



Université de Fribourg, Suisse

Département d'informatique

Bachelor en informatique de gestion

INTÉGRATION CONTINUE

Concept théorique et développement d'une plate-forme d'intégration automatisée pour un projet informatique.

Travail de Bachelor

Auteur:

Chatton Marc

Rte des Acacias 9

1700 Fribourg

Responsable:

Dr Stefan Hüsemann

Fribourg, Septembre 2010

Table des matières

1	INTRODUCTION	1
1.1	Motivations	1
1.2	Objectifs	1
1.3	Structure du travail.....	3
1.4	Conventions	3
2	PRÉSENTATION GÉNÉRALE DU CONCEPT D'IC	5
2.1	Qu'est-ce que l'IC?	5
2.2	Historique.....	5
2.3	Avantages.....	6
2.4	Désavantages.....	8
2.5	Les difficultés de la mise en place	9
3	PRÉSENTATION DES DIFFÉRENTS ENVIRONNEMENTS DANS L'IC.	11
4	PRÉSENTATIONS DES 6 DISPOSITIFS QUI DÉFINISSENT L'IC.	13
4.1	Compilation du code-source	13
4.2	Intégration de la base de données	13
4.2.1	« <i>Local Sandbox</i> »	15
4.2.2	<i>Les bonnes pratiques</i>	15
4.3	Lancer les tests	16
4.3.1	<i>Tests unitaires</i>	17
4.3.2	<i>Tests d'intégration</i>	17
4.3.3	<i>Tests système</i>	17
4.3.4	<i>Tests fonctionnels (ou d'acceptation)</i>	18
4.3.5	« <i>Defect Driven Development</i> »	18

Table des matières	2
<hr/>	
4.4	Inspection..... 19
4.5	Déploiement..... 22
4.5.1	<i>Etiqueter les données du « repository »</i> 23
4.5.2	<i>Produire dans un environnement sain</i> 23
4.5.3	<i>Générer et étiqueter un build issu du « repository »</i> 23
4.5.4	<i>Lancer des tests de tous niveaux dans un clône de l'environnement de production</i> 24
4.5.5	<i>Créer des feedbacks pour le build</i> 24
4.5.6	<i>Possibilité de revenir à une version précédente</i> 24
4.6	Feedback 24
4.6.1	<i>Les bonnes informations</i> 25
4.6.2	<i>Les bonnes personnes</i> 25
4.6.3	<i>Le bon moment</i> 26
4.6.4	<i>La bonne manière</i> 26
5	OUTILS ET AUTOMATISMES À ADOPTER POUR UNE IC IDÉALE 27
5.1	Réflexes à avoir pour le repository 27
5.1.1	<i>Utiliser le repository pour stocker tous les fichiers</i> 27
5.1.2	<i>La « Staging Area»</i> 27
5.2	Réflexes à avoir pour le build 28
5.2.1	<i>Types de build</i> 29
5.2.2	<i>Les façons de le lancer</i> 30
5.2.3	<i>Automatiser le build</i> 30
5.2.4	<i>L'exécuter via une seule commande</i> 30
5.2.5	<i>Créer de plus petites équipes en séparant les étapes de développement</i> 31
5.2.6	<i>Attribuer un poste de « Chef de build »</i> 31
5.2.7	<i>Séparer le build de l'environnement de développement IDE</i> 31

5.2.8	<i>Faire en sorte que les builds échouent rapidement</i>	31
5.2.9	<i>Faire qu'un build soit compatible peu importe l'environnement</i>	32
5.2.10	<i>Utiliser une machine dédiée pour le build d'intégration</i>	32
5.2.11	<i>Lancer des builds rapides</i>	33
5.2.12	<i>Refactorer les builds pour optimiser le processus</i>	35
6	LES OUTILS DE DÉVELOPPEMENT	37
6.1	Le serveur d'IC	37
6.1.1	<i>AnthillPro</i>	37
6.1.2	<i>CruiseControl .NET</i>	38
6.2	Les Builds scripts	38
6.2.1	<i>(N)Ant</i>	38
6.2.2	<i>Maven</i>	39
6.3	Le système de contrôle de versionnement	39
6.3.1	<i>Subversion</i>	39
6.3.2	<i>TortoiseSVN</i>	40
6.4	Les Feedbacks	40
6.4.1	<i>Doxygen</i>	41
7	L'AVENIR DE L'IC ET SES ALTERNATIVES	42
7.1	Avenir de l'IC	42
7.2	Les alternatives	45
8	ENVIRONNEMENTS DE DÉVELOPPEMENT DE LA BSU	47
8.1	Situation sans IC	47
8.2	Situation après mise en place de l'IC	48
8.3	Analyse SWOT	50
9	CONCLUSION	56

Table des matières	4
<hr/>	
10 GLOSSAIRE ET TABLE D'ABRÉVIATION.....	58
10.1 Glossaire	58
10.2 Table d'abréviation	60
11 BIBLIOGRAPHIE	61

Graphiques

FIGURE 3-1 : LES COMPOSANTS D'UN SYSTÈME D'IC, [DUVALL 2007, P. 5].....	11
FIGURE 4-1: EXEMPLE DE L'UTILISATION D'UN «LOCAL SANDBOX» POUR CHAQUE DÉVELOPPEUR, [DUVALL 2007, P. 118].....	15
FIGURE 4-2: LE FEEDBACK COMME PAUL M. DUVALL LE CONÇOIT, [DUVALL 2007, P. 206]	25
FIGURE 5-1 : LA STAGING AREA [CROSSROADS 2004].....	28
FIGURE 5-2 : LE BUILD SCRIPT CONFIGURÉ POUR S'ADAPTER À CHAQUE ENVIRONNEMENT, [DUVALL 2007, P. 79]	32
FIGURE 5-3 : TABLEAU INDIQUANT LES AMÉLIORATIONS POSSIBLES ET SES CONSÉQUENCES (L'ÉVOLUTIVITÉ, LA PERFORMANCE ET LES DIFFICULTÉS QUI EN RÉSULTENT), REPRISE DU GRAPHIQUE DE [DUVALL 2007, P. 90].....	35
FIGURE 7-1 : « CODAGE-VÉRIFICATION-COMMIT » SELON TEAMCITY [TEAMCITY 2009].....	42
FIGURE 7-2 : COMMIT À 2 PHASES, [DUVALL 2007, P. 224].....	43
FIGURE 8-1 : SITUATION DE L'ENVIRONNEMENT DE DÉVELOPPEMENT DE LA BSU AVANT LA MISE EN PLACE D'UN SYSTÈME D'IC.....	48
FIGURE 8-2 : MISE EN PLACE DE L'IC POUR LA BSU	49
FIGURE 8-3 : CHEMINEMENT DE L'ANALYSE SWOT [GROUPERESSOURCES.COM 2010].....	50

1 Introduction

1.1 *Motivations*

J'avais déjà quelques idées d'orientation pour le travail qui m'attendait : travailler avec des bases de données, aborder des technologies telles que celles utilisées par Bluewin TV (Teleclub) pour la retransmission des événements sportifs (« achat » du match par le téléspectateur depuis son canapé) et d'en faire une étude de marché, etc. Cependant, c'est suite aux discussions que j'ai eues avec le Dr. Stefan Hüsemann, que j'ai rencontré via l'association d'étudiants « BSU » (Börsenspiel der Schweizer Universitäten), et dont il est le Président du « Vorstand » (sorte de « conseil d'administration » des anciens Présidents du jeu) que j'ai opté pour une autre voie. Il m'a proposé de travailler sur l'environnement de développement du logiciel qui permet de générer ce jeu boursier. C'est d'ailleurs là que j'ai pris pour la première fois la connaissance du terme « Continuous Integration » (Intégration continue). Après quelques recherches sur le sujet, j'ai pu remarquer que le terme était relativement récent et que d'ailleurs, quasiment aucune information n'était disponible en français. Au fur et à mesure de mes recherches, j'ai aussi pu constater que même si le terme n'était que peu utilisé, il n'en restait pas moins indispensable pour chaque entreprise de développement informatique de mettre en place un environnement de développement à la hauteur de leurs ambitions. Ce paradoxe (peu d'informations et application du concept quasi-systématique pour les développeurs) m'a intrigué et poussé à approfondir le sujet, et même de l'appliquer à une dimension raisonnable pour de petits projets informatique développé par la «BSU».

1.2 *Objectifs*

L'objectif principal est de présenter les tenants et aboutissants du concept d' « intégration continue » dans un cadre théorique dans un premier temps, puis de l'appliquer avec les outils à disposition pour un projet informatique de petite envergure. Découvrir de nouveaux langages de programmation ainsi que la création d'un environnement de développement viable pour un projet informatique, faisaient de ce travail un défi très intéressant.

En démarrant ce travail, j'ai essayé de me poser les questions les plus pertinentes et ciblées possibles sur le sujet. C'est d'ailleurs en me basant sur celles-ci que j'ai

structuré mes chapitres dans l'objectif de pouvoir répondre à toutes ces questions dans un ensemble concis et cohérent.

Partie théorique:

- *Qu'est-ce que l'intégration continue?*

Je présenterai le concept d'IC qui sera exposé dans sa globalité avant d'être analysé en détail.

- *Quelles sont les composants principaux pour la mise en place?*

L'environnement de développement représentatif sera présenté à l'aide d'un graphique et développé point par point.

- *Quelles étapes pour y parvenir?*

J'ai relevé 6 étapes qui constituent l'IC, à savoir : la compilation, l'intégration de la base de données, les tests, l'inspection, le déploiement et le processus de « feedback ». Une partie sera entièrement consacrée à la présentation de ces diverses étapes.

- *Quelles sont les avantages/désavantages auxquels on peut s'attendre?*

Il sera expliqué tous les avantages que l'IC peut apporter au développement d'un projet, ainsi que les inconvénients inhérents à sa mise en place.

- *Quels outils pour l'IC?*

Une présentation non-exhaustive de quelques outils pour le versionnement (repository), pour le serveur d'intégration, pour les tests, pour l'inspection et pour les « feedbacks ».

- *Que peut-on espérer de plus pour l'IC ?*

Le futur de l'IC sera présenté, avec les optimisations qui en découlent.

Partie pratique:

- *Quelles sont les apports envisagés pour la BSU ?*

Il sera exposé la situation avant et après l'installation de cette plateforme d'IC pour cette association. Une analyse SWOT permettra de mieux cerner les avantages et désavantages qui en découlent.

- *Comment l'IC peut être mise en place pour un petit projet informatique utilisant des technologies .NET?*

Cette question représentera l'essentiel du travail pratique, on y répondra via la réalisation de l'infrastructure de l'IC pour un projet informatique réalisé pour la BSU.. Cette partie sera présente dans une annexe de ce travail et sera présentée comme un guide pour la mise en place d'une IC.

1.3 Structure du travail

Ce travail se compose principalement en 2 parties : une première théorique puis une seconde pratique.

On se concentrera, dans la partie théorique, sur l'approfondissement des outils qui permettent l'application de l'IC à proprement parler. Ceci servira de passerelle avant l'application concrète du concept dans la partie pratique.

Cette seconde partie abordera le cas de la BSU, où une plateforme d'IC a été développée sur le serveur d'intégration de l'association. Un guide d'installation et d'utilisation sera également disponible dans une annexe, ce avec certains codes de la partie pratique (il s'agit-là des fichiers de configurations des différents programmes utilisés, l'essentiel du travail étant dans la configuration du serveur). Il ne contient que très peu d'informations vu que, une fois implémenté, le serveur d'IC n'exige que peu de manipulations pour qu'il soit fonctionnel.

Les derniers points se résument à un support où vous seront expliqués les différents termes abordés ainsi que la bibliographie qui a été utilisée tout au long du travail.

1.4 Conventions

- On donnera la signification d'un acronyme entre parenthèses, lors de sa première utilisation.
- Inversement, lorsqu'on créera un acronyme pour alléger le texte, il sera donné en relation avec sa signification.
- Les termes anglophones qui n'ont pas d'équivalent en français seront mis entre crochets «...».

-
- Les notions abordées qui seront (ou étaient) développées dans un chapitre ultérieur (resp. précédent) seront accompagnés d'un lien en italique vers le chapitre en question (par exemple (*cf. Chapitre 5.2.12 pour les détails*)).
 - L'URL (Uniform Resource Locator) complet d'un site Web sera référencé dans la partie « Bibliographie » en fin de document, avec la date de la dernière visite.
 - Le numéro des figures utilisées dans le document correspondent à leur ordre d'apparition dans un chapitre.
 - Une citation d'un auteur se fera en *italique* et comportera une référence à son auteur (par exemple [*Martin Fowler 2006*]).
 - Une idée (ou concept) évoquée par un auteur et reprise dans ce travail sera référencée par un « cf. » (par exemple *cf. [Duvall 2007, p. 111]*).
 - Les néologismes (par exemple des termes anglais francisés) seront également placés entre crochets «...».
 - Les parties de code seront mises en *italique*.

2 Présentation générale du concept d'IC

Avant de se lancer dans une explication plus technique ou d'approfondir l'architecture nécessaire à sa mise place, il est important de définir la notion d'IC, de comprendre pourquoi les programmeurs ont commencé à s'en servir, ainsi que les avantages et désavantages qui en découlent. Ce chapitre nous offre donc un premier aperçu du sujet.

2.1 Qu'est-ce que l'IC?

« Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. » [Fowler 2008]

L'intégration continue est une pratique de développement informatique qui consiste à devoir « intégrer » (donc entre autres : compiler, tester et déployer) les changements appliqués sur les lignes de code du programme avant de commencer une nouvelle tâche. Le but est de proposer un « feedback » rapide si un défaut a été détecté dans le code-source, le problème pouvant être identifié et corrigé dans les plus brefs délais. Il en résulte un programme plus stable et développé plus rapidement. L'expression « intégration continue » est plus une accroche syntaxique qu'une description formelle. On utilise ce terme pour mettre en exergue le fait qu'un build journalier ou hebdomadaire n'est simplement pas suffisant.

Le terme « continue » (« continuous » en anglais) est par ailleurs techniquement incorrect puisque cela implique qu'il ne s'arrête jamais dans le temps, que le système intègre de façon incessante, ce qui est physiquement impossible. Il aurait été plus juste d'utiliser le mot « continue » (« continual ») qui aurait le sens de quelque chose qui se répète à intervalles très rapprochés.

2.2 Historique

Avant, on pouvait parler de « Big Bang Integration », avec ce concept (qu'on peut comparer à l'approche « Run it and see ») les différents modules n'étaient pas intégrés avant qu'ils ne soient tous opérationnels. On intégrait ensuite le programme sans véritables tests préalables et on le lançait pour s'assurer de son bon fonctionnement. Les défauts de cette méthode prouvaient que cette voie n'était pas satisfaisante: haute

probabilité de découvrir des défauts critiques dans la phase de production, difficultés à isoler le défaut (Provient-il de l'interface? Du composant?), etc.

Kirk Knoernschild, auteur d'un article sur l'IC, présentait le « Big Bang Integration » comme une méthode qui « *retarde la découverte des erreurs et la réduction du risque ce qui peut avoir des résultats désastreux sur l'effort de développement, incluant un appauvrissement de la qualité du logiciel, un retard de livraison, et des solutions qui ne satisfont pas les besoins du client.* » [Knoernschild 2008]

L'intégration continue est une réponse logique aux problèmes précités, et a beaucoup évolué depuis sa conception. Au départ, on parlait de « Daily Build » qui était devenu un standard. Maintenant, on part du principe qu'un « build » est effectué chaque jour et/ou chaque fois qu'un changement significatif est apporté au projet. [Extreme Programming 2009]

Le terme « Continuous Integration » est apparu la première fois dans une des douze pratiques du processus de développement de « Extreme Programming ». Les principes peuvent être cependant appliqués à chaque modèle de programmation itérative (comme Agile par exemple). Kent Beck, le créateur d'XP (Extreme Programming) ainsi que Martin Fowler, spécialiste reconnu des méthodologies de développement, ont été les premiers à écrire des articles traitant du sujet à la fin des années 1990. Le processus d'intégration n'est pas un problème récent, mais comme la complexité des projets augmente avec le temps, le besoin d'intégrer régulièrement et de s'assurer que les modules fonctionnent ensemble est devenu prioritaire. La conséquence de ce besoin fut le développement de pratiques pour s'assurer une intégration régulière et fiable, et donc la naissance du concept « d'intégration continue ».

2.3 Avantages

On peut citer de nombreux avantages découlant d'un usage efficient de l'IC:

- L'argument principal reste le fait que l'IC réduit les risques → les tests et les inspections permettent de détecter automatiquement les défauts dès qu'ils sont « engagé » dans le « repository », et donc de les corriger directement. Le fait

d'instaurer une routine de tests, dans un environnement stable, permet aussi de réduire les doutes quant à la fiabilité du système.

- Une meilleure visibilité dans le projet → l'IC a pour vocation première une participation accrue des différents acteurs : mettre en place une infrastructure pour le partage d'informations, centraliser les données et les labéliser dans le « repository » afin d'en avoir une meilleure vision d'ensemble, privilégier le code commenté pour que d'autres développeurs puissent y accéder plus facilement, recevoir les informations liées au résultat du « build », etc. Toute cette mise en place permet de prendre des décisions en connaissance de cause étant donné que toutes les informations pertinentes sont disponibles.
- Cette visibilité permet en outre d'acquérir une certaine confiance des développeurs, ceux-ci se sentent rassurés puisqu'ils connaissent l'impact que leur code, même corrompu, peut avoir sur le système. Ils savent que le corriger devient plus aisé, qu'il est même possible de revenir à une version antérieure. Ils prennent donc plus souvent l'initiative de faire un changement. Il faut aussi savoir que de nombreux outils, souvent associés au serveur d'intégration, sont disponibles afin d'avoir une vue globale du code-source, et de pouvoir y associer une meilleure analyse (par exemple un éditeur UML qui génère des diagrammes à partir du code-source dans Javadocs, un générateur de documentation technique (Doxygen), un outil d'analyse statique des données métriques (Panopticode), etc.).
- Réduire les processus répétitifs → le propre de l'IC est d'instaurer un processus ordonné, en automatisant des étapes on évite un surplus de travail au développeur.
- Générer un logiciel « déployable » → l'avantage indéniable de l'IC est le fait de pouvoir générer un exécutable du logiciel n'importe quand, dès qu'un changement est effectué sur le « repository » une nouvelle version du logiciel est créée. Cela permet donc de détecter les erreurs au plus vite. Cette méthode influe aussi sur la confiance des développeurs: il arrive parfois qu'ils avancent dans leur travail sans même savoir si le « build » va se faire, on pourrait presque parler dans ce cas de procrastination des programmeurs qui repoussent sans cesse l'échéance du build. C'est pourquoi on parle souvent d' « integration hell » (l'enfer de l'intégration) [Duvall 2007] puisque la phase d'intégration a toujours été la cause d'importants retards, rarement comptabilisée à l'échéance et ceci peut être évité avec l'IC. Ces

exécutables sont également essentiels pour le client qui peut constater au jour le jour les avancées de son programme et ainsi redéfinir au mieux ses besoins.

- Programme de meilleure qualité → la qualité est ce qui est recherchée au travers de l'IC. On peut définir la qualité comme la pondération des spécifications d'un logiciel suivantes:

Extensibilité, sécurité, performance, capacité à être maintenu, lisibilité, etc. cf. [Duvall 2007, p.28]

- Plusieurs caractéristiques de l'IC contribuent à obtenir un code fiable. On peut par exemple détecter une duplication du code, appliquer un standard ou une convention dans le développement du code, etc. De nombreux points seront développés plus loin, dans les dispositifs à mettre en place pour l'élaboration de l'IC.

2.4 **Désavantages**

(Ou les idées reçues qui incitent les équipes de développement à opter pour le statu quo)

- Le changement → bien entendu le changement n'est jamais aisé à gérer pour un développeur qui a déjà ses habitudes. L'idée serait de faire une transition pas-à-pas (par exemple diminuer le nombre de tests, intégrer à des fréquences moins importantes ou revenir au standard qu'était le daily build le temps de l'adaptation). Travailler avec quelqu'un qui a de l'expérience dans l'IC permet aussi de passer le cap avec plus de facilité.
- Coûts supplémentaires → le seul investissement serait une machine pour l'intégration, de nombreux logiciels pour la mise en place du dispositif étant disponibles gratuitement sur internet. Il va de soi que l'investissement en vaut la chandelle comparé à tous les coûts qu'implique la découverte tardive d'un défaut (bug). De plus, il n'est pas non plus nécessaire d'acheter un ordinateur dernière génération pour un projet de petite/moyenne ampleur. On perçoit ici l'importance d'une vision à long terme pour une entreprise : en investissant pour une machine, la comptabilité de l'entreprise ne percevra pas immédiatement un retour sur investissement, d'autant qu'une période d'adaptation à cette méthodologie est indispensable pour en tirer les premiers bénéfices (on en revient à la problématique du changement citée auparavant).

- Le nombre de « builds » qui échouent ou leur lenteur d'exécution → s'ils échouent, c'est souvent les tests qui en sont la cause. La solution serait d'effectuer un « build privé » avant de « commiter » dans le « repository », les tests seraient donc effectués en interne. Ce désavantage n'en est évidemment pas un, vu que le « build privé » fait partie des bonnes pratiques de l'IC. De plus, il est plus facile de corriger un « build » effectué régulièrement, qui ne contiendrait que peu d'erreurs (vu que les changements ne sont pas importants) qu'un « build » hebdomadaire dont les changements ne sont pas connus de tous les développeurs. Pour ce qui est de la lenteur d'exécution, on peut facilement parer à ce problème en partitionnant les builds (*cf. Chapitre 5.2.12 pour les détails*) et en les invoquant selon le besoin.
- Migration → le passage d'un large volume de projets de développement dans un environnement d'IC (projets qui sont souvent répartis dans de nombreux environnements de développement) requiert un planning et une coordination efficace pour que la migration soit réussie.

2.5 Les difficultés de la mise en place

L'intégration continue est une solution idéale pour les petites équipes de développement. Comme nous allons le voir, il aide à maintenir tous les acteurs informés des derniers changements. Il permet aussi à toutes les équipes gravitant autour du projet d'obtenir l'information de la version qu'elles recherchent. Cependant, de nombreux facteurs pourraient amener à des difficultés quant à la mise en place d'un tel système:

Sur un plan matériel:

- Les capacités de stockage ou de bande passante
- La puissance du processeur du serveur d'intégration
- La taille et la performance du repository
- Les dépendances entre les différents fichiers lors du build
- Etc.

Sur un plan organisationnel:

- Les relations entre les sous-équipes et le rythme où chaque intégration est lancée

-
- Les équipes dispersées géographiquement (horaires différents)
 - Etc.

Les projets de plus grande envergure amènent souvent à réorganiser les équipes afin de leur attribuer des « sous-projets ». Cette particularité change la façon dont on structurera l'environnement de développement. Chaque équipe, chargée d'un sous-projet, sera responsable de délivrer un code ou des exécutables fonctionnels afin que toutes les autres équipes puissent, au besoin, les réutiliser. Chacun sera cependant responsable de son propre code et ne modifiera pas celui des autres.

3 Présentation des différents environnements dans l'IC.

Il est important, dans un premier temps de visualiser et comprendre les étapes qu'implique la programmation d'un logiciel dans un environnement d'IC typique. La Figure 3-1 en donne un parfait aperçu. Elle s'articule autour de 5 étapes décrites ci-dessous.

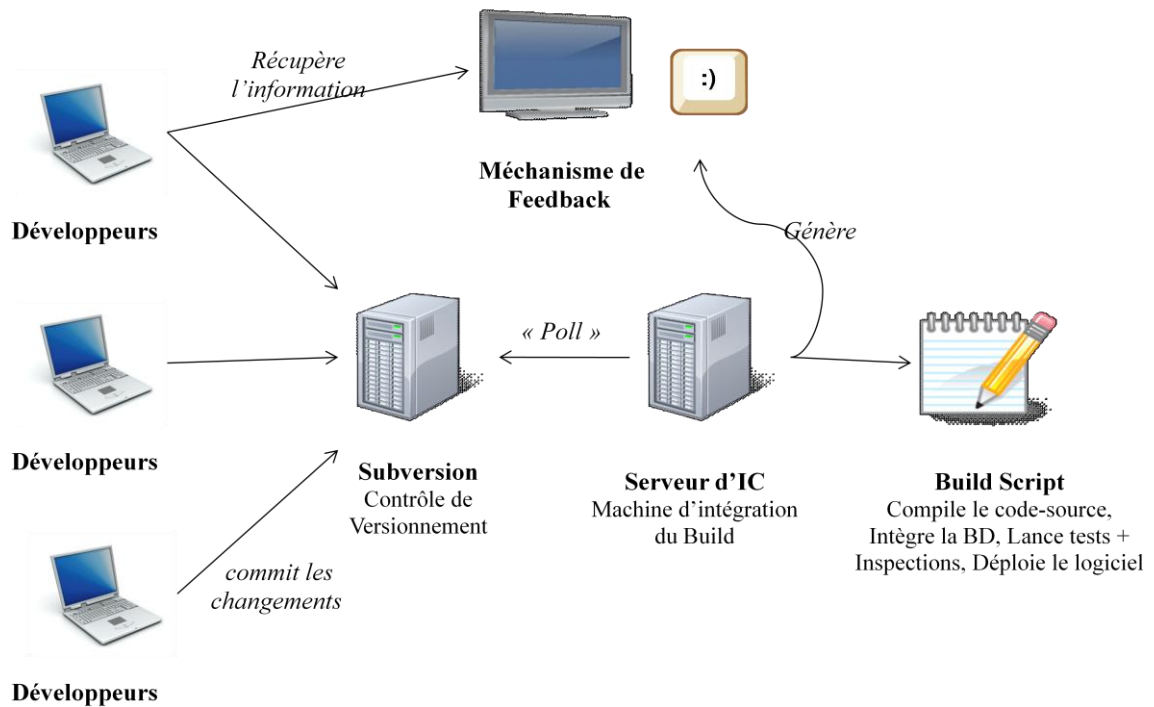


Figure 3-1 : Les composants d'un système d'IC, [Duvall 2007, p. 5]

Les termes utilisés sont pour la plupart conservés en anglais, puisqu'ils sont entrés dans le jargon d'informatique et que souvent, aucun terme français ne leur correspond.

1. Chaque développeur travaille sur son propre poste. Il devra dans un premier temps effectuer un « check out » du contrôle de versionnement, ce qui signifie qu'il importerá sur sa machine la dernière version du programme qui réside dans le « repository ».

2. Une fois qu'il a effectué un changement significatif sur le code-source, il effectuera le chemin inverse: c'est-à-dire qu'il exportera son travail dans le « repository » pour le mettre à disposition des autres développeurs. On dit qu'il

« commit » son code. Habituellement, le développeur passe par une étape intermédiaire qui consiste à lancer un « private build » avant son « commit », c'est-à-dire qu'il contrôlera sur son poste que les changements qu'il a effectué permettront de conserver une version saine du programme avant de mettre son travail en commun.

3. Le Serveur IC est la machine qui fera le « build », il s'agit en fait de compiler, tester, inspecter et déployer le code-source, donc de vérifier que le logiciel fonctionne de manière fiable. Le rôle de ce serveur sera de contrôler (« poll », scruter en français) régulièrement s'il y a une nouvelle version du code-source présente dans le « repository » auquel cas il exécutera le « Build Script » qui consiste à lancer les différentes étapes du « build ». Ce serveur n'est pas indispensable au bon fonctionnement de l'IC, puisqu'il est possible d'écrire manuellement tous les « Build Script » et de forcer le « build » d'intégration quand un changement a lieu dans le « repository ». Il est cependant plus aisé de profiter des avantages que le serveur met à disposition, quitte à devoir adapter certains scripts manuellement, d'autant plus que de nombreux serveurs d'intégration sont disponibles, certains même gratuitement (comme CruiseControl.NET qui sera utilisé pour la partie pratique).

4. Le « Build Script » est un fichier de configuration utilisé pour compiler, intégrer la base de données, tester, lancer l'inspection et déployer le programme. On peut automatiser le cycle du « build » du programme sans forcément que ce soit dans le cadre d'une implantation de IC : Ant, Nant, etc. en sont un parfait exemple.

5. Une fois le « build » effectué, le serveur d'IC génère des feedbacks pour informer les développeurs du résultat. Il peut s'agir d'un courriel, d'un sms, d'un son, etc. avertissant les utilisateurs concernés de la réussite ou de l'échec du « build », de nombreuses autres informations peuvent également être envoyées. Ce point sera développé plus en profondeur dans la partie traitant des dispositifs (*cf. Chapitre 4.6 pour les détails*).

4 Présentations des 6 dispositifs qui définissent l'IC.

Après les différents environnements, nous allons maintenant nous attarder sur ce qu'implique le fait d'utiliser un système d'IC : il s'agit en fait de 5 étapes successives par lesquelles le code-source va passer, puis du feed-back qui informera le développeur de l'issue du build.

4.1 *Compilation du code-source*

Selon le Larousse, la définition de « compilation » est « *la traduction d'un programme écrit en langage évolué en un programme équivalent en langage machine* » (c'est-à-dire en un code binaire). [Larousse, 2010]

L'objectif est donc qu'il soit lisible et autonome (donc exécutable) pour le processeur. Une fois compilé, le programme sera plus rapide à l'exécution. Il faudra néanmoins le recompiler à nouveau si on modifie le fichier source afin que toutes les modifications puissent être effectives.

Pour les langages dynamiques que sont PHP ou Ruby, qui sont des langages dits interprétés, on ne peut pas véritablement parler de compilation puisqu'ils ne génèrent aucun code binaire. Ils sont traduits au fur et à mesure par un « interpréteur » qui est simplement un programme auxiliaire. [Duvall 2007, p.12]

Idem pour les langages dits intermédiaires qui peuvent subir une phase de compilation intermédiaire vers un fichier écrit dans un langage qui n'est pas intelligible et non-exécutable (par exemple un applet Java qui nécessite un navigateur internet pour être exécuté, alors même qu'il est compilé).

Dans le cadre de l'IC, on ne fera pas de différenciation entre ces langages. Même si la compilation n'est pas effective. Cette étape permet pour certains langages dynamiques d'exécuter un processus de contrôle qui peut être assimilé à une compilation dans ce contexte particulier.

4.2 *Intégration de la base de données*

Dans cette étape, on parlera de « Continuous Database Integration » (CDBI), qui est simplement l'application de l'IC au niveau de la base de données (BD). En d'autres termes, le processus d'intégration reste le même que pour le code-source: on appliquera

les changements à la BD dans le « repository » pour qu'un nouveau build puisse se faire, on testera (les procédures, les fonctions, les interactions des contraintes, etc.), on inspectera (l'intégrité des données, la performance, les conventions/standards de nommage, etc...), on déploiera (la BD aux instances de développement et de tests de la BD) et on recevra un feedback en retour.

Le problème ici reste bien entendu que le code-source et la BD ont souvent des dépendances qui compliquent le processus d'automatisation.

Le but de l'automatisation est de soulager les membres de l'équipe de développement, leur éviter des tâches répétitives qui représentent, à terme, une perte de temps évitable. Paul M. Duvall a énuméré toutes les activités répétitives que l'on pourrait associer au « CDBI », celles-ci représentent quasiment toutes les commandes de bases liées à la manipulation des données dans une BD:

supprimer/créer une BD, supprimer/insérer une donnée dans le système, migrer une BD/donnée, modifier une donnée/procédure/permission d'accès, etc... *cf. [Duvall 2007, p. 111]*

Ces étapes permettent surtout de décharger une personne en particulier: le DBA (pour DataBase Administrator, ou Administrateur de BD). Donner la possibilité aux développeurs de modifier eux-mêmes certaines tâches basiques lui permet de se concentrer davantage sur la normalisation des données, le développement de standard ou l'amélioration des performances. Cela permet en outre d'intégrer au mieux le DBA comme membre de l'équipe de développement et cela évite aussi au développeur de perdre du temps à contacter un intermédiaire pour un changement qu'il est capable de faire lui-même. Ce changement implique certaines conditions néanmoins:

- Que le programmeur travaille dans un environnement privé, et qu'il ne « commit » qu'une fois que tous les tests passent avec succès. Il prend donc la responsabilité de délivrer au repository une BD stable.
- Que le DBA supervise les changements une fois qu'ils ont eu lieu, et qu'il détecte les causes d'une éventuelle erreur.

4.2.1 « Local Sandbox »

Comme il a été dit plus haut, le danger de l'intégration de la BD, c'est les dépendances système. Un changement effectué dans la structure d'une BD partagée peut amener les « private build » des autres développeurs à échouer. Cette problématique a incité les chefs de projet à revoir l'environnement de développement. Il s'agit de concevoir la BD comme le code-source: lors du « check out » du repository, on inclut la BD afin de travailler en « vase clos » le temps que les tests confirment que les changements n'affecteront pas le build des autres développeurs. Le DB Sandbox consiste donc en une BD locale qui est une copie conforme de la BD originelle. cf. [Duvall 2007, p. 117]

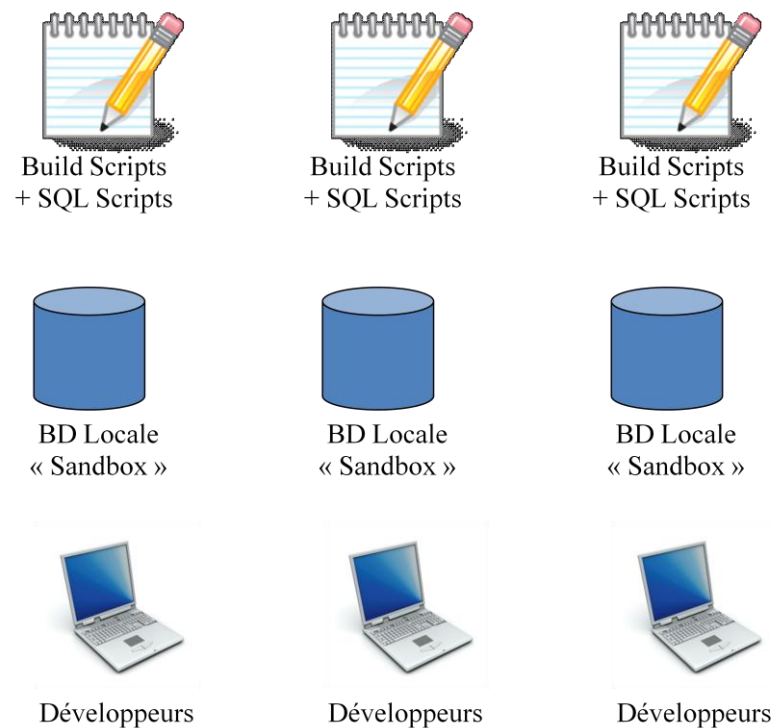


Figure 4-1: Exemple de l'utilisation d'un «local sandbox» pour chaque développeur, [Duvall 2007, p. 118]

4.2.2 Les bonnes pratiques

1. Une des pratiques à adopter pour le CDBI c'est l'envoi dans le repository de tous les fichiers liés à la BD:

les configurations spécifiques de la BD, les procédures et les fonctions, les scripts « Data Definition Language » (DDL), les scripts « Data Manipulation Language » (DML), les diagrammes entité-association, ou les tests.

L'objectif est de pouvoir reconstruire la BD à partir de zéro, grâce aux fichiers laissés sur le repository. *cf. [Duvall 2007, p. 120]*

2. Une autre pratique serait de faire une mise à jour automatique des BD « Sandbox » des développeurs. Dès qu'un changement est effectué dans la BD principale, un « check out » s'effectue chez chacun afin de rester synchroniser avec le repository. Ceci implique forcément de rester connecté au réseau, ou du moins de faire une requête de mise à jour manuelle régulièrement. *cf. [Duvall 2007, p. 110]*

3. Refactoriser la BD: on peut le définir comme la maintenance du code de la BD, c'est-à-dire supprimer les redondances, simplifier le code, limiter la complexité, etc. C'est une technique bien connue mais qui implique beaucoup plus d'attention en ce qui concerne les BD, puisque la refactorisation implique 3 changements simultanés: *cf. [Duvall 2007, p. 120]*

- Changer le schéma de la BD
- Migrer les données dans la BD
- Changer les codes d'accès de la BD

4. Intégrer la BD « Sandbox » régulièrement: de nombreuses pratiques peuvent être reprises de l'IC appliquée au code-source. On peut donc se référer au *Chapitre 5* pour plus d'informations.

4.3 Lancer les tests

Dans cette étape, le but sera d'incorporer des tests automatiques dans le processus du build. L'échec d'un seul des tests provoquera l'échec complet du build. Le but ici sera de rendre un programme fiable à 100%. Comme le précise Paul M. Duvall, la fiabilité théorique totale d'un système se calcule par le produit de la fiabilité de tous ses composants. Par exemple, si la valeur de fiabilité de 3 composants-système est évaluée à 90%, le système entier ne sera pas fiable à 90%, mais bien à 73% ($0.9 * 0.9 * 0.9$). Les routines de tests sont là justement pour rendre un niveau de fiabilité optimal: plus le code-source est encadré de tests pertinents, plus il sera considéré comme « fiable ». *cf. [Duvall 2007, p. 131]*

« Imperfect tests, run frequently, are much better than perfect tests that are never written at all. » [Fowler 2008]

On peut catégoriser les tests en 4 groupes différents selon les phases de développement dans lesquels on se situe:

4.3.1 Tests unitaires

Par unitaire on suppose qu'on a affaire à une portion de programme. On teste un module pour s'assurer qu'il fonctionne en toutes circonstances par rapport aux spécificités exigées. Ces modules peuvent avoir des dépendances les uns avec les autres. Dans ce cas, on supposera que ces dépendances sont minimales et n'impliquent en aucune manière des objets graphiques ou des bases de données. La caractéristique principale des tests unitaires est leur temps d'exécution rapide.

4.3.2 Tests d'intégration

Le but ici est de vérifier l'interaction entre les différents composants que chaque développeur a conçus pour produire un ensemble dont le comportement correspond aux modifications effectuées. Les différences principales avec les tests unitaires résident dans le fait que la base de données sous-jacente, les dépendances système ou le réseau seront nécessaires. Ils ne sont cependant pas considérés comme des tests de haut niveau.

***Tests d'intégration** : tests effectués pour montrer des défauts dans les interfaces et interactions de composants ou systèmes intégrés. [CFTL 2009]*

4.3.3 Tests système

Ces tests vérifient que les interfaces externes comme les services Web fonctionnent comme prévu. Les dépendances externes rendent ce genre de test beaucoup plus coûteux au niveau du temps d'attente, on les effectue d'ailleurs principalement de nuit ou durant une période de pause.

Quand les tests d'intégration sont développés dans le but de tester les interfaces entre les composants, les tests système évoluent dans un environnement contrôlé et configuré, les différents tests sont donc développés pour simuler des scénarios de la « vraie vie » qui sont effectués dans un environnement de test ayant justement pour charge de simuler cette « vraie vie ».

Une autre différence réside dans le fait que l'objectif des tests d'intégration est de s'assurer que les différents composants de l'application répondent aux exigences du

client, tandis que celui des tests système, c'est de valider l'exactitude et la perfection de l'application dans l'accomplissement des fonctions prévues et dans le test de toutes les fonctions requises dans la « vraie vie ». cf. [Duvall 2007, p. 136]

Tests système : le processus de test d'un système intégré pour vérifier qu'il réponde à des exigences spécifiques [CFTL 2009]

4.3.4 Tests fonctionnels (ou d'acceptation)

On cherchera ici à tester le système dans les conditions définies par le client. C'est donc à ce stade qu'on testera si l'exécutable répond aux définitions du besoin que l'acheteur a émis. On tâchera donc de mettre les incohérences ou les omissions en évidence avant la mise en œuvre opérationnelle du système.

Test d'acceptation : test formel en rapport avec les besoins, exigences et processus métier, conduit pour déterminer si un système satisfait ou non aux critères d'acceptation et permettre aux utilisateurs, clients ou autres entités autorisées de déterminer l'acceptation ou non du système [CFTL 2009]

L'objectif de la catégorisation des tests de développement dans le cadre de l'IC est simplement de permettre un résultat plus rapide: plus il y a de dépendances entre les composants (bases de donnée, Web ou fichiers système par exemple), plus les tests prendront du temps à être effectués. A partir de là, on lancera des tests unitaires beaucoup plus souvent que les tests d'intégration ou que les tests système (qu'on effectuera lors de « build secondaire » (cf. Chapitre 5.2))

4.3.5 « Defect Driven Development »

Une des bonnes pratiques de l'IC est de faire échouer le build le plus rapidement possible (afin de détecter au plus vite les erreurs). Le Defect Driven Development peut paraître tout aussi paradoxal: pourquoi commencer à écrire des tests destinés à échouer? Paul M. Duvall écrit, dans le livre déjà cité: « *Defects don't drive development--preventing those nasty aberration drives development!* » [Duvall 2007, p. 144]. Donc pour progresser dans le développement, il faut pouvoir prévenir la survenue des défauts, car on pardonne plus facilement 1000 erreurs différentes que 1000 fois la même erreur. Il s'agit donc d'éviter de corriger une erreur et qu'elle réapparaisse sous une même forme

quelques temps plus tard. Il faudra par conséquent lancer un test qui expose le problème au grand jour et à partir de là, il pourra être corrigé.

La difficulté consiste donc à écrire un test qui échoue selon les conditions voulues. On corrigera dès lors le code source de façon à ce que ce test échoue. L'objectif sera non seulement de corriger le défaut, mais surtout de prévenir des erreurs récurrentes.

4.4 Inspection

L'un des avantages résultant de l'utilisation de l'IC était que le programme ressortait souvent de meilleure qualité. Cette qualité on la doit essentiellement à la batterie de tests et d'analyse qui découlent de cette étape. Quand on parle de « lancer l'inspection », il s'agit en fait de contrôler si les standards de « grammaire » du code sont respectés, contrôler l'adhérence des différentes couches architecturales ou le degré de duplication du code, ainsi que générer des données qui permettront d'améliorer le code.

Il est parfois difficile de différencier le test de l'inspection. On remarquera simplement que les deux étapes ne changent pas le code lui-même, elles mettent simplement en exergue les anomalies, bugs ou changements à apporter pour améliorer le programme. Les différences sont dans la systématique des outils utilisés. Les tests sont dynamiques: on exécutera le programme afin de tester ses fonctionnalités. L'inspection agit de manière plus systématique, en fonction d'un set de règles prédéfinies: c'est un ensemble d'outils d'analyse statique et dynamique, le but étant d'arriver à un standard auquel tous les développeurs adhèrent (ce standard pouvant varier selon les équipes de développement, ces outils sont donc parfaitement paramétrables).

Le gros avantage de l'inspection telle qu'on l'applique dans l'IC, c'est qu'on met de côté le subjectivisme de l'Homme au profit de la logique implacable de l'ordinateur. Le « pair programming » (c'est-à-dire la programmation par équipe de deux qui est l'une des méthodes de développement proposées dans XP) illustre parfaitement le côté subjectif de la méthode. Ces tests ont le mérite d'être rigoureux, répétitifs et surtout appliqués de façon à rendre une réponse standard à un même problème rencontré plusieurs fois. Automatisés et fiables, ces outils n'ont donc plus besoin d'intervention humaine. On permet ainsi, comme dans la partie « test » et « intégration de la BD », d'épargner au programmeur les tâches fastidieuses liées à cette étape afin qu'il puisse se consacrer à la correction à proprement parler.

Observons plus en détails les différents tests dévolus à l'inspection du code:

- Réduire la complexité du code:

La question est: « Comment identifier un code complexe? ». On utilisera la donnée appelée « Cyclomatic Complexity Numbers » ou CCN (issue des recherches de Thomas McCabe abouties en 1970) qui est le nombre qui mesure la complexité en comptant le nombre de chemins distincts d'une méthode. Par exemple, chaque « if », « while », « for », etc. rajoutera 1 au degré de complexité de la méthode. *cf. [rkcole 2009]*

Son étude débouche sur le fait que plus le degré de complexité est élevé, plus le risque d'erreur est important. Quelques outils (comme JavaNCSS) permettent de déterminer la longueur des méthodes ainsi que le nombre de lignes de code non-commentées.

Il y a deux moyens de réduire les risques liés à la complexité :

- Augmenter le nombre de tests unitaires pour les méthodes plus complexes (l'idéal serait d'avoir un nombre de tests équivalent au degré de complexité, du moins de s'en rapprocher)
- « Refactorer » le code en plusieurs parties, cela permet de partager la complexité en plus petites étapes, plus gérable et donc en méthode qu'on pourra tester avec plus de facilité. Exemple:

```
void printOwing() {  
    printBanner();  
  
    //print details  
    System.out.println ("name: " + _name);  
    System.out.println ("amount " + getOutstanding());  
}
```

```
void printOwing() {  
    printBanner();  
    printDetails(getOutstanding());  
}
```

```
void printDetails (double outstanding) {  
    System.out.println ("name:    " + _name);  
    System.out.println ("amount  " + outstanding);  
}
```

[Refactoring 2009]

- Appliquer des standards de développement.

Avoir des règles permettant aux développeurs d'avoir une méthodologie de travail est quelque chose d'essentiel. Il permet une certaine indépendance de programmation vu que si l'on reprend le code d'un collègue, on pourra s'y retrouver peu importe les circonstances. Plusieurs outils d'analyse sont disponibles, le plus connu étant PMD pour la plateforme Java. Il permet de régler ses préférences au niveau des conventions de nommage, de design, d'architecture ou simplement de code qui n'est pas utilisé. *cf. [Duvall 2007, p. 173]*

- Réduire le code dupliqué.

Le phénomène « copier-coller » n'est pas nouveau dans le domaine de la programmation. Même si le développement de langage orienté-objet ou de la programmation procédurale permettrait, en théorie, d'éradiquer le problème, le programmeur tombe souvent dans la facilité. La perte de temps à trouver, à analyser et à corriger un code qu'on retrouve à de nombreux endroits, suivi de l'incertitude de les avoir tous identifiés, ainsi que les coûts inhérents au développement de tests du code additionnel, devraient suffire à inciter le développeur à se concentrer davantage sur le problème. A nouveau des programmes de tests permettent d'identifier ces portions de codes qui se répètent. PMD (cité auparavant) offre d'ailleurs un « Copy-Paste Detector » (CPD) dans son escarcelle de paramètres personnalisables. *cf. [Duvall 2007, p. 176]*

- Connaître la couverture du code.

Il s'agit simplement de calculer la quantité de lignes impliquées lors d'un test d'une méthode. On peut mettre ce chiffre en relation avec le nombre total de lignes de la méthode pour connaître la couverture du code. Par exemple si une méthode à 100 lignes, et 35 seulement sont utilisées pour le test, on dira qu'elle a une couverture de 35%. Il va sans dire que l'objectif est de se rapprocher des 100%. De nombreuses autres techniques permettent d'estimer la couverture des tests: « fonction » qui calcule le nombre de fonctions utilisées, « branch » qui calcule si toutes les différentes options d'un « if/else » (par exemple) sont prises en compte, « condition », « entry/exit », « loop » ou « path ». Toutes ces informations peuvent être utiles selon les besoins du programmeur qui souhaite optimiser ses tests. De nombreux outils sont d'ailleurs disponibles: Junit, Ncover, Clover.NET ou EMMA, pour ne citer qu'eux. *cf. [Di-ENS-Labatut 2007]*

Cette information peut être particulièrement utile pour le Chef de projet, qui fixe à l'avance un seuil ou un quota de couverture du code par des tests afin de garantir une certaine qualité du programme. Il peut dès lors comparer la différence de couverture d'une intégration à l'autre, et analyser le pourquoi d'une éventuelle baisse (le programme augmentant en terme de lignes de code, mais la quantité de tests de suit pas, ou bien certains tests ont été enlevés ou n'ont plus de référence claire au code)

Il y a une quantité incroyable de données qui pourraient être analysées pour améliorer la qualité du programme. L'important étant de décider quelle information est pertinente pour l'équipe de développement, pour l'équipe de l'Assurance Qualité, pour le manager ou le client. Le seul objectif doit être la recherche de la qualité: via des routines de tests et des feedbacks rapides, les développeurs peuvent se concentrer sur des tâches nécessitant une analyse plus fine du code.

4.5 Déploiement

Lorsqu'on parle de développement de logiciel, il va sans dire que l'unique objectif est de déployer un logiciel fonctionnel. Toutes les étapes qui précèdent celle-ci (et celle qui lui

succède) n'ont pour but que de faciliter le déploiement, et de le rendre opérationnel peu importe l'environnement sur lequel il tourne.

Paul M. Duvall parle de 6 étapes caractéristiques lors du déploiement d'un logiciel dans le cadre d'une IC cf. [Duvall 2007, p. 191]:

4.5.1 Etiqueter les données du « repository »

Etiqueter les différentes versions permet d'identifier et suivre l'évolution d'un groupe de fichiers.

4.5.2 Produire dans un environnement sain

Le but ici est de s'assurer que le système d'exploitation, les logiciels installés, les scripts, la configuration du système ou du serveur n'entrent pas en conflit avec le logiciel déployé. La solution est de partir de zéro (rien sur le système) puis de tester pas-à-pas le logiciel en appliquant une couche après l'autre, ou plus fiable encore, de lancer un processus automatisé qui appliquerait et supprimerait toutes les couches pour avoir la certitude qu'elles sont bien compatibles les unes avec les autres.

4.5.3 Générer et étiqueter un build issu du « repository »

« Labeling a build is as simple as performing a full build and assigning an identification to it » [Duvall 2007, p. 196]

La différence entre labéliser un build et une version dans le repository réside simplement dans le fait que le build est un exécutable, un fichier .zip ou .jar déjà compilé. Le contenu du repository n'est qu'un groupement de fichiers non-binaires qui n'est souvent pas encore compilés.

Comme pour le repository, le but est une meilleure communication entre les différents organes de production, pouvoir faire un historique de chaque build et bien sûr, la possibilité de revenir en arrière en cas d'échec.

4.5.4 Lancer des tests de tous niveaux dans un clône de l'environnement de production

Il s'agit là de tester sur un environnement sain (qui sera le clone de l'environnement de production) que tous les tests passent avec succès. Cette batterie de tests (déjà effectué dans une autre phase de l'IC) permet simplement de s'assurer (une nouvelle fois) que le logiciel est performant, ceci dans un environnement différent. Ces tests sont aussi effectués pour « rassurer » les développeurs: un développeur confiant est souvent un développeur performant.

4.5.5 Créer des feedbacks pour le build

Le feedback est la base de l'IC. Avoir un retour de l'information permet :

- aux développeurs d'avoir un rapport sur ce « quoi faire ? » pour corriger les éventuels défauts
- aux clients de savoir dans quelle phase de développement se trouve leur commande
- au chef de projet d'avoir une vue d'ensemble sur ce qui exige encore des améliorations.

On ne le dira jamais assez, une bonne communication à tous les échelons de production permet d'éviter les malentendus et de gérer au mieux les échéances.

4.5.6 Possibilité de revenir à une version précédente

La capacité de revenir à une version précédente est une option très importante dans le processus de développement, également pour comprendre les erreurs qui ont mené à un « broken build ». Cette étape implique que les builds ou le repository ont bien été étiquetés.

4.6 *Feedback*

Afin de gérer au mieux l'émission de feedbacks, on articulera notre choix sur quatre branches distinctes, comme l'a présenté Paul M. Duvall dans son livre.

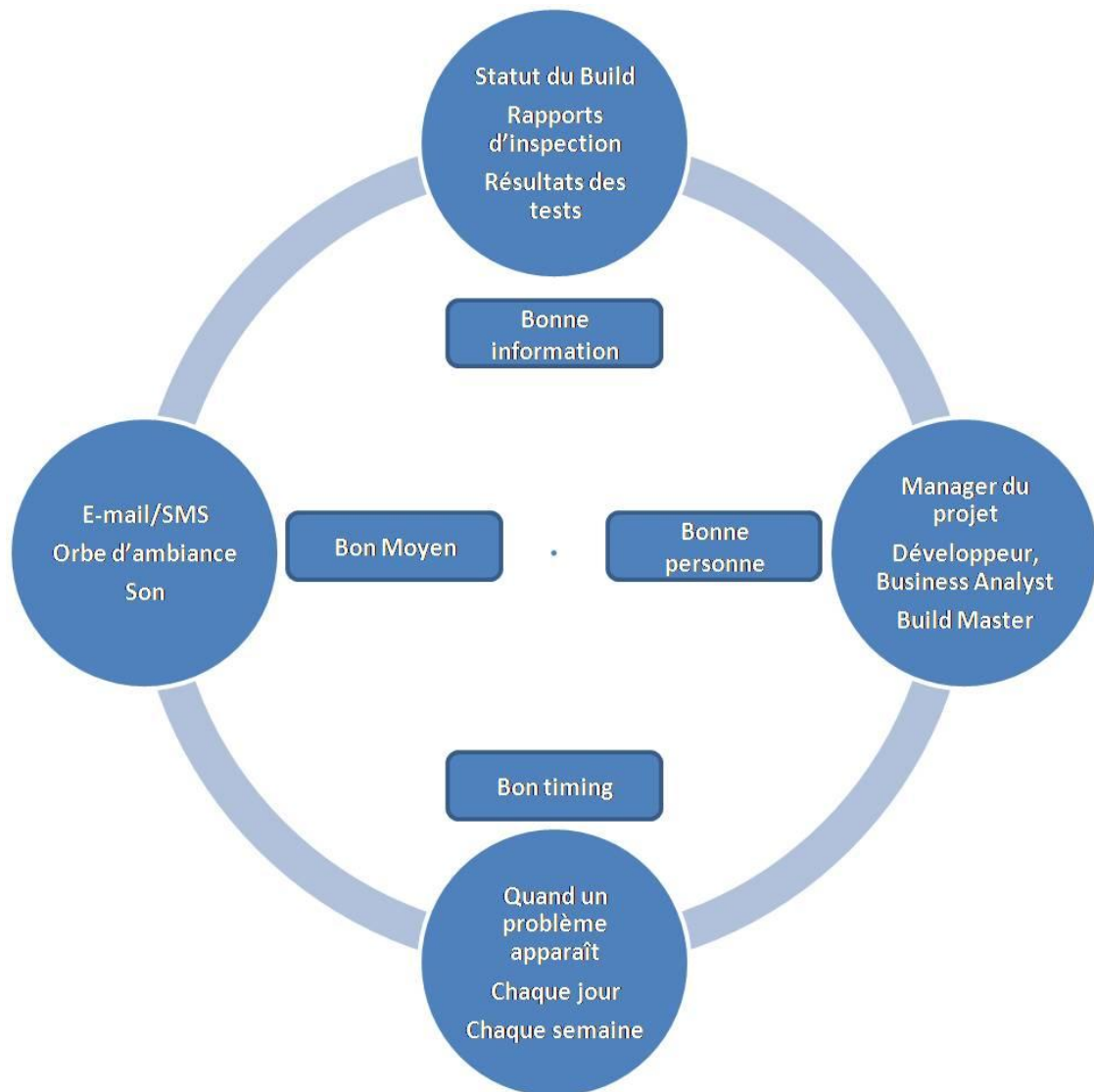


Figure 4-2: Le feedback comme Paul M. Duvall le conçoit, [Duvall 2007, p. 206]

4.6.1 Les bonnes informations

De nombreuses informations sont générées: les résultats de tous les tests de la phase du même nom, ceux des outils d'analyse statique de la phase d'inspection, ou simplement celui de la réussite ou non du « build ». Toutes ces informations n'intéressent bien sûr pas tous les acteurs du projet et on peut facilement répartir les informations pertinentes pour chacun.

4.6.2 Les bonnes personnes

De nombreuses personnes sont intéressées au résultat d'un build, à commencer par le testeur qui s'intéressera aux résultats de la phase de « tests » et à celle de l'« inspection » afin de peaufiner ses propres requêtes. Le développeur, le chef de projet,

le responsable du build ou simplement le client, tous ces acteurs ont un intérêt dans la bonne marche du projet. Certains préféreront recevoir une alerte quand il y a un problème dans le processus d'intégration, d'autres uniquement quand le « build » réussit: l'essentiel étant de ne pas envoyer trop d'informations aux acteurs au risque qu'ils ignorent carrément les messages d'alerte, tellement ils en reçoivent.

4.6.3 Le bon moment

L'objectif principal de l'IC, au-delà de la qualité, est le gain de temps. Réduire au maximum le temps entre la découverte d'une faille ou d'un code incorrect, et sa correction. On découvre les erreurs grâce aux routines de tests, on les corrige grâce aux feedbacks qui sont générés. Donc, plus on attend pour envoyer l'information, plus le problème tardera à être résolu. Ici le bon moment sera donc « le plus vite possible » pour les personnes qui sont concernées par la correction de ces erreurs.

4.6.4 La bonne manière

C'est peut-être la partie la plus ludique du processus: chacun à la possibilité de choisir la manière dont il veut que l'information lui soit transmise. Certains opteront pour un simple e-mail ou un sms (de telles options peuvent être choisies d'un simple clic via la majorité des serveurs d'intégration du marché). D'autres gadgets permettent aussi d'informer sur le statut du build: l'orbe qui s'illumine (et dont la couleur pourrait même éclairer sur la réussite ou l'échec du build), un son, une icône sur la barre des tâches ou même une fenêtre qui s'ouvre via sa messagerie instantanée (MSN, AIM, Yahoo, etc...). Au-delà de l'aspect récréatif de la manière dont l'information nous parvient, il est clair que si l'on travaille dans un bureau bruyant, on n'optera pas pour un son qu'on ne pourra peut-être même pas entendre, la même chose pour un sms dans un immeuble qui ne capte pas de réseau. Le but est bien entendu de limiter le temps entre la découverte et la résolution. On choisira dès lors l'outil le plus adapté à nos besoins, avec de la fantaisie si cela est possible.

5 Outils et automatismes à adopter pour une IC idéale

Ce chapitre représente une sorte de « guide des bonnes pratiques » pour installer un système d'IC optimal et les outils informatiques nécessaires pour y parvenir. On tâchera ainsi de répondre aux questions suivantes, présentées dans l'introduction.

- *Quelles sont les habitudes à acquérir, les pratiques à adopter pour une IC idéale?*
- *Quels outils pour l'IC?*

5.1 **Réflexes à avoir pour le repository**

5.1.1 Utiliser le repository pour stocker tous les fichiers

On pourrait s'attendre à ce que les fichiers du code-source soient les seuls ayant besoin d'être centralisés. Cependant, et afin d'éviter des commentaires du genre « pourtant ça marchait sur ma machine! », il est important d'inclure dans son repository tous les fichiers tels que :

- les JAR, les librairies, les DLLs
- les fichiers de configurations
- les builds scripts ainsi que les paramètres d'environnement du build
- les fichiers de données pour initialiser une application

Le fait que l'environnement soit connu et que les fichiers soient les mêmes pour tous permet d'améliorer la standardisation de la production et que tous les développeurs aient un environnement stable de production.

« Everything should be in the repository » [Fowler 2008]

5.1.2 La « Staging Area »

Le meilleur moyen de coordonner des dépendances multiples liées au build est d'utiliser une « Staging Area ». On pourrait assimiler cette pratique à une « sandbox » (*cf. Chapitre 4.2.1*) réservée à tous les composants dont sont dépendants le build et les tests.

Comment ça marche?

- Chaque équipe développe son sous-projet sans se soucier des autres: lance des builds, compile et commit dans le repository commun.
- A une certaine échéance, ces équipes mettent en commun les bibliothèques, les APIs, ou le fichier de configuration dont les autres pourraient avoir besoin.
- Lorsque le build est effectué dans l'espace de travail personnel du développeur ou même dans le serveur d'intégration de l'équipe, tous les fichiers qui concernent l'environnement de développement du build pointent vers la « Staging Area » qui permet de centraliser les informations. [Crossroads 2004] (voir la Figure 5-1 : La Staging Area [Crossroads 2004] ci-dessous)

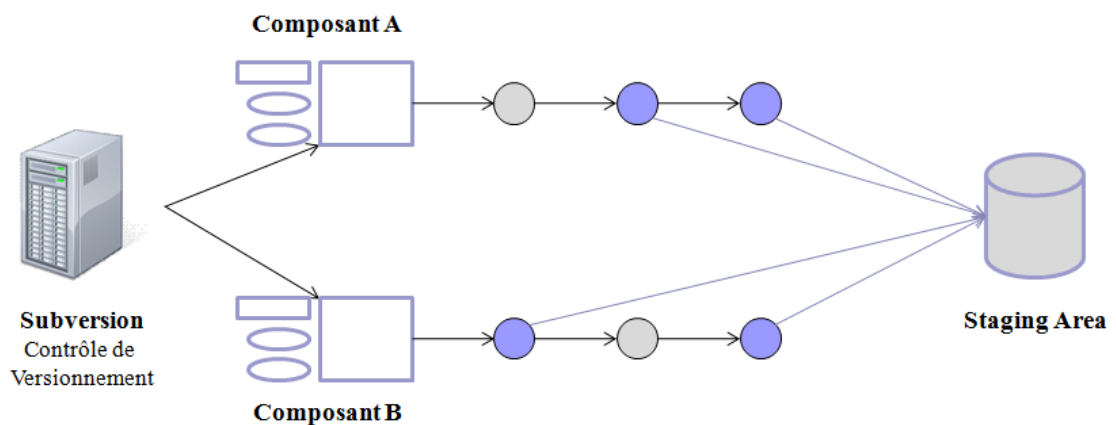


Figure 5-1 : La Staging Area [Crossroads 2004]

La « Staging Area » est donc simplement une technique pour éviter les dépendances d'interface des différentes équipes lors du build. On sépare donc leur implémentation. L'environnement choisi permet donc de satisfaire les dépendances build et tests des différentes équipes de développement.

5.2 **Réflexes à avoir pour le build**

Eviter de se retrouver face à un build qui échoue, voilà un objectif fondamental pour les équipes de développement. Le plus important reste la fréquence et la durée de ces échecs puisque le processus d'IC s'arrêtera tant que le build posera problème. Le but ici est de ne pas démoraliser les programmeurs avec des interruptions généralisées dues à

l'échec du build. La plupart du temps ces échecs ne donnent pas véritablement de clés quant à leur résolution ou à l'état actuel du build. Ceci amène le développeur à éviter de faire un « check out » pour mettre son espace de travail à jour, et choisit donc d'intégrer son code le moins souvent possible. On se retrouve donc à ignorer la règle No 1 des bonnes pratiques à adopter pour l'IC : « un commit régulier du code ». Plus on repousse l'échéance du build, plus les risques d'un échec sont grands, plus les équipes de développement sont craintives quant au résultat du build et plus ils éviteront de « commiter » leur code. On tâchera donc d'éviter de se retrouver face à ce cercle vicieux en adoptant quelques bonnes pratiques qui aideront à la mise en place d'une IC idéale.

5.2.1 Types de build

- **Le « private build »**

Ce build est lancé à titre personnel et dans l'environnement local du développeur. Il a pour but de s'assurer que le code ne contient aucune erreur avant sa mise en commun sur le repository. Il s'agit plus d'un build préventif et correspond parfaitement à cette volonté de partitionner au maximum l'environnement de développement.

- **L' « integration build »**

Ici on obtient un build complet destiné à toute l'équipe de développement, et dont le code-source provient du repository. Martin Fowler parlera ici de « staged build » qui consiste en un partitionnage des étapes du build. *cf. [Fowler 2008]* D'abord le « commit build », qui lancera la compilation ainsi que tous les tests unitaires. Ensuite, et seulement si le « commit build » est concluant, le « secondary build » lancera les tests composants, système ou de performance (*cf. Chapitre 4.3*), ainsi que tous les tests d'inspection prédéfinis. Ces builds secondaires (qui peuvent eux-mêmes être partitionnés) déclenchent quant à eux une routine de tests plus détaillée, il faudra donc plus de patience pour leur résolution. Le « staged build » permet d'obtenir plus rapidement le résultat du build afin d'être plus réactif en cas d'erreur.

- **Le « release build »**

Il s'agit du build destiné au client ou au département qualité. Une autre routine de tests est lancée telle que celle d'acceptation ou celle liée à la performance de chargement. Ce build met en exergue un des grands avantages de l'IC: créer un logiciel déployable en tout temps. Ce build n'intervient en général qu'à la fin d'une itération ou d'une autre étape importante. *cf. [Duvall 2007, p. 80]*

5.2.2 Les façons de le lancer

- Sur demande

L'utilisateur choisit lui-même le moment de son intégration, c'est donc un processus manuel.

- Agendé

On prévoit une date et une heure précise à laquelle l'intégration sera effectuée.

- A chaque « commit »

Le serveur d'intégration scrutera un changement dans le repository et activera le processus d'intégration s'il en perçoit un.

- « Event-driven »

Il s'agit cette fois du repository qui déclenche le processus en lieu et place du serveur d'intégration. Celui-ci s'activera dès qu'un changement prédéfini a lieu. *cf. [Duvall 2007, p. 81]*

5.2.3 Automatiser le build

Le but de l'automatisation est toujours le même: limiter au maximum l'intervention humaine pour les processus répétitifs et sujets à erreur. En choisissant l'automatisation pour le build, on admet également que de nombreuses options sont suggérées:

Est-ce un build privé? Est-ce qu'on recherche juste un feedback rapide? Est-ce que les changements qui ont été effectués sont significatifs et méritent un build complet? Etc. Il faut donc déterminer quels processus seront inclus dans le « build script ». *cf. [BuildMeister 2009]*

5.2.4 L'exécuter via une seule commande

Fowler et Duvall imaginaient le concept d'intégration comme un simple bouton qu'on presserait et qui générerait toutes les étapes déjà citées dans la partie des dispositifs (*cf. Chapitre 4*) en arrière fond, et dont la résultante serait le logiciel. Le site internet officiel dédié au livre de Paul M. Duvall porte même ce nom (« www.integratebutton.com »). *[Duvall WS 2009], cf. [Duvall 2007, p. 144]*

5.2.5 Créer de plus petites équipes en séparant les étapes de développement

Le but ici est de séparer le projet en sous-projet, et donc de diviser l'équipe de développement en sous-équipes. Si un build venait à échouer avec une équipe de 100 personnes qui dépendent de ce résultat, il est évident que les heures de développement perdues seraient plus importantes que pour un build impliquant 10 personnes. On limite donc l'impact de l'échec et par conséquent, on optimise les ressources à disposition. L'autre avantage d'une équipe réduite est la communication entre développeurs. Le site AnthillPro (société de développement de serveur d'IC) présente la communication comme « *un facteur-clé dans le support des trois piliers du build sans douleur (pain free)* » [Anthillpro 2009]. Ces trois piliers seraient la réduction de la durée d'un projet se trouvant dans un état d'échec, l'appui au travail engendré par cet échec, et la limitation du nombre de personnes concernées.

5.2.6 Attribuer un poste de « Chef de build »

Une bonne communication rend ce statut quelque peu inutile, néanmoins il est important de pouvoir compter sur quelqu'un en cas de problème. Son rôle sera simplement de s'enquérir du résultat du build et d'informer le responsable de l'échec que du travail l'attend afin de rendre le logiciel à nouveau fonctionnel. Une bonne communication rend ce statut quelque peu inutile, néanmoins il est important de pouvoir désigner un responsable en cas de problème.

5.2.7 Séparer le build de l'environnement de développement IDE

On parlait précédemment, dans le déploiement du logiciel, de l'importance d'avoir un environnement sain, une sorte de clone de l'environnement de production. Cette précision a son importance quand il s'agit d'élaborer le « build script ». Celui-ci ne doit en aucun cas dépendre de son IDE: le problème résultant étant la dépendance du script à sa propre configuration, alors que le but recherché est la standardisation de l'intégration par le serveur d'IC, et que celui-ci n'a que le système opérationnel d'installé. cf. [Duvall 2007, p. 74]

5.2.8 Faire en sorte que les builds échouent rapidement

Bien sûr il est préférable de réussir son build, néanmoins il faut partir du principe que le build a de grandes chances d'échouer, et que, dans ce cas, il vaut mieux qu'il échoue le

plus vite possible. Il faut dans un premier temps déterminer l'ordre dans lequel les différentes intégrations ou tests vont se dérouler, en choisissant les parties susceptibles d'échouer le plus rapidement, ce qui peut varier selon le type de projet.

5.2.9 Faire qu'un build soit compatible peu importe l'environnement

Il s'agit donc d'avoir un build efficace qui ne dépendrait pas de l'environnement. Pour cela on peut passer par différentes valeurs de configurations qui seront disponibles dans la plupart des environnements comme la configuration du serveur d'application, les informations liées à la connexion de la base de données ou la configuration du « Framework » (espace de travail modulaire → p.ex. Eclipse). La *Figure 5-2* nous montre que même si les environnements dans lesquels on travaille peuvent être différents, le script utilisé pour le build peut être paramétré unilatéralement pour améliorer la configurabilité du système, afin qu'il puisse s'adapter à toutes les variations d'environnement.

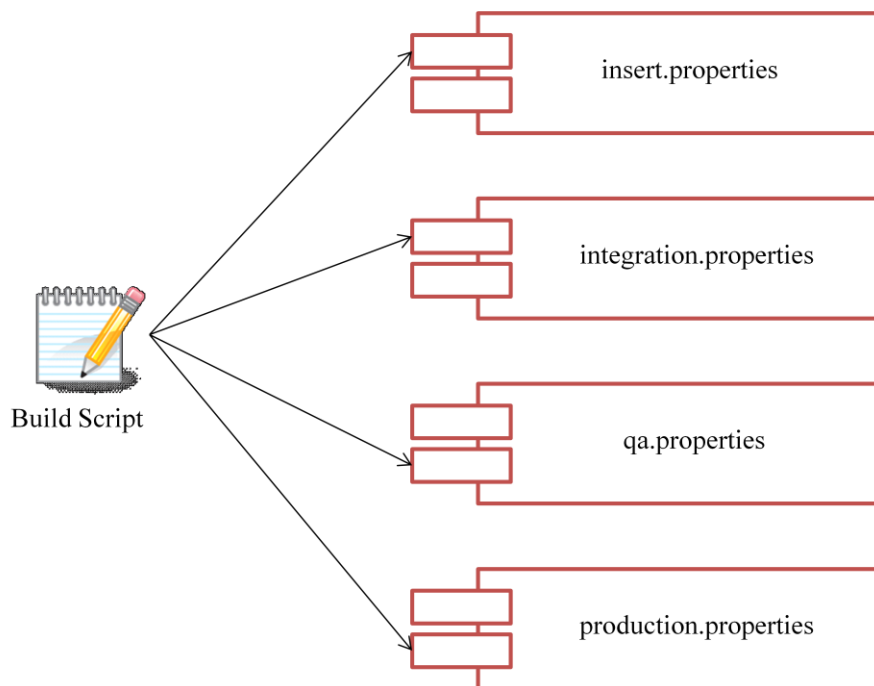


Figure 5-2 : Le Build Script configuré pour s'adapter à chaque environnement, [Duvall 2007, p. 79]

5.2.10 Utiliser une machine dédiée pour le build d'intégration

Dans la partie des réflexes à adopter pour le repository, le problème du « Mais ça marchait sur ma machine ! » était abordé. Réduire voir supprimer les problèmes liés à la configuration est un défi réalisable via une machine dédiée pour le build d'intégration.

Le danger est naturellement que la configuration de sa machine ne corresponde pas à celle des autres développeurs et que le code-source qu'on importe sur le repository et dont l'intégration était réussie, ne fonctionne plus sur la machine des autres développeurs. L'idée est de mettre en place une machine externe dont l'environnement serait « propre », c'est-à-dire un environnement d'intégration sans dépendance de code issu d'une intégration précédente (un script pourrait se charger de rendre l'environnement à un état prédéfini avant le commit du code). Ce dispositif permettrait également de savoir si le développeur a omis de « commiter » tous les fichiers nécessaires (fichiers de configuration, build script, script de la base de données, etc...) dans le repository.

On l'utiliserait donc comme un filet de sécurité afin de s'assurer que le logiciel réponde à toutes les exigences et fonctionne comme il devrait. *cf. [Duvall 2007, p. 81]*

5.2.11 Lancer des builds rapides

La base de l'IC, ce sont les feedbacks, savoir où en est le processus d'intégration, quel test a échoué, si le programme répond aux attentes, etc. Ces informations se doivent d'arriver à l'intéressé (développeur, client, responsable qualité,...) dans les plus brefs délais. Le thème du *paragraphe 5.2.8* (Faire en sorte que les builds échouent rapidement) est une des conséquences de cette pratique : plus la durée du build est courte, plus le feedback est rapide. Paul M. Duvall différencie le concept d'évolutivité et celui de performance dans le cadre d'un build. « *L'évolutivité du build indique à quel point le système du build est capable de gérer l'augmentation du nombre de codes qu'il intègre et analyse. La performance du build se réfère à la durée du build.* » [Duvall 2007, p. 87] Il précise que le système d'IC devrait être capable de gérer une augmentation du code sans grandes dégradations sur les performances. Il décrit également un processus en 4 points permettant de réduire au maximum la durée du build.

1. Rassembler les données du build

Une liste de données qui peuvent être pertinentes pour l'amélioration de la durée du build est collectée. Paul M. Duvall en dresse une liste qui paraît exhaustive :

Temps de compilation, nombre de lignes de code, nombre et types d'inspections, génération du temps d'assemblage moyen, temps d'exécution des tests, ratio entre les builds réussis et ratés, temps d'inspection, temps de déploiement, temps de « rebuild » de la base de données, ressources et utilisation de la machine dédiée au build d'intégration, chargement du système de versionnement. *cf. [Duvall 2007, p. 88]*

2. Analyser les données du build

On analysera ces données en fonction des techniques d'amélioration de la *Figure 5-3*, ainsi que des 3 critères évoqués (évolutivité, performance et difficulté d'implémentation).

3. Choisir et appliquer les améliorations

On choisira les améliorations spécifiques en fonction de la stratégie voulue, des moyens à disposition et des ressources humaines disponibles. La *Figure 5-3* donne une bonne idée des tenants et aboutissants du choix du développeur.

Techniques d'amélioration	Priorité	Evolutivité	Performance	Difficulté
Utiliser une machine dédiée pour les builds d'intégration	1	↑	↑	↓
Augmenter les capacités physiques de la machine dédiée au build d'intégration	2	↑	↔	↓
Améliorer les tests de performance	3	↓	↑	↔
Garder une ligne pour le build d'intégration	4	↓	↔	↓
Optimiser l'infrastructure	5	↔	↔	↑
Optimiser le processus du build	6	↓	↔	↓

Faire les builds des composants du système séparément	7	↓	↔	↔
Améliorer les performances de l'inspection du programme	8	↓	↑	↓
Faire des builds d'intégration distribués	9	↑	↑	↑

Légende : impact sur l'évolutivité, la performance et la difficulté : ↑ haute ↓ basse et ↔ égale

Figure 5-3 : Tableau indiquant les améliorations possibles et ses conséquences (l'évolutivité, la performance et les difficultés qui en résultent), reprise du graphique de [Duvall 2007, p. 90]

4. Réévaluer et répéter si nécessaire

Tout est dans le titre, il s'agit de voir si les changements opérés correspondent aux objectifs fixés.

5.2.12 Refactorer les builds pour optimiser le processus

Paul M. Duvall propose 5 étapes pour optimiser le processus. cf. [Duvall 2007, p. 94]

1. Examiner l'infrastructure

On parle ici uniquement du matériel ou de sa configuration: d'un serveur inadapté, des performances du réseau, de systèmes dispersés géographiquement qui ralentissent le processus du build, etc. L'idée est donc de rechercher et améliorer si nécessaire les composants techniques qui génèrent le build.

2. Optimiser le processus du build

Si le temps de compilation des fichiers est trop important, on peut parer à ce problème en choisissant un build incrémental qui ne compilera que les fichiers qui ont été modifiés depuis le dernier build.

3. Faire un build pour chaque composant du système

(Donc le séparer en sous-systèmes)

Le seul point décisif pour que ces sous-systèmes soient opérationnels, c'est qu'un sous-système « maître » soit désigné afin que chaque changement dans un sous-système dépendant engendre un build complet.

4. Améliorer la performance de l'inspection du logiciel

Faire en sorte que les informations recherchées soient les plus pertinentes possibles. Plus on est concis dans le choix des valeurs qui nous intéressent, plus l'inspection sera optimisée.

5. Appliquer des builds d'intégration distribués

Méthode qui ralentit plus qu'elle n'apporte des résultats. On peut néanmoins l'utiliser dans le cas où toutes les solutions précédentes n'ont pas apporté satisfaction.

6 Les outils de développement

N'ayant pas eu l'occasion de tester tous ces différents outils de développement, ils vous seront présentés selon certains critères fonctionnels présentés sur les pages Web des différentes firmes ayant développées ces logiciels, et non sur des critères subjectifs comme l'ergonomie ou la facilité de prise en main.

6.1 Le serveur d'IC

Un serveur d'IC pertinent est un serveur qui doit posséder un minimum de fonctionnalités essentielles (l'exécution du « build », l'intégration du contrôle du versionnement, le feedback, la labélisation du « build ») et plusieurs fonctionnalités qui lui permettent de se démarquer de ses concurrents (la gestion des dépendances inter-projets, l'interface, la sécurité, etc.). Dans les serveurs qui répondent aux exigences principales et qui fournissent même d'autres prestations, on s'intéressera aux deux principaux serveurs disponibles sur le marché, dont fait parti celui qui sera utilisé dans la partie pratique du travail: AnthillPro et CruiseControl (.NET). Une matrice des différentes possibilités de chacune de ces deux éditions est disponible sur le site du développeur de CruiseControl [*ThoughtWorks 2009*], datant du 15 avril 2010. Ne cherchant pas à être exhaustif mais simplement à comparer ces deux serveurs, plus de 20 autres serveurs sont également disponibles dans cette matrice à la page Web mentionnée dans la bibliographie.

6.1.1 AnthillPro

Distribué par Urbancode et utilisant la plateforme Java, c'est un produit commercial qui dispose d'un grand nombre de fonctionnalités. Autrefois, Urbancode proposait un logiciel libre s'appelant Anthill OS. Le succès aidant, ils ont décidé de développer le concept avec des améliorations importantes comme une configuration beaucoup plus flexible, la possibilité d'utiliser différents versions de JDK (Java Development Kit) pour le « build » de projets différents, un support plus robuste pour les dépendances du projet ou la possibilité de lancer le « build » de n'importe quelle version antérieure. cf. [*CM Crossroad 2009*] Pour certains, les améliorations les plus significatives restent l'authentification et les paramètres d'autorisation [*Duvall 2007, p. 264*] qui permettent aux administrateurs de gérer les accès en matière de visualisation ou les options de configuration.

6.1.2 CruiseControl .NET

Distribué par ThoughtWorks et utilisant, elle, la plateforme Microsoft .NET, CruiseControl.NET (CCNET) est le « petit frère » du doyen des serveurs d'IC (CruiseControl, pour la plateforme Java). C'est un logiciel libre (c'est-à-dire téléchargeable gratuitement sur le site Web du distributeur [*SourceForge 2010, lien 1*]).

L'un des grands avantages est la quantité de support et de documentations disponibles pour sa mise en place qui compense le fait que la configuration est essentiellement manuelle dans un fichier XML, et qui la rend souvent peu intuitive pour le profane. De nombreux « add-on » permettent de faciliter la prise en main. On pourra citer le programme CCNetConfig qui permet une écriture plus intuitive du fichier de configuration ou CCTray qui offre une meilleure visualisation des résultats du build. Le « WebDashboard » peut également être installé afin d'interagir avec le serveur via son navigateur WEB.

6.2 Les Builds scripts

6.2.1 (N)Ant

Disponible depuis 2001, NAnt (pour *Not Ant*) est un programme permettant l'automatisation de la compilation du code-source .NET. C'est ce programme qui va être utilisé pour ce projet. On cherchera, via ce logiciel, à optimiser et assouplir l'intégration en évitant de devoir passer la compilation par une ligne de commande, le but étant d'unifier le mode de compilation et d'actions qu'on effectuera sur les sources, tout en intégrant XML.

En plus de s'occuper de la compilation, NAnt peut effectuer de nombreuses tâches, parmi lesquelles:

- Compilation de sources en C, C++, C#, J#, Visual Basic.NET, et Jscript.NET
- Lancement des tests unitaires NUnit
- Manipulation de répertoires ou de fichiers
- Accès en lecture/écriture dans la base de registre
- Accès à CVS (contrôle de versionnement), ...

NAnt possède de nombreux avantages en comparaison à d'autres logiciels correspondant. On citera déjà le fait que NAnt est multi-plateformes, c'est-à-dire qu'il est opérationnel aussi bien sur Linux, MacOS ou Windows. D'ailleurs aujourd'hui, la plupart des outils « OpenSources » en .NET sont distribués avec un script NAnt. Toutes les informations complémentaires ainsi que les liens de téléchargement sont sur la page WEB du développeur [*SourceForge 2010, lien 2*].

6.2.2 Maven

Apache Maven [*Maven, 2010*] est un outil logiciel libre pour la gestion et l'automatisation de production des projets logiciels Java. [*Wikipedia 2010, lien 5*] Il est considéré comme bien plus simple à configurer que Ant (la version Java de NAnt) : l'écriture et la modification des scripts de Ant sont souvent fastidieuses. De plus, si les besoins divergent ou simplement la structure du projet diffère, le script Ant n'est plus directement réutilisable et nécessite des retouches. En ce sens-là, la standardisation est nettement plus aisée sur Maven. Il n'est d'ailleurs plus question de créer un fichier XML décrivant précisément les tâches que l'on doit exécuter pour construire son projet : Maven propose de décrire son projet, il s'adapte alors à cette description pour le construire. [*Développez.com, 2009*] Finalement, Maven est plus flexible que Ant puisqu'il donne la possibilité à ses utilisateurs de créer leurs propres plugins, contrairement au très statique Ant.

Maven utilise un paradigme connu sous le nom de *Project Object Model* (POM) afin de décrire un projet logiciel, ses dépendances avec des modules externes et l'ordre à suivre pour sa production. Il est livré avec un grand nombre de tâches prédéfinies, comme la compilation de code Java ou encore sa modularisation. [*Wikipedia 2010, lien 5*]

6.3 **Le système de contrôle de versionnement**

6.3.1 Subversion

Subversion est un système de contrôle de versionnement (*cf. Chapitre 10 pour le glossaire*) axé client-serveur (Subversion étant le serveur) qui permet à un grand nombre de développeurs de travailler sur un même projet. Particulièrement apprécié pour les projets « OpenSource », il fonctionne en s'appuyant sur une base centralisée hébergée sur le serveur qui contient la totalité des ressources (différentes versions, noms

de l'auteur du changement, dates de mises à jour, etc.). Cette base centralisée est aussi appelée « repository ».

Jusqu'ici, cette définition du programme pourrait être celle de CVS (*Concurrent Versions System*). En réalité, Subversion s'appuie sur le même principe de fonctionnement en maintenant la plupart des conventions de travail de CVS, y compris une grande partie de l'ensemble des commandes. Par contre, les améliorations sont notables par rapport à son prédécesseur. Subversion propose, contrairement à CVS, des commits atomiques, ce qui évite de se retrouver dans un état contradictoire si une opération sur le repository est interrompue en plein milieu. Ceci est permis en utilisant de façon sous-jacente une base de données capable de gérer les transactions atomiques (p.ex. Berkeley DB) [Wikipedia 2010, lien 4]. On parlera aussi du déplacement ou le renommage des fichiers/répertoires qui sont possibles sur Subversion sans même en perdre l'historique. Enfin on peut définir les permissions d'accès sur les différentes parties du repository grâce au protocole WebDAV, tandis que CVS ne le peut que de façon limitée. Le site suivant [Better SCM 2010] propose un comparatif quasi exhaustif des différents logiciels de contrôle de versionnement et est mis à jour de façon régulière (au 18 mai 2010 au moment d'écrire ces lignes). Il apparaît que Subversion est un logiciel très compétitif sur ce marché.

6.3.2 TortoiseSVN

Un logiciel de contrôle de versionnement centralisé implique l'interaction serveur-client. Subversion étant le serveur, TortoiseSVN est son client open-source qui va gérer les fichiers et les répertoires à travers le temps. Outre un accès très intuitif à Subversion au travers du menu contextuel de l'explorateur, TortoiseSVN y ajoute également son propre menu. On repère également très rapidement le statut de la copie de travail grâce à de petites icônes de recouvrement. Dans la mesure où ce n'est pas une intégration pour un IDE spécifique tel que Visual Studio ou Eclipse, il est possible de l'utiliser avec tous les outils de développement disponibles. [TortoiseSVN 2010] et [Optus.net 2007]

6.4 Les Feedbacks

C'est un fait que la qualité de la documentation technique destinée aux développeurs responsables de la maintenance de l'application influe grandement sur la pérennité d'un projet de développement.

6.4.1 Doxygen

Doxygen ou littéralement « générateur de documentations » (DOCuments GENerator) est un logiciel permettant de créer des documentations techniques pour notamment le C, C++, C# ou Java, ce dans différents formats (HTML, PDF, LaTeX, RTF, PostScript et prochainement en XML).

En extrayant l'information à partir du code source, il permet de systématiser le comportement du développeur afin qu'il prenne des habitudes allant dans ce sens.

Le site officiel [*Doxygen 2009*], rédigé par Dimitri van Heesch, créateur de Doxygen, donne une liste des caractéristiques principales de ce logiciel.

Parmi elles, on pourrait citer le fait que Doxygen permet l'insertion des définitions de fonctions, de membres ou de classes dans la documentation. Il permet aussi de détecter automatiquement les fonctions privées, publiques ou protégées, de même que Qt, une bibliothèque logicielle orientée objet et développée en C++. Tout a été mis en place pour faciliter l'utilisation et la lecture, comme les fragments de code source qui sont mis en évidence de façon syntaxique. [*Doxygen 2009*],

Doxygen ne sera pas utiliser pour la partie pratique de ce travail, cette présentation étant simplement là pour définir les outils disponibles dans le cadre d'une pratique de développement informatique « idéal », dans le sens où le temps investi à la mise en place sera rentabilisé sur le long terme.

7 L'avenir de l'IC et ses alternatives

L'intégration continue est un terme nouveau. Son principe repose cependant sur certaines méthodologies qui, elles, sont appliquées depuis plusieurs années déjà. On sait que l'intégration est un problème qui devrait persister dans les années à venir. Malgré toutes les améliorations qu'on peut imaginer, l'objectif à terme restera donc toujours le même : faire de l'intégration un non-événement. Nous allons voir dans ce chapitre l'avenir que peut nous réserver l'IC et les possibles alternatives à disposition.

7.1 Avenir de l'IC

Nous pouvons déjà percevoir les différents virages que l'IC va entreprendre à l'avenir, certains développeurs n'ayant pas attendu la fin de ce travail pour créer des logiciels encore plus performants et mieux agencés. Il est plus aisé de se baser sur un logiciel payant pour apprécier l'évolution du secteur. Prenons, par exemple, le cas de « TeamCity » : depuis juin 2009 et sa version 4.5, il propose de changer le scénario « codage-commit-vérification » en « codage-vérification-commit » (cf. Figure 7-1 : « Codage-Vérification-Commit » selon TeamCity [TeamCity 2009]) le commit se faisant automatiquement quand la vérification est effectuée. On évite ainsi de se retrouver avec un build ne fonctionnant pas dans le logiciel de contrôle de versionnement.

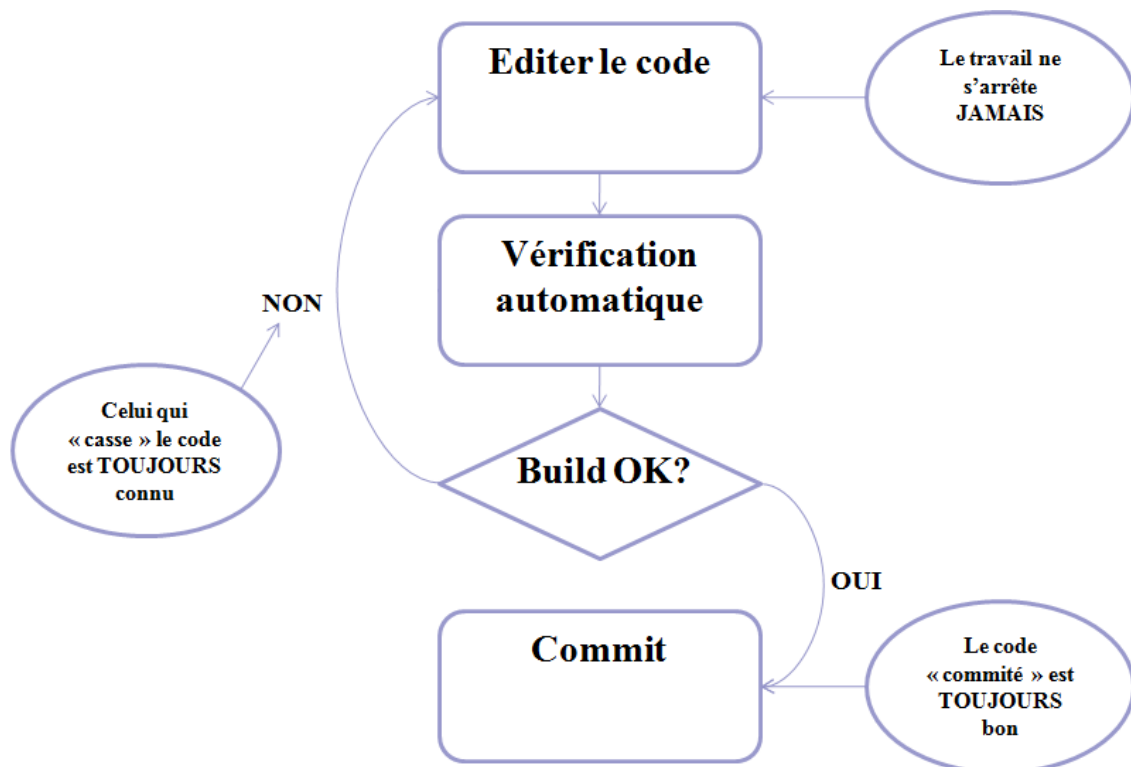


Figure 7-1 : « Codage-Vérification-Commit » selon TeamCity [TeamCity 2009]

Cette pratique laisse cependant nombres de développeurs perplexes, l'assimilant à celle déjà bien ancrée dans le monde de la programmation, la « branch per task » (littéralement : une branche par tâche), dont Brad Appleton a été un des précurseurs [Appleton 1998]. Ceux-ci prétendent que les avantages qu'apporteraient cette solution sont précisément les mêmes que ceux d'un système avec un bon « branching ». Le problème de l'argumentation de Paul M. Duvall est qu'il se focalise sur une programmation « globale », c'est-à-dire qu'il n'y a qu'une « branch » principale pour toutes les tâches (ou presque). cf. [Codicesoftware, 2008]

Le « branching », dans un contrôle de versionnement, est simplement la duplication d'un code source, d'un fichier, etc. de façon à ce que les modifications prennent effet en parallèle parmi ces deux branches. Ce « branching » peut prendre cinq formes différentes selon les besoins : physique, fonctionnel, environnemental, organisationnel et procédural. La « branch per task » exécute quant à elle chaque tâche sur sa propre branche d'activité. Cette approche permet de développer parallèlement des projets de large envergure impliquant de nombreux acteurs. Elle permet aussi l'isolation des changements, comme la résolution des « bugs » ou l'intégration des versions, sans déstabiliser le code, pour ensuite resynchroniser ces changements après tous les tests. cf. [Wikipedia 2010, lien 2]

Il est certain qu'une parallélisation standardisée des différents tests, couplée à un environnement plus complexe permettrait d'exploiter au maximum la croissance de la performance des machines qui s'améliorent à une vitesse quasiment exponentielle (ce, même si les logiciels voient eux-aussi leur complexité augmenter).

Where do you see CI going in the future?

« ...I think there is much more we can make "automatic for the people". I foresee future development shops to look like a NASA or television control room including real-time monitoring of the software under development, and if anything goes beyond our identified thresholds, we are notified and can take immediate action. I see the future of CI increasing the efficiency and quality of software we develop for our users. »

Interview avec Paul Duvall, [Stelligent 2008]

7.2 Les alternatives

Git [Git 2010] est un SCV (système de contrôle de versionnement) développé initialement par Linus Torvalds (le créateur du système d'exploitation « libre » Linux) et sorti en 2005. L'une de ses caractéristiques principales est que, contrairement à la plupart de ses concurrents qui empruntent un modèle centralisé basé sur un seul et même serveur, Git est complètement décentralisé. Cette particularité implique que chaque collaborateur dispose d'une version complète du dépôt distant (et non plus de la dernière version uniquement) sur son ordinateur, appelé alors « dépôt local ». Une connexion internet n'est donc plus indispensable pour continuer à travailler. [maximebornemann.com 2010]

Git possède également de nombreux avantages, tels que :

- Création de branches : on peut fusionner ou créer des branches très facilement
- Transaction atomique : lorsque vous commitez des fichiers modifiés, Git s'assure qu'ils seront tous pris en compte. Si ce n'est pas le cas, le processus s'annule, permettant ainsi à votre dépôt d'éviter la corruption.
- La Staging Area (voir chapitre 5.1.2) : quand la plupart des VCS possèdent deux états pour leurs fichiers (commités ou modifiés), Git rajoute un état transitoire (staged). A la place de commiter vos fichiers une fois une modification effectuée, Git place un snapshot (qui fait référence à la copie actuelle de l'état d'un système [Wikipedia 2010, lien 6]) dans une zone intermédiaire (staging area, ou index). Vous pouvez donc continuer à modifier vos fichiers sans vous préoccuper d'un précédent commit. cf [Fedora 2010]

Maintenant que les spécificités de Git ont été explicitées, la question serait : en quoi ce logiciel pourrait être une alternative à l'IC? Tout simplement parce qu'il permet de mettre en place une sorte d'IC sans serveur distant, donc complètement gratuit. Il suffirait de dupliquer (Git Clone) le code-source du répertoire de travail vers un second répertoire depuis lequel le build serait lancé, et, en cas de succès, le commit serait effectué. De plus, avec Git Clone, Git copie les fichiers via des liens qui permettront ensuite de mettre à jour le répertoire très rapidement. Pour la resynchronisation du répertoire de travail « Git Pull » permet au répertoire courant de contenir exactement les mêmes fichiers et les mêmes commit que ceux qui ont été poussés sur le repository partagé, tout ça grâce au « hash » des fichiers. [Javabien.net 2010]

Cette solution telle qu'elle est présentée est véritablement prometteuse, puisqu'elle n'implique aucune maintenance, une exécution via une seule commande et plus d'erreur dans le repository, donc tous les avantages d'un système d'IC, le serveur en moins. Il faudrait néanmoins la tester pour identifier les probables désavantages qu'elle implique.

8 Environnements de développement de la BSU

La BSU (Börsenspiel der Schweizer Universitäten) est une association d'étudiants de l'Université de Fribourg qui propose chaque année une simulation de jeu boursier depuis 1992. Si le concept séduit en accueillant de plus en plus d'étudiants années après années (l'édition 2009 étant une exception), il faut admettre que le programme d'administration du jeu ainsi que l'architecture de ses bases de données n'ont pas beaucoup évolué ces 10 dernières années. L'objectif des années à venir pour la « Team IT » (équipe chargée de l'informatique dans l'organisation de la BSU) sera de faire avancer le jeu en proposant une nouvelle architecture, l'intégration de logiciels plus performants ou un nouveau processus de configuration du jeu, ceci afin d'améliorer les performances générales du jeu. C'est sur cette dernière proposition que se concentrera ce travail. Pour donner une idée des performances accrues qui découleraient de la mise en place de l'IC pour un futur projet de développement, il fallait un projet informatique en cours de développement voir terminé. C'est donc sur le travail de Master d'Hélène Martenet [Martenet 2009] que vont être concentrées toutes les étapes de cette partie pratique.

8.1 Situation sans IC

Actuellement les projets informatiques ne sont gérés que par un simple logiciel de contrôle de versionnement (Subversion sur le serveur mis à disposition par l'Université, et TortoiseSVN [TortoiseSVN, 2010] sur les ordinateurs de chaque développeur) (cf. Figure 8-1 : Situation de l'environnement de développement de la BSU) et un logiciel de travail collaboratif (Groove, issu de Microsoft Office, ou plus récemment, Wuala [Wuala, 2010] qui permet un stockage en ligne et développé par LaCie) afin d'échanger les informations sur la programmation, les délais et autres dates de réunion. Cette configuration pourrait suffire pour le développement d'un petit projet concernant 1 ou 2 personnes, mais si les membres de l'équipe planchent ensemble sur un programme plus complexe, il serait judicieux de revoir l'environnement.

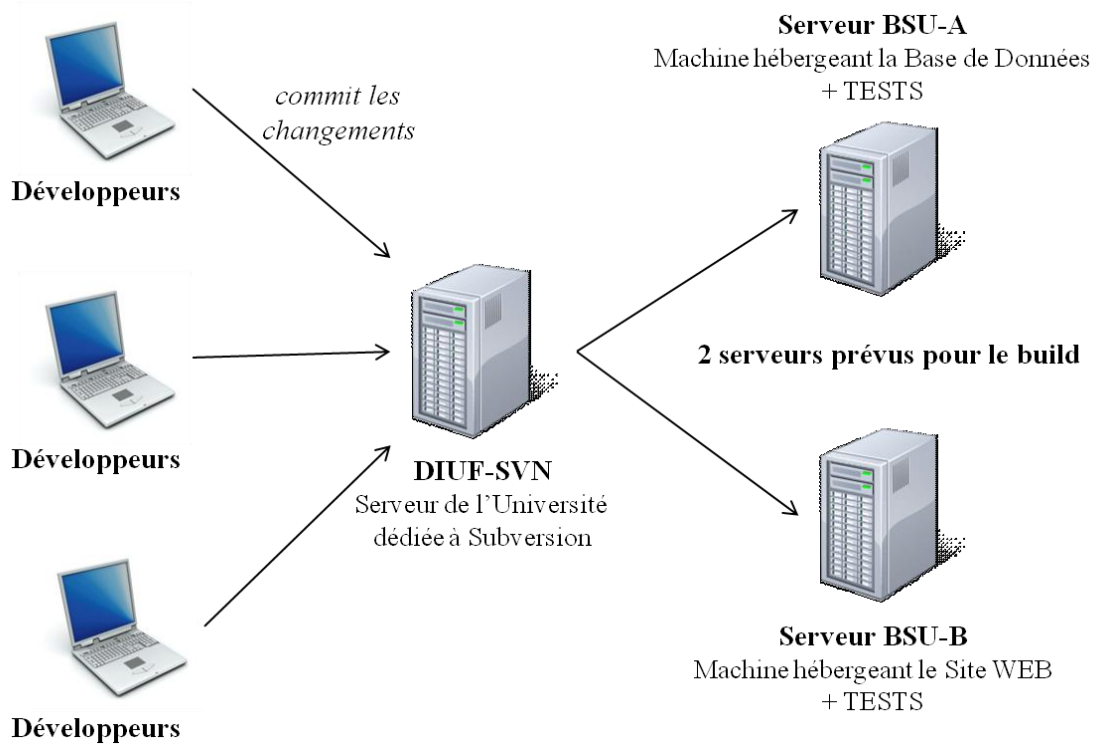


Figure 8-1 : Situation de l'environnement de développement de la BSU avant la mise en place d'un système d'IC

8.2 Situation après mise en place de l'IC

La mise en place d'une architecture différente pour l'IC destinée à optimiser la qualité et le temps de production final d'un projet pour le BSU a été l'objet de ce travail. La *Figure 8-2* donne un bon aperçu de ce à quoi l'architecture ressemble.

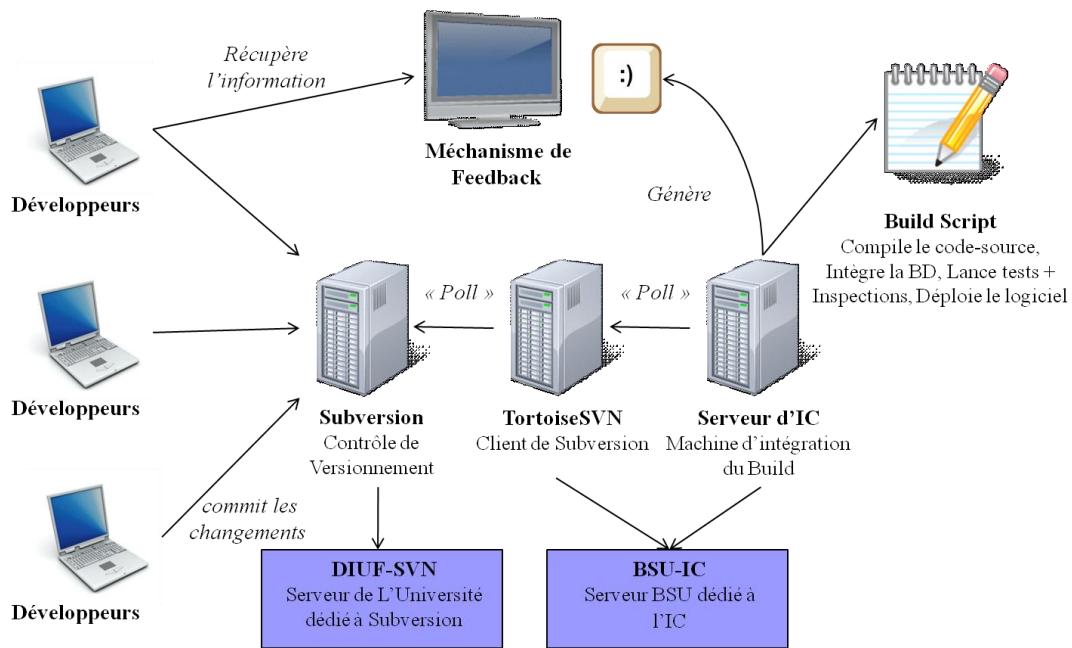


Figure 8-2 : Mise en place de l'IC pour la BSU

CruiseControl.NET (le serveur d'IC) ainsi que TortoiseSVN sont installés sur le serveur *BSU-IC*. Les développeurs auront également chacun TortoiseSVN installé sur leur poste de travail et feront un commit régulier de leur travail. Subversion ne sera donc pas installé sur la même machine que CC.NET (ce qui aurait pu être fait, vu que tous les pré-requis avaient été mis en place dans cette optique). Le choix d'un système de contrôle basé sur un simple test de modification du fichier se justifie par le fait que l'administration pour un système Serveur-Client comme Subversion est assez complexe à mettre en œuvre au niveau des autorisations d'accès (pare-feu), et vu que l'Université met à disposition un serveur dédié SVN [*DIUF-SVN 2010*], le choix s'est porté sur une autre solution. L'information nécessaire au fichier « .build » (le Build Script qui ordonne toutes les tâches, lancé par CC.NET et exécuté par NAnt) sera donc le chemin d'accès au dossier racine du projet, dossier géré par TortoiseSVN.

Le but était de mettre en place un système d'IC similaire à celui qu'une équipe de développement pourrait utiliser. Il était donc important que les « updates » de TortoiseSVN se fassent de manière automatique afin d'éviter de devoir se connecter sur le serveur *BSU-IC* pour le faire manuellement. Dans cette optique, un petit fichier .BAT sera exécuté par NAnt pour des updates automatique et ainsi vérifier si un fichier a été rajouté à Subversion sur le serveur *DIUF-SVN*, et le cas échéant, le télécharger via

TortoiseSVN afin que le nouveau build se fasse. L'intégration aura alors lieu ainsi que le mécanisme de feedback qui informera les développeurs de l'issue du build.

Il a été question dans les chapitres 2.3 et 2.4 des avantages et désavantages de la mise en place d'un système d'IC dans une équipe de développement. Ces avantages se retrouvent bien sûr pour des projets un peu moins ambitieux que ceux d'une équipe de programmation professionnelle. En prenant le cas de la BSU, qui compte 3 à 4 développeurs dans ses rangs, ses projets n'ont encore jamais inclus plus de 2 intervenants en même temps (pour ce qui est de ces deux dernières années du moins). Un des principaux atouts de l'IC étant qu'il permet de limiter les problèmes que peut poser l'interaction du travail, il est certain que la BSU n'exploitera pas tout le potentiel d'amélioration que peut apporter un système d'IC. La mise en place d'un tel système aide néanmoins à mieux visualiser son propre travail et celui des autres, à assumer ses erreurs et à connaître avec précision l'avancée du projet. Enfin, ces changements seront précieux lorsqu'il s'agira de rendre la version finale du programme.

8.3 Analyse SWOT

L'analyse SWOT (de l'anglais Strengths (forces), Weaknesses (faiblesses), Opportunities (opportunités), Threats (menaces)), ou MOFF (Menace, Opportunités, Forces et Faiblesses) pour les francophiles, est un outil de stratégie d'entreprise couramment utilisé lorsqu'il est nécessaire d'analyser une situation présente d'une manière structurée. [Atkinson 1998]

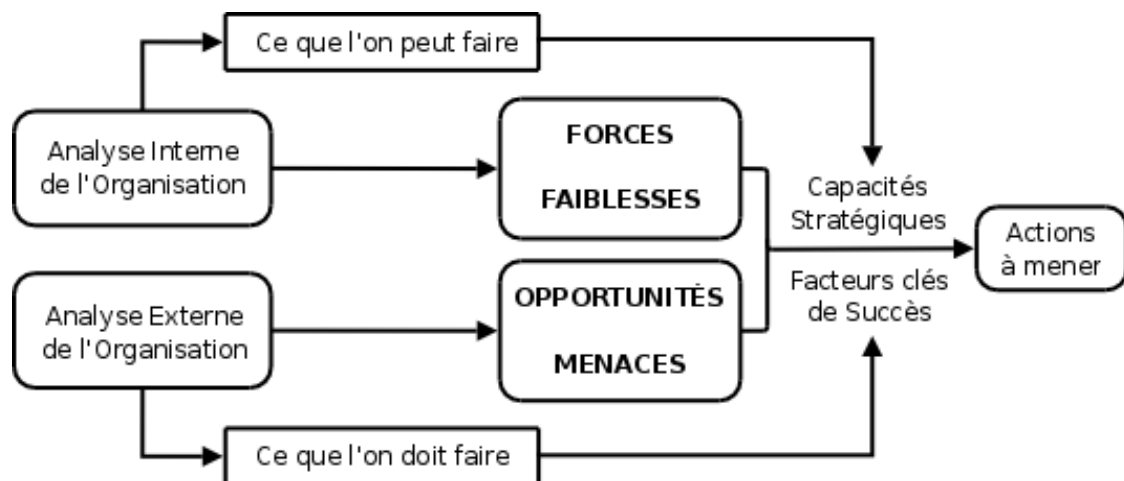


Figure 8-3 : Cheminement de l'analyse SWOT [GroupeRessources.com 2010]

SWOT donne accès à l'identification des forces et des faiblesses du projet en regard des opportunités et des menaces de l'environnement externe. La méthode vise à identifier les stratégies maximisant le potentiel de forces et d'opportunités, tout en minimisant l'impact des faiblesses et des menaces. Elle constitue ainsi une charnière entre analyse et diagnostique. [COTA 2006]

Voici les 4 plans de la matrice SWOT et leurs significations dans ce contexte :

Forces : Il s'agit des avantages compétitifs que l'association BSU peut acquérir grâce à la mise en place du système d'IC, ceci en regard de l'ancien environnement de développement.

Faiblesses : Il s'agit des désavantages à la mise en place du système d'IC auxquels l'association BSU devra faire face, ceci en regard de l'ancien environnement de développement.

Opportunités : Il s'agit du domaine d'action dans lequel la BSU peut espérer jouir d'un avantage différenciant grâce à l'IC (par rapport à l'ancienne solution).

Menaces : Il s'agit des perturbations de l'environnement de la BSU qui ont des chances de se réaliser suite à la mise en place de la plateforme d'IC.

Forces

- *Réduit les risques*

Grâce aux tests et aux inspections déclenchés dès le commit du fichier dans le repository, on réduit les risques d'un programme corrompu.

- *Meilleure visualisation*

Grâce aux feedbacks immédiats après le build, les développeurs (et autres intéressés) ont une visualisation parfaite de l'état du projet ainsi que de son avancement.

- *Gain de temps en générant un logiciel déployable (éviter l'Integration Hell)*

Le gain de temps sera surtout perceptible en fin de projet. Les développeurs plus aguerris, ayant déjà eu affaire à une intégration tardive, pourront en témoigner : repousser l'échéance du build amène souvent à de mauvaises surprises.

- *Obtenir un code fiable*

De nombreux outils développés pour l'IC permettent d'éviter la duplication du code, d'appliquer un standard dans la convention du code, etc. et donc d'optimiser la fiabilité du code (il s'agit en général de paramétrer dans le logiciel, les critères qui semblent importants pour le développeur).

- *Meilleure qualité*

Comme définit dans le glossaire au chapitre 10, la qualité est en fait la pondération de spécificités telles que l'extensibilité, la sécurité, la performance, la capacité à être maintenu et la lisibilité. L'IC mise en place pour la BSU permettra d'obtenir un programme de meilleure qualité.

Faiblesses

- *Script NAnt peu intuitif*

NAnt est un programme difficile à intégrer. Son utilisation dans un projet utilisant des technologies du Framework 3.5 impliquent des problèmes de compatibilité à foison. De plus, il est assez peu intuitif et sa prise en main prendra logiquement plus de temps pour l'approprier complètement.

- *Mise en place rébarbative*

La mise en place d'un système d'IC pour un projet demande de la patience pour les profanes. Peu de fichiers sont à développer mais il faut quand même du temps pour se familiariser avec ces nouveaux outils. La *migration* (voir point suivant) est également assez fastidieuse, il serait donc judicieux de commencer son projet directement avec l'IC, et non pas de l'intégrer par la suite.

- *Migration*

La coordination qu'implique le passage d'un gros projet sur une plateforme d'IC est souvent démotivante, d'autant plus que ces projets sont souvent répartis dans de nombreux environnements de développement (voir le chapitre 2.4 sur les désavantages de l'IC).

- *Beaucoup plus de règles de codage*

Auparavant dans la BSU, aucune règle ne régissait le développement d'un code. Avec l'IC, pour que CC.NET lance sa routine de tests (par exemple), il faut en écrire ! C'est une lapalissade que de le mentionner, mais beaucoup de règles sont indispensables pour profiter des avantages que procurent l'IC. Même si on peut considérer ça comme une « faiblesse » dans un premier temps, il est certain que ces habitudes de développement, aussi contraignantes soient-elles, doivent être perçues comme une « force » dans le futur.

- *Lenteur d'exécution du build*

Il s'agit d'un problème récurrent pour les builds de projets importants. De nombreuses techniques pour optimiser le temps du build sont proposées au chapitre 5.2 (dans le cas où celui-ci s'éterniserait). En ce qui concerne la BSU, le problème pourrait être d'autant plus flagrant que le build implique dans son processus un check-out de TortoiseSVN (le client de Subversion).

Opportunités

- *Motivation supplémentaire*

La mise en place d'un serveur d'IC ainsi qu'un serveur dédié au build permettra certainement un regain de motivation quant à la création de nouveaux logiciels pour l'association BSU. La nouvelle infrastructure ainsi que le guide d'utilisation permettront de mieux préparer le terrain pour les développeurs de la BSU.

- *Découverte d'un nouveau concept*

Ce nouvel outil de travail permettra aux développeurs d'élargir le spectre de leurs expériences en matière de développement.

- *Confiance des développeurs*

Peur de commiter un code corrompu, ou, au contraire, peur de récupérer un code corrompu développé par un collègue ? Ce problème n'existe plus avec le serveur d'IC de la BSU. Cet état de fait contribue à donner confiance aux développeurs et à conforter l'esprit d'équipe. Ceux-là n'hésiteront plus à apporter leur contribution au projet.

- *Acquérir de bonnes habitudes de développement*

Chacun à sa manière de coder, mais le développement en équipe implique quelques règles de « bonne conduite », de telle sorte qu'un programmeur puisse reprendre le travail de son collègue sans avoir à lui demander des explications. Le développement via un système d'IC incite les développeurs à acquérir de nouveaux réflexes en matière de développement.

Menaces

- *Aversion au changement*

Changer d'habitude de programmation n'est pas chose aisée. Il existe de nombreux développeurs ayant testé l'IC qui ont préféré revenir à une programmation « traditionnelle ».

- *Problèmes de compatibilité au Framework 3.5*

Les programmes nécessaires au bon fonctionnement de l'IC étant déjà installés, cette « menace » ne devrait pas vraiment en être une. Néanmoins, plusieurs outils n'ont pas été intégrés dans l'architecture d'IC actuelle. Chaque nouvelle installation révélant son lot de surprise, il faudrait dès lors s'attendre à des problèmes à ce niveau. De même, l'installation automatique des Updates de Windows implique le chargement des Frameworks les plus récents (le numéro 4 est déjà disponible). Ceux-ci pourraient à nouveau engendrer des problèmes de compatibilité.

- *Choix technologiques de l'équipe IT de la BSU*

CruiseControl.NET est un outil dédié au langage .NET. Ce langage n'étant pas des plus utilisés dans le milieu étudiant de l'Université de Fribourg (aucun cours ne traite de ce langage), il est probable que Java (ou un autre) lui soit préféré. Il faudrait alors remettre en place un nouveau système d'IC avec des programmes différents (CruiseControl et Ant par exemple).

- *Nouveaux systèmes simplifiés*

L'avenir de l'IC est développé au chapitre 7.1. Il y est proposé certaines voies d'amélioration qu'ont entrepris (ou pourraient entreprendre) les développeurs. L'une d'elle, Git, est particulièrement intéressante et pourrait peut-être bien suppléer le système mis en place pour la BSU dans un avenir proche.

- *Abandon du concept*

Existant depuis 1994, la BSU est une petite association avec peu de développeurs. Le programme PMS étant à la base du jeu n'a, par exemple, pas été retouché depuis plus de 10 ans. On pourrait s'imaginer que l'évolution de l'IT (Information Technology) demeure au statu quo avec quelques évolutions anecdotiques, auquel cas le concept d'IC ne serait évidemment plus d'actualité. Néanmoins, les projets informatiques planifiés pour ces prochaines années laissent présager des changements importants. L'IC de la BSU aura alors de beaux jours devant elle.

L'analyse SWOT du système d'IC pour la BSU permet de rendre compte des avantages et désavantages inhérents à son installation. Il est par contre très difficile de juger quelle sera l'évolution de son environnement. Toutefois cette analyse, bien qu'assez subjective, permettra sans doute de mieux cibler les différents scénarios auxquels la plateforme d'IC de la BSU pourrait faire face.

9 Conclusion

Nous avons pu constater que malgré tous les outils mis à disposition pour réaliser une IC efficace, cela reste un concept ou plutôt une attitude à adopter. Toutes les étapes peuvent parfaitement être entreprise de façon manuelle et/ou avec une architecture différente afin de se passer des outils mentionnés au paragraphe 6 . Pour être cohérent avec cette affirmation, on peut simplement considérer une autre façon de percevoir CruiseControl .NET : nous l'avons décrit comme un serveur d'intégration, ce qu'il est, maintenant un ordinateur externe qui viendrait récupérer la dernière version du code sur le repository et s'assurerait que ce code soit viable pour le build aurait le même résultat. Cette « machine d'intégration dédiée » possède les mêmes avantages que CruiseControl, l'automatisation en moins.

L'objectif de cette digression est de mettre en avant le rôle de l'IC. Son origine ne provient que d'une seule et unique règle des projets XP et Agile : « le code-source du repository doit passer tous les tests, et le build doit se faire ». A partir de là, on comprend que tous les moyens sont bons pour arriver à ses fins, sachant que toute sa structure est parfaitement modulable. Tous les outils évoqués dans ce travail sont donc utiles mais pas indispensable, l'important étant que l'équipe de développement prennent de nouvelles habitudes de programmation. De plus, la quantité/qualité de l'information ainsi que des programmes dédiés à l'IC se sont grandement développés durant l'écriture de ce travail, d'autant que celui-ci a été rédigé sur une période plus importante que prévue. Ceci constitue le meilleur exemple des efforts mis en place par les sociétés informatiques afin de proposer des outils d'IC optimaux.

Concernant la BSU, l'objectif de chaque membre actif de cette association est d'acquérir un peu d'expérience « de la vie professionnelle » et d'appliquer ses connaissances sur un projet commun : pour des étudiants en informatique, le fait de disposer d'un outil collaboratif aussi performant permettra sans doute d'oser se lancer dans de nouveaux projets, en équipe, d'acquérir de nouveaux réflexes de programmation (Agile), de ne plus appréhender de retrouver un logiciel « buggé » par un de ses collègue ou au contraire de « commiter » son travail sur Subversion, et surtout d'avoir une vision globale du projet et de son avancée.

D'un point de vue plus personnel, l'écriture de ce travail m'a permis de connaître un concept qui m'était encore inconnu et qui me permettra sans aucun doute de mieux appréhender les dangers de l'intégration durant ma vie professionnelle. Les différents langages ou les configurations (qu'elles soient système ou liées aux programmes) m'ont également permis de percevoir au mieux toutes les relations qu'exigent l'écriture d'un programme à grande échelle : m'étant toujours borné à des petites applications ne nécessitant en aucune façon l'aide de tierces personnes, il était essentiel de comprendre l'importance des outils de travail collaboratif.

10 Glossaire et Table d'abréviation

Voici un petit récapitulatif des termes et abréviations qui ont jalonné ce travail.

10.1 Glossaire

Version Control Repository (ou Repository) / Version Control System

Un SCV (système de contrôle de versionnement) est un élément indispensable pour la création d'un logiciel. Il s'agit en fait d'un dossier commun dans lequel résident tous les codes-source, le but étant de pouvoir y accéder en un point central afin de pouvoir récupérer la version la plus récente du code, ou revenir dans le temps si une nouvelle version est corrompue. Les logiciels de contrôle de versionnement (VCS, Version Control System) fonctionnent justement grâce au concept de repository et donnent la possibilité à une ou plusieurs ressources informatiques de stocker leurs données afin de les labéliser et de pouvoir, par la suite, identifier et récupérer toutes les versions intermédiaires des ressources. Le SCV s'occupe donc des mises à jour du code-source et conserve les différentes versions lors d'un changement, la nouvelle version étant simplement incrémentée de 1 (lié au développement linéaire) par rapport à la précédente (révision). Ce même concept sera aussi utile (et utilisé) lors des builds. De nombreux systèmes permettant le contrôle des versions (ou « versionnement ») sont disponibles (comme Subversion ou TortoiseSVN). [Wikipedia 2010, lien 3]

Automatisé:

On parle d'automatisation lorsque le processus est complètement automatisé et ne requiert pas d'intervention d'un utilisateur. En anglais, on parle d'un processus « headless » ou « hands-off ». cf. [Duvall 2007, p. 27]

Build:

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

Environnement de développement

Il s'agit de l'environnement dans lequel le programme est écrit. Le plus souvent on parle d'un IDE (Integrated Development Environment) comme Eclipse ou Visual Studio, mais

les bibliothèques, les fichiers de configuration, les serveurs ou les « builds scripts » en font aussi partie. *cf. [Duvall 2007, p. 28]*

Intégration

C'est l'acte de combiner des parties de codes-source séparées, pour déterminer comment elles fonctionnent comme un tout. *cf. [Duvall 2007, p. 28]*

Inspection

Il s'agit de l'analyse du code-source pour les attributs de la qualité interne. On considérera les aspects automatiques (analyses statiques et du temps d'exécution) comme une inspection de logiciel. *cf. [Duvall 2007, p. 28]*

JDK (Java Development Kit)

Le Java Development Kit (JDK) est l'environnement dans lequel le code Java est compilé pour être transformé en bytecode afin que la JVM (machine virtuelle de Java) puisse l'interpréter. *cf. [Java.Net 2010]*

Qualité

La qualité est une valeur donnée aux différentes spécificités du produit. Ce terme est très subjectif et peut varier en fonction des priorités de chacun et de la pondération qu'on attribue à ces spécificités. Parmi elles on trouve : l'entretien, la sécurité, les performances, l'extensibilité, les tests opérés durant la conception du programme, etc. *cf. [Duvall 2007, p. 28]*

Risque

Le risque est la potentialité qu'une erreur survienne. On observera que la plupart des programmeurs les classent en fonction d'un rapport entre leurs typologies (degrés de « dangerosité ») et d'une estimation de leur probabilité d'apparition. On peut ainsi réduire les risques en suivant un schéma basique : d'analyse du risque, de réduction/prévention et de suivi. *cf. [CNRS 2010]*

10.2 Table d'abréviation

Acronyme, sigle ou abréviation	Signification
BD	Base de Données
BSU	Börsenspiel der Schweizer Universitäten
CCNET	CruiseControl .NET
Cf	confer
DBL	Data Base Administrator
DDL	Data Definition Language
DML	Data Manipulation Language
IC	Intégration Continue
IDE	Integrated Development Environment
IIS	Internet Information Services
IT	Information Technology
JVM	Java Virtual Machine
SCV	Système de contrôle de versionnement
SVN	Subversion
SWOT	Strengths Weaknesses Opportunities Threats
URL	Uniform Resource Locator
XML	Extensible Markup Language
XP	eXtreme Programming
VS	Visual Studio

11 Bibliographie

[**AnthillPro 2009**], Serveur d'IC payant, <http://www.anthillpro.com/blogs/>, <http://www.anthillpro.com/html/resources/build-pain-relief/notification.html> (consulté le 28.05.2010)

[**Appleton 1998**], Brad Appleton, Stephen Berczuk, Ralph Cabrera, and Robert Orenstein, 1998: Streamed Lines: *Branching Patterns for Parallel Software Development*, <http://www.cmcrossroads.com/bradapp/acme/branching/> (consulté le 20.05.2010)

[**ASP.NET Forum 2005**], Forum d'aide pour ASP.NET, <http://forums.asp.net/p/869604/869604.aspx> (consulté le 08.06.2010)

[**Atkinson 1998**], Kim Atkinson, 1998: *SWOT analysis: a tool for continuing professional development*, *International Journal of Therapy and Rehabilitation*

[**Better SCM 2010**] , Site de comparaisons de système de contrôle de versionnement, <http://better-scm.berlios.de/comparison/comparison.html> (consulté le 09.06.2010)

[**Blog Développeur 2009**], Partage d'informations entre développeurs, <http://blogs.developpeur.org/redo/archive/2004/08/16/2286.aspx> (consulté le 28.09.2009)

[**BuildMeister 2009**], Site qui propose une présentation des outils pour le build, <http://www.buildmeister.com/> (consulté le 24.05.2009)

[**CFTL 2009**], Comité français des tests logiciels, définitions, <http://www.cftl.net/cms/files/Dokumente/frGlossaire%20des%20tests%20de%20logiciel%20-%20ISTQB.pdf> (consulté le 26.05.2009)

[**CM Crossroad 2009**], Crossroads the Configuration Management Community, <http://www.cmcrossroads.com/>, <http://www.cmcrossroads.com/cgi-bin/cmwiki/view/CM/AnthillPro> (consulté le 20.05.2010)

[**CNRS 2010**], Centre National (français) de Recherches Scientifiques, <http://www.dsi.cnrs.fr/conduite-projet/phasedefinition/qualite/risques/basdefqual.htm> (consulté le 07.06.2010)

[CodeBetter 2009], Partage d'informations entre développeurs, <http://codebetter.com/blogs/jeffrey.palermo/archive/2007/11/28/upgrade-nant-for-use-with-vs2008-solutions-and-net-3-5.aspx> (consulté le 16.09.2009)

[Codicesoftware, 2008], Site officiel de Codice Software, créateur de logiciels de développement, <http://codicesoftware.blogspot.com/2008/03/continuous-integration-future.html> (consulté le 26.07.2009)

[COTA 2006], Site d'une ONG de droit belge : article sur l'analyse SWOT, http://www.cota.be/SPIP/IMG/pdf/Fiche_1A_GCP_SWOT.pdf (consulté le 14.09.2010)

[Crossroads 2004], Magazine informatique pour le management de la configuration, <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.83.6098&rep=rep1&type=pdf> (consulté le 26.05.2010)

[Développez.com 2009], Site d'aide aux développeurs, <http://linsolas.developpez.com/articles/java/outils/builds/#LV-A-1> (consulté le 29.08.2010)

[Di-ENS-Labatut 2007], Pour les tests unitaires et la couverture des codes, <http://www.di.ens.fr/~labatut/ED6/cours-6.pdf> (consulté le 20.05.2009)

[DIUF 2010], Site du département d'informatique de l'Université de Fribourg, <https://diuf.unifr.ch/wiki/Subversion> (consulté le 16.09.2009)

[DIUF-SVN 2010], Serveur SVN de l'Université de Fribourg, <http://diuf-svn.unifr.ch> (consulté le 18.05.2010)

[Doxygen 2009], Site officiel du générateur de documents, <http://www.stack.nl/~dimitri/doxygen/features.html> (consulté le 26.05.2010)

[Duvall 2007], Paul M. Duvall, 2007: *Continuous Integration, Improving Software Quality and Reducing Risk, 1st Edition*. Addison-Wesley, Pearson Education 2007.

[Duvall WS 2009], Support Internet du livre de Paul Duvall, <http://integratebutton.com> (consulté le 20.05.2009)

[Extreme Programming 2009], Site officiel sur l'Extreme Programming, <http://www.extremeprogramming.org/> (consulté le 20.05.2009)

[Fedora 2010], Site de la communauté francophone de Fedora, système d'exploitation « libre » basé sur Linux, <http://doc.fedora->

fr.org/wiki/Gestion_et_contr%C3%B4le_de_versions_avec_Git (consulté le 13.09.2010)

[**Fowler 2008**], Site officiel de Martin Fowler, <http://martinfowler.com/articles/continuousIntegration.html> (citation p.2/12, consulté le 29.05.2009)

[**Git 2010**], Site officiel du logiciel Git, <http://git-scm.com/> (consulté le 08.09.2010)

[**GroupeRessources.com 2010**], Site d'une agence de placement, article sur l'analyse SWOT, <http://www.grouperessources.com/index.php?p=Offre2> (consulté le 14.09.2010)

[**Java.Net 2010**], Informations sur la technologie Java, <https://jdk.dev.java.net/> (consulté le 07.06.2010)

[**Javabien.net 2010**], Blog concernant Java et Agile, <http://blog.javabien.net/tag/hudson/> (consulté le 12.09.2010)

[**Knoernschild 2008**], Kirk Knoernschild, 2008: Continuous Integration: The Project Heartbeat, <http://www.burtongroup.com/research/PublicDocument.aspx?cid=1500> (accès « Client » indispensable pour visualiser ce document, consulté le 05.06.2009)

[**Larousse 2010**], Dictionnaire Larousse en ligne, <http://www.larousse.fr/dictionnaires/francais/compiler> (consulté le 11.05.2010)

[**Martenet 2009**], Hélène Martenet, 2009 : *Analyse des exigences d'un jeu boursier en ligne et implémentation d'un prototype d'amélioration. Travail de Master, Université de Fribourg.*

[**Maven, 2010**], Site officiel d'information et de téléchargement du logiciel Maven, <http://maven.apache.org/> (consulté le 29.08.2010)

[**maximebornemann.com, 2010**], Site/Blog sur les nouvelles technologies, <http://maximebornemann.com/introduction-au-contrôle-de-version-avec-git> (consulté le 08.09.2010)

[**MSDN 2010**], Ressources Microsoft pour technologie .NET, <http://msdn.microsoft.com/fr-fr/library/a99txfy5%28VS.80%29.aspx> (consulté le 11.05.2010)

[**NUnit 2010**], Site officiel de téléchargement du logiciel NUnit, <http://www.nunit.org/> (consulté le 26.07.2010)

[**Optus.net 2007**], Site proposant des tutoriaux, <http://mirror.optus.net/sourceforge/t/to/tortoisesvn/TortoiseSVN-1.4.5-fr.pdf> (consulté le 20.05.2009)

[**Page Brooks 2008**], Partage d'informations entre développeurs, <http://pagebrooks.com/archive/2008/05/20/using-nant-with-.net-3.5.aspx> (consulté le 16.09.2009)

[**Refactoring 2009**], <http://www.refactoring.com/catalog/extractMethod.html> (consulté le 06.05.2009)

[**rkcole 2009**], Code Metrics: Cyclomatic Complexity and Unit Tests, <http://www.rkcole.com/articles/other/CodeMetrics-CCN.html> (consulté le 05.05.2009)

[**SourceForge 2010**], Site pour le téléchargement du serveur d'IC CC.NET, <http://cruisecontrol.sourceforge.net/download.html> (lien 1, consulté le 18.05.2010), Site pour le téléchargement de Nant, <http://nant.sourceforge.net> (lien 2, consulté le 19.05.2010)

[**Stelligent 2008**], Site officiel de Stelligent, serveur d'IC payant, <http://www.stelligent.com/content/view/54/71/1/2/> (consulté le 16.05.2009)

[**TeamCity, 2009**], Site officiel de TeamCity, serveur d'IC payant, http://www.jetbrains.com/teamcity/delayed_commit.html (consulté le 08.06.2009)

[**Test early 2009**], Site/blog pour le Management des Builds et l'IC, <http://www.testearly.com/>

[**TortoiseSVN 2010**], Site officiel d'informations et de téléchargement de TortoiseSVN, <http://tortoisesvn.net/about> (consulté le 09.06.2010)

[**ThoughtWorks 2010**], Site de partage d'informations, axé IC, <http://confluence.public.thoughtworks.org/display/CC/CI+Feature+Matrix> (consulté le 19.05.2010)

[**Wikipedia 2010**], Encyclopédie « libre », http://fr.wikipedia.org/wiki/Internet_Information_Services (lien 1, consulté le 11.05.2010), http://en.wikipedia.org/wiki/Branching_%28software%29 (lien 2, consulté le 20.05.2010), http://fr.wikipedia.org/wiki/Syst%C3%A8me_de_gestion_de_versions (lien 3, consulté le 07.06.2010), [http://fr.wikipedia.org/wiki/Subversion_\(logiciel\)](http://fr.wikipedia.org/wiki/Subversion_(logiciel)) (lien

4, consulté le 11.05.2010) , http://fr.wikipedia.org/wiki/Apache_Maven (lien 5, consulté le 29.08.2010), http://en.wikipedia.org/wiki/Snapshot_%28computer_storage%29 (lien 6, consulté le 14.09.2010)

[**Wuala 2010**], Site officiel d'informations et de téléchargement de Wuala, <http://www.wuala.com/> (consulté le 25.08.10)