Livre Blanc

L'importance des modèles de branching dans la gestion des configurations logicielles

IdeoLogiciels
www.ideologiciels.com

Parmi les modèles de branching utilisés dans la gestion des configurations logicielles, le modèle ***branch-by-purpose*** *offre un meilleur support aux efforts de développements en parallèle et améliore non seulement le contrôle sur les versions planifiées mais aussi sur les versions produites en urgence.*

Par *Chuck Walrad*, Davenport Consulting
Et *Darel Strom*, Expert Support

Si vous désirez améliorer la qualité de votre logiciel, vous devez d'abord le comprendre. Quels sont ses composants ? Comment sont-ils organisés et liés les uns aux autres ? Si vous ne comprenez pas votre code source, vos chances de le faire évoluer sans casser quelque chose sont faibles.

Trop souvent, nous voyons des projets capituler en essayant de générer une version pour l'équipe de test, ou pour les clients à des fins de tests de recette. Les développeurs ont travaillé fiévreusement pour intégrer toutes les fonctionnalités, mais le processus échoue au stade de l'intégration. Les composants logiciels ne sont pas cohérents entre eux, et certains composants peuvent manquer. De mauvaises versions sont distribuées, et des défauts précédemment corrigés réapparaissent sans explication.

Pourquoi les organisations n'ont-elles pas un meilleur contrôle sur leurs logiciels ? Le problème dépend-il du volume du code et de sa complexité, est-il inhérent aux efforts de développement en parallèle, ou est-ce simplement le résultat de la rotation du personnel ou de raccourcis pris sous la pression des plannings ?

Tous ces facteurs peuvent contribuer à cette situation, mais le vrai problème vient d'une incompréhension fondamentale de la gestion de configuration logicielle, telle qu'elle s'applique au monde du développement d'applications.

La Gestion de Configuration Logicielle expliquée

La gestion de configuration logicielle a deux objectifs différents :

- Support au management pour contrôler les modifications apportées aux produits logiciels. Cette fonction inclut les activités traditionnellement associées à la gestion de configuration logicielle[1-3] – spécifiquement, identifier les composants logiciels, contrôler les changements qui sont réalisés sur ceux-ci, enregistrer et tracer l'état des configurations et des composants, et mener des audits et des revues.
- Support au développement pour coordonner les modifications sur les fichiers entre les développeurs.[4,5] Ces activités incluent l'identification des versions de fichiers, la construction des logiciels et la gestion des révisions.

Le branching fait partie intégrante de la gestion de versions, de la construction précise des logiciels, et de la gestion des révisions. Il autorise le développement en parallèle d'un nouveau système, et fournit le support concurrent à des révisions multiples en affectant un label à chaque instance d'un item de configuration appartenant à une branche, et en établissant une relation entre le label et les révisions du module, tel que décrit dans l'encadré *Glossaire de la gestion de configuration logicielle.*

Les bonnes décisions sur l'opportunité et l'utilité de créer une branche peuvent être facilitées pour les développeurs et les responsables de versions, afin de coordonner les modifications sur les produits. Une bonne stratégie de branching facilite la fourniture du bon code, la re-création de versions anciennes, et – si nécessaire – le retour en arrière vers une version antérieure.

Adopter le bon modèle de branching facilite un développement rapide, améliore la qualité globale du produit et l'efficacité du processus, réduit l'impact des dysfonctionnements logiciels, et améliore la performance organisationnelle.

Glossaire de la gestion de configuration logicielle

Quelques définitions de base sont utiles pour acquérir la connaissance fondamentale de la gestion de configuration logicielle dans le monde du développement d'applications.

Baseline >> En développement de logiciel, les standards IEEE définissent une baseline comme « *la spécification ou le produit qui a été revu et accepté, qui sert alors de référence pour les développements futurs, et qui peut être modifié seulement au travers de procédures de contrôles formalisées.*» [1] Une baseline peut également être définie comme un « *ensemble de composants logiciels formellement désignés, et identifiés à une date précise durant le cycle de développement du logiciel* » ou peut « *référer à une version particulière du logiciel sur laquelle les intervenants se basent. Dans tous les cas, la baseline peut seulement être changée en respectant des procédures de contrôles de modifications formalisées.*» [2]

Branche >> « Une branche est un accord sur la division d'un objet (composant, produit ou système) en itérations multiples identifiant chaque instance de composant, produit ou système, fournissant une correspondance exacte entre un label de version et les révisions de modules. »[3]

Gestion de configuration ou Gestion de configuration logicielle >> (1) « *La gestion de configuration logicielle identifie la configuration d'un logiciel à des dates données, en contrôlant systématiquement les changements à la configuration, et en maintenant l'intégrité et la traçabilité de la configuration tout au long du cycle de vie du logiciel. Les éléments placés sous contrôle de configuration incluent les produits logiciels livrés aux clients et les éléments qui les accompagnent ou qui servent à les générer.* »[4] Ceci comprend les outils utilisés pour créer, tester et maintenir le produit. (2) « *Une discipline définissant une direction administrative et technique, ainsi qu'une procédure de surveillance pour : identifier et documenter les caractéristiques physiques et fonctionnelles d'un composant de configuration, contrôler les changements apportés à ces caractéristiques, enregistrer et reporter le traitement des modifications et l'état d'implémentation, et vérifier la conformité avec les besoins spécifiés.* »[1]

Révision >>La distribution d'une configuration logicielle en dehors de l'équipe de développement. Ceci inclut les révisions internes, aussi bien que les distributions commerciales.[5]

Ingénierie de version >>Processus de gestion des révisions et des versions tout au long du cycle de vie du logiciel, et fourniture du produit fini dans le format et sur le média appropriés.

Gestion de version >> Identification, packaging, et livraison des éléments du produit tels que l'exécutable, la documentation, les notes de versions et les données de configuration.[5]

Composant de configuration logicielle >> Un composant de configuration logicielle agrège des objets identifiés et les traite comme une entité unique dans le processus de gestion de configuration logicielle.[1]

Version >> (1) « *Une révision initiale ou la révision d'un composant de configuration logicielle, associé à une compilation ou une recompilation complète du composant de configuration logicielle.* »[1] (2) « *Une révision initiale ou complète d'un document, à l'inverse d'une révision résultant de la modification des pages d'une révision précédente.* »[1] (3) « *Un élément logiciel particulier, identifié et spécifié.* »[5]

Références

- 1. *IEEE Std. 610.12-1990, Standard Glossary of Software Engineering Terminology, IEEE Press, Piscataway, N.J., 1990.*
- 2. *Software Engineering Body of Knowledge, trial version, IEEE Press, Piscataway, N.J., 2001, p. 108.*
- 3. *M. Ben-Menachem, Software Configuration Guidebook, McGraw Hill, Maidenhead, Berkshire, UK, 1994.*
- 4. *M. Paulk, et al., Key Practices of the Capacity Maturity Model, Version 1.1, SEI-93-TR-25, Software Eng. Inst., Carnegie Mellon University, Pittsburgh, 1993.*
- 5. *Software Engineering Body of Knowledge, trial version, IEEE Press, Piscataway, N.J., 2001, p. 111.*

Le modèle de branching

Un modèle de branching reflète la logique adoptée pour la réplication d'un élément de configuration – que ce soit un module de programme ou un sous-système – en instanciations multiples, chacune d'elle supportant son label d'identification propre et unique.

La sélection d'un modèle de branching approprié permet au responsable de version de satisfaire plusieurs intervenants qui ont souvent des intérêts ou des priorités différents : l'équipe de développement, l'équipe de test, et l'équipe support (qui représente l'utilisateur final du produit). Pour déterminer la justesse d'un modèle de branching, nous évaluons ses capacités à :

- maintenir une base stable pour les nouveaux développements, en supportant l'intégration continue ou quotidienne, par des régénérations globales,
- délivrer des versions d'urgences – qui contiennent toutes les corrections nécessaires et aucune autre modification – aux équipes de test et aux clients,
- tester des versions qui contiennent toutes les corrections nécessaires, et aucune autre modification,
- minimiser l'impact des versions d'urgence sur les nouveaux efforts de développement,
- revenir en arrière, sur une version précédente si nécessaire,
- supporter des versions séquentielles multiples
- supporter des versions concurrentes multiples, telles que des versions alternatives pour des plateformes ou des clients différents.

Le modèle « Branch-by-release »

La sagesse conventionnelle et les standards existants nous incitent à gérer la configuration logicielle comme une série de baselines successives.[2,3] Le modèle branch-by-release de gestion de code instancie cette approche. Dans ce modèle conventionnel, le code est branché lors d'une décision de produire une nouvelle version du produit. La nouvelle branche sert de baseline pour continuer le développement. Comme le montre la figure 1, la branche ancienne contient la version produite – la baseline historique actuelle utilisée comme point de référence. Cette branche est laissée de coté, et disparaîtra peu à peu.

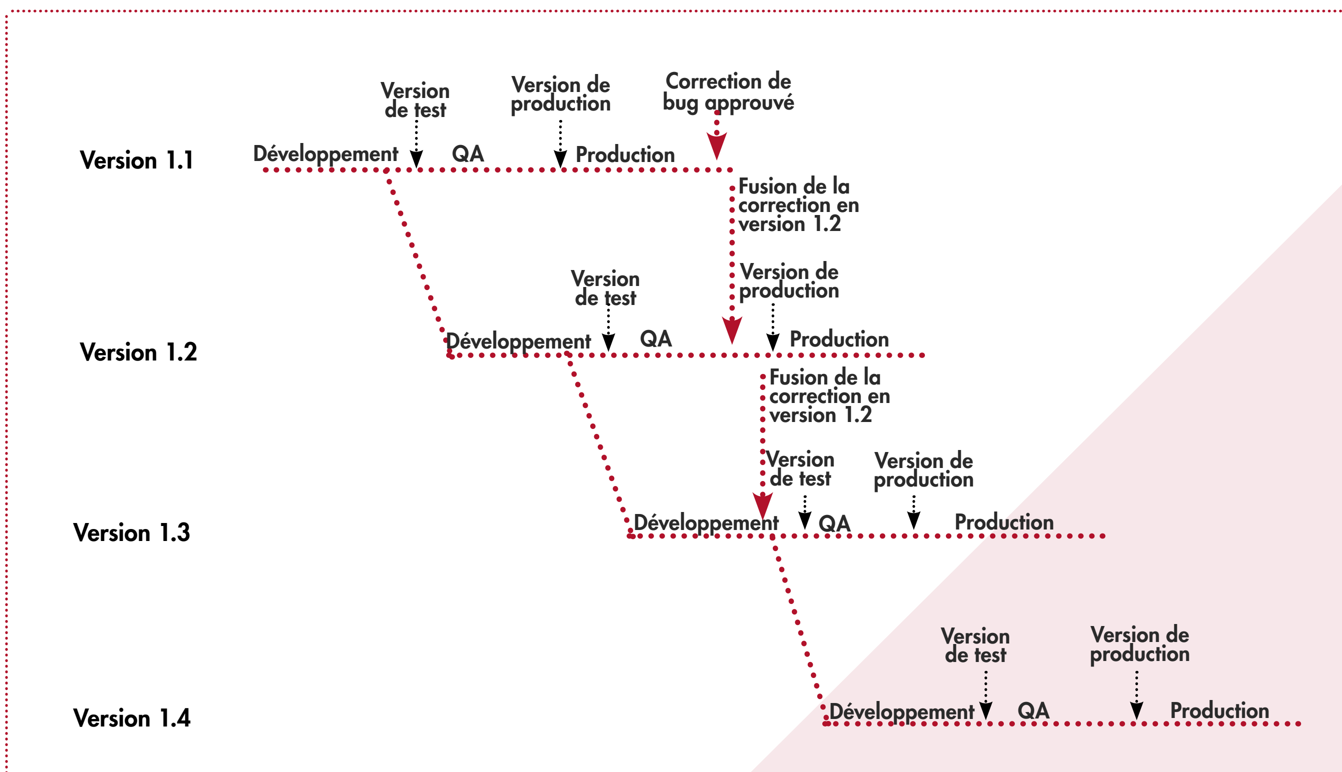

Figure 1. Le modèle branch-by-release. Dans ce modèle conventionnel, la branche est créée lorsque le responsable de version délivre une nouvelle version. La nouvelle branche sert de baseline pour continuer le développement tandis que l'ancienne – la référence historique – est laissée de coté.

Le modèle branch-by-release fournit une série de baselines successives que la gestion de configuration logicielle requiert conventionnellement. Il fournit une base commune pour les développeurs qui souhaitent apporter des modifications ultérieures au code. Cependant, il a deux inconvénients importants :

- Il requiert généralement des modifications de code en série, telles que des check-in et des check-out successifs, plutôt que du développement en parallèle,
- Il ajoute de la complexité au support des versions.

Le modèle branch-by-release est facile à comprendre – en tous cas avant que les premières versions correctives n'apparaissent. Mais réaliser une correction sur une version antérieure présente beaucoup de risques de perdre des corrections de bugs vérifiées, ou d'autres modifications. Tant que des versions restent supportées, la nécessité de correction de bugs – et donc de générer des versions d'urgence – reste possible. En ce cas, les développeurs doivent réaliser cette correction dans la vieille branche et créer une nouvelle version à partir d'elle. Cela semble assez simple, mais les complications surviennent vite. Les développeurs doivent propager la correction au travers des autres branches, pour s'assurer que le bug ne réapparaîtra pas dans les versions futures, où il n'a pas encore été corrigé.

Isoler des changements spécifiques et confirmer la nécessité de propager chacun d'eux à toutes les versions en aval crée une surcharge de communication et de coordination. Cet environnement change constamment, quand les développeurs passent d'une version à l'autre. Le modèle ne supporte alors plus les développements parallèles sur le long terme – tout le code ayant fait l'objet d'un check-out doit être vérifié avant chaque nouvelle version.

Si les développeurs ne vérifient pas le code avant chaque nouvelle version, le modèle n'autorise pas la reconstruction facile d'une version de façon globale. Lorsqu'un développeur réalise un check-in de modifications après la validation de la version, le responsable de version doit s'assurer que ces modifications non testées ne risquent pas de s'immiscer dans d'autres versions telles que des versions d'urgence, créées à partir du même code. C'est particulièrement problématique car le fait de ne pas reconstruire de versions de façon globale augmente considérablement le risque de builds incorrects pour les versions d'urgence. Cela impacte également le principe même de baseline, parce que des modifications qui n'ont jamais été livrées sont maintenant introduites dans le code de cette baseline.

Au bout du compte, le modèle *branch-by-release* ne fournit pas une façon efficace de générer et maintenir de multiples versions concurrentes.

Le syndrome de la génération par numéro de bug

L'utilisation du modèle *branch-by-release* conduit au syndrome redouté de la génération par numéro de bug. Ceci se produit lorsque on a réalisé un check-in dans une ancienne branche, après la génération de la version, de telle façon que le code de la branche ne correspond plus à la version qui a été distribuée. Le responsable de version doit sélectionner les portions de code associées aux corrections de bugs spécifiques que la direction du projet a décrété comme étant nécessaire aux versions futures. Cette situation arrive en général au pire moment, lorsque l'entreprise a besoin d'une version d'urgence. Ceci ajoute une pression supplémentaire à l'équipe, qui doit déjà produire rapidement une correction pour remédier à une situation d'urgence.

Fastidieux au possible, et souvent difficile, le processus de génération par numéro de bug met à l'épreuve le responsable de version, qui doit s'assurer que la version produite ne contient que les portions de code nécessaires à chaque correction, et rien de plus. Ce processus réclame souvent la participation peu enthousiaste de l'équipe de développement, augmentant ainsi le nombre de personnes impliquées dans cette situation tendue.

Comment le projet en est-il arrivé là ? Souvent, la direction de projet détermine que les modifications initialement prévues pour une version ne peuvent pas être inclues sans un risque trop important. Par exemple, la modification peut avoir été estimée comment potentiellement déstabilisante, et le temps nécessaire à la tester n'a pas été suffisant. Les développeurs n'ont peut être simplement pas pu effectuer les modifications souhaitées dans les temps.

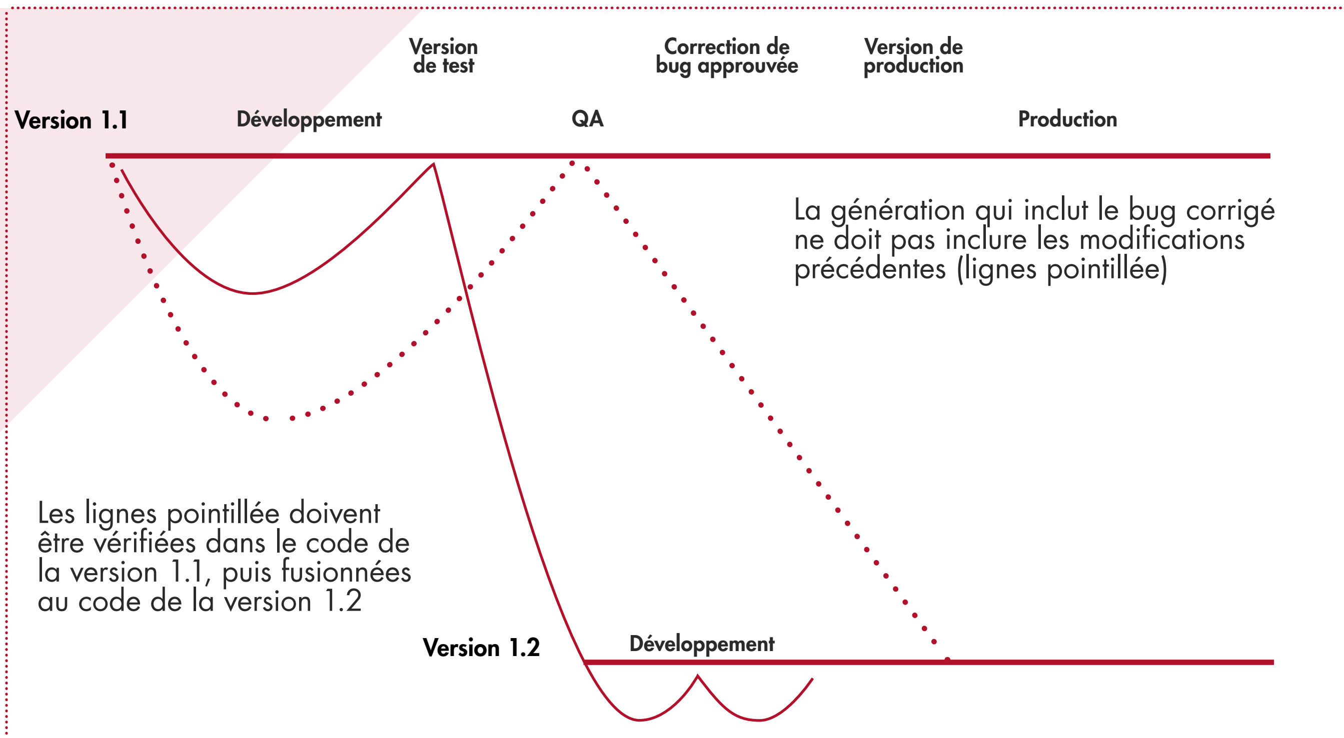

Figure 2. *Le syndrome de la génération par numéro de bug. Ce phénomène se produit lorsque le responsable de version doit sélectionner à la main les morceaux de code associés à des corrections de bugs spécifiques, puis s'assurer que seuls les morceaux nécessaires pour la correction sont inclus dans la génération.*

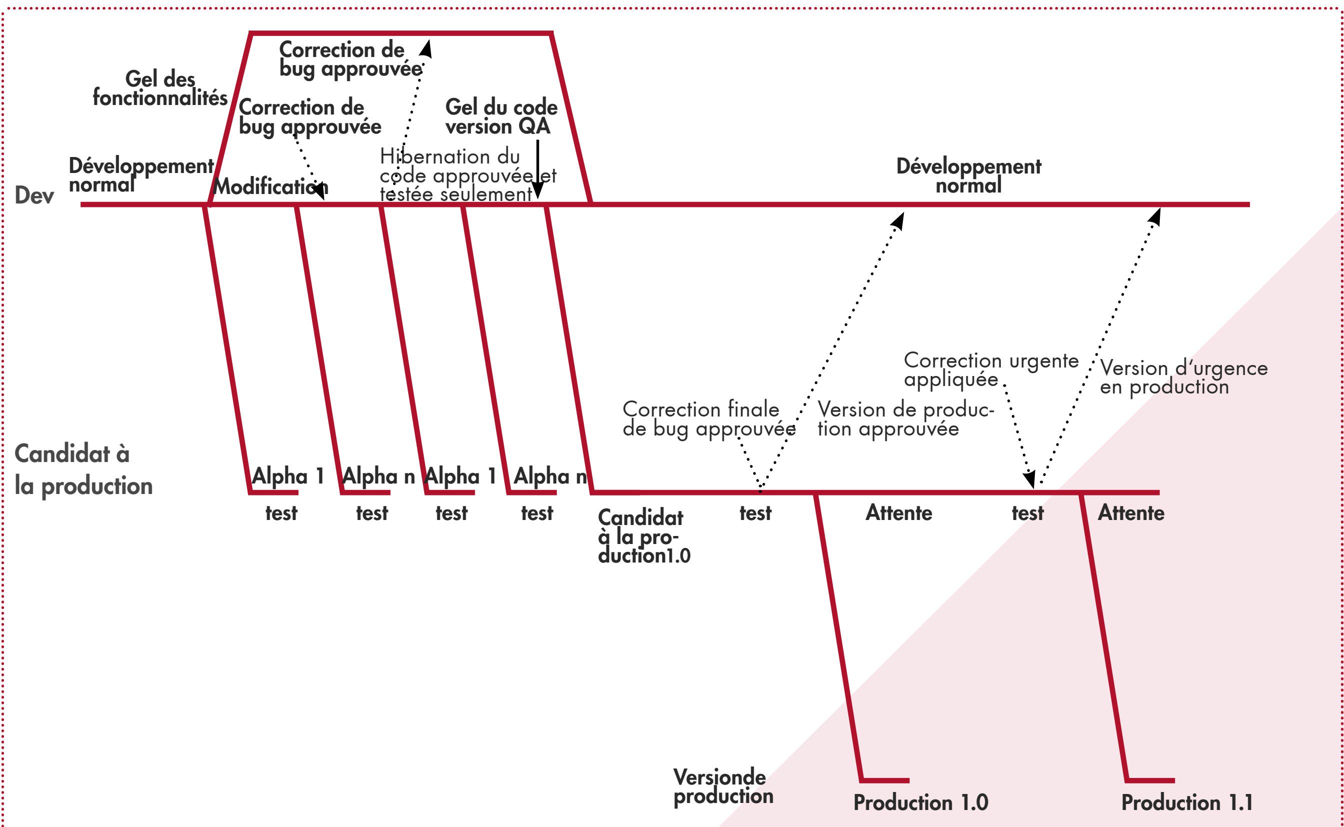

Figure 3*. Le modèle branch-by-purpose. Le responsable de version crée de nouvelles branches pour des raisons spécifiques, telles que les versions de test alpha ou bêta, mais l'équipe de développement continue à travailler dans la branche de développement principale.*

La figure 2 montre le dilemme que les développeurs rencontrent lorsqu'ils sont confrontés à cette situation. Les outils de contrôle de version demandent que le code qui a fait l'objet d'un check-out subisse un check-in dans la même branche. Si le développeur ne réalise pas un check-in du code avant la génération de la version du produit, il ne peut pas ré appliquer ce code directement dans la branche de la nouvelle version. Il est nécessaire de réaliser un check-in dans la branche originale, puis de la migrer dans la branche plus récente.

Une autre possibilité pour le développeur est d'abandonner ces modifications, et de recommencer dans la nouvelle branche. Evidemment, le développeur n'acceptera pas cette option et la perte de travail qu'elle entraîne.

Par conséquent, chaque version d'urgence réalisée sur cette ancienne branche sera concernée par la génération par numéro de bug. Pour s'assurer que seules les modifications nécessaires et aucune autre apparaissent dans les versions d'urgence, le responsable de version doit sélectionner chaque partie de code associée à chaque correction. Cette approche empêche la génération globale, parce que cette dernière implique la récupération de toutes les modifications en cours. Evidemment, ce scénario augmente les risques d'erreurs.

Dans certains cas, lorsqu'un code de mauvaise qualité demande de nombreuses corrections postérieures à la version, cela peut conduire à l'abandon d'un processus de génération de version discipliné, en faveur d'un développement à vue, tel que décrit dans l'encadré ***Développement à vue***.

Un modèle de gestion de configuration amélioré

Dans le modèle branch-by-purpose, décrit dans la figure 3, le responsable de version appuie sa décision de créer une branche sur le besoin de satisfaire un but spécifique. En général, ce but implique de générer le logiciel et ses éléments associés, tels que la documentation, à l'extérieur de l'équipe de développement. Ces versions marquent typiquement des étapes significatives du projet, telles qu'une version de test alpha, une version de test bêta, etc.

Le modèle branch-by-purpose supporte les versions ordinaires, aussi bien que les versions d'urgence. De plus, il évite les problèmes causés par le branch-by-release, et amplifiés par le syndrome de la génération par numéro de bug. Le modèle branch-by-purpose s'adapte à toutes les problématiques.

> *Le développement à vue*

Les équipes de développement internes dans les entreprises informatiques finissent souvent par tomber dans le piège du développement à vue, plutôt que de suivre une vraie discipline de version. Lorsque cela arrive, la notion de baseline est inversée. Pour une chose, il n'y a pas de version de produit déclarée. A la place, les dossiers de version utilisés par ces équipes ne contiennent que des morceaux de code à un instant donné, plutôt que la version réelle. Ces clichés tentent de capturer l'état de l'application pour créer des baselines, sur lesquelles effectuer des changements et synchroniser la ligne de développement avec la ligne de production.

Les développeurs réalisent cette activité après coup, de telle façon que ce qui se trouve dans les dossiers de version guide le système opérationnel. Cette approche a simplement comme objectif de capturer et d'archiver ce qui existe en production, quoi que ce puisse être. La véritable gestion de configuration, en revanche, identifie et sélectionne à l'avance les composants pour la génération d'une nouvelle version de l'application. La version est alors générée en utilisant cette information.

Cette différence est cruciale. Dans le modèle ***capture-and-archive***, l'utilisation de l'application en production est l'incarnation de l'état réel du produit, et le processus de gestion de configuration suit péniblement, en essayant de garder trace des modifications et de fournir aux développeurs une baseline fiable sur laquelle travailler. Dans le modèle ***baseline-and-release***, le responsable de version crée la baseline a priori, en utilisant le système de gestion de configuration pour construire une nouvelle version à partir d'objets de configuration préalablement identifiés, puis génère la version qui correspond exactement à la baseline.

Les organisations qui utilisent un processus de développement à vue comptent en général sur le modèle de baseline ***capture-and-archive***. Ces entreprises n'offrent pas de vraies versions de produits logiciels en tant que telles, juste une suite continue de mises à jour – ou de patchs – au logiciel en production. Bien que ce puisse être une méthode satisfaisante pour les corrections d'urgence, cela peut vite devenir la seule méthode de mise en production de toutes les modifications.

Lorsqu'une organisation glisse vers le développement et la fourniture à vue, elle ne remplace jamais l'application entière. A la place, elle la met à jour de façon incrémentielle, ce qui crée des problèmes complexes :

- Ce modèle continu rend difficile voire impossible la régénération de versions antérieures. Il n'y a pas de façon de comparer les applications, puisque

l'application est maintenant considérée comme un ensemble de mises à jours consécutives, de même qu'il est pas possible de revenir rapidement et facilement à une étape précédente.

- Tester les incarnations fugitives de cette chaîne continue peut être difficile ou impossible voir même très coûteux. Il y a rarement un processus en place pour tester l'application dans son ensemble, avec chaque fichier individuellement modifié – et juste ce fichier – pour refuser les modifications supplémentaires jusqu'à ce que les défauts découverts dans la livraison aient été corrigés et que la non-régression ait été validée.
- Le modèle de développement continu peut également présenter un problème de sécurité majeur. Une fois qu'un fichier source entre en production, il ne sera pas remplacé jusqu'à ce que la prochaine correction sur ce fichier n'entre elle-même en production. Si quelqu'un insère par erreur une copie spéciale de ce fichier, cette copie peut rester longtemps en production sans que personne ne s'en rende compte. Bien que ceci puisse arriver avec le modèle de remplacement d'application, au moins dans ce cas, de telles itérations peuvent-elles laisser des traces dans le code source?
- L'aspect de modification continue, sans aucun remplacement du modèle de correction continue, peut conduire au soupçon que certaines pièces en production ne viennent pas du dossier source de la version actuellement en développement, comme par exemple une correction d'urgence qui n'aurait pas été réintégrée dans le dossier source. Ceci conduit à la peur de la régénération totale de l'application, qui peut ne pas recréer le système en production. Si vous ne pouvez pas recréer le système en production, vous ne contrôlez pas votre produit.
- Ayant atteint cette situation inconfortable, l'équipe de développement choisit de recréer la baseline de son code source en capturant et en archivant les systèmes en production. Il n'y a pas ici de système de branching. Cette approche remplace simplement une ligne de code par une autre, et le système recommence.

Le modèle de correction continue présume que les développeurs peuvent correctement gérer un par un des centaines de fichiers en production. C'est néanmoins très difficile. Même si un seul fichier est déplacé dans un processus complètement automatisé et infaillible, il est toujours nécessaire de gérer, tester, et surveiller des centaines de composants interdépendants qui progressent en production. Même si 10% des fichiers sont modifiés entre deux générations, il reste un nombre impressionnants de composants à maîtriser.

Ce modèle offre également l'avantage de simplifier la vie du développeur en le laissant travailler dans le même environnement dans la branche de développement principale. Ceci limite les confusions entre les membres de l'équipe de développement sur la localisation des modifications et permet de livrer des versions d'urgence plus robustes. Tous ces avantages réduisent considérablement l'angoisse de l'équipe causée par le gel du code.

Nous avons à plusieurs reprises observé des équipes de développement raisonnablement mûres et disciplinées, qui résistaient au gel du code dans le modèle du ***branch-by-release***, parce qu'il ne leur permettait plus de réaliser le check-in de leur travail en cours. Reporter le gel du code réduit en général le cycle de test, et diminue la qualité globale du projet. Une fois de plus, il est préférable de créer une branche pour la nouvelle version, plutôt que de démarrer une nouvelle branche et continuer le travail dans celle-ci.

D'un autre coté, gérer ce modèle est plus complexe, d'abord parce qu'il demande une compréhension plus poussée de la gestion de configuration logicielle, et une plus grande maîtrise des outils de gestion de configuration. De plus, il renverse complètement l'approche traditionnelle : comme précédemment, le responsable de version crée une nouvelle branche pour la nouvelle version, mais le développement reste dans la branche principale.

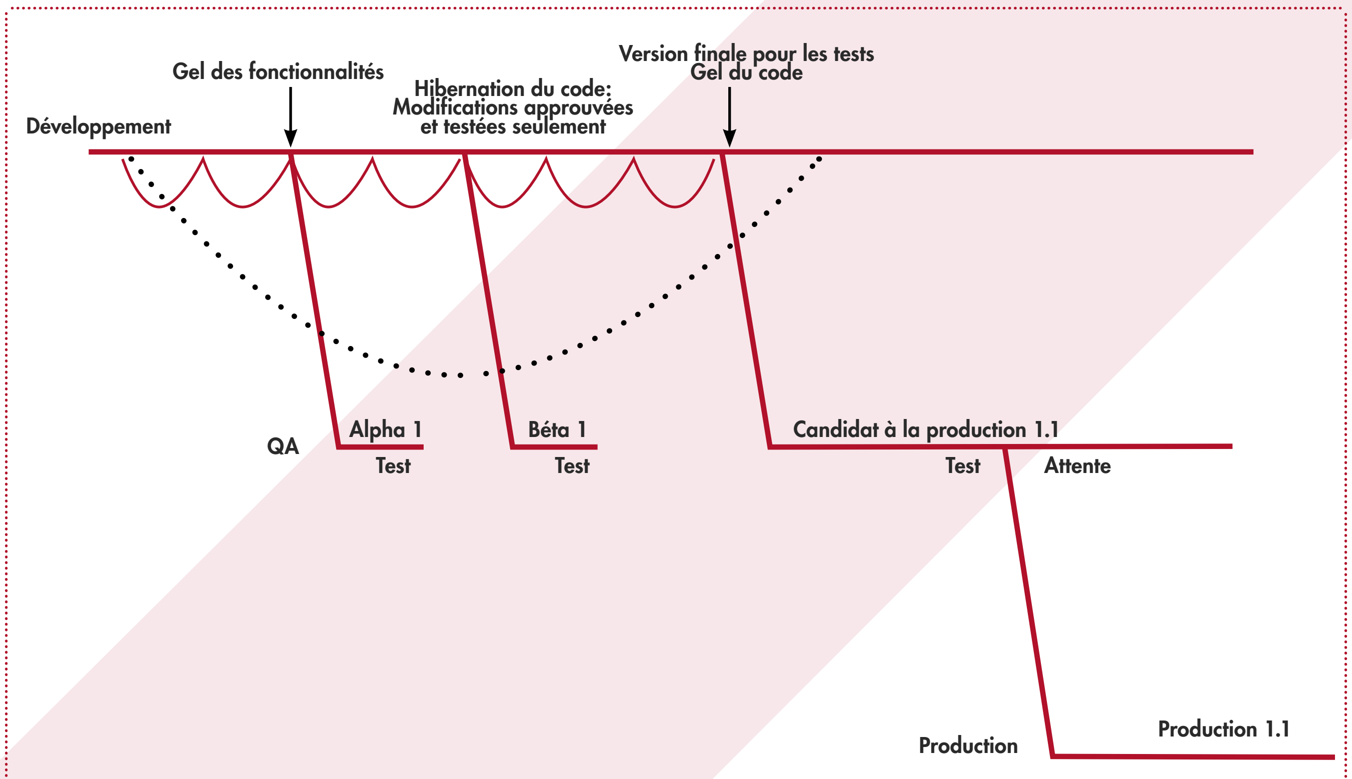

Figure 4*. Eviter le piège de la génération par numéro de bug. Dans le modèle branch-by-purpose, les développeurs peuvent refaire un check-in dans la branche de développement principale après le gel du code, sans toucher le code de la version, lui conservant ainsi toute son intégrité.*

Le produit peut être généré pour l'équipe de test, et dans ce cas le responsable de version crée une branche de test. Ou, lorsque les développeurs et les testeurs ont terminé leurs activités de découvertes et de corrections de défauts, le responsable de version peut créer une version du produit pour les besoins de production (directement pour le client dans le cas de projets au forfait, vers un groupe opérationnel dans le cas d'un développement en interne, ou, dans le cas d'un logiciel commercial, vers l'unité de manufacturing). Lorsque la nouvelle version est produite, le responsable de version crée une branche de production – branche spéciale dont le seul propos est de supporter cette version.

Le modèle **branch-by-purpose** pré suppose que le cycle des versions du produit inclut une étape de gel des fonctionnalités, après laquelle aucun développeur n'ajoute de nouvelles fonctionnalités et n'apporte d'améliorations aux fonctionnalités existantes sans une évaluation et un contrôle prudents des modifications. Ce jalon marque l'entrée de l'activité de développement dans la période d'hibernation du code. Durant l'hibernation du code, les développeurs peuvent faire des corrections dans la branche de développement, et envoyer périodiquement de nouvelles versions contenant les corrections à l'équipe de test. Chaque version envoyée à l'équipe de test a sa propre branche, ce qui permet de vérifier si un bug particulier a déjà été reçu par l'équipe de test, et d'identifier à la fois dans quelle version il a été trouvé, et dans laquelle il a été corrigé.

Au fur et à mesure où le logiciel gagne en stabilité tout au long du cycle de test, de moins en moins de corrections deviennent nécessaires. La période d'hibernation cède alors la place au gel du code. Lors de cette étape, l'équipe juge le produit prêt pour la version finale, même s'il reste des tests de recette à effectuer.

A partir de là, le développeur applique toutes les modifications approuvées par la direction de projet directement sur cette branche de test, et migre les corrections vers la branche de développement principale. Après vérification par les testeurs du bon fonctionnement des modifications, le responsable de version peut générer le produit et créer une branche de production. L'équipe utilise le même genre de cycle lorsque un défaut trouvé par un utilisateur final implique une version d'urgence, tel que décrit dans le ***build Production 1.1*** de la figure 3.

Supporter les développements en parallèle

Le climat actuel en faveur du développement rapide demande de paralléliser les développements. Les projets doivent accepter des améliorations qui n'étaient pas prévues dans la définition de la version en cours de développement. Même dans les cas rares où le développement rapide n'est pas de rigueur, le projet doit garder des ressources disponibles pendant l'hibernation et le gel du code, pour implémenter des corrections futures éventuelles.

Pour satisfaire tous ces besoins de développement parallèle, les responsables de versions peuvent créer des branches *passerelles* temporaires (une pour les corrections de bugs en attente et une autre pour les améliorations, ou bien une seule regroupant à la fois les corrections et les améliorations). Ils peuvent alors fusionner cette branche avec la branche principale après la génération de la version de production.

Dans le modèle **branch-by-release,** un problème peut surgir lorsqu'un changement sur le développement en cours doit être différé après une version. Dans le cas de fichiers qui ont fait l'objet d'un check-out, les développeurs se trouvent face à un cruel dilemme, parce qu'ils doivent :

- soit effectuer un check-in du code dans la branche dont il vient, ce qui signifie le remettre dans la branche d'une version qui ne le contenait pas, et donc détruire l'intégrité de cette branche, et ensuite le migrer vers la nouvelle branche de version,
- ré effectuer un check-out à partir de la nouvelle branche, refaire les modifications et refaire un check-in dans cette nouvelle branche.

Le modèle **branch-by-purpose** évite ce dilemme. Comme le montre la figure 4, puisque la branche de code principale n'a pas changé, le développeur peut juste faire un check-in du code dans la branche principale dont il vient, et il sera inclus dans une version future du produit. Le code de la version du produit, Release 1.1, reste identique car il réside sur sa propre branche.

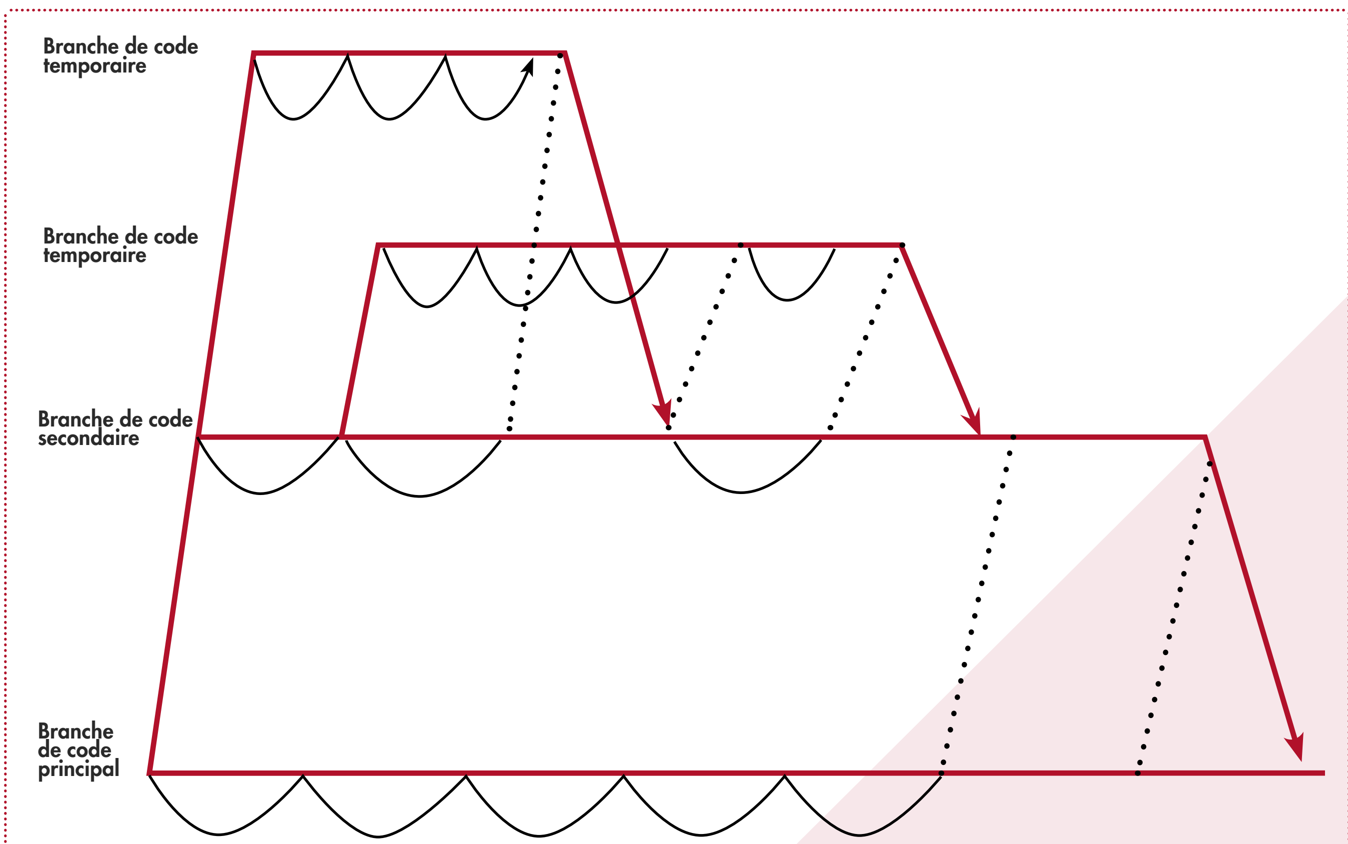

Figure 5. *Modifications de branches de code temporaires. Le responsable de version établit une branche de code secondaire pour gérer les modifications majeures du produit. Les développeurs peuvent alors créer une ou plusieurs branches temporaires à partir de cette branche secondaire et les utiliser pour travailler sur des modifications mineures.*

Groupes multiples et sous projets

Le modèle **branch-by-purpose** s'accommode mieux au besoin d'avoir plusieurs groupes travaillant sur plusieurs sous-projets en parallèle. Le branching permet de créer un environnement totalement répliqué, dans lequel les développeurs peuvent modifier le code et tester les modifications sans gêner les autres développements qui affectent simultanément le code. Créer de petites branches adaptées permet aux développeurs de réaliser des check-in de leur travail dans ces branches temporaires sans affecter la branche principale.

Le concept de développement parallèle implique le besoin de fusionner fréquemment les changements d'une branche dans une autre. Le modèle **branch-by-purpose** anticipe ce besoin, et rend aussi indolore que possible la gestion de configuration logicielle et les processus de gestion de versions.

A l'inverse, le modèle **branch-by-release** part de l'hypothèse que les versions sont linéaires et séquentielles, chaque version suivant immédiatement son prédécesseur. Par conséquent, les modifications apportées à une version existante de la baseline courante – ou à son prédécesseur s'il est toujours supporté – sont mal gérées, et réaliser ces modifications demande un effort particulier pour s'assurer que :

- chaque régénération d'une ancienne version contient uniquement les changements souhaités – typiquement, des corrections de bugs – et rien d'autre
- le responsable de version fusionne proprement ces modifications dans la nouvelle version en cours, et dans toutes les versions entre la version ancienne et la version courante.

Dans le modèle **branch-by-purpose**, à l'inverse, les développeurs partent de l'hypothèse qu'ils ne créent qu'une seule ligne principale de code et que, à des moments prévus, le responsable de version créera une branche séparée lorsqu'il créera une nouvelle version du produit. Cette approche autorise les générations globales fréquentes sur la ligne de développement principale, qui contient donc toutes les modifications disponibles. Les bonnes pratiques des sociétés de développement actuelles mettent l'accent sur les intégrations fréquentes au travers des générations globales. De telles générations intègrent les modifications *bit par bit*, au fur et à mesure que les développeurs et les testeurs réalisent des check-in. Les générations peuvent être quotidiennes ou continues. Cette pratique assure une intégration continue, et évite les gros problèmes que beaucoup de modifications simultanées peuvent causer lorsque l'équipe projet diffère l'intégration jusqu'à la fin du cycle de la version.

Disposer d'un environnement complètement dupliqué pour modifier le code permet la construction quotidienne ou continue de chaque ligne de développement, et par conséquent les tests de ces modifications, sans ralentir la ligne de développement principale. Il permet à plusieurs développeurs de réaliser plusieurs modifications du code pour des buts multiples.

Pendant que les équipes projet réalisent le branching pour les développements en parallèle à un niveau très fin – chaque correction de bug – ils utilisent deux façons principales pour créer les branches pour les lignes de développement alternatives :

- lorsque un ou plusieurs développeurs travaillent sur une modification ou un ensemble de modifications de faible importance sur plusieurs semaines, ou
- lorsque de nombreux développeurs travaillent sur une modification importante sur plusieurs mois.

Les variations du modèle **branch-by-purpose** peuvent s'adapter à ces deux cas.

Modifications des lignes de développement

Lorsque les équipes de test mettent en évidence que les modifications fonctionnent comme attendu, les développeurs fusionnent ces modifications dans la ligne de développement principale. Comme le montre la figure 5, les développeurs peuvent également fusionner périodiquement les corrections de bugs et autres modifications de la ligne principale vers les lignes temporaires si elles doivent exister plus de quelques semaines.

Que les développeurs fusionnent les modifications de la branche principale ou pas, ils doivent fusionner les modifications apportées dans les branches temporaires avant de fusionner la branche temporaire avec la branche principale. L'intégration est effectuée sur la branche temporaire, assurant ainsi la compatibilité des modifications avant l'insertion dans la branche principale, dont dépend le reste du développement.

Pour réaliser des modifications plus conséquentes et plus complexes, les développeurs établissent une deuxième ligne de développement pour développer cette évolution majeure du produit, comme un changement d'architecture. Ils fusionnent les changements de la branche principale vers leur ligne secondaire chaque semaine, par exemple.

La figure 5 montre deux lignes temporaires dérivant de la branche secondaire, permettant ainsi aux développeurs de travailler sur des petites modifications qui dépendent des changements apportés à la branche

secondaire. Les développeurs fusionnent les modifications vers les branches intermédiaires si elles doivent vivre plus de quelques semaines.

En dépit de sa large utilisation, et de son adéquation avec les standards conventionnels des baselines séquentielles, le modèle branch-by-release impose des charges inutiles aux équipes projet, en supportant les versions diffusées tout en en développant de nouvelles. Générateur d'erreurs, il demande souvent des opérations manuelles pour gérer les bugs des versions. De plus, son défaut principal est une complexité inutile dans la gestion des corrections des versions déjà diffusées et la gestion des changements en cours, qui ne peuvent pas être introduits facilement en urgence.

A l'opposé, le modèle **branch-by-purpose** évite ces pièges. En plus, il fournit un mécanisme structuré pour gérer plusieurs lignes de développement, supportant les besoins des organisations pour fournir plusieurs versions aussi bien que les besoins des équipes de développement pour réduire les cycles, en utilisant correctement le développement en parallèle et l'intégration continue.

Remerciements

Nous remercions *Matt Foley* et *Adonica Gieger* pour leurs commentaires utiles dans la revue de cet article.

Références

- 1. *P.H. Fieler, Configuration Management Models in Commercial Environments, tech. Report CMU/SEI-91-TR, Software Eng. Inst., Carnegie Mellon University, Pittsburgh, 1991.*
- 2. *ANSI/IEEE Std. 828-1983, IEEE Standard for Software Configuration Management Plan, IEEE Press, Piscataway, N.J., 1983.*
- 3. *ANSI/IEEE Std. 1042-1987, IEEE Guide to Software Configuration Management Content, IEEE Press, Piscataway, N.J., 1987.*
- 4. *R. Conradi and B. Westfechtel, "Version Models for Software Configuration Management", ACM Computing Surveys, vol. 30, no. 2, 1998, p.233.*
- 5. *I. Sommerville, Software Engineering, Addison-Wesley, Reading, Mass., 1996, p.657.*

Chuck Walrad est directeur chez *Davenport Consulting*. Ses centres d'intérêt incluent la stratégie pour la propagation et l'implémentation de bonnes pratiques dans l'ingénierie logicielle. *Chuck Walrad* a obtenu un Master of Science en théorie linguistique à l'*Université de Californie* de San Diego. Elle est membre de l'IEEE et de l'ACM. Il est possible de la contacter à cwalrad@daven.com.

Darrel Strom est ingénieur de développement senior chez *Expert Support*. Ses centres d'intérêt incluent les mécanismes de développement logiciel qui automatisent les processus critique requis pour générer et fournir des produits logiciels fiables. *Darrel Strom* a obtenu un *Bachelor of Science* en informatique à l'*Université du Mississipi* Sud. Il est membre de l'IEEE et de l'ACM. Il est possible de le contacter à dstrom@xs.com.

Traduit de l'anglais par *Ideo Technologies.*
Document non-contractuel.