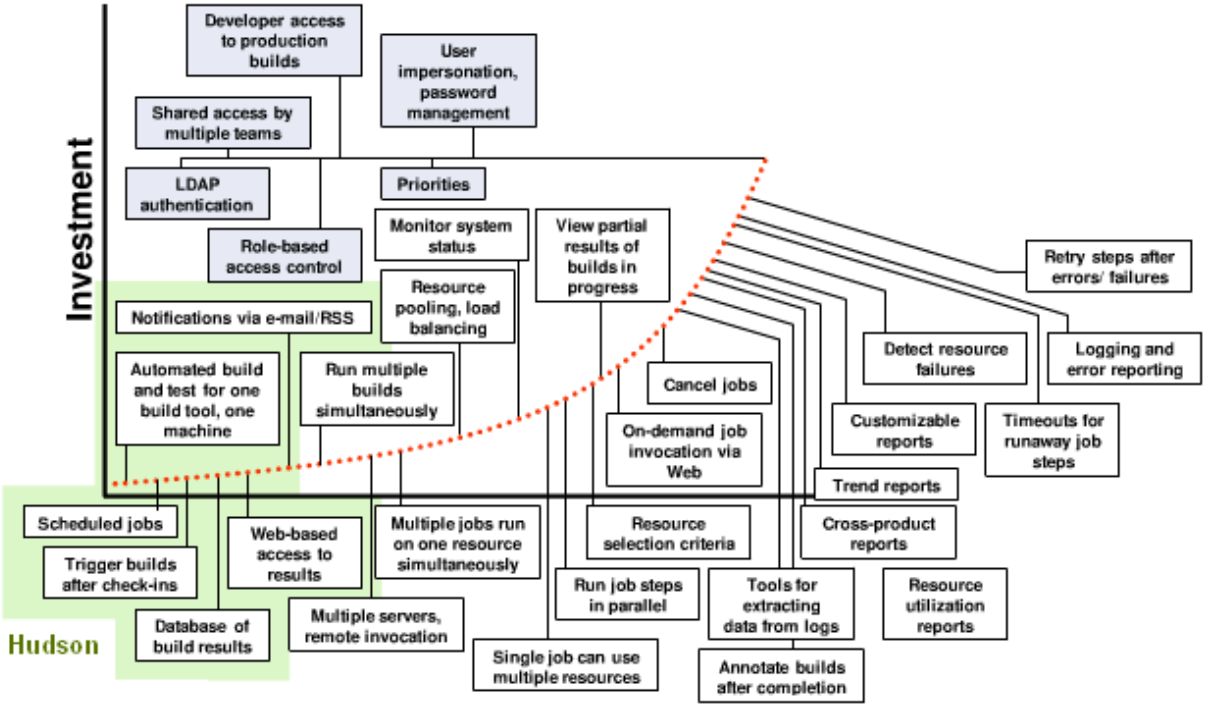
5. Un meilleur reporting…



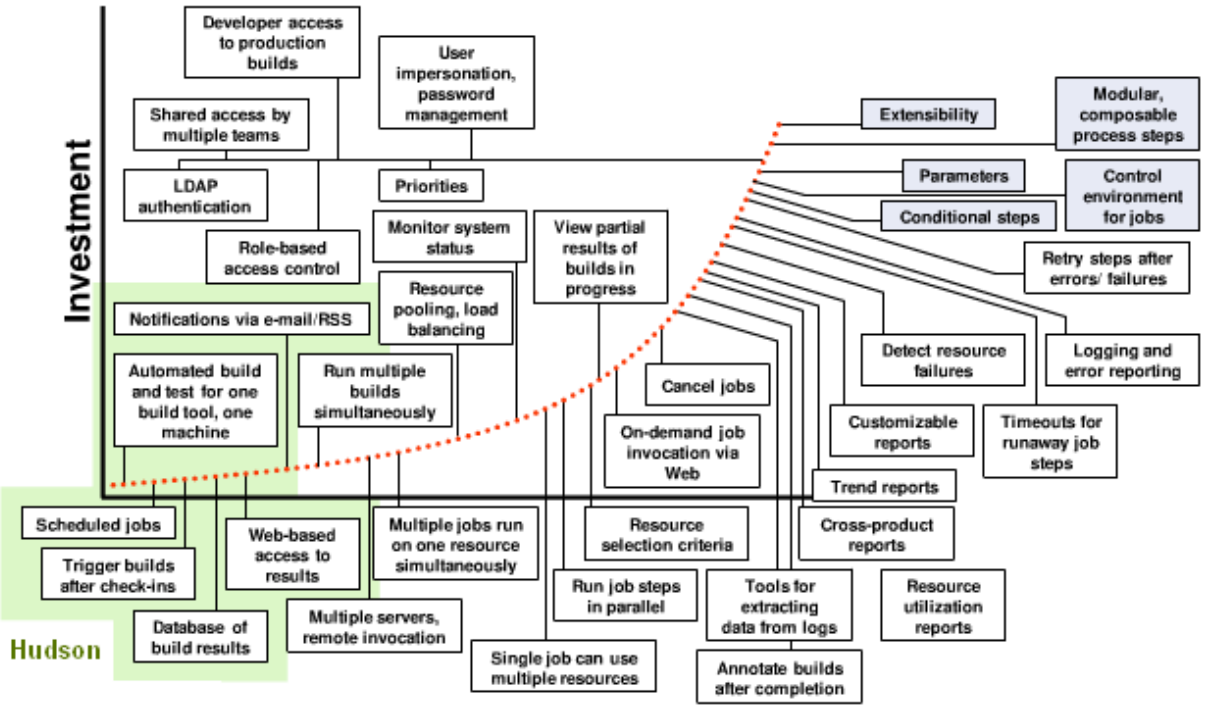
6. Une meilleure gestion des erreurs…



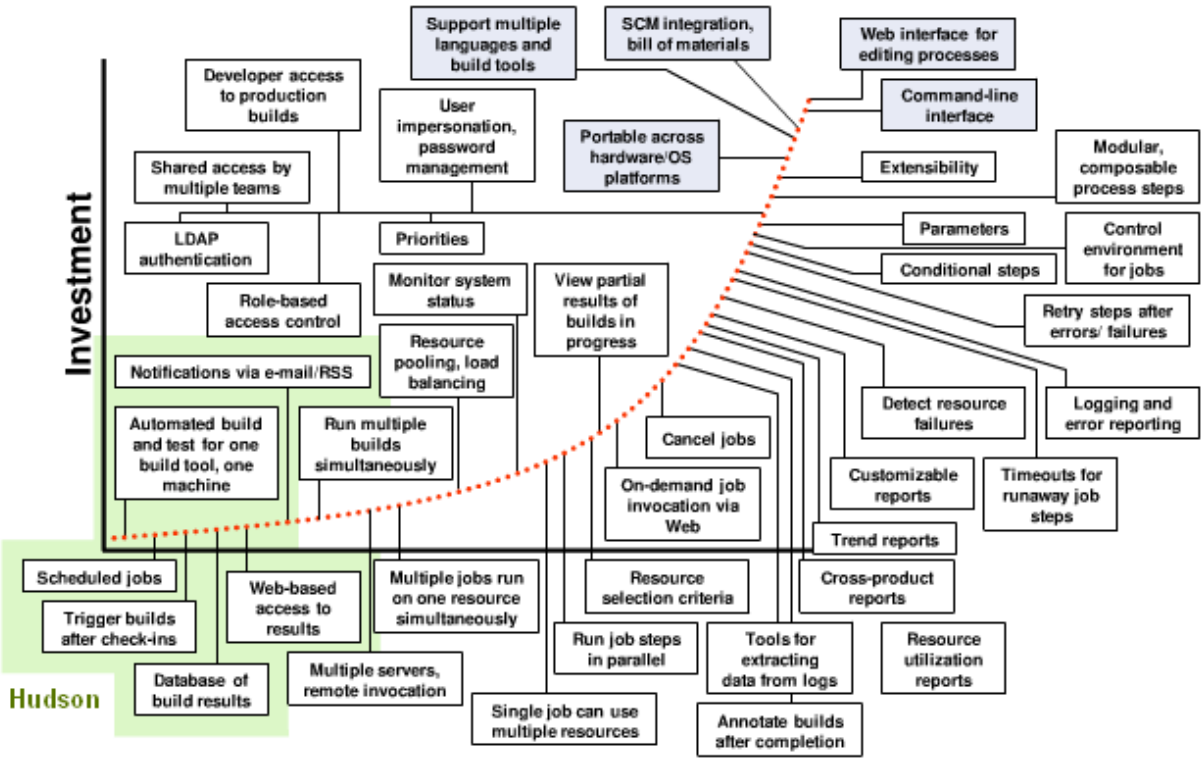
7. Ajout de la gestion des utilisateurs (authentification, rôle, accès…)



8. Ajout d'outils pour le développement…



9. "State-of-the-art" CI… (niveau optimisé)



Ce dernier schéma présente l'intégration continue en version "high-tech". A ce niveau, l'ensemble des processus d'intégration est formalisé, documenté et sécurisé. Toutes les fonctions sont mises en place et utilisées quotidiennement par l'équipe.

Arriver à un tel niveau d'optimisation se fait progressivement, par l'ajout constant de nouvelles briques logicielles, matérielles et humaines.

Il est important de remarquer que le logiciel du serveur d'intégration ne fait pas tout, il ne représente qu'une petite partie (en vert) du schéma.

9. CONCLUSION

9.1. Communication et Stratégie

Pour que le projet d'intégration continue réussisse, il faudra l'implication de tous.

L'intégration continue est un concept relativement nouveau qui peut bouleverser les habitudes du développeur. Une conduite au changement est indispensable. Il faut sensibiliser les membres de l'équipe. Il faut communiquer sur les intérêts pour que tout le monde sache que cela existe. Il faut rédiger des supports : guide d'utilisation, bonnes pratiques, aide à la mise en place, méthodologie… Ces documents doivent répondre aux questions Qui, Quoi ? Quand ? Effet ? Où ?... Il faut intégrer une culture d'entreprise de l'intégration continue, il faut accompagner les projets (prise en compte du contexte et de la maturité des équipes).

Ce dossier en fait partie, mais ce n'est qu'un point de départ. Néanmoins, j'espère qu'il portera ses fruits et qu'il vous donnera envie de vous documenter davantage sur le sujet.

Je ne le dirais jamais assez, l'intégration continue doit être progressive et itérative. Les résultats ne seront pas instantanés, mais apparaitront assez rapidement. Dans la plupart des cas, les développeurs ne prennent pas en compte les données issues de CheckStyle ou PMD, même s'ils sont déjà installés sur le projet ! Un bon usage préalable de ces outils est un pré requis avant de se lancer dans l'intégration continue.

Il ne suffit pas de disposer d'outils très performants pour savoir les utiliser à bon escient.

9.2. L'intégration continue comme un jeu

Pourquoi ne pas prendre l'intégration continue comme un jeu, une sorte de rituel divertissant. Par exemple, celui qui casse le build de nuit amène les croissants pour l'équipe… ou bien, mettre en place le jeu de l'intégration continue ("CI Game"). Sous forme de plug-in pour Hudson, les développeurs perdent ou gagnent des points en fonction des résultats des builds et de l'origine des erreurs. Tous les mois, on peut faire une sorte de classement des développeurs et, expliquer aux perdants comment améliorer leurs scores.

Pour en savoir plus : http://wiki.hudson-ci.org/display/HUDSON/The+Continuous+Integration+Game+plugin

9.3. Ce qu'il faut retenir !

L'intégration continue permet de développer mieux, plus rapidement, à un coût moins élevé, avec plus de confiance : "better, faster, cheaper". Ce n'est pas un outil, mais un processus qui utilise des outils et fournit une vraie valeur ajoutée à vos projets informatiques. La communication est un des piliers centraux pour la mise en place d'un tel projet.

Gardez en mémoire qu'il n'y a pas qu'une seule intégration continue. L'intégration continue est un processus sur mesure qui s'adapte à vos besoins. L'intégration continue mise en place dans l'entreprise A ne sera pas la même que celle mise en place dans l'entreprise B.

Better

On construit des builds de meilleure qualité.
On teste souvent et en amont ("early & often").
On respecte les bonnes pratiques et les conventions de codage.

Faster

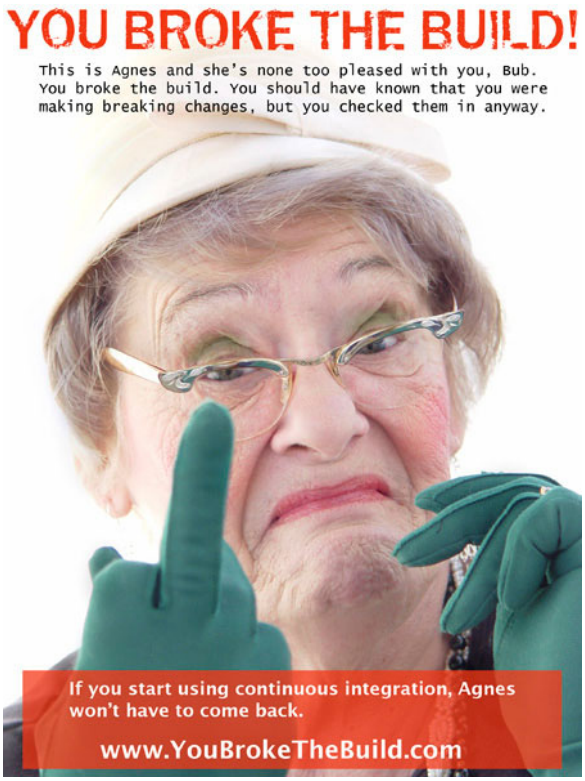
Les tests sont exécutés parallèlement au développement, pas à la fin.
Il n'y a plus de point d'intégration.
La construction d'un build ne devient plus un "évènement".

Cheaper

On identifie les défauts en amont.
On les résout quand il coûte le moins cher.
Le test devient facile et répétable.

Pour conclure, l'intégration continue est une porte ouverte à d'autres principes de l'*eXtreme Programming* et notamment sur le *Test-driven development* (commencer par écrire son test, ensuite, écrire le code suffisant pour passer le test et, enfin, refactoriser le code que l'on vient d'écrire). En fait, l'intégration continue est fortement liée à ce concept.

Voici une page bonus, je ne peux m'empêcher de terminer ce dossier par une pointe d'humour (noire) venant de nos amis américains, images trouvées sur le web. A noter que les sites web mentionnés sur les affiches existent vraiment !



© Michael Fransen