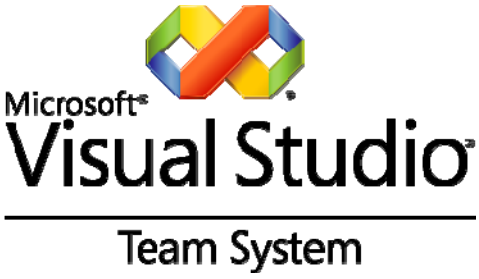


Team Foundation Server 分岐のガイダンス

John Jacob、Mario Rodriguez、Graham Barry 共著

Microsoft Corporation

目次

本ドキュメントは『Microsoft Team Foundation Server Branching Guidance』の日本語翻訳版です。原文ドキュメントは以下のサイト上にて公開されています。

http://www.codeplex.com/BranchingGuidance

はじめに

組織に Team Foundation Server を導入する場合、実際に試行錯誤を繰り返しながらの習得が必要なこともあれば、導入に際して行う決定が後で大きく影響する可能性があるため、事前の調査や計画が必要なこともあります。この資料は、こうした影響の大きな分野の 1 つである、ソフトウェアの分岐とマージについての参考資料を提供することを目的としています。

ソフトウェアの分岐とマージは非常に大きなテーマで、ソフトウェア業界で大きな成長を遂げている分野です。この分野を確立してきたソフトウェア会社では、長年にわたり、分岐とマージの実践方法を進化させてきているため、この業界で共有すべき知識が豊富にあります。デザイン パターンに対して行われてきたことと同様に、役に立つと思われるほぼすべてのシナリオに対応する分岐パターンが文書化されています。この資料は、こうした概念の入門書になることを目的としています。また、Team Foundation Server の採用を実施または検討している組織に対しては、分岐とマージの実践的手引きとなることを目的としています。ただし、この魅力的で幅広い分野を包括する情報源ではありません。この資料を一読後、さらにさまざまな視点で詳しい調査を行う場合は、www.cmcrossroads.com (英語) を参照してください。このサイトで提供される詳細ドキュメントでは、分岐とマージを含め、ソフトウェア構成管理に関するトピックを詳しく掘り下げて説明しています。また、この資料の最後にも、いくつかの参照先へのリンクを記載してあります。

この資料では、あらゆる規模のチームに関連する考え方、小規模チーム (20 人弱の開発者) に関連する考え方、大規模チーム (20 ～ 100 人の開発者) に関連する考え方など、いくつかの特定の概念を扱っています。きわめて大きなチーム (数千人の開発者) に対しては追加のガイダンスが用意されています。これについては、付録 (「バーチャル ビルド ラボ」) で説明します。この資料では、あらゆる規模のチームに関連する概念を最初の数章で説明してから、小規模チーム向けのアドバイスを示します。その後、大規模チーム向けのアドバイスを取り上げ、最後の章ですべてのチームを対象にこれらのアドバイスをまとめます。

並行開発

分岐とマージの機能が作成された主な理由は、並行開発を可能にしたいという願望にあります。並行開発とは、同じソフトウェア プロジェクトのさまざまな分野で作業する開発チームが、同時に作業を進めながら、各開発者によって行われた変更が競合する可能性を最小限に抑えることができる手法です。並行開発が実現できれば、やり直しや問題解決のための無駄な作業が最小限に抑えられます。では、並行開発の例をいくつか紹介しましょう。

例 1：
40 人の開発者で構成されるチームが 1 つのアプリケーションを構築しているとします。機能ごとに構成された 4 つのチームがあり、各チームに開発リーダーがいます。各機能チームの規模は、開発者 2 ～ 15 人とさまざまです。また、機能ごとにマイルストーンも異なります。この場合、各機能チームを明確に分離すると同時に、アプリケーションの共通領域への変更を、信頼性のある管理された方法で可能にするメカニズムが必要です。この解決策としては、分岐が非常に適しています。この資料では、これを「機能チームを分離するための分岐」と呼びます。

例 2：
つい最近、開発チームによって、Web サイトの初回リリースが公開されたところです。チームでは、このサイトの次のバージョンの開発に着手しています。ところが、公開したばかりの運用中のサイトで、大切な顧客から深刻なバグを指摘されました。中核となる開発チームは、次のバージョンに向けてサイトの改良を進めながら、公開済みサイトのメンテナンスのためにバグの修正を行う必要があります。つまり、これら 2 つの作業の流れを分離するメカニズムが必要です。この資料では、これを「メンテナンスのための分岐」と呼びます。

上記の例をまとめると、並行開発は、次のように定義することができます。

- 同じコード ベースに対してさまざまな機能やバグ修正を実装する開発作業
- 同じソフトウェア コード ベースの異なるリリースを扱う開発作業

並行開発で主に必要となるのは、同時進行中の他の開発に悪影響を与えることなくコードを分離して安定させることができるメカニズムと、ある分岐に分離した変更を他の開発分岐に戻して統合できるメカニズムです。

分岐の定義

分岐によって、開発作業 (バグの修正、機能の開発、コードの安定化など) ごとに、必要なソース、ツール、外部依存関係、およびプロセスの自動化に対する完全に独立したスナップショットが提供されるため、並行開発が可能になります。各開発作業でこのような完全に独立したスナップショットを使用することで、事実上、他の作業に依存することなく、独自のペースで作業を続行できるようになります。その後、関連性のある特定の開発作業 (バグの修正、機能の実装、重大な変更の安定化) に沿って、これらのスナップショットに含まれる各ソースを分岐できるようになります。

通常、分岐には次の処理が必要です。

1. ソース コードのスナップショットを取得して分離を行います。ある特定の時点で取得したソース コード、または安定した状態や既知の状態のソース コード (最新の正常なビルドなど) をスナップショットにすることができます。結果として作成されるコピーを「子分岐」、作成元を「親分岐」と呼びます。
2. 子分岐に分離したスナップショット内で変更を行い、安定させます。
3. 変更を親分岐と双方向に同期します。通常、統合またはマージと呼びます。これが重要になる理由については、この資料の後半で説明します。

Team Foundation Server での分離の作成

既に説明したように、上記のような状況では分離メカニズムとマージ メカニズムが必要になりますが、どちらのメカニズムも Team Foundation Server の機能によって提供されます。最初に、Team Foundation Server で提供される分離メカニズムについて説明しましょう。

個人レベルでの分離

個人レベルでコードの分離が必要な場合があります。つまり、開発者は、特定のソース コード ベースの中のさまざまな状態やスナップショットで作業することが必要になる場合があります。このような場合、Team Foundation Server の機能を効率的に使用して、同一ソース コードから複数のワークスペースを作成することができます。ワークスペースとは、ローカル ディスク上で保持される、ソース コードのクライアント側のマッピングです (「ソース管理ワークスペースの使用」参照)。ローカル ハード ディスクにはこのようなマッピングを複数保持することができ、各マッピングを異なる状態に同期させることができます。たとえば、最新の正常ビルドのラベルや、サーバー上の最新バージョンのファイルにワークスペースをマップすることができます。こうした各ワークスペースにより分離が提供され、ワークスペース間で相互に悪影響を及ぼすことなく変更を行えるようになります。これが可能になるのは、Team Foundation Server が、作成したインスタンスの数に関係なく、各ワークスペース固有の変更 (チェックアウト、保留中の変更、編集など) を追跡するためです。さらに、ワークスペースをサーバーにマップすると、各ワークスペースで保留中の変更をチェックインすることにより、暗黙のうちに変更をサーバーにマージできます。別のワークスペースに切り替える必要があり、保留中の編集をチェックインできない場合は、Team Foundation Server のシェルブ機能を使用して、変更をシェルブセットとして安全に保存できます。

共同作業向けの分離

ワークスペースは、コーディングのさまざまな状態や段階に必要な分離を個人のローカル ディスク領域上で提供しますが、現在のソフトウェア開発の多くはチーム作業として行われます。そのため、チーム レベルの分離のニーズにも対応する必要があります。つまり、他の開発者が現在作業しているソース コードの安定性を損なうことなく、複数の開発者が修正、機能追加、または重大な変更に関する共同作業を行えることが必要です。この場合にも、Team Foundation Server の優れた分岐機能が役に立ちます。

分岐

Team Foundation Server では、分岐を、一連のファイルを複数の分岐パスに展開できる機能と定義しています。分岐は、グループでの共同作業や、分離したその場所での変更のバージョン管理を提供する唯一の分離メカニズムであるという点において、独特な機能であると言えます。

Team Foundation Server では、「パス空間」で分岐を管理します。これは、エクスプローラでのフォルダのコピーに似ています。ただし、分岐はコピーとは異なり、後で変更をマージできるように、ソースをターゲットに関連付けた履歴を管理します。

Team Foundation Server の分岐では新しいコピーが作成されるため、分岐への移動は容易です。実際、ソース管理エクスプローラでは、分岐がソース項目とまったく同様に表示されます。また、同じコード ベースの分岐での同時作業を、必要な数だけ非常に簡単に行うことができます。これを行うには、該当するフォルダを他のフォルダと同じように取得するだけです。

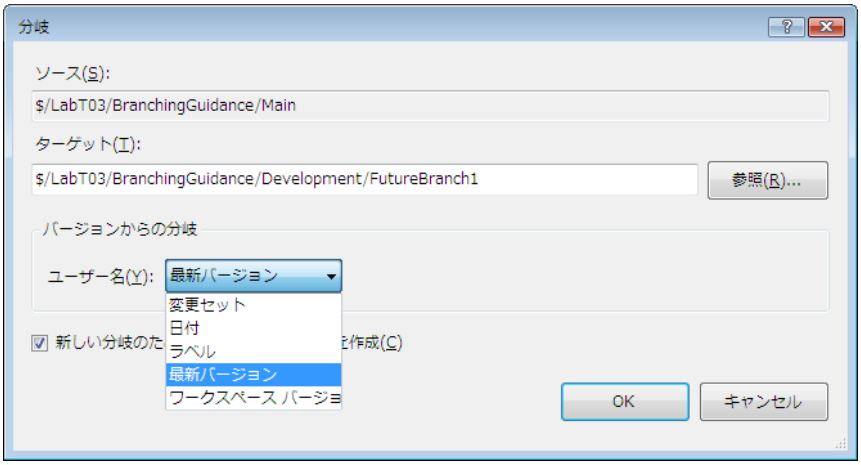
また、分岐のアクセス許可を維持するために、サーバー内の任意のフォルダに容易にアクセス許可を設定することができます。分岐は通常のパスであり、ソース ツリー内の他のフォルダと同様に、パスに基づくアクセス許可を使用してアクセスを制御します。これを行うには、ソース管理エクスプローラを開き、そのフォルダに移動して右クリックし、ポップアップ メニューの [プロパティ] をクリックして、[セキュリティ] タブをクリックします。コマンド ラインからは、Permission コマンドを使用できます。

分岐の作成に新たな記憶領域が必要になることはほとんどありません。サーバーでは、フォルダに含まれるファイルの数や種類に関係なく、同じ内容のコピーが 1 部だけ保持されるため、必要な記憶域が最小限に抑えられます。そのため、1 MB のファイルのコピーが 100 部あっても、すべてのファイルの内容が同じであれば、サーバーには 100 MB ではなく、1 MB しか保存されません。新しい分岐を作成してコミットすると、

新しい分岐内でソース分岐内のファイルと等しいすべてのファイルは、同じ内容を参照します。その結果、分岐によって新たな記憶領域が必要になることはほとんどなくなり、分岐したファイルがソースと異なる場合のみ、記憶領域が拡張されます。また、ファイルが変更された場合も、Team Foundation Server は差分エンジンを使用してファイル間の違いを分析し、再度記憶領域の最適化を行います。

分岐を作成する場合、日付、ラベル、ワークスペース バージョン (最後にワークスペースに取得したバージョン)、または最新のサーバー バージョンに基づいた分岐を選択できます。また、必要に応じて、これらを組み合わせることもできます。たとえば、あるフォルダを日付に基づいて分岐し、別のフォルダを項目の最新バージョンに基づいて分岐することができます。

ソース コードの最新バージョンの分岐を作成する方法を次に示します。



Team Foundation Server での分岐の詳細については、「Team Foundation ソース管理の分岐およびマージ」を参照してください。

マージ

マージとは、Team Foundation Server における分岐の論理的な帰結点です。バージョン管理におけるマージは、異なる 2 つの分岐で発生した変更を集約するプロセスです。マージ操作では、ソース分岐で行われた変更を取得して、その変更をターゲット分岐に統合します。マージでは、名前の変更、ファイルの編集、ファイルの追加、ファイルの削除、削除の取り消しなど、ソース分岐で行われたあらゆる種類の変更が統合されます。Team Foundation Server では、子分岐と親分岐の関係など、すべての変更の記録が保持されるため、マージ エンジンでこうした処理が可能になります。

ソース分岐とターゲット分岐の両方で項目が変更されると、バージョンの競合が記録され、それらの競合を解決するように求められます。

いったん分岐を作成すれば、変更をソース分岐からターゲット分岐に移動したり、ターゲット分岐からソース分岐に戻すことは非常に簡単です。Team Foundation Server では、分岐した項目間の関係とマージ履歴が保持されます。
複数の分岐間でマージを行うと、ソース分岐とターゲット分岐で発生したすべての変更がマージされます。こうした変更には、追加、削除、削除の取り消し、移動 (実際には名前の変更操作) などがあります。たとえば、ソース分岐内の項目の名前が A.TXT から B.TXT に変更された場合、マージを実行すると、その変更により、ターゲット分岐内の対応する項目の名前も変更されます。

ソースとターゲットの両方のファイルに変更を行った場合、それらの変更をマージすると、競合が発生します。このような競合はサーバーで管理され、クライアントには (コマンド ラインと Visual Studio のどちらからも) ダイアログが表示されます。このダイアログでは、競合が発生していること、およびその競合を解決する手順が示されます。競合ごとに、ソースとターゲットのどちらの変更を受け入れるか、または両者を組み合わせて受け入れるかを選択する必要があります。ファイルを編集した場合に、変更が自動的にマージされるようにすることもできます。変更を自動的にマージできない場合またはファイルを手作業でマージする場合は、競合解決のダイアログから、3 方向のグラフィカルなマージ ツールを起動します。

既に分岐のセクションでも説明したように、マージは一連の保留中の変更としてワークスペース内に収容されます。マージした変更をコミットする前であれば、ファイルの変更や移動、ビルドの問題の解決など、マージの結果をさらに変更することができます。マージした変更全体は、1 つの変更セットとして自動的にコミットされます。

サーバーにコミットされたマージは、関連する項目のマージ履歴に組み込まれます。次に一連の変更のマージが必要になったときに、Team Foundation Server によって、まだマージされていない変更が示されます。マージを追跡する必要はありません。マージの追跡はシステムで管理されます。この情報は、コマンド ライン、またはチーム エクスプローラのソース管理エクスプローラからいつでも取得できます。

また、マージする変更セットを選択することもできます。変更セットのサブセットのみをマージすることもできます。これにより、1 つのバグの修正をマージする際に、それ以前に行われたすべての変更がマージされ

ないようにすることができます。どの変更によって特定のバグが修正されたかがわかると、統合作業項目トラッキングが簡単になります。開発者は変更セットをチェックインするときに、バグや機能を表す関連作業項目にその変更セットをリンクさせることができます。したがって、バグの修正を反映するためにマージする必要のある変更セットを見つけるには、作業項目の [リンク] タブをクリックし、マージする適切な変更セットを探すだけで済みます。

チーム エクスプローラ内では、簡単にマージを実行できます。マージ ウィザードの手順に従って、マージする変更を選択できます。前回のマージ以降のすべての変更をマージしたり、特定の変更のみを選択したりできます。ソース管理エクスプローラで、マージする変更が含まれているフォルダ (分岐のソース) を右クリックし、ポップアップ メニューの [マージ] をクリックすると、マージ ウィザードが起動します。マージ ウィザードでは、候補となるターゲットが既に認識されているため、一覧から単にターゲットを選択するだけで済みます (フォルダの分岐を 1 回しか行っていなければ、エントリは 1 つしかありません)。すべての変更をマージするか、選択した変更だけをマージするかを選択できます。選択した変更をマージする場合は、まだマージされていない変更が表示されます。マージする変更を選択してから [完了] をクリックすると、関連するファイルごとに保留中の変更がマージされます。マージのビルドが成功したことを確認したら、マージをチェックインします。これで変更のマージは完了です。

使用可能な変更セットのサブセットのみをマージする場合、マージ ウィザードで、変更セットの連続する範囲を選択する必要があることに注意してください。変更セットの連続していないサブセットをマージする場合は、サブセットの範囲ごとにマージ ウィザードを起動する必要があります。

分岐のシナリオ

ここまでは、Team Foundation Server の分岐機能を、ソフトウェア開発におけるグループの共同作業を分離するためのメカニズムとして説明してきました。ここで、開発作業をグループに分けて分離を行う一般的な方法を見てみましょう。

- **リリースの分離：** 複数のリリースに対して並行作業を行う必要がある場合、各リリースのソース コードをそれぞれ別の分岐に分離することを考えます。これを「リリースの分離」といいます。これは、組織がソースの分岐を必要とする最も一般的な状況です。ここでいう "リリース" とは、必ずしも製品の新しいバージョンを意味するとは限らないことに注意してください。特定用途の分岐からテスト チームにアプリケーションをリリースすることを意味する場合もあります。これにより、開発分岐で新規開発を継続しながら、それとは別にそのリリースの問題点を調べることができます。

- **機能の分離：** チームが新機能に着手する場合、その機能を試験用と見なしたり、同一の分岐で開発作業を行うには危険すぎると考えることがよくあります。このような場合、「機能の分離」により、アプリケーションの他の部分を不安定な状態にすることなく、特定の機能に共同で取り組むことができます。
- **チームの分離：** サブチームを独立した状態で作業させることもよくあります。「チームの分離」により、このようなチームは他のチームが取り組んでいる大きな変更の影響を受けることなく、共同作業を行うことができます。場合によっては、チームの開発者ごとに専用の分岐を作成することもあります。既に説明したクライアント側のワークスペースの特性を考慮すると、Team Foundation Server では、各開発者専用の分岐を作成する必要はありません。
- **統合の分離：** 分岐間で変更をマージすると、動作が非常に不安定になることがよくあります。このような状況で、"統合" の分岐を管理すると役に立つ場合があります。この分岐は、実際の開発が行われる場所ではなく、マージを準備する場所です。他の分岐で発生した変更をこの分岐にマージした後、安定化を図ります。変更が安定したら、その変更を他の分岐に再度マージすることができます。一部の組織では、バグの修正やマージによってもたらされる可能性のある不安定化のことを「重大な変更 (braking changes)」と呼びます。このような組織では、「統合の分離」のニーズに対応するために "重大な変更用の分岐" を用意します。

上記は、大部分のソフトウェア開発組織に共通のシナリオです。Team Foundation Server の堅牢かつ多様な分岐機能によって、こうしたシナリオごとのさまざまなレベルの分離要件に適切に対応できます。一方、よくあるミスですが、過剰な分岐には注意が必要です。そのため、分離によるメリットとそのコストを比較し、開発プロセスでどれぐらいのオーバーヘッドを許容できるかを適切に判断するようにしてください。分岐にかかるコストは、次のように分類できます。

- **マージ：** 分離によるメリットにかかわらず、マージ プロセスでは、潜在的な競合の解決や、マージ操作によって生じる新しいバグの修正にかなりの作業が必要になります。マージ操作では、多くの場合、さまざまな分岐で発生した変更の一連の競合を調整するために、1 人以上のチーム メンバが必要になります。
- **待機時間：** 分岐間で変更を移動する作業には時間がかかります。一般的なマージ操作では、(まだ安定していない場合) ソース分岐で安定化を図り、マージを実行し、競合する変更を解決してから、ターゲット分岐で安定化を図ります。ソース ベースが小規模であればこの操作は数分で実行できますが、大規模になると何日もかかることがあります。分離領域間で変更を移動するために、複数の分岐を経由してマージを実行する場合、システム全体に変更を移動するのに数週間かかることもあります。

ここでは、このような分離メカニズムの大部分について、それらのコストも併せて説明することによって、考慮すべき項目がバランスよく含まれた一連のガイダンスを提供します。

分岐の構造に関する一般的なガイダンス

ここまでは、Team Foundation Server の優れた分離メカニズムである分岐とマージを使用して並行開発を実現する方法について説明してきました。ただし、こうした機能を最大限に活用するには、分岐操作とマージ操作の方針を確立することも必要です。作業内容や共通の目標と成果を基にチームを編成するのとまったく同様に、予測と計画を Team Foundation Server 内の分岐の構造と構成にまとめると便利です。

分岐のチェックリスト

分岐の作成には、次のチェックリストを使用することをお勧めします。

1. 分離モデルを選択します。このモデルは、次のどのニーズ (1 つまたは複数) が存在するかによって決まります。
   i. ソフトウェア開発作業に関する共通ワークフローの確立。これらの作業には、機能開発、品質保証、安定化、リリース準備、継続的なエンジニアリングなどがあります。
   ii. 密接に関連付けられた機能。相互に依存関係があり、まとめてコンパイルする必要がある (たとえば、共通のヘッダーやライブラリを共有している) コードを含む機能、実行時に相互に依存する複数のバイナリを必要とする機能などが挙げられます。
   iii. 開発作業で担当する分野の明確な区別。
   iv. ソース コードの厳密な不可侵性、責任、および監査の確認。
   v. 開発ライフサイクルのさまざまな段階の表現。
2. 開発作業のコード昇格パスを定義します。このパスは、並行開発作業がメインの分岐に最終的にどのように統合されるかを定義します。
3. コードをマージして分岐間の昇格を行うために満たす必要がある品質基準を定義します。
4. 分岐ごとに主担当者を定義します。この担当者は、分岐のユーザーに付与するアクセス許可を決定して、その分岐へのチェックインを管理および規制し、定義済みの品質基準を適用して、別の分岐との間のマージを受け入れます。
5. 4 番目の項目を決定したら、分岐ごとにアクセス許可 (読み取り、書き込み、分岐、ラベルなど) を定義します。

分岐を確立する最も簡単な方法は、一般的な組織の開発作業に幅広く採用されている次の 3 つの段階に従うことです。

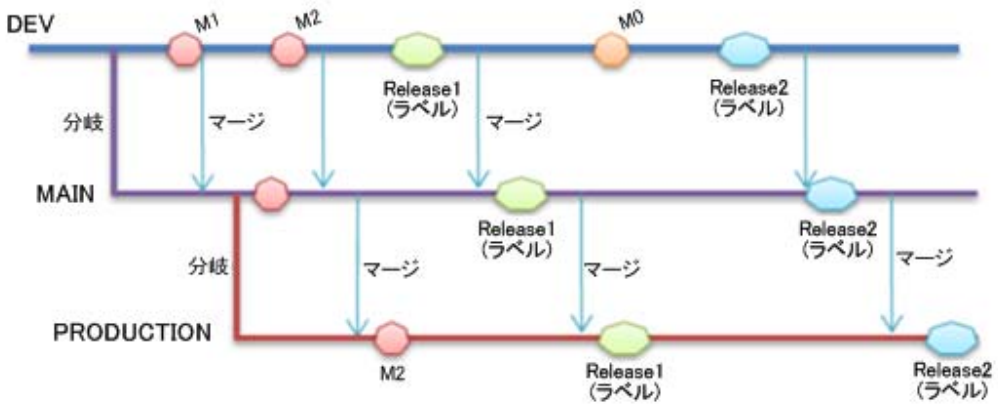
- DEV – 新機能の開発やバグの修正に関連する作業
- MAIN (別名 TEST または INTEGRATION) - 製品の安定化とリリースの準備に関連する作業
- PRODUCTION – リリース済みの製品に対する継続的なエンジニアリング サービスの提供に関連する作業

DEV 分岐には、機能の開発、バグの修正、重大な変更の統合など、すべての新規開発作業が分類されます。この領域は、新規開発の分離、格納、安定化を行うことを目的として設計されています。

MAIN 分岐は、大部分の組織に存在しますが、いくつか異なる名前で呼ばれています。"TEST" とも "INTEGRATION" とも呼ばれますが、一般的には "MAIN" と呼ばれます。この資料では、この分岐を MAIN と呼んでいますが、MAIN、TEST、INTEGRATION のいずれで呼んでも、この分岐の目的は同じであることを覚えておいてください。この分岐は、機能の分岐間で変更を統合するための貯蔵タンクとして使用されるため、できるだけ安定した状態に保つ必要があります。

PRODUCTION 分岐では、リリース済みの製品、またはリリース間近の製品のソースが保持されます。この分岐ではリリース済みの製品やリリース間近の製品のソースが保持されるため、コードの品質が非常に安定していて、徹底した変更の管理によってその品質が保証されます。

この方針に従ったアプリケーションのリリースは次のようになります。



重要点のまとめ：

- 機能とバグ修正用のコードは DEV 分岐で開発されます。
- 重要なマイルストーンに達したり、適切に定義された品質基準を満たした時点で、コードが定期的に MAIN にマージされます。
- MAIN のコードは、機能の完全性、セキュリティ、パフォーマンスなどのリリース基準に従って安定化が図られます。
- コードは、運用環境での追加テストを行うのに十分な品質に達した時点で、MAIN から PRODUCTION にマージされます。リリースは PRODUCTION で行われます。
- 各分岐のコードが次の分岐に昇格するかリリースされると、所定の分岐を次のマイルストーンに移行できるようになります。

これは、分岐ガイダンスを最も簡単に表現したものであるため、以降のセクションで紹介するいくつかのシナリオに基づいて、さらに説明を加えながら、より詳しい内容にしていきます。特筆すべきは、Team Foundation Server を使用すれば、「分岐のチェックリスト」で定義したすべての基準を簡単に満たすことができるということです。以降の説明の中で、これらの実装を紹介していきます。ここからは、「分岐のシナリオ」で紹介したシナリオに基づくさまざまな実装に加え、サービスを提供する必要があるチームの規模について詳しく説明します。

分岐の方針

「分岐のシナリオ」で説明したさまざまな分離のシナリオを思い出してください。これらのシナリオを上記の分岐構造にまとめてみましょう。この資料で推奨される分岐構造 (DEV と PRODUCTION) に対応付けて、

さまざまな分離のシナリオを次の表にまとめました。MAIN を意図的にこの表から外したことに注意してください。この理由は後で説明します。

サポートされる分離条件	**DEV**	**PRODUCTION**
機能の分離	√	
チームの分離	√	
統合の分離	√	√
リリースの分離		√

この分岐ガイダンスでは、DEV 分岐でいくつかの条件をサポートしています。DEV 分岐の下でいくつかの分岐を使用して、機能の分離や統合の分離を提供したり、より大まかなレベルでチームの組織的な方針に従って分離を提供したりできます。同様に、製品のさまざまなバージョンが運用環境やユーザーのデスクトップ上に配置されている可能性があり、これらをメンテナンスする必要があります。そのため、PRODUCTION 分岐の下で、サポートする特定のリリースに対応する 1 つ以上の分岐を提供します。この資料の目的から、DEV と PRODUCTION に 1 つ以上の分岐が存在することを示すために、これらを「コンテナ ノード」と呼ぶことにします。一方、MAIN には、必ず 1 つの分岐が存在し、常にこの状態にしておく必要があります。これは、MAIN を DEV と PRODUCTION の両方で行われるすべてのコーディング作業の合流点とするためです。MAIN は、合流してくるコードを別方向に接続する、鉄道の分岐点のようなものと考えてください。そのため、次期リリースに向けての安定化という 1 つの目的しかないため、MAIN は 1 つしか存在しません。

分岐の方針は他にもあり、チームの文化やプロセスを反映するものもありますが、この資料では取り上げていません。こうした他の方針には次のようなものがあります。

- タスク単位の分岐
- コンポーネント単位の分岐
- テクノロジ単位の分岐

このような方針の詳細については、MSDN で公開されている、ホワイトペーパー (Chris Birmele 著、http://www.microsoft.com/japan/msdn/vs05/vsts/BranchMerge.aspx) を参照してください。このセクションは情報提供のみを目的としているため、これらの方針を実装する前に、必要に応じて調査を行うことをお勧めします。

分岐によって実現されるさまざまな作業の分離

分岐	DEV	MAIN	PRODUCTION
種類	分岐またはコンテナ	必ず分岐	分岐またはコンテナ
目的	機能の開発、統合、重大な変更の安定化	製品の次期主要リリースの出荷準備、パフォーマンスとセキュリティの安定化、エンド ツー エンドの受け入れ	サービスの提供、リリース済み製品の継続的なエンジニアリング、目前に迫った次期リリースの最終修正
担当	開発マネージャ	リリース管理と QA	リリース管理
アクセス許可	各機能を担当する開発者、機能レベルの統合テストに重点を置いている一部のテスト エンジニアには読み取り/書き込みアクセスを許可します。	大半のユーザーに読み取り専用のアクセスを許可します。マージ、統合、分岐間や MAIN へのソース コードの昇格を行う上級開発者には、読み取り/書き込みアクセスを許可します。日次ビルドに緊急の修正プログラムを適用する場合、状況によっては読み取り/書き込みアクセス許可が必要です。大半のユーザーに読み取り専用のアクセスを許可します。	読み取り/書き込みアクセス許可は、修正プログラム、QFE、その他継続的なエンジニアリング作業に取り組む、特定の開発者とテスタに限定します。それ以外のメンバは、出荷コードの安定性を確保し、継続的なエンジニアリングによって提供される修正プログラムに対応できるように厳しく管理されます。
ビルド作業	大量のコード チャーンによる、継続的な統合。頻繁にビルドとテストを繰り返すことで、コードの状態について定期的にフィードバックす	コードが DEV 分岐から MAIN に昇格するタイミングに基づく日次ビルド。	DEV/TEST のニーズ/準備に基づく要求時のビルド。

	る必要があります。		
テスト対象	機能の完成、機能の安定化、機能の統合、主要製品のマイルストーンに関する要件。頻繁にビルドとテストを繰り返すことで、コードの状態について定期的にフィードバックする必要があります。	エンド ツー エンドの受け入れテスト、パフォーマンスやセキュリティのテスト。製品の出荷準備ができるまで繰り返します。	製品または修正プログラムのリリース時にサインオフします。

分岐方針の作成

ここまでは、Team Foundation Server の分岐とマージを支える概念について説明してきました。ここで、どのようにして開発ニーズに最も合った分岐方針を表現すればよいかということに注意を向けてみましょう。

1 回のリリースのシナリオ

上図は、分岐の物理レイアウトと論理表現の両方を表しています。物理レイアウトは、分岐が Team Foundation Server のソース管理ビューに実際にどのように表示されるかを示し、論理表現は分岐の相互関係を示します。これらの関係は、分岐の作成方法を表します。矢印は、親から (→) 子への関係で子分岐の方向を指します。コードの昇格 (前述のように、マージによって実現されます) は、親子関係によって確立された関係パスに沿って行われます。Team Foundation Server のマージ機能では、昇格パスに沿った双方向のマージがサポートされます。

これらの分岐で行われる開発作業の代表的なワークフローについて説明します。

- 機能とバグ修正用のコードは DEV 分岐で開発されます。
- 重要なマイルストーンに達したり、適切に定義された品質基準を満たした時点で、コードが MAIN にマージされます。MAIN が存在しない場合は、既知の状態 (ラベル、特定の日付、変更セットなど) に基づいて $/MyProduct/DEV から分岐することで、MAIN を作成します。
- MAIN のコードは、機能の完全性、セキュリティ、パフォーマンスなどのリリース基準に従って安定化が図られます。
- コードは、運用環境での追加テストを行うのに十分な品質に達した時点で、MAIN から PRODUCTION にマージされます。PRODUCTION が存在しない場合は、既知の状態 (ラベル、日付など) に基づいて $/MyProduct/MAIN から分岐することで、PRODUCTION を作成します。
- 各マイルストーンが次の分岐に昇格するかリリースされると、所定の分岐を次のマイルストーンに移行できるようになります。
- このサイクルを、リリースに対して設けられた品質基準を満たすまで繰り返します。基準を満たした時点で、リリース (Release1.0) は $/MyProduct/Production から公開されます。
- $/MyProduct/Production/Release1.0 から $/MyProduct/Safekeeping/Release1.0 の下に読み取り専用の Safekeeping 分岐を作成します。
- この時点で、$/MyProduct/DEV と $/MyProduct/MAIN は製品の次期リリース (2.0) に使用できるようになり、$/MyProduct/PRODUCTION は Release1.0 の継続的なエンジニアリングに使用できるようになります。

複数回のリリースのシナリオ

上記の構造は、1 回のリリースのみをサポートする比較的規模の小さいチームに対応しています。次に、常に複数回のリリースをサポートする必要がある作業環境について考えてみましょう。$/MyProduct/PRODUCTION の下で複数のリリース ノードをサポートするように分岐の方針を変更し、この資料の前半で定義したように、PRODUCTION を事実上のコンテナ ノードにします。

この場合の物理的および論理的な分岐構造を次に示します。

これは「分岐のシナリオ」で説明したリリースごとの分離を示す良い例です。ワークフローの大部分は「1 回のリリースのシナリオ」で説明したものと同じですが、リリースの分岐を作成する作業が異なります。$/MyProduct/MAIN は、リリース可能な状態になった後、リリースを表す $/MyProduct/Production/Release2.0 や $/MyProduct/SafeKeeping/Release2.0 などのバージョン フォルダに分岐します。

複数の機能/チームのシナリオ

次に複雑な作業は、1 回のリリースに含まれる複数の機能を並行開発するための分離に対応することです。この場合、DEV をコンテナにし、その中に複数の機能分岐を MAIN から分岐することで作成します。たとえば、次のような分岐を作成します。

- $/MyProduct/MAIN → $/MyProduct/DEV/FeatureBranch1
- $/MyProduct/MAIN → $/MyProduct/DEV/FeatureBranch2

これによって、$/MyProduct/DEV/ はさまざまな機能分岐のコンテナ ノードになります。また、$/MyProduct/MAIN はこれらの機能分岐間でコードの昇格を行う際の合流点になります。

コード昇格モデルの定義

ここまでは、分岐の作成とそれらの分岐間の関係 (子分岐の方向を指す矢印によって表されます) の確立に取り組み、複数の開発作業用に分離を用意したり複数のリリースをサポートしたりして、さまざまなシナリオで分離を提供してきました。また、このような関係を確立することによって、「コード昇格パス」を暗黙に定義しました。コードはこのパスに沿って次の分岐に昇格します。

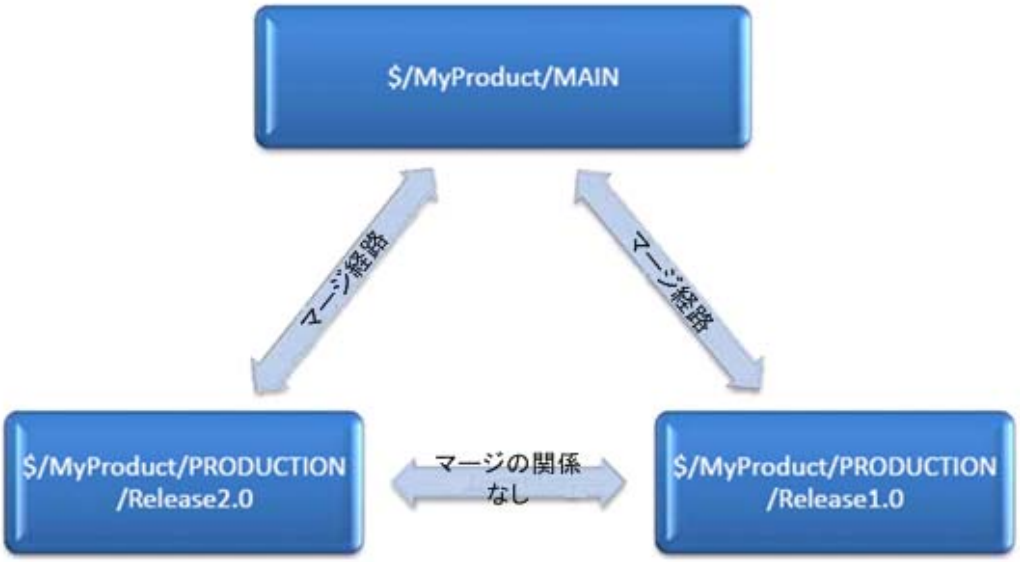
上記では、次のような分岐を作成しました。

$/MyProduct/MAIN → $/MyProduct/DEV/FeatureBranch1
$/MyProduct/MAIN → $/MyProduct/DEV/FeatureBranch2

このとき、MAIN → FeatureBranch1 と MAIN → FeatureBranch2 との間に親子関係を確立しました。この関係の矢印は子分岐の方向を指していますが、実際に Team Foundation Server に格納される関係を表す場合は双方向の矢印で表す方が適切です。つまり、上記の図は次のように変更されます。

双方向の矢印は、コードを親から子、子から親のどちらの方向にもマージできることを示しています。この方向は「マージ パス」と呼ばれ、Team Foundation Server でマージ操作を行うときに表示されるダイアログボックスに既定で提供されます。

マージ パスによってコード昇格パスが定義されるため、この後説明する、コンテナ内の分岐とそれらの親ノードとの間で発生するコードの昇格に関するベスト プラクティスで、このマージ パスが大きな重要性を持ちます。このことは特に、MAIN から作成された、DEV コンテナ内の分岐に対して当てはまります。次の図を使用して、これらの分岐間の関係を説明しましょう。



コードの昇格基準

MAIN 分岐と DEV コンテナ内の分岐 (FeatureBranch1、FeatureBranch2) との間に親 ←→ 子関係を作成したときに、これらの分岐間でマージ操作を行うために満たす必要がある一定の基準が含まれたコード昇格パスを確立しました。この関係におけるコードの昇格基準を次に示します。

1. MAIN のコードには常に高い品質基準を適用し、その品質を維持します。これは次の 2 つの観点から重要です。
   a MAIN では次のリリース向けに準備が整ったコードが収容されるため、常に出荷できる状態か、少なくともマイルストーンに見合った状態にしておく必要があります。
   b MAIN は、DEV コンテナ内の FeatureBranch1 と FeatureBranch2 という分岐でそれぞれ個別に行われる開発作業の合流点として機能します。つまり、FeatureBranch2 で FeatureBranch1 に含まれている修正が必要になった場合、これらの修正は FeatureBranch1 から MAIN にマージされた後でなければ取得できません。MAIN の動作が安定していない場合は、MAIN に依存する他の分岐が悪影響を受ける可能性があります。
2. コードの不安定要素を DEV コンテナ内の分岐でできるだけ多く解決し、コードを再度 MAIN にマージする前に安定化を図る必要があります (理由は 1 つ目の昇格基準に示したとおりです)。

品質ゲート

1 つ目のコードの昇格基準は、コードに高い品質基準を適用し、その品質を維持することでした。これを実

現するためのベスト プラクティスは、厳密に定義された、合意に基づく一連の公開品質基準を用意し、MAIN にマージする DEV コンテナ内の分岐がこれらの基準を満たすようにすることです。マイクロソフトでは、この基準を「品質ゲート」と呼び、通常は次の条件を設定しています。

1. すべてのバージョン (デバッグ版、製品版)、および該当するすべてのプラットフォーム (x86、AMD64 など) でソース コードが正常にビルドされた。
2. MAIN にマージする前に実行するすべてのビルド検証テスト (主要なシナリオを組み合わせたテスト、パフォーマンス テスト、セキュリティ テストなど) で、ビルドが正常に実行された。

順方向統合と逆方向統合

2 つ目のコードの昇格基準は、DEV コンテナ内の分岐を MAIN に再度マージする前に、不安定要素を含む変更点 (新機能、バグ修正、統合作業など) に対処し、コードを安定させることでした。この点に関して、マイクロソフトでは順方向統合 (FI) と逆方向統合 (RI) という 2 つの新しい概念を導入します。

"順方向統合" とは、最新の変更が行われた親分岐を子分岐と同期させることにより、子分岐を更新する操作を示します。つまり、次の方向で統合を行います。

$/MyProduct/MAIN → $/MyProduct/DEV/FeatureBranch1

反対に、"逆方向統合" とは、最新の変更が行われた子分岐を親分岐と同期させることにより、親分岐を更新する操作を示します。つまり、次の方向で統合を行います。

$/MyProduct/MAIN ← $/MyProduct/DEV/FeatureBranch1

コードの昇格に関するベスト プラクティス

上記で説明した順方向統合と逆方向統合のマージ操作は、事実上この分岐構造のコード昇格プロセスを示しています。このため、コードの昇格という点に重点を置いたベスト プラクティスが必要です。

MAIN で最大限の安定性を確保する必要があることを考えると、MAIN にコードを昇格させる子分岐には、逆方向統合の基準を厳しく適用する必要があります。この基準を次に示します。

1. 上記で定義した MAIN からの順方向統合を行うと、MAIN のコードの現在の状態が子分岐にすべて同期されます。最終的な逆方向統合を行う前に、MAIN からの順方向統合を頻繁に行い、統合されたコードのビルドを継続的に行うことをお勧めします (「Continuous Integration Using Team Foundation

Build」(英語) を参照してください)。これにより、逆方向に統合する機能分岐で、マージによって発生する競合を解決したり、コードの安定化を図ったりするために行う作業を大幅に減らすことができます。また、順方向統合とビルドを何回か繰り返すことで、最終的な MAIN への逆方向統合の前に、ビルドとテストに関する問題を発見して解決できます。このため、ビルドは "早めかつ頻繁に" 行います。

2. コードを統合先の分岐にチェックインする前に、そのコードが品質ゲートの基準を満たしていることを確認します。

これらのガイドラインに従うことにより、チェックインする子分岐より先に他の分岐がチェックインしていない限り、最終的な MAIN への逆方向統合で、マージによる競合が発生する可能性は低くなります。通常は、逆方向統合のスケジュールを設定することをお勧めします。これは、先に他の分岐からの逆方向統合が行われた場合、そのたびに逆方向統合の実行基準を再度満たす必要があるためです。さらに重要なことには、逆方向統合のスケジュールを設定することにより、共同で作業している他の開発チームが逆方向統合の作業に優先順位を付け、そのスケジュールを調整できるようになるためです。通常、逆方向統合のスケジュールの設定は、リリース管理チームや QA チームが個々の開発チームの要望に基づいて行います。

ベースレス マージ

上記で説明したように、分岐コマンドを実行すると、分岐間でコードをマージする際に使用できる、2 つのフォルダ間の関係が作成されます。この関係は双方向なので、コードをどちらの方向にマージすることもできます。しかし、現状では Team Foundation Server で子分岐間のマージを行うことは容易ではありません。では、ベースレス マージについて理解するために、次の構造について考えてみましょう。

上記のシナリオでは、次の分岐間のコード昇格パスを適切に定義しました。

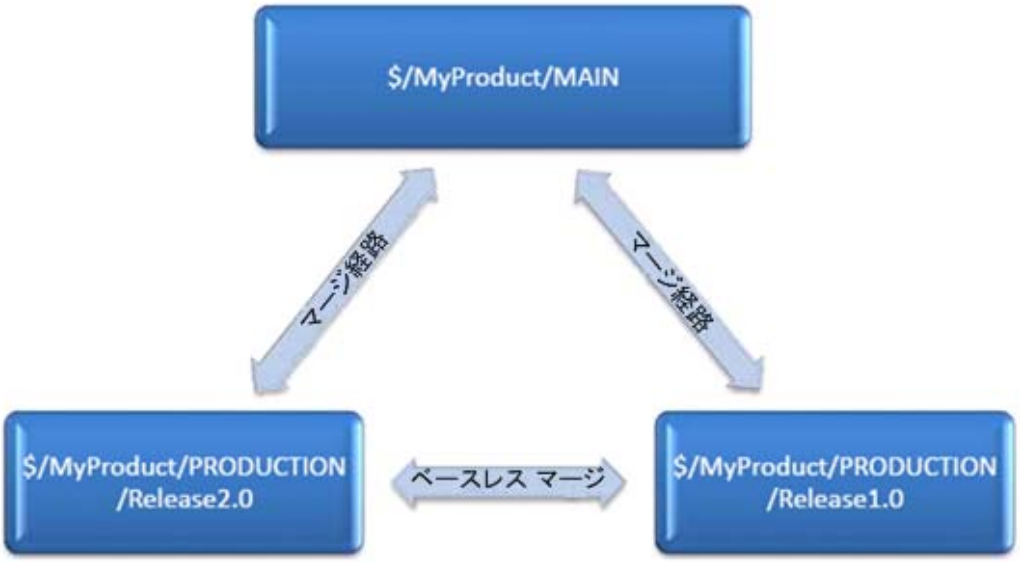
$/MyProduct/MAIN ⟷ $/MyProduct/PRODUCTION/Release1.0
$/MyProduct/MAIN ⟷ $/MyProduct/PRODUCTION/Release2.0

これらのコード昇格パスが存在するのは、PRODUCTION の下にある Release1.0 と Release2.0 が MAIN の子であるためです。残念なことに、子分岐の Release1.0 と Release2.0 の間に直接の関係はありません。このため、$/MyProduct/MAIN を経由せずに Release1.0 から Release2.0 に修正プログラムを適用することは困難です。

分岐コマンドによって確立されたマージ パスをたどる場合、Release1.0 の修正プログラムを Release2.0 分岐に適用するには、次の順序でマージを行う必要があります。

$/MyProduct/PRODUCTION/Release1.0 → $/MyProduct/MAIN → $/MyProduct/PRODUCTION/Release2.0

MAIN を経由することが適切ではない場合もあります。これは、MAIN が Release1.0 と Release2.0 の基になったバージョンから大幅に変更されている可能性があり、その場合、MAIN に修正プログラムをマージし、MAIN のコードをビルドしてテストするのに膨大な時間がかかることが予想されるためです。このような状況で、「ベースレス マージ」が必要になります。次の図は、既存の関係とベースレス マージを示しています。



MSDN では、ベースレス マージを「基になるバージョンが存在しないマージ」と定義しています。つまりユーザーは、ベースレス マージにより、分岐やマージの関係を持たないファイルやフォルダをマージできるようになります。ベースレス マージの実行後はマージの関係が確立されるため、次回のマージはベースレス マージにはなりません。

ベースレス マージは、Team Foundation Server によって提供され、コマンド ラインからのみ使用できる強力な機能 (「コマンド ラインからのみ実行できる操作 (Team Foundation ソース管理) [Visual Studio 2005] / [Visual Studio 2008]」を参照してください) ですが、慎重に、きわめて例外的な状況でのみ使用することをお勧めします。ほとんどの場合、コード昇格モデルを適切に定義することにより、ベースレス マージを使用しなくても、他の方法で変更をマージできます。

機能チーム：マイクロソフトの成功事例

ここでは、これまで学習してきた高度な概念を基にして、大規模なチームに最適な分岐構造に関するガイダンスを提供します。マイクロソフトの開発部門では、Visual Studio 2008 のリリースから「機能チーム モデル」を使用しています。このプロセスは Microsoft Office チームが始めたもので、社内全体で成功を収めていました。このセクションのガイダンスは、機能チーム モデルと対応する機能分岐に基づいています。

マイクロソフトのチームは、過去の出荷経験から得た次のような重要な教訓を受けて、機能チーム モデルを採用しました。

- プロジェクトのライフサイクルの終盤になると機能の安定化の完了は常に予定より遅れるため、製品の出荷日を正確に予測することはできませんでした。このため、安定化が完了して製品を出荷できる状態になるタイミングを判断できませんでした。
- コード化されていても安定していない機能を削除することは困難であるため、機能を縮小してリリース日に間に合わせることもできませんでした。また、新しく開発された機能によって生じる依存関係を容易に解消できなかったため、機能の縮小はさらに困難になりました。さらに、新機能の多くは既に製品デモなどでユーザーに紹介されているため、公式リリースでこうした機能が提供されることを期待するユーザーもいます。
- 機能が開発されてからそのテストが完了するまでの時間が長すぎることにより、効率が低下していました。

以前の出荷経験から得たこれらの教訓を基にして、機能チーム モデルの指針となる重要な原則を決定しました。これらの原則を次に示します。

- 品質を向上させます。機能を MAIN に再度マージする前に、特定の品質ゲートに合格することと、コードの安定化を図ることを各チームに要請することによって、確実に効率を高めます。
- 作業の遅延を防止します。作業を開発フェーズの後半まで保留するのではなく初期段階で行うようにして、早期にバグを検出および解決できるようにします。
- MAIN 分岐を常に出荷できる状態にしておきます。これにより、機能のコードが不安定または不完全な状態で出荷される危険性がなくなり、より多くのマイルストーンで製品を提供できるようになるだけでなく、短期間で最終リリースの準備を行うことができます。
- 品質ゲートに関する共通の一貫した定義と、プロジェクトの進捗状況を測定するための明確な基準を用意することにより、完了状態の共通定義をすべてのチームに適用します。

機能チームと機能の定義

"機能チーム" は、さまざまな専門分野のメンバから構成される、1 つの機能の開発を担当する小規模のチームです。"機能" は、ユーザーが直接目にすることができるか、インフラストラクチャとして他の機能によって使用される、単独でテスト可能な作業単位です。機能のサイズは、1 つの機能チームが開発できる小ささと、単独でテストすることが妥当な大きさの両方を満たしている必要があります。

機能は、短い周期 (3 ～ 6 週間) で実装、安定化、および MAIN への統合を完了することによって生産性を最大限に高めることを目的として定義します。これにより、安定した周期で機能のコードが MAIN に統合されるため、Customer Technology Preview (CTP) の出荷回数を増やし、開発ライフサイクル全体を通して貴重なフィードバックを早期にユーザーから受け取ることができます。

また、機能を設計する場合、その機能の開発前に MAIN にチェックインする必要がある他の依存機能を正確に把握し、明確に示す必要があります。このため機能は、大規模な機能を、単独でテストできる小規模の管理しやすい出荷単位に分割するのに役立ち、大規模なエンド ツー エンドの製品として出荷した場合に生じる不安定さを解消します。

チームの構成と重要性

一般的な機能チームは、プログラム マネージャ、5 人以下の開発者、およびそれに対応する数のテスタから構成され、3 ～ 6 週間の間、機能を完成 (以下で定義します) させるという共通の目的を持つチームとして、緊密に連携して作業します。

- 機能チームのメンバ全員が同じ期間内に作業します。
- 開発者とテスタが並行作業します。つまり、開発の進行中にテストが行われるため、機能の開発期間と品質の検証期間との間に生じる遅延を最小限に抑えることができます。機能の開発は機能分岐で行われ、機能チームに必要な分離が提供されます。
- 機能は、実装が完了し、安定し、十分にテストされ、他のユーザーが使用できる状態になった時点で "完成" と見なされます。他のユーザーによる使用とは、内部テスト (ドッグフーディングとも呼ばれます) や、CTP による外部ユーザーとの共有などを指し、インフラストラクチャの機能の場合は、別の機能チームがその機能に依存する機能のコードを開発できるようになった状態を "完成" と見なします。

マイクロソフトで機能チーム モデルを使用することによって得られたメリットは明確です。このモデルにより、厳密に定義された明確な目標に基づいて作業を行うことができるため、チームにやる気と協調性がもたらされます。また、ターンアラウンド時間や、開発者、テスタ、プログラム マネージャの間で行う繰り返し作業にかかる時間を短縮できました。さらに、MAIN 分岐の安定性を損なうことなく他のチームとコードを共有できます。そして、明確な目標が設定された共同作業環境でチーム作業を行うことにより、開発部門の個々のメンバの士気も向上しました。

これまで説明した、分岐とマージに関するベスト プラクティスのガイドラインは、機能チーム モデルにもほ

とんど当てはまりますが、機能チーム モデルの独創性と効果を高めるために、以下に重要な技術的手法を要約して示します。

- 単にコードを完成させるのではなく、機能を完成します。機能は製品に追加する前に完成します。つまり、機能の仕様決定、設計、実装、自動化、テスト、およびバグの修正を完了します。
- 品質ゲートを使用して、部門全体に一貫性を導入します。このために、機能の完成基準を部門で定義します。機能を製品に追加する前に、これらの品質ゲートに合格する必要があります。
- 機能分岐を使用して新機能の開発を分離します。つまり、各機能を別々の分岐に分離します。
- 次のうち一方の条件を満たした時点で、機能分岐を PU 分岐 (以下で説明します) に逆方向統合します。
    - ◊ 機能が完成した。
    - ◊ 品質ゲートを満たした。

機能チーム モデルのもう 1 つの特徴は、機能分岐と MAIN 分岐の間の中間分岐として PU (製品単位) 分岐を導入したことです。これにより、一連の品質ゲートを満たした後、機能分岐 ← → PU 中間分岐 ← → Main という昇格パスに沿ってマージが行われます。これらの品質ゲートは、作業の遅延を防ぐだけでなく、前もって行う必要があることがわかっている作業と、製品サイクルの中で後から状況に応じて行う作業のバランスを適切に取ることができるように設計されています。マイクロソフトで適用した品質ゲートには、機能仕様、開発設計、テスト計画、脅威モード、コードでの知的財産権の保護などがあります。

次の図は、機能チーム モデルの物理的な分岐構造を示しています。

エンド ツー エンドの分岐の実装

今回のシナリオとして、ポートフォリオに製品が 1 つだけ含まれている企業を想定します。この企業の物理的背景を以下に示します。

沿革

企業名：AdventureWorks

事業内容： アドベンチャー、映画撮影、およびスポーツ関連の産業を専門に扱う保険会社

開発者数： 130

アプリケーション数： 10

Team Foundation Server のインスタンス数： 1

所在地： 全世界に分布

構成管理者：John Smith

メモ：AdventureWorks では、全社で使用されている 10 個のアプリケーションを 3 つの開発チームで担当しています。これらのアプリケーションの 1 つに、事業対象の分野ごとにカスタマイズされた、セキュリティが確保された Web サイトがあります。

適切なカスタムの分岐構造を作成することは、構成管理者の仕事の中で最も重要な職務の 1 つです。建築家

が建物が時の経過に耐え得るよう気を配るのと同様に、構成管理者は分岐構造が複数回のリリース、並行開発、および製品の拡張に耐え得るよう気を配る必要があります。批判的に思考したり、事前調査を実施することによって、適切な構造が用意できるだけでなく、チームの効率や生産性がこれまでにないレベルまで向上します。

このホワイトペーパーを作成するにあたって、マイクロソフト、複数の企業、および複数の小規模組織のメンバにアンケート調査を行いました。この調査から、分岐構造を作成する担当者の大半が、知識や経験に基づいた判断を導き出すのに必要なあらゆるデータを得るために実行する一連の手順についてのガイダンスを求めていることがわかりました。これらの手順を、以下のチェックリストにまとめました。

- ✓ チーム プロジェクトの構造を決定する。
- ✓ 各チームの文化と要件を把握する。
- ✓ 要件を論理ガイダンスと物理ガイダンスにマップし、論理ガイダンスの初期草稿を作成する。
- ✓ 必要に応じて構造を変更する (必要であればベースレス マージを実行する)。
- ✓ 論理図と物理図を完成する。
- ✓ これらの図のリリース シナリオを確認する。
- ✓ チーム プロジェクトと物理的なバージョン管理構造を作成する (既存のコードがある場合はインポートする)。
- ✓ ベスト プラクティスを作成し、トレーニングを用意する。
- ✓ 経験から学び、分岐構造をカスタマイズして、自分の状況に合った構造にする。

手順 1: チーム プロジェクト構造の決定

John が最初に行う手順は、Team Foundation Server 内でこれらのチームやプロジェクトをどのように配置するかを決定することです。決定するにあたって最初に問題となるのは、各チーム プロジェクトへの分割方法です。この資料はチーム プロジェクトではなく分岐に関するものなので、チーム プロジェクト ガイダンスのホワイトペーパー (近日公開予定) を一読することをお勧めします。

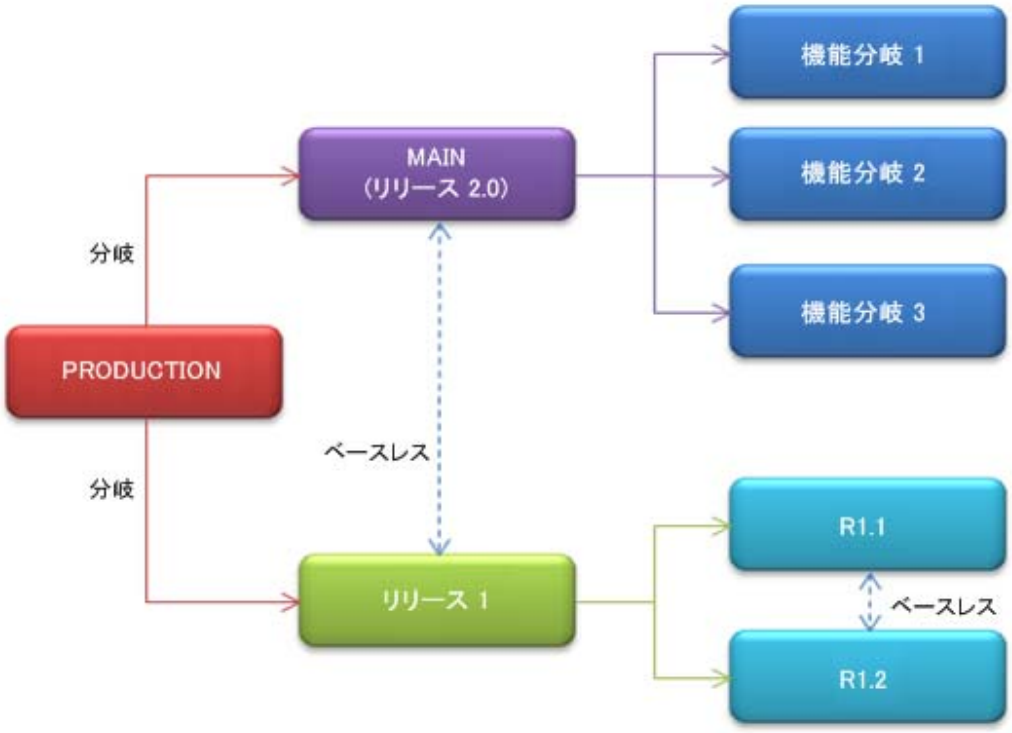
John はこの資料を一読し、企業の製品ポートフォリオをアプリケーションの種類 (顧客保険、内部分析、および Web サイト) に基づいていくつかに分類することにしました。このように決めた主な理由は、この分類方法が、各アプリケーションの出荷に採用されているプロセスと、各コンポーネントのビルドを任されている開発者の所在地にうまく適合するためです。今回の場合、顧客保険アプリケーションはすべてスクラム プロセスを採用しているのに対し、内部分析チームでは大きく異なるカスタム プロセスを採用しています。そ

のため、この 2 つを同じチーム プロジェクトに入れるのは困難です。

手順 2： 分岐構造の決定

チーム プロジェクトの構造が決まったら、John が次に行う手順は、プロセスをはじめとした組織の開発文化を文書にして理解することです。これを行うことによって、この資料で推奨している機能チームの分岐構造ガイダンスと比較して確認を行う際に使用する要件の一覧を作成できます。John はチームを分析して、1 チームを除く他の全チームでは機能別に分岐を用意するのが適切であると判断しました。

John は分岐構造用に以下の論理図を作成しました。



上図では、John は元のアプローチを修正し、MAIN と複数の機能分岐との間に関係を作成しました。John がこのような構造にすることを決めた根拠を理解するために、ここで示した要素のいくつかを分析してみましょう。

1. 昨今の企業では既に昇格モデルが採用されているので、機能分岐を反映するようにその昇格モデルを変

更するのは大きな変更にはなりません。

2. 開発者は、通常、同時に 2 つのリリース作業に従事します (リリース 1.1 とリリース 1.2)。リリース スケジュールは約 6 か月単位なので、各リリースの開発期間は約 3 か月強です。リリース 1 フォルダからリリース 1.1 と リリース 1.2 に分岐すると、必要な分離が作成されます。
3. チームでは、それらの分岐間の変更をマージするために、ベースレス マージ コマンドを使用してリリース 1.1 とリリース 1.2 の間にマージ関係を確立します。リリース 1.1 で行われた変更はリリース 1.2 にも必要なので、修正のコードがリリース 1 の分岐に反映されるまで待ちたくはありません。その変更をリリース 1.2 にマージして戻すには、関係を確立する必要があります。
4. 多くの場合、中間バージョンに対して行われた変更は、リリース 2 の機能分岐に反映する必要があります。これを実現し、全分岐でコードの変更を手動で行うのを避けるために、ベースレス マージ コマンドを使用してマージ関係を確立します。マージ関係がいったん確立されると、リリース 1.1 に対して行われた変更を簡単にリリース 2 に再適用することができます。

以上の分析から、ベースレス マージ機能を使用すべき理由を検討してみましょう。

今回のシナリオに話を戻すと、コード修正を迅速に処理し、コードを複製しないようにするために、リリース 1 の分岐とリリース 2 の分岐の間でコードを移動する方法が必要です。しかし、これらの分岐間にはマージ関係がありませんでした。この問題の解決方法は、2 つの分岐間でベースレス マージを実行することです。直接のマージ関係を確立するために、この操作を 1 回実行する必要があります。

もう 1 つ注意すべき点があります。これで関係は作成されますが、チーム エクスプローラの UI には関係が表示されないので、結果として関係が確立されたこれらの分岐間でマージを行う場合は、コマンド ラインから実行する必要があるという点です。また、最初に実行するベースレス マージによって多くの競合が発生するので (同名のファイルごとに 1 つの競合が発生します)、発生したすべての競合を解決するのに十分な時間を確保しておく必要があります。

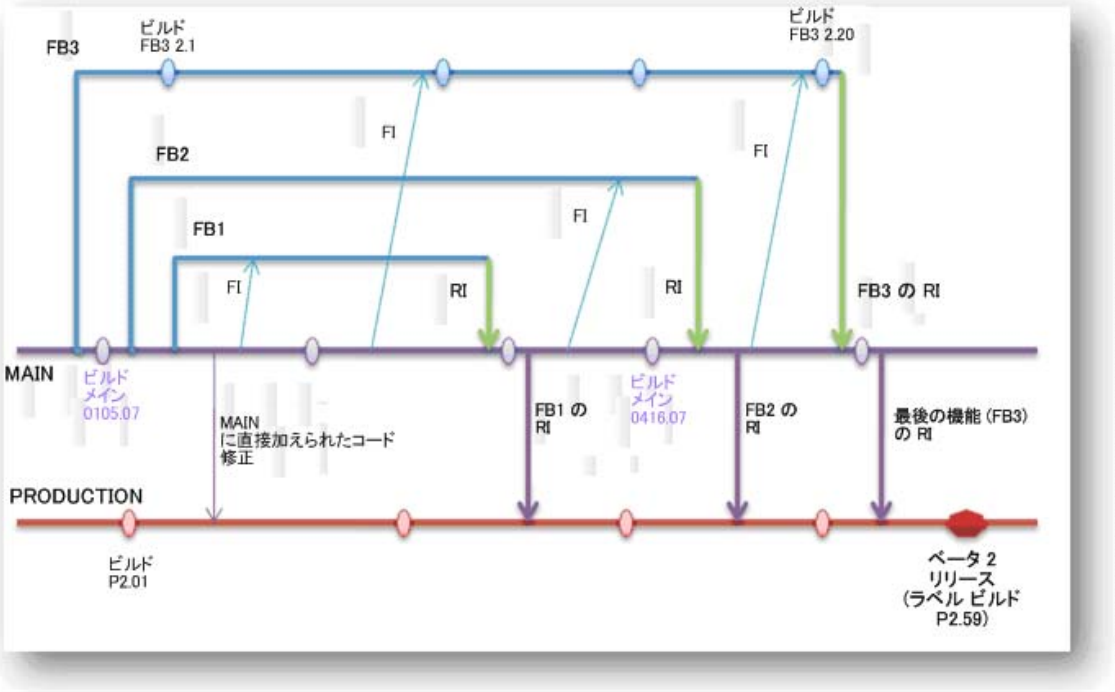
手順 3： 紙の上でのリリースの確認

論理図と分岐構造が完成したので、プロセスの次の手順は、抜けている点がないことを確認するために、リリースでこれらの分岐がどのように利用されるかを紙の上でたどることです。これを「昇格とマージのパイプライン」と呼びます。

このパイプラインの説明に最適なのは、一連のシナリオを紹介することです。昇格とマージのパイプラインがどのように進んでいくかを示す一連の図を以下に示します。

シナリオ 1：リリース 2 のベータ版のリリース

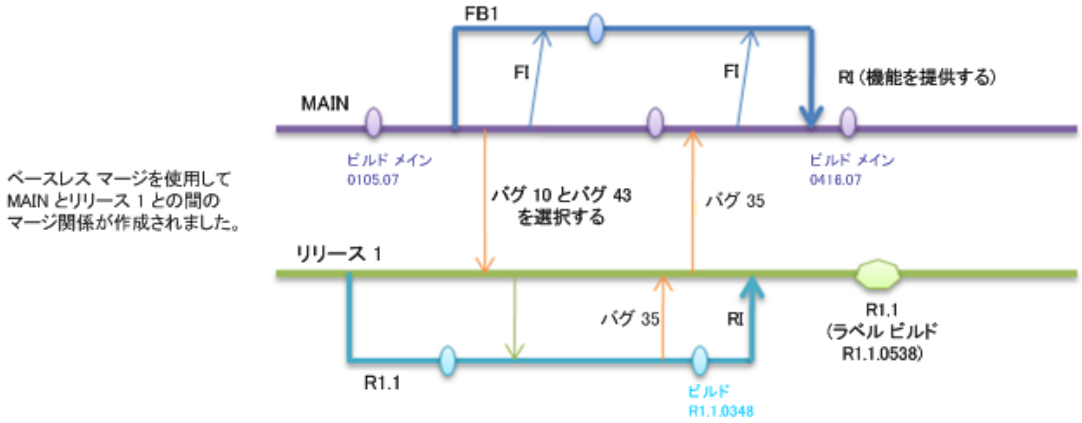
このシナリオでは、「コード昇格モデルの定義」で説明した概念の実例を示します。特に、パイプラインの機能分岐で行われる一連の機能開発を進めていくうえで決め手となる、順方向統合と逆方向統合の操作に重点を置いて説明します。以下の図は、リリース 2 のベータ版のリリースに至るまでの作業工程を表現したものです。



- リリース 2 は MAIN で安定化が図られているのに対して、数本の機能分岐 (FB1、FB2、FB3) ではベータ 2 リリース向けに設定された機能の実装と安定化のための開発作業を行います。
- これらの機能分岐はそれぞれ、「コードの昇格に関するベスト プラクティス」で定義した基準を基に、一連の順方向統合 (FI) 操作 (それぞれの分岐を指す上向きの矢印で示されます) によって、MAIN に加えられた最新の変更まで定期的に同期します。
- ある時点で FB1 の機能が完成すると、FI と逆方向統合 (RI) のベスト プラクティス ガイダンスと品質ゲートの基準を満たした後で、FB1 の変更が MAIN に RI されます。

- FB1 の RI の後に 2 回の重要な FI 操作が続いていることに注意してください。この 2 回の FI 操作によって、FB2 分岐と FB3 分岐に FB1 の最新機能が反映されるため、FB2 分岐と FB3 分岐が MAIN に RI する際、それらの変更に、機能開発の作業結果と FB1 の変更の両方が含まれるようになります。「コードの昇格に関するベスト プラクティス」で強調したとおり、MAIN からの順方向統合を定期的に行うと、機能分岐から変更を統合する際に発生するマージの競合数が減少します。また、統合の問題を安定化するために必要な繰り返し作業の回数が減少し、許容可能な短い期間で作業を完了できるようになります。これらの操作と並行して統合ビルドを継続的にスケジュールすると、FI 操作の成功に基づいてさらにコードを強化することができます。
- また、MAIN では、機能分岐から入ってくる RI の結果を PRODUCTION 分岐に反映するために、同様の FI 操作を行う必要があります。前述の機能分岐の場合と同様、PRODUCTION 分岐に関してもこの操作を行うことにより、統合の問題を安定化するために必要な繰り返し作業の回数が減少し、許容可能な短い期間で作業を完了できるようになります。このような理由で、機能分岐から MAIN への各 RI のすぐ後に、MAIN から PRODUCTION への一連の FI ("FB1 の RI"、"FB2 の RI"、"最後の機能 (FB3) の RI" というタイトルが付いた下向きの矢印で示されています) が行われます。
- これらの分岐が RI を受け入れるたびにリリース ビルドを行うよう、MAIN と PRODUCTION の両方にスケジュールを設定します。このようなビルドは、品質保証 (QA) チームによって、ベータ 2 マイルストーンの出荷基準 (機能の完全性、パフォーマンス、およびセキュリティの考慮事項) を満たすように徹底的にテストされます。
- “最後の機能 (FB3) の RI” というタイトルが付いたマージの後のある時点で、ベータ 2 (ビルド P2.59) マイルストーンの出荷基準に到達します。

シナリオ 2 : 1.1 のリリースと一部のコード修正のリリース 2 へのマージ



このシナリオでは、必要な修正だけを選択するという方法を使用することに注目します。ライフサイクルの中では、ある分岐で行った修正を別の分岐に迅速に反映することが必要になる場合が多々あります。このような修正の例としては、製品のあるバージョンでは行われていても、他のあるバージョンでは行われていない深刻なバグの修正があります。

ご記憶のように、1.1 で行った修正を MAIN (リリース 2) に移動できるように分岐間の関係を作成しました。MAIN によって確立された関係がない場合、あまり理想的とは言えない次のようなマージ パスを使用する必要があります。

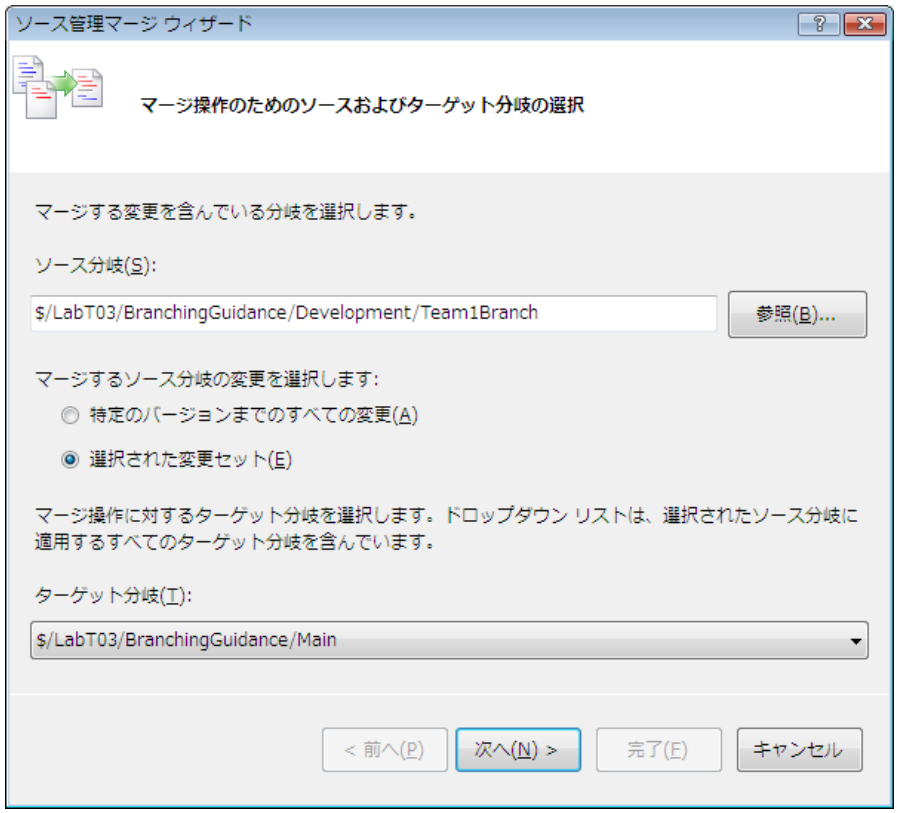
1.1 → Release 1 → PRODUCTION → MAIN

修正の移動はこのようなパスではなく、次のようなより自然なパスで行われます。

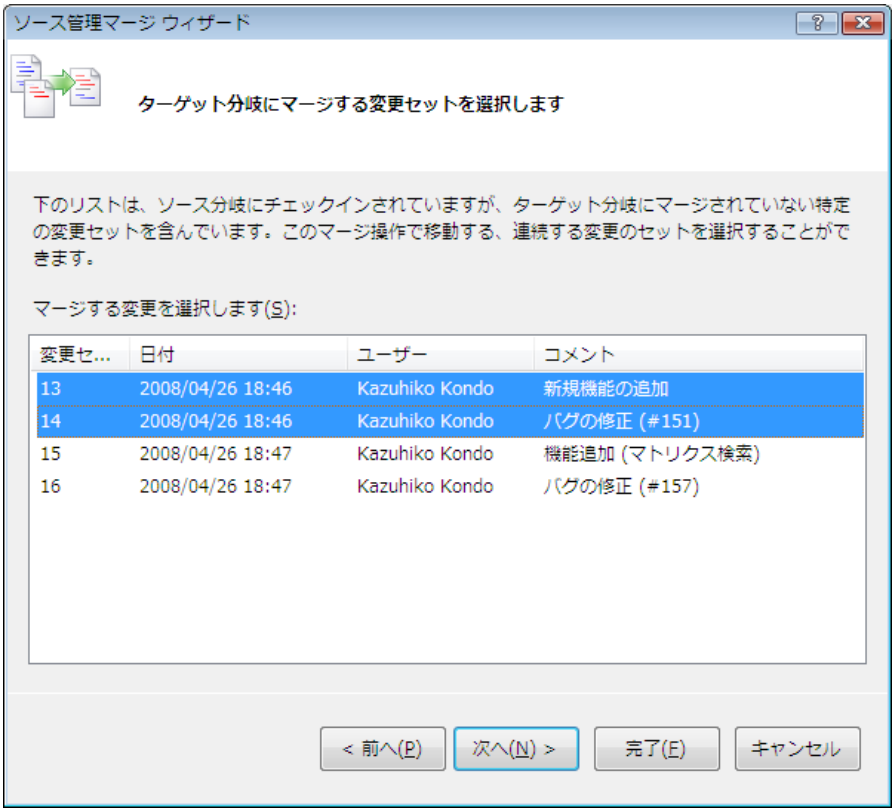
1.1 → Release 1 → MAIN

Team Foundation Server では、マージする変更セット (1 つの変更セット、または複数の変更セットの連続する範囲) をマージ ウィザードで指定することにより、これらの変更のうち必要なセットだけを選択することができます。

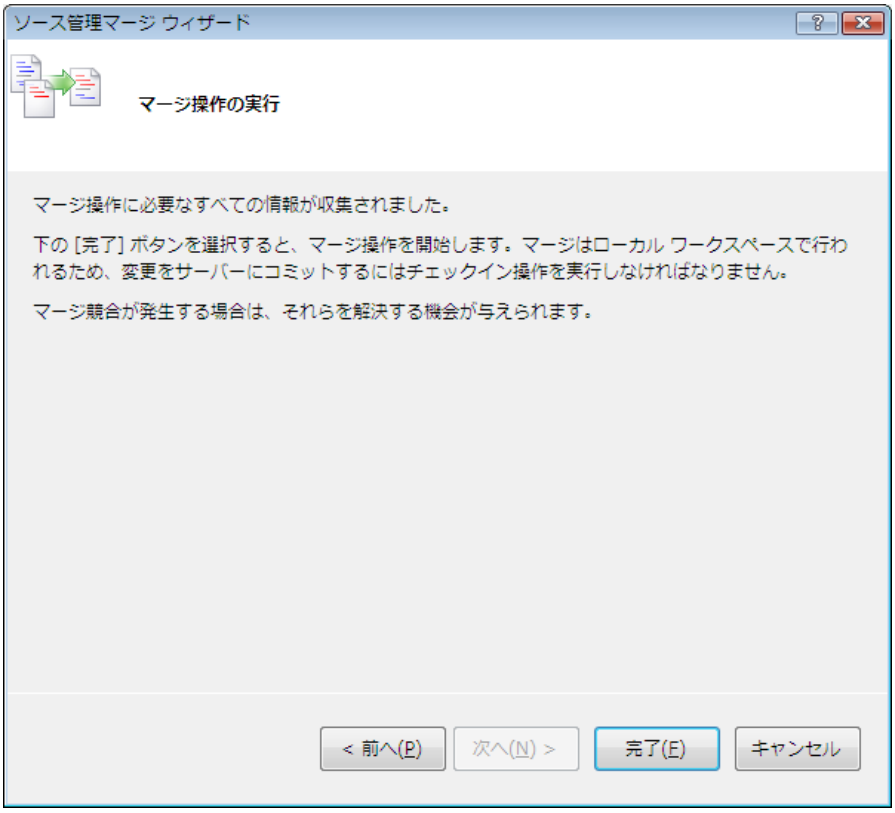
マージ ウィザードを使用して、マージする変更セットの選択を行う例を以下に示します。



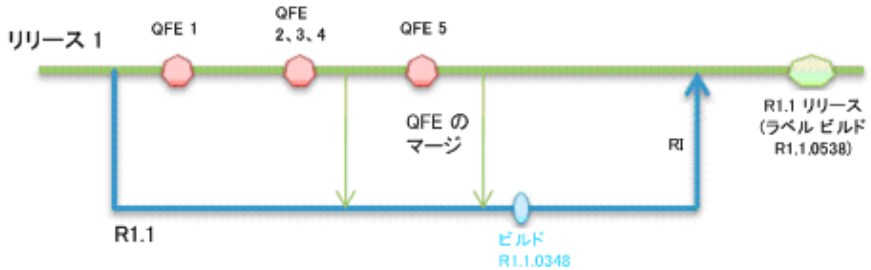
マージする変更セットを選択します。


ソース管理マージ ウィザード
ターゲット分岐にマージする変更セットを選択します
下のリストは、ソース分岐にチェックインされていますが、ターゲット分岐にマージされていない特定の変更セットを含んでいます。このマージ操作で移動する、連続する変更のセットを選択することができます。
マージする変更を選択します(S):
変更セ...
日付
ユーザー
コメント
13
2008/04/26 18:46
Kazuhiko Kondo
新規機能の追加
14
2008/04/26 18:46
Kazuhiko Kondo
バグの修正 (#151)
15
2008/04/26 18:47
Kazuhiko Kondo
機能追加 (マトリクス検索)
16
2008/04/26 18:47
Kazuhiko Kondo
バグの修正 (#157)
< 前へ(P)
次へ(N) >
完了(F)
キャンセル

マージを完了します。



シナリオ 3：継続的なエンジニアリング：リリースに対するメンテナンス作業



このシナリオでは、John のチームが 1.1 リリースと 1.2 リリースに対してどのようにサービスを提供するかを示します。継続的なエンジニアリングと QFE (Quick Fix Engineering, 緊急修正プログラム) は、どちらも運用環境でのリリースのメンテナンスをサポートする活動を指します。これらは、以下のバグの修正が必要になったときに行われます。

1. 初回リリース時に検出されなかったバグ
2. 運用環境に追加された別のソフトウェアによってもたらされたバグ

3. 前のリリース サイクルの際には修正が延期されたが、現在は修正が不可欠と考えられるバグ
4. ハードウェアの変更によってもたらされたバグ

このガイダンスでは、QFE 修正を直接リリース 1 (継続的なエンジニアリング) 分岐に適用して、分離を分岐レベルではなくワークスペース レベルで提供することをお勧めします。その後、必要なテストと検証が行われ、1.X リリースにマージされたら、これらの QFE にラベルを設定することができます。

この例で使用している分岐構造では、最新のリリースに対する継続的なエンジニアリングのサポートを考慮しています。サポートする過去のリリース バージョンがいくつかある場合、分岐構造は異なり、リリース 2 の分岐が加わりますが、同じ原則が適用されます。

手順 4： 物理構造の作成

John は、ここまでのところで骨の折れる作業を行ってきましたが、今度は紙の上に書き留めてきたことをすべて Team Foundation Server に移す必要があります。最初の手順はチーム プロジェクトの作成です。

John は、チーム プロジェクトの作成に関するホワイトペーパーに記載されている理論を応用し、アプリケーションごとのアプローチとチームごとのアプローチの両方を利用することに決めました。この目的で、Client Insurance、Internal Analytics、Web site という 3 つのチーム プロジェクトを作成しました。

John の会社では各アプリケーションに携わる開発者の数が非常に多いので、分岐構造に機能分岐モデルを採用することに決めました。John は次のような分岐構造を用いることにしました。

付録

ここでは、この資料に記載されている推奨事項を理解するのに役立つ補足資料として、追加のトピックをいくつか扱います。

ソフトウェア構成管理

ソフトウェア構成管理 (SCM) は、「ソフトウェア システム内の構成アイテムの識別と定義、ソフトウェア システムのライフ サイクルの中で行うこれらのアイテムのリリース、バージョンと変更の管理、構成アイテムと変更要求の状態の記録とレポート、および構成アイテムの完全性と正確性の検証を行うプロセス」です (出典 : Infosys)。

SCM の実践により、開発チームの全メンバが効果的に共同作業を行うのに役立つ基盤が提供されます。高層ビルを支えるのに堅固な基盤が重要であることはだれも疑いませんが、SCM がソフトウェア開発で同様の役割を果たすということを理解している人は少ないようです。

経験している苛立ちや失敗についてチームのメンバが話すのをよく耳にしますが、たいてい、根本原因は、間違った構成のテスト、コードの不適切なマージ、欠陥のあるビルドの展開など、SCM の実践が不適切であることがわかります。

SCM を適切に実践すると、次のようなメリットがあります。

- 知的財産 (ソフトウェア資産) が保護されます。
- チーム メンバ間のコミュニケーションの向上に役立ちます。
- 明確な担当と責務を確立する方法が提供されます。
- トレースと再現が可能になります。
- 再利用が容易になります。
- 一貫性、信頼性、および整合性が提供されます。

SCM が最も影響を与えるものは、バージョン管理です。バージョン管理の一般的な目的は、同じようなハードウェア構成、OS 構成、ネットワーク構成でプロセスの自動化を行った場合に同一のソフトウェアができあがるように、独立し、バージョン管理が行われ、必要なソース、ツール、外部の依存関係、プロセスの自動化を備えたリポジトリを提供することです。Team Foundation Server の分岐機能とマージ機能は、ソフトウェア構成管理のこのような重要な領域に対応します。

分岐とマージのアンチパターン

MSDN で公開されている、Chris Birmele のホワイトペーパー (http://www.microsoft.com/japan/msdn/vs05/vsts/BranchMerge.aspx) については既に紹介しました。

上記のホワイトペーパーから引用したアンチパターン (非推奨の事項) の一覧をここに挙げておきます。組織で実施する分岐とマージについての話し合いを成立させるのにさらに役立つことでしょう。開発環境で以下の現象の 1 つ以上が発生するとしたら、それは懸念すべきことである可能性があります。

- マージ恐怖症 : 何としてでもマージを避けること。たいていは、結果を恐れていることが原因。

- マージ マニア : ソフトウェア資産の開発ではなくマージに多くの時間をかけすぎること。
- ビッグ バン マージ : 分岐のマージを開発作業の最後まで先延ばしにし、すべての分岐を同時にマージしようとすること。
- 終わりなきマージ : マージするものは常にあるため、絶え間なくマージ作業を行うこと。
- 間違った方向へのマージ : あるソフトウェア資産のバージョンを、以前のバージョンとマージしてしまうこと。
- 分岐マニア : 明白な理由もなく、多くの分岐を作成すること。
- 連鎖分岐 : MAIN にマージして戻らずに分岐すること。
- 一時分岐 : 分岐を行う理由が変化するため、分岐が永久的に一時ワークスペースになってしまうこと。
- 不安定な分岐 : 他の分岐と共有されていたり、他の分岐にマージされたりする、不安定なソフトウェア資産で分岐を行うこと。
    - 注 : 分岐が独立した分岐として存在する限り、ほとんどの場合、それらは不安定です。そこに分岐を行う意義があります。違いは、不安定な状態にある間は分岐を共有またはマージするべきではないという点です。
- 開発の凍結 : 分岐、マージ、および新しい基本ラインのビルドを行う間、すべての開発作業が停止すること。
- 分割を目的とした壁 : メンバが携わっている機能に基づいて分割を行うのではなく、開発チームのメンバを分割するために分岐を使用すること。

自分や自分のチームが上記の状況のいずれかに該当する場合は、ある程度時間をかけて、根本原因を理解し対処法を特定することをお勧めします。

Team Foundation Server での分岐と Visual SourceSafe での共有の比較

Team Foundation Server での分岐は、Visual SourceSafe の共有機能とは異なります。分岐ではファイルやフォルダの相違が許可されており、ソース ファイルおよびソース フォルダと、ターゲット ファイルおよびターゲット フォルダとの間の変更は、明示的にマージ操作が実行されるまで自動的に同期されることはありません (つまり、ソースを変更してもターゲットは更新されず、またターゲットを変更してもソースは更新されません)。Visual SourceSafe では、変更は自動的にマージされます。Team Foundation Server では、分岐の使用が堅牢性の高い変更管理方法として推奨されているため、共有はサポートされていません。

ラベル付けと分岐の比較

Team Foundation Server の "label" コマンドを使用すると、ユーザーが保存および取得できる一意のラベルを使用して、Visual SourceSafe と同様に、特定の状態 (ワークスペース バージョンやタイムスタンプ) に基づいたソース コードのスナップショットを保存することができます。ラベルを使用する典型的な状況は、ソース コードのスナップショットが特定の日に正常にビルドされたことを示す場合です (これは、チーム ビルドによってビルドごとに自動的に行われます)。特定のユーザーに与えられたアクセス許可によっては、ラベルの変更が可能です (ラベルからファイルを変更、追加、削除することができます)。ラベルは強力な機能ですが、以下のことを考慮すると、ラベルの乱用には注意が必要です。

- Team Foundation Server では、ラベルに加えられた変更の履歴は保持されません。
- 特定のアクセス許可が与えられると、ラベルを削除したり、ラベルに対する変更を無効にしたりすることができ、このような変更を監査する方法はありません。
- ある 1 つのラベルやそのラベルに含まれるファイルを、複数の開発者が使用または変更する場合、競合が発生することがあります。

このような理由から、ソースのスナップショットが短時間だけ必要な場合、またはラベルが変更されないことをアクセス許可を使用して保証できる場合にだけラベルを使用することをお勧めします。

このような場合以外で、ソースのスナップショットを比較的長時間にわたって使用する必要があったり、変更の履歴や監査が必要であったり、また、特に、数人のユーザーがスナップショットにアクセスし、ファイルやそのコンテンツを変更する可能性がある開発作業については、分岐を使用することを強くお勧めします。分岐を使用すると、その名のとおり、ファイルやフォルダを元の状態から分化させたり (元の状態も保存されます)、ファイルやフォルダに履歴を含めたり、ファイル レベルやフォルダ レベルの優れたアクセス許可を使用して変更を管理したりすることができます。

分岐と準拠

Team Foundation Server の分岐は、高度なセキュリティ機能を備えており、米国企業改革法 (Sarbanes-Oxley Act of 2002、略称 SOX 法) などの規制の遵守がきわめて容易になります。これがまさに、さまざまなバージョンのリリースに対応する保管用の読み取り専用分岐を推奨する理由です。リリース済みのソフトウェアに対応するソース コードの監査や不可侵性の要件に応えるだけでなく、以前のリリースをデバッグするためのより単純なメカニズムを提供します。

準拠と監査が必要な場合にラベルを使用することはお勧めしません。ラベルは変更可能で履歴が保持されないため、ラベルへの変更の監査結果を記録することはできません。保管用の分岐は、こうした機能をすべて提供するため、ほとんどの準拠の要件に対して推奨される解決策です。

環境の特定

分離のバージョンを管理するだけでなく、ソフトウェアを対応するハードウェア環境に展開して、さまざまな段階でソフトウェアの状態をプレビューしたり、運用環境でソフトウェアをサポートしたりできるようにする必要があります。環境は、1 台のデスクトップ コンピュータのように単純な場合や、分散アプリケーション向けの大規模サーバー群のように複雑な場合もあります。それぞれの環境を作成する動機は、分岐を作成する動機と同じです。つまり、管理された方法でソフトウェアに対する変更から環境を分離します。通常、大半の企業では、次の 1 つ以上の環境が用意されています。

- **開発** – すべての開発作業が行われる環境。すべての変更がこの環境で行われるため、最も不安定な環境です。開発環境には、MAIN にチェックインされる最新バージョンのソフトウェアが反映されている必要があります。
- **テスト** – すべてのテストがこの環境で行われます (多くの場合、この環境はバーチャル サーバーで構成されます)。通常、テスト環境には安定した "ドロップ" (多くの場合は 1 週間または 2 週間に 1 度) のみが提供されるため、開発チームは、ビルドをテスト可能な状態にしてからテスト環境にドロップすることに焦点を絞って作業に取り組むことができます。
- **ユーザー受け入れテスト (UAT)** – UAT 環境は、開発環境やテスト環境よりも安定しています。これは、デモを行う際によく使用される環境で、プロジェクト担当者はソフトウェアをテストしてサインオフする際に使用します。
- **運用前** – この環境に伴うコストは高くなる可能性があるため、この環境を使用しない企業もあります。運用前環境では、運用環境のみで発生する問題 (データベース クラスタの問題や Web ファームの問題など) を早期に特定できるように、運用環境と同じか類似したハードウェアが必要です。
- **運用** – 実際のアプリケーションの使用がサポートされる環境。
- **メンテナンス** – アプリケーションがリリースされると、開発チームは引き続きそのアプリケーションや製品の次期メジャー バージョンのビルドを行います。その際、メンテナンス環境は、メンテナンス用のコード ベースに対して行われたバグ修正や変更のテストに使用されます。

次の表では、上記の各環境に対してリリースを作成する際に一般的に使用される分岐を示します。

環境	分岐	補足説明
開発	DEVELOPMENT	通常、開発環境は、DEVELOPMENT ラインのいずれかから最新の適切なビルドを反映します。
テスト	通常は MAIN、場合によっては DEV 分岐	通常、テスト環境には、MAIN ラインからのソフトウェアのビルドを含めます。場合によっては、DEVELOPMENT ラインからのビルドも含めることができます。
ユーザー受け入れテスト	通常は MAIN。ユーザー受け入れテストに合格する必要のあるサービス リリースをプレビューするときは PRODUCTION の場合もあります。	ほとんどの場合、UAT 環境では、MAIN ラインからの高品質のビルドを実行します。
運用前	PRODUCTION	ほとんどの場合、運用前環境には PRODUCTION ラインからのビルドを含めます。
運用	PRODUCTION	PRODUCTION には、必ず PRODUCTION ラインからのビルドを含めます。

バーチャル ビルド ラボ

マイクロソフトの非常に大規模なチームは、機能分岐で必要なレベルの分離が提供される規模ではなくなったため、1 つ上のレベルの分離であるバーチャル ビルド ラボ (VBL) を採用しました。機能分岐モデルとバーチャル ビルド ラボ モデルの主な違いは分離の範囲です。VBL では、チームの分離を提供します。その他の点では、分岐とコード昇格に関するベスト プラクティスの考慮事項がすべて当てはまります。また、上位レベル (機能レベルとは対照的なチーム組織レベル) で分岐を開始すると、コード昇格パスが複雑になるため、結果として、マージ方法が複雑になると同時に、分岐間の待ち時間が長くなります。待ち時間が長くなる主な

理由は、コードを MAIN パイプラインに昇格する際に、品質ゲートのいくつかの基準を満たす必要があるためです。このようなマージの複雑さと待ち時間を回避するには、深く入れ子になった構造ではなく、よりフラットな分岐構造を選択することをお勧めします。

バーチャル ビルド ラボ (VBL) は多数の分岐を使用して、800 人以上の開発者が所属する組織やチームに必要な分離を提供します。VBL の詳細については、Brian Harry のブログ (http://blogs.msdn.com/bharry/archive/2005/11/12/492198.aspx) (英語) を参照してください。