



## ***Team Foundation Server Branching Guidance***

***John Jacob, Mario Rodriguez, and Graham Barry***  
***Microsoft Corporation***

## **Contents**

Introduction.....	4
Parallel Development.....	4
Branching Defined .....	5
Creating Isolation in Team Foundation Server .....	5
Isolation at Individual Level.....	5
Isolation for Collaboration.....	6
Branching .....	6
Merging.....	7
Branching Scenarios.....	9
General Branching Structure Guidance .....	10
Branching Checklist .....	10
Branching Strategies .....	12
Broad Areas of Branching Isolation.....	13
Creating Your Branching Strategy .....	14
Single Release Scenario .....	14
Multiple Release Scenario.....	15
Multiple Feature/Team Scenarios:.....	15
Defining Your Code Promotion Model.....	16
Code Promotion Criteria.....	18
Quality Gates.....	18
Forward/Reverse Integration.....	18
Code Promotion Best Practices .....	19
Baseless Merges.....	19
Feature Crews: How Microsoft Does It .....	21
Feature Crew and Feature Defined.....	22
Team Composition & Values:.....	23
End-to-End Branching Implementation: .....	24
Step1 – Deciding on a Team Project structure .....	25
Step2 – Deciding on a branch structure .....	25
Step3- Following a release on paper.....	27

Scenario 1: Releasing a Beta for Release 2 .....	27
Scenario 2: Releasing 1.1 and merging some code fixes to Release 2 .....	29
Scenario 3: Sustained Engineering: Maintenance Activities on Releases .....	31
Step 4- Creating the physical structure.....	32
Appendix.....	33
Software Configuration Management .....	33
Branching and Merging Anti-Patterns.....	33
Branching in Team Foundation Server vs. Sharing in Visual SourceSafe .....	34
Labeling vs. Branching.....	34
Branching and Compliance .....	35
Identifying Environments .....	35
Virtual Build Labs.....	36

## Introduction

As you adopt Team Foundation Server in your organization, there are some things that you just need to dive in and learn by doing, and there are other things that you need to research and plan for because decisions that you make in these areas can have a big impact down the road. This paper is intended to provide some advice on one of those areas, branching and merging of software.

Branching and merging of software is a very large topic, and it's an area where there is a lot of maturity in the software industry. Established software organizations have been evolving branching and merging practices for many years, and there is a lot of wisdom out there that people in our industry have shared. Similar to the spirit of design patterns, there are documented branching patterns that cover just about any scenario that you could conceive of. This paper is intended to be a primer of these concepts, and is intended to be a practical introduction to branching and merging for organizations adopting or thinking about adopting Team Foundation Server. This is not meant to be a comprehensive source on this large and intriguing topic area. Once you have read this paper and want to get more viewpoints and dig deeper, please visit [www.cmcrossroads.com](http://www.cmcrossroads.com). This site includes detailed papers which go very deep into Software Configuration Management topic areas including branching and merging. We have also included some specific links at the end of this paper.

There are certain concepts in this paper that are relevant to teams of all sizes, some that will be more relevant to smaller teams (around 20 or fewer developers), and some that will be more relevant for larger teams (20 to 1,000 developers). There is additional guidance for extremely large teams (thousands of developers) which we will touch on in the appendix (Virtual Build Labs). We will address some concepts in the first couple of chapters that are relevant to teams of all sizes, then we get into advice specific to smaller teams, then we cover advice for larger teams, and finally tie it all together with the final section intended for all teams.

## Parallel Development

The desire to enable parallel development is the primary reason that branching and merging was created. Parallel development is a practice which enables development teams working on different areas of the same software project to work simultaneously while minimizing the chance of conflicting changes, thus minimizing wasted effort due to rework or defect resolution. Here are some examples of parallel development:

*Example 1:*

A team of 40 developers is building an application. There are four feature teams, each led by a development lead. The feature teams vary in size from two developers to fifteen developers. The milestones for each feature area also vary. There needs to be a mechanism to provide isolation to each of the feature teams while allowing for changes to the common areas of the application in a reliable and controlled fashion. Branching is a very good solution here; this is referred to in this document as Branching for Feature Team Isolation.

*Example 2:*

A development team has just gone live with the first release of a Web site. The development team has started working on the next version of the site, but a critical bug is found by an important customer on the live site. A strategy needs to be introduced to allow the core development team to continue evolving the next version of the site while bug fixes can be made for maintenance of the released site. There needs to be a mechanism to isolate these two work streams. This is referred to in this document as Branching for Maintenance.

From these examples, it is easy to generalize parallel development as being either:

- Development activity on the same code base, implementing different features or bug fixes
- Development activity on different releases of the same software code base.

The key necessity for parallel development is a mechanism by which code can be isolated and stabilized without the danger of cross-contamination of other ongoing development, and a mechanism by which such isolated changes in one branch can be integrated back into other development branches.

## Branching Defined

Branching enables parallel development by providing each development endeavor (bug fix, feature development, stabilization of code, etc.) its own, self contained snapshot of needed sources, tools, external dependencies and process automation. Such a self-contained snapshot effectively enables each development activity to proceed at its own pace, without taking any dependency on another. It follows that these snapshots are allowed to diverge their respective sources along the particular development activity they are involved with – fixing a bug, implementing a feature or stabilizing a breaking change.

Branching typically involves:

1. Taking a snapshot of source code to create isolation – a snapshot may be as of a certain point in time, or a stable or known state of source code (such as the last successful build). The resulting copy is the **child branch**, and the source from which it was created is called the **parent branch**.
2. Containing and stabilizing changes within the isolated snapshot in the child branch.
3. Bi-directional synchronization of changes with the parent branch – usually referred to as integrating or merging. We will explain why this is important later in this document.

## Creating Isolation in Team Foundation Server

As noted above, there is a need for an isolation mechanism and a merging mechanism – both of which can be addressed by functionality found in Team Foundation Server. First, let's discuss some isolation mechanisms available in Team Foundation Server.

### Isolation at Individual Level

In case you need code isolation on an individual basis – that is, as a developer you need to work on different states or snapshots of a given source code-base, you can effectively use Team Foundation Server's capabilities for creating multiple workspaces off the same source code. Workspaces are your

client-side mappings of source code onto your local disk – see [Working with Source Control Workspaces](#). You can have several such mappings on your local hard disk, each synchronized to a different state. For example, a workspace might be mapped to the label of the last successful build, or the latest versions of files on server. Each such workspace provides you the isolation for you to make changes without the dangers of cross-contamination between workspaces. This is because Team Foundation Server tracks changes (check outs, pending changes, edits, etc.) specific to each of your workspaces, no matter how many instances you have created. And, by virtue of having the workspaces mapped to the server, you implicitly have a way to merge your changes onto the server by checking in the pending changes in each of the given workspaces. In case you have to switch between workspaces and are not ready to check in your pending edits, you can safely put away your changes as a shelveset using Team Foundation Server's [Shelving](#) feature.

## Isolation for Collaboration

While workspaces provide the necessary isolation for different states or phases of coding on your individual local disk space, modern software development is more often than not a team effort. As such, we also need to address the need for isolation at a team level – that is, where two or more developers need to collaborate on a fix, feature, or breaking change, without risking the stability of source code that other developers are actively working on. This is where Team Foundation Server's rich branching functionality comes into play.

## Branching

Team Foundation Server defines branching as a feature that allows a collection of files to evolve in two or more divergent paths. Branching is unique in that it's the only isolation mechanism that provides for collaboration amongst a group of people and versioning of changes within that isolated space.

Team Foundation Server branches in "path space," which is analogous to copying a folder in Windows Explorer. However, unlike copying, branching maintains the history relating the source to the target to allow merging future changes.

Because branches in Team Foundation Server create new copies, they are easy to navigate; in fact branches appear in the Source Control Explorer just like the source items. It also makes it very easy to simultaneously work on as many branches of the same code base as you need - just grab that folder as you would any other folder.

When it comes to maintaining permissions for branches, it's easy to set permissions on any folder in the server. Branches are regular paths and use the same path-based permission access control as any other folder in your source tree. Open Source Control Explorer, navigate to the folder, right click on it, choose Properties from the popup menu, and click on the Security tab. From the command line, you can use the [permissions](#) command.

Creating branches uses very little additional storage space. The server minimizes the storage required by only keeping one copy of identical content no matter how many different files are contained in the folder. So, if you have 100 copies of a 1 MB file and all of the files are identical, the server will store only 1 MB, not 100 MB. When you create a new branch and commit, all of the files in the new branch that are

identical to the files in the source branch point to the same content. The result is that a branch consumes very little additional storage space, and that storage space expands only when the branched file becomes different than the source. And even when files change, Team Foundation Server employs a differencing engine to analyze changes between files and once again optimize storage space.

To create a branch, you can choose to branch by date, label, workspace version (i.e., the versions you last retrieved into your workspace), or the latest server version. Furthermore, you can mix and match as needed. For example, you can branch one folder based on a date and another folder based on the latest version of its items.

Here is how to create a branch of the latest version of the source code:



For more detailed documentation on branching with Team Foundation Server see: [Branching and Merging](#) [Team Foundation Source Control](#)

## Merging

Merging is the logical corollary to branching in Team Foundation Server. Merging in version control is the process of combining changes that have transpired in two distinct branches. A merge operation takes changes that have occurred in the source branch and integrates them into the target branch. Merging integrates all types of changes in the source branch including name changes, file edits, file additions, file deletions, and undeletions. The merge engine is able to do this because Team Foundation Server keeps a record of all changes including the relationships of a child branch to the parent branch.

If items are modified in both the source and target branches, a version conflict will be filed and you will be prompted to resolve those conflicts.

Once you have created a branch, moving changes from the source branch to the target branch or from the target branch back to the source branch is very easy. Team Foundation Server maintains the relationships of branched items and the merge history.

When merging across branches, all of the changes that have been made in the source and target branches are merged. This includes additions, deletions, undeletions, and moves (which is effectively a rename operation). For example, if an item in the source branch has been renamed from A.TXT to B.TXT, when performing a merge that change will also rename the corresponding item in the target branch.

For files where you have made changes to both the source and the target, merging the changes will result in conflicts. These conflicts are maintained by the server, and the client provides a dialog (both from the command line and Visual Studio) that shows the conflicts and leads you through the process of resolving them. For each conflict, you need to choose whether to accept the source or target changes or a combination of them. If there are edits to the file, you can choose to have the changes merged for you automatically. If the changes cannot be merged automatically or you would like to manually merge the files, the conflict resolution dialog will launch a graphical three-way merge tool.

As mentioned with branching, your merge becomes a set of pending changes that are contained within your workspace. You can make additional changes to the result of the merge, such as changing or moving files or fixing build issues, before committing the merged changes. The entire set of merged changes is committed atomically as a single changeset.

Once a merge has been committed to the server, it becomes part of the merge history of the items involved. The next time you need to merge a set of changes, Team Foundation Server will indicate which changes have not yet been merged. You don't have to keep track of the merges - the system takes care of this for you. At any time, you can get this information from either the command line or from the Source Control Explorer in Team Explorer.

You can also choose what changesets you want to merge. You may want to choose to merge only a subset of the changesets so that you can merge a single bug fix without merging all of the changes that occurred before it. Knowing what change fixed a particular bug becomes simple with the integrated Work Item Tracking. When a developer checks in a changeset, he or she can link that changeset to a relevant work item which might represent a bug or a feature. So finding what changesets need to be merged to propagate a bug fix simply requires clicking on the Links tab of the work item to find the appropriate changeset or changesets to merge.

Merging is a painless process within Team Explorer. The Merge Wizard leads you through the steps of choosing which changes to merge, allowing you to merge every change since the last merge or to pick only particular changes. In the Source Control Explorer, simply right-click on the folder from which you want to merge changes (the source of a branch). Clicking the Merge item on the popup menu will bring up the Merge Wizard. The Merge Wizard already knows what the possible targets are, so you simply select the target from a list (there's only one entry if you've only branched the folder once). You can elect to merge all changes or just selected changes. If you choose selected changes, you will be presented with changes that haven't yet been merged. After selecting the changes to merge, click Finish, and you will have merges pending for each file involved. Check in the merge when you are satisfied that it builds correctly. That's all it takes to merge changes.

It is worth noting that if you are merging only a subset of the available changesets the Merge Wizard will require that you select a consecutive range of changesets. If you wish to merge a non-consecutive subset of changesets you can do so by re-launching the Merge Wizard for each individual range of changesets.

## Branching Scenarios

We've just been introduced to Team Foundation Server's branching feature as an isolation mechanism for group collaboration in software development. Let's now look at typical ways development activity can be grouped together to provide isolation:

- ➔ **Release Isolation:** If you need to work on multiple releases in parallel, you will probably want source code for those releases to be isolated in their own branch – hence **release isolation**. This is the most common need that causes organizations to start branching their sources. Keep in mind that the term "release" here does not necessarily refer to a new version of your product. It may be appropriate to release your application to your test team from a dedicated branch so that you can work through issues with that release independently from the new development that is continuing on your development branch.
- ➔ **Feature Isolation:** It is common for teams to embark on new functionality that is either considered experimental or is considered risky enough to merit that work being performed in its own branch. In such cases, **feature isolation** allows the people working on a given feature to collaborate without exposing the rest of the application to any instability that may result.
- ➔ **Team Isolation:** It is also common that organizations elect to have sub-teams work in isolation from each other. **Team isolation** provides the ability for these teams to collaborate without being subjected to the potentially breaking changes that other teams are working on. Some organizations go so far as to create a dedicated branch for every developer on the team. This is not required with Team Foundation Server because of the nature of client-side workspaces discussed earlier.
- ➔ **Integration Isolation:** Merging changes between branches is often a very destabilizing event. It is sometimes beneficial to maintain an **integration** branch where active development does not take place but rather serves as a staging area for merges. Changes from other branches can be merged into this branch and then stabilized. Once stable, the changes can be merged back into the other branches. In some organization, the term "breaking changes" refers to the potential destabilization bug fixes or merges can introduce – accordingly they provide for a "breaking changes branch" to accommodate the need for integration isolation.

The scenarios above are common to most software development organizations – the various levels of isolation requirements for each of these are capably addressed by Team Foundation Server's rich and robust branching functionality. On the other hand, we should also note that a common mistake that some organizations make is to over-branch. The benefits of isolation, therefore, should be weighed with their costs so that you can make an appropriate decision about the amount of process overhead you're willing to incur in your development. The costs of branching can be broken down as follows:

- ➔ **Merging:** Despite the benefits offered by isolation, the process of merging still involves a fair amount of effort to resolve potential conflicts and fix any new bugs which may arise from the

merge operation. During a merge operation, you often require one or more people from your team to reconcile a set of conflicting changes that have been made in different branches.

- ➔ Latency: It takes time to move changes between branches. A typical merge operation involves stabilizing the source branch (if it's not already stable), executing the merge, resolving conflicting changes, and then stabilizing the target branch. For small source bases, this can be done in minutes, but for large source bases it can take days. If moving changes between isolation areas involves merging through multiple branches, it can begin to take weeks to move changes through the system.

We will discuss many of these isolation mechanisms together with their costs in order to provide a balanced set of guidance for you to consider..

## General Branching Structure Guidance

So far we have seen how Team Foundation Server enables parallel development through its rich collaborative isolation mechanism – branching and merging. But in order to best leverage these capabilities we also need to establish a strategy for branching and merging operations. Just as you would organize a team of people based on activities, shared goals and outcomes, it helps to put some forethought and planning into the structure and organization of your branches in Team Foundation Server.

### Branching Checklist

We recommend the following checklist for branch creation:

1. Choose an isolation model – which could be informed by the need for one or more of the following:
  - i. Consolidating common workflows with respect to your software development activities. This might include feature development, quality assurance, stabilization, release readiness, and sustained engineering.
  - ii. Features that are tightly coupled – have code that is inter-dependent and needs to be compiled together (sharing common headers, libraries, for example) and/or require binaries that have dependencies on each other at runtime.
  - iii. Providing well-demarcated areas of responsibility for development activity
  - iv. Ensuring strict inviolability, accountability and auditing of source code
  - v. Representing different phases of your development lifecycle
2. Define a code promotion path for development activity which defines how your parallel development activity eventually get integrated into the mainline branch.
3. Define a set of quality criteria that need to be met before code can be merged for promotion between branches.
4. Define the key stakeholders for each branch – these are people who will decide on granting permissions to users of the branch, control and regulate check-ins into that branch, and enforce the quality criteria you've defined for accepting merges to and from another branch.
5. Once #4 is decided, define the permissions for each branch – read, write, branch, label, etc.

The simplest way to establish branches is along the three broad phases of development activity in a typical organization, namely:

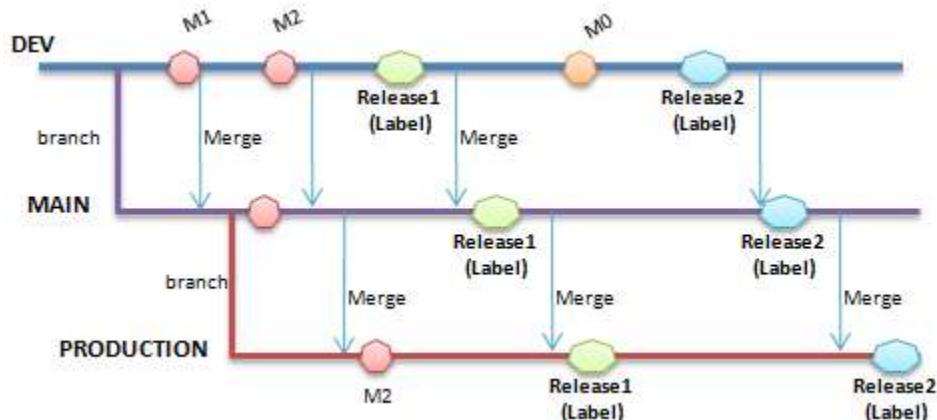
- DEV – activity around new feature development and bug fixing
- MAIN (also called TEST or INTEGRATION) – activity around ensuring product stabilization and readiness for release
- PRODUCTION – activity around providing sustained engineering services for released products

The DEV branch is for staging all new development activity whether it is for feature development, bug fixing, or integration of breaking changes. This area is designed to isolate, contain, and stabilize new development .

The MAIN branch exists in spirit in most organizations, but it goes by a few different names. Some organizations refer it as "TEST," others as "INTEGRATION," and another common name is "MAIN." We'll refer to this branch in this document as MAIN, but please keep in mind that the intent is identical for this branch whether you call it MAIN, TEST, or INTEGRATION. It's the branch that is used as a holding tank to integrate changes to and from feature branches, and it should be kept as stable as possible.

The PRODUCTION branch contains the sources of your released products, or soon to be released products. Since this branch houses sources of released products or soon to be released products, quality of code is very stable and guaranteed to be as such via strict control of changes.

A release of an application that follows this strategy will look like this:



Highlights summary:

- Code for features and bug fixes is developed in the DEV branch.
- At significant milestones and upon meeting some well-defined quality criteria, code gets merged into MAIN periodically.
- Code in MAIN gets stabilized for release criteria such as feature completeness, security, and performance.
- Code is merged into PRODUCTION from MAIN when the quality is high enough for additional testing in production. Releases happen from PRODUCTION.

- As code from each branch gets promoted to the next branch or released, the given branch becomes open for the next milestone.

This is the most basic representation of our branching guidance, one that we will elaborate and improve upon based on several scenarios discussed in the following sections. It is important to note that all of the criteria we've defined in the **Branching Checklist** can be easily accomplished using Team Foundation Server. We'll demonstrate those implementations as we go. In the following sections we will delve into the various implementations of this branching guidance based on the scenarios we described under **Branching Scenarios**, as well as the size of teams we need to serve.

## Branching Strategies

Recall our discussion under **Branching Scenarios** where we described different isolation scenarios – let's consolidate those scenarios with the branching structure described above. The following table summarizes the various isolation scenarios accommodated by our recommended branching structure – DEV and PRODUCTION. Note that we have intentionally left MAIN out of this table. We will explain why this is so shortly.

Isolation Criteria Allowed	DEV	PRODUCTION
Feature Isolation	✓	
Team Isolation	✓	
Integration Isolation	✓	✓
Release Isolation		✓

Our branching guidance allows for several criteria for branching under DEV – we can use it to house our branches for feature isolation, or integration isolation, or at a much higher level provide for isolation along team organizational lines. Similarly, we could potentially have different versions of our product in production or on users' desktops that we need to provide maintenance for – accordingly we provide for one or more branches under PRODUCTION, each pertaining to a particular release we intend to support. For the purposes of this paper, we will start referring to DEV and PRODUCTION as **container nodes** to denote the fact that they contain one or more branches under them. MAIN on the contrary always remains a single branch – and should always be kept as such. This is because MAIN is intended to be the confluence point for all coding activity happening both under DEV and PRODUCTION. Think of it as a railway junction where code meets and makes connections to different destinations. As such, there can only be one MAIN with the singular purpose of stabilization for the next release.

There are other branching strategies that may be appealing for the culture and process of your team that aren't covered in this document. Some of these are:

- Branch per Task
- Branch per Component
- Branch per Technology

There is an excellent whitepaper published on MSDN by Chris Birmele that can provide you with more information on these strategies: [http://msdn2.microsoft.com/en-us/library/aa730834\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/aa730834(VS.80).aspx). Be aware that this section is for information only and you should do the necessary amount of research before implementing any of these strategies.

## Broad Areas of Branching Isolation

<b>Branch:</b>	<b>DEV</b>	<b>MAIN</b>	<b>PRODUCTION</b>
<b>Type</b>	Branch or Container	Always a Branch	Branch or Container
<b>Goal</b>	Feature development, integration and stabilization of breaking changes	Achieving ship-readiness of the next major product release, performance and security stabilization, end-to-end acceptance	Servicing, sustained engineering of released products, or final fixes for the most imminent next release
<b>Ownership</b>	Development Managers	Release Management and QA	Release Management
<b>Permissions</b>	Read-write for developers working on respective features, and some test engineers focused on feature/function level/integration testing.	Read-only for most users. Senior, expert developers responsible for merging/integrating and promoting source code between branches and into MAIN have read-write permissions. Urgent hot fixes to daily build will necessitate read-write permissions on a case-by-case basis.	Read-only for most users. Read-write restricted to selected developers and testers working on hot fixes, quick fix engineering and other sustained engineering efforts. Otherwise very tightly controlled to ensure stability of shipped code, and readiness to provide sustained engineering fixes.
<b>Build Activity</b>	Continuous integration due to heavy code-churn. Need constant feedback on code health via frequent builds and testing iterations.	Daily builds based on whenever code is promoted from the DEV branches into MAIN.	On-demand builds based on DEV/TEST need/readiness.
<b>Test Focus</b>	Feature completion, feature stabilization, feature integration, and core product milestone requirements. Need constant feedback on code health via frequent builds and testing iterations.	End-to-end acceptance testing, performance and security testing. Iterate until the product ready to ship.	Sign-off on release of product or hot fixes.

## Creating Your Branching Strategy

Now that we've described the concepts behind branching and merging in Team Foundation Server, let's turn our attention to how you can describe an optimal branching strategy for your development needs.

### Single Release Scenario



The diagram above represents both the physical layout of the branches as well as their logical representation. The physical layout describes how the branches actually appear on your Team Foundation Server Source Control view, while the logical view represents their inter-relationships. These relationships represent how these branches were created – the arrows point to the child branch in a parent → child relationship. Code promotion – which is accomplished by merging as we described earlier - happens along the relationship path established by the parentage. Team Foundation Server's merge functionality supports a bi-directional merge along these promotion paths.

Let's describe the typical workflow of the development activity staged in these branches.

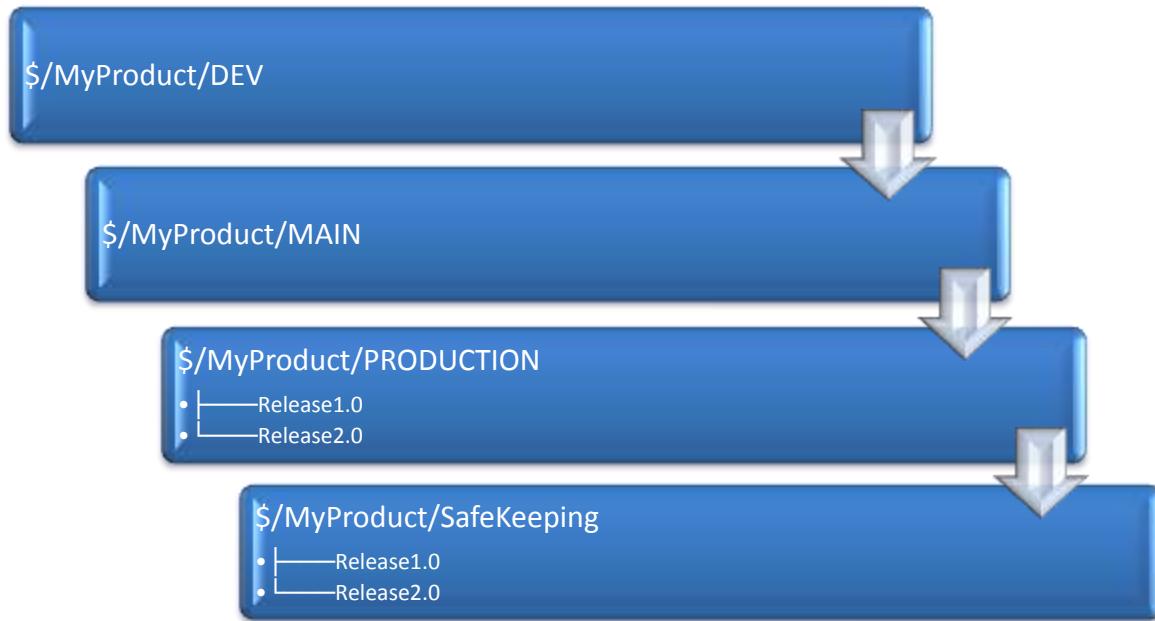
- Code for features and bug fixes is developed in the DEV branch.
- At significant milestones and upon meeting some well-defined quality criteria, code gets merged into MAIN. If MAIN doesn't exist, we create MAIN by branching it from \$/MyProduct/DEV based on some known state - a label, a given date, a changeset, etc.
- Code in MAIN gets stabilized for release criteria such as feature completeness, security and performance.
- Code is merged into PRODUCTION from MAIN for release when the quality is high enough for additional testing and the product is close to release-quality. If PRODUCTION doesn't exist, we create PRODUCTION by branching it from \$/MyProduct/MAIN based on some known state – label, date, etc.
- As each milestone gets promoted to the next branch or released, the given branch becomes open for the next milestone.
- We iterate on this cycle until we reach a quality bar established for release – at this point, a release (Release1.0) is declared from \$/MyProduct/Production.
- We create a read-only Safekeeping branch under \$/MyProduct/Safekeeping/Release1.0 from \$/MyProduct/Production/Release1.0.

- At this point, `$/MyProduct/DEV` and `$/MyProduct/MAIN` become available for the next release (2.0) of the product, while `$/MyProduct/PRODUCTION` becomes available for sustained engineering of Release 1.0.

## Multiple Release Scenario

Our representation above corresponds to a relatively small sized team with just one release to support. Now we will consider a shop that needs to support more than one release at any given time. We modify our branching strategy to support multiple Release nodes under `$/MyProduct/PRODUCTION`, effectively making it a container node, as defined in earlier in our document.

The physical and logical view of the branching structure can be represented as follows:



This is a good example of Isolation by Release as discussed under the Branching Scenarios section. Most of the workflow descriptions under Single Release Scenario still apply, with the exception of branch creation operations for releases. At the time `$/MyProduct/MAIN` is ready for release, it is branched into a version folder to denote the release - `$/MyProduct/Production/Release2.0` for example, and `$/MyProduct/SafeKeeping/Release2.0`.

## Multiple Feature/Team Scenarios:

The next degree of complexity we need to address is to accommodate isolation for parallel development of features for a release. It now makes sense to make `DEV` a container and introduce multiple feature branches, created by branching from `MAIN`. For example:

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

`$/MyProduct/MAIN` → `$/MyProduct/DEV/FeatureBranch2`

`$/MyProduct/DEV` now becomes a container node for various feature branches, while `$/MyProduct/MAIN` becomes the junction point for code promotion between these feature branches.



## Defining Your Code Promotion Model

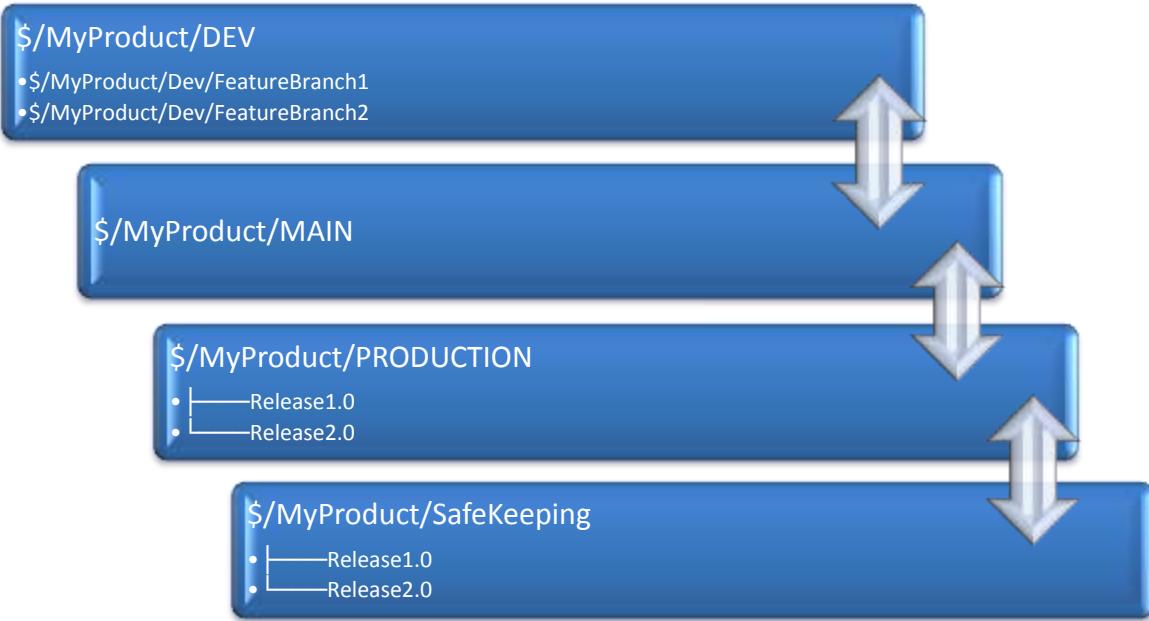
We've so far dealt with the creation of branches and establishing relationships between the branches (denoted by the arrows pointing to the child branch) to provide isolation for various scenarios such as providing isolation for multiple development endeavors or supporting multiple releases. By establishing these relationships, we've also implicitly defined the code promotion path – that is, the path along which code can be promoted to the next branch.

When we created the following branches:

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

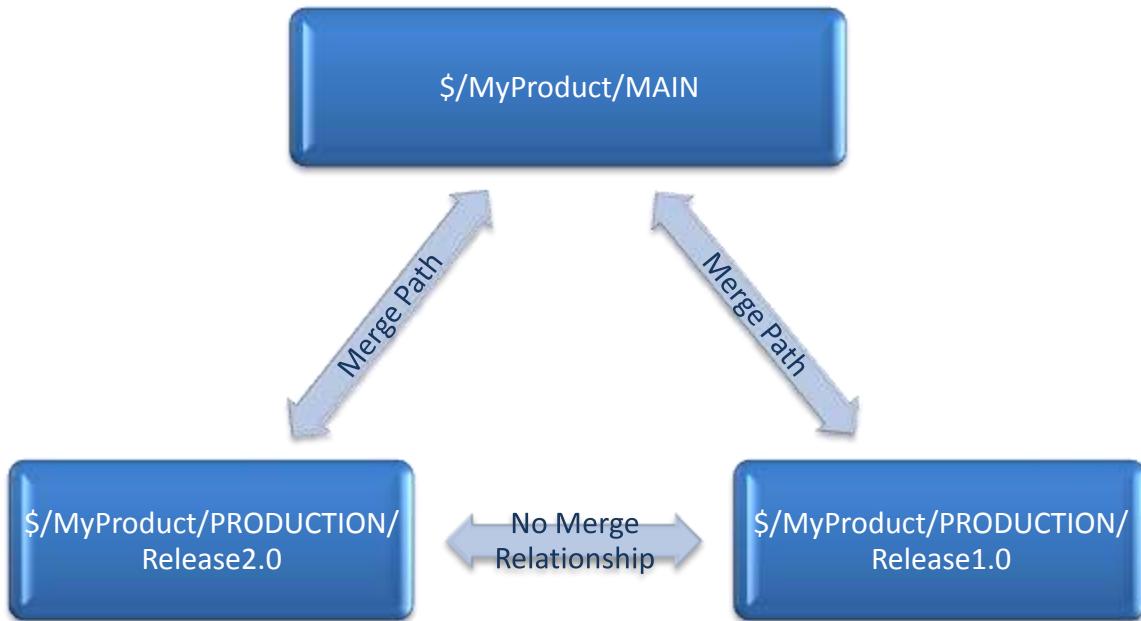
`$/MyProduct/MAIN` → `$/MyProduct/DEV/FeatureBranch2`

we established a parent-child relationship between `MAIN` → `FeatureBranch1`, and `MAIN` → `FeatureBranch2`. While the arrows point towards the child branch in the relationship, realistically it is a bi-directional arrow that best sums up the relationship as stored by Team Foundation Server. The modified diagram illustrates the point:



The bi-directional arrows denote that code can be merged in either direction – parent → child, or child → parent. This is referred to as the “merge path” which Team Foundation Server offers as the default in the merging dialog for a merge operation.

The merge path defines the code promotion path and has great significance with respect to the best practices we are going to describe in dealing with container branches vis-à-vis their parent node. This is especially true in the case of the DEV container branches created from MAIN. Let’s illustrate it with the following diagram that explores the relationship between them:



## Code Promotion Criteria

In creating the parent ↔ child relationships between MAIN and the DEV container branches (FeatureBranch1, FeatureBranch2) we've established a code-promotion path which needs to follow certain criteria for merging operations between them. Let's look at the criteria for code-promotion in this relationship:

1. Enforce and maintain a high code quality bar in MAIN at all times. This is important from two standpoints:
  - a. MAIN hosts the code that is being readied for the next release – we need to guarantee its ship-readiness or at least milestone readiness at all times.
  - b. MAIN serves as the junction/confluence point for development activity happening in isolation in the various DEV branches – FeatureBranch1, FeatureBranch2. That is, if FeatureBranch2 needs fixes contained in FeatureBranch1, it can only get them after those fixes have been merged into MAIN from FeatureBranch1. Any destabilization of MAIN could potentially contaminate other dependent branches.
2. Frontload as much code destabilization in the DEV container branches, and stabilize them before merging back to MAIN (for reasons mentioned in #1).

## Quality Gates

The first criterion we defined for code-promotion was to enforce and maintain a high code quality bar – the best practice guidance here is to have a well-defined, agreed upon and publicized set of quality criteria that need to be satisfied by any DEV container branch intending to merge into MAIN. At Microsoft, these have come to be known as "Quality Gates" which typically consist of the following conditions being met:

1. A successful build of the source code in all flavors (debug, retail) and on all applicable platforms (x86, AMD64, etc.).
2. A successful test run using all the build verification tests that are run on builds off MAIN – these could be a combination of core scenarios, performance and security tests.

## Forward/Reverse Integration

Our second criterion for code-promotion was to frontload and stabilize destabilizing changes (new features, bug fixes, integration efforts) in the DEV container branches before merging back into MAIN. In this respect, we are introducing two new concepts: Forward Integrations (FI) and Reverse Integrations (RI).

**Forward Integration** refers to the operations where the child branches sync up to the latest changes from its parent branch as in the example:

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

Conversely, **Reverse Integration** refers to the operations where the child branch updates its latest changes into its parent branch as in the example:

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

## Code Promotion Best Practices

The merge operations described above – Forward Integration and Reverse Integration – effectively describe the code promotion process for this structure. Certain best practices in this regard need to be emphasized with respect to code promotion.

Given the need to ensure utmost stability of MAIN, Reverse Integration criteria should be stringently enforced for child branches intending to promote code to MAIN. This should involve:

1. A Forward Integration from MAIN, which as defined earlier, is a full sync of the current state of code in MAIN into the child branch. Forward Integrating frequently from MAIN as well as continuous integration builds (see: [Continuous Integration Using Team Foundation Build](#)) well in advance of the final Reverse Integration are recommended as these greatly reduce the effort needed to resolve merge conflicts and stabilize code in the feature branch which will be Reverse Integrated. Effectively, this front-loads the discovery and fixing of build and test issues via several iterations before the final Reverse Integration into MAIN – enforce a policy of “building early and building often”.
2. Satisfying the Quality Gates criteria for the merged code before it is checked into the target branch.

Following these guidelines, the final Reverse Integration into MAIN is less likely to have merge conflicts - unless another branch has checked-in ahead of the current child branch. It is usually prudent to set up a Reverse Integration schedule to avoid being repeatedly overtaken by Reverse Integration from other branches and having to repeat the Reverse Integration passing criteria. More importantly, it allows other development teams to prioritize and coordinate their Reverse Integration efforts in a spirit of collaboration. Release Management and/or QA teams usually own the scheduling of RIs based on requests from individual development teams.

## Baseless Merges

As we've discussed above, the branch command will create a relationship between two folders which you can leverage to perform merges of code between the branches. The relationship is bi-directional so code can be merged both ways but Team Foundation Server currently doesn't facilitate merging between child branches. To begin to understand baseless merges, let's consider the following structure:



In the scenario above, we have well-defined code promotion path between:

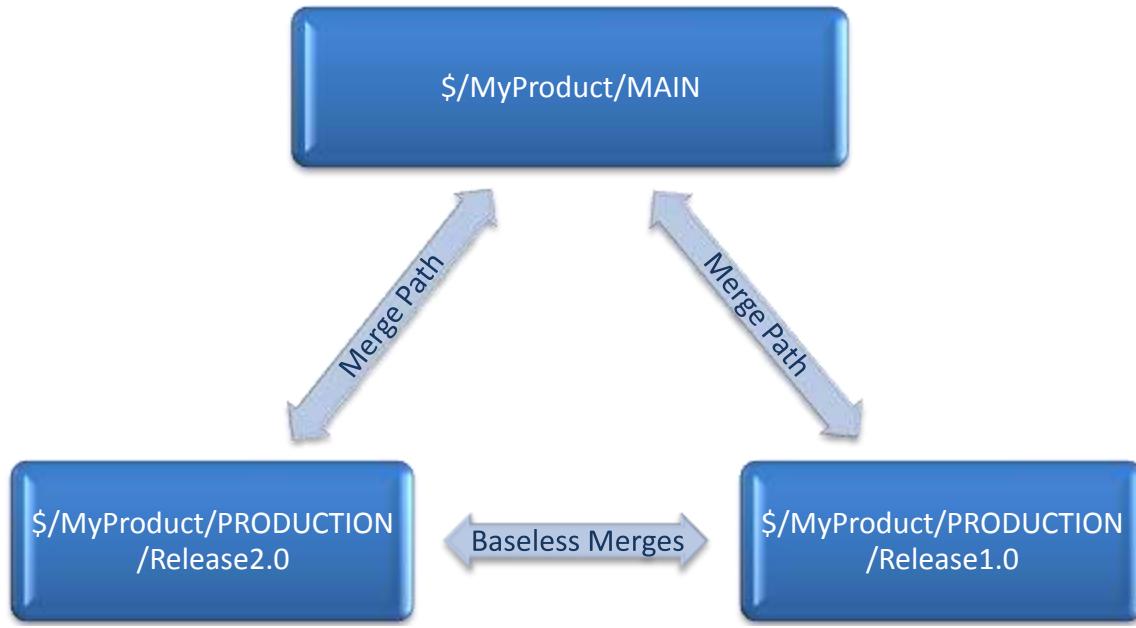
`$/MyProduct/MAIN ↲→ $/MyProduct/PRODUCTION/Release1.0`  
`$/MyProduct/MAIN ↲→ $/MyProduct/PRODUCTION/Release2.0`

These code promotion paths exist because both Release1.0 and Release2.0 under production are children of MAIN. Unfortunately, there is no direct relationship between the child branches - Release1.0 and Release2.0, which makes it difficult for us to merge, for example, a hotfix from Release1.0 into Release2.0 without having to traverse through \$/MyProduct/MAIN.

If we followed the merge path established by the branch command, a fix from Release1.0 will need to be merged in the following sequence for it to be available in the Release2.0 branch:

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

In some cases it may not be feasible to traverse through MAIN because MAIN could potentially have moved far ahead of the source code versions in both Release1.0 and Release2.0, and it would take a significant amount of work to merge, build, and test the hotfix in MAIN. Here is where **baseless merges** become necessary. The following diagram explores the existing relationship and baseless merges.



MSDN defines a baseless merge as “a merge (performed) without a basis version.” That is, it allows the user to merge files and folders that do not have a branch/merge relationship. After a baseless merge, a merge relationship exists and future merges do not have to be baseless.

While baseless merges are a powerful feature that Team Foundation Server provides (only via the command line; see: [Operations Available Only From the Command-Line \(Team Foundation Source Control\)](#)) we recommend that you use this feature sparingly and only under very exceptional circumstances. In most cases, a well-defined code promotion model should provide you with a good way to merge changes without having to resort to baseless merges.

## Feature Crews: How Microsoft Does It

With the advanced concepts behind us it is time for us to discuss our guidance for branching structures which are optimized for large teams. With the release of Visual Studio “Orcas,” Microsoft’s Developer Division went to a **feature crew model**. This process was originated by the Microsoft Office team and has proven successful across the company. The feature crew model and corresponding feature branches will be the basis for our guidance in this section.

The adoption of the feature crew model within Microsoft’s teams is informed by valuable lessons from past shipping experiences in regards to:

- Inability to reliably predict the ship date of a product because, invariably, the stabilization of features was deferred towards the end of the project lifecycle. This made it impossible to predict how long the stabilization would last before the product could be deemed ship-ready.

- Inability to reduce the scope of features to hit a release date because it was difficult to scale back on features which, although had been coded, had not been stabilized. Scaling back on scope was further exacerbated by the fact that newly developed features had introduced other dependencies around them that were hard to back out. Additionally many of those features had already been shown to customers via product demos and there were certain expectations set about them shipping in the official release.
- Inefficiencies due to the significantly long time between when the feature was developed and when it was fully tested.

The lessons learned from the prior shipping experiences shaped the key guiding principles behind the feature crew model. These principles are:

- Driving quality upstream – ensuring efficiency by requiring teams to meet certain quality gates and to stabilize their code before their feature can be merged back into MAIN.
- Preventing deferred work – ensuring work up front to facilitate early bug detection and resolution, rather than deferring it until later in the development phase
- Keeping the MAIN branch customer-ready which allows for more frequent milestone deliveries to customers without the risk of shipping unstable or incomplete feature code, as well as reducing the time required to prepare the final release.
- Enforcing a common definition of what completion means across various teams via a common, consistent definition of quality gates with meaningful metrics on project progress.

## Feature Crew and Feature Defined

A **feature crew** is a small interdisciplinary team responsible for producing a feature. A **feature** is an independently testable unit of work that is either directly visible to a customer or is infrastructure that is going to be consumed by another feature. Features should be small enough that they can be worked on by a feature crew – but large enough that it actually makes sense to test it independently.

Features are defined with a goal to optimize for productivity via short cycles – typically 3-6 weeks - of implementation, stabilization and integration into MAIN. This ensures a steady flow of features into MAIN and enables shipping of frequent Customer Technology Previews (CTPs) for valuable, early customer feedback throughout the lifecycle.

A feature should also be designed with a clear understanding and explicit statement of other feature dependencies that will need to have been checked into MAIN before the given feature can be developed. So it helps to break down big features into smaller, manageable deliveries that are independently testable, and prevents destabilization of the product which may otherwise occur if the features are delivered as a big, end-to-end endeavor.

## Team Composition & Values:

A typical feature crew is a program manager, five or less developers, and a corresponding number of testers that will work together for 3-6 weeks as a closely knit team with the common goal of hitting feature complete (as defined below).

- Everyone in the feature crew works together within the same timeframe.
- Developers and testers work in tandem – code gets tested as it is developed, so we minimize the lag time between feature development and its quality validation. Feature development happens in feature branches which provide the code isolation needed for the feature crew.
- A feature is “Feature Complete” when it has been fully implemented, is stable, is well tested, and it is either ready for others to consume. This consumption may be in the form of internal testing (aka “dogfooding”), by sharing with external customers in a CTP, or for infrastructure features it is ready to be coded against by another feature crew.

It is easy to see the benefits realized by the feature crew model at Microsoft. It provides teams with a sense of empowerment and collaborative spirit by working on a well-defined, focused goal. The feature crew model enables faster turnaround and iterations between developers, testers, and program managers. It also enables sharing of code between different teams without risking the destabilization of the MAIN branch. Individually, too, the developer division observed improvements in morale due to the focused and collaborative environment which teams worked in.

While most of the best practice guidelines we have discussed with respect to branching and merging still apply in the feature crew model, it helps to summarize and underscore the following key engineering approaches that make the approach unique and effective:

- Drive to Feature Complete, not simply Code Complete. Finish a feature before adding it to the product - spec, design, implement, automate, test and fix bugs
- Use Quality Gates to provide consistency across the division. Define the divisional requirements for Feature Complete. These Quality Gates must be satisfied before adding a feature to the product
- Use Feature Branches to isolate new feature development. Each feature gets its own branch.
- Only Reverse Integrate the feature branch into the PU branch (covered below) once each of the following is met:
  - The feature is complete
  - The Quality Gates are satisfied

One additional characteristic of the Feature Crew model is the introduction of a PU (Product Unit) branch as an intermediate staging branch between the Feature Branch and MAIN. As such, the promotion path for merging now becomes: Feature Branch  $\leftrightarrow$  PU Staging branch  $\leftrightarrow$  Main, after satisfying a set of Quality Gates. These Quality Gates are designed to prevent work from being deferred whilst striking a sensible balance between the work that we know should be done upfront and the work that can only realistically happen later in the product cycle. Some examples of Quality Gates enforced at Microsoft

include Functional Specification, Dev Design, Test Plan, Threat Mode, and IP Protection Code Coverage just to name a few.

The following diagram represents the physical structure of Feature Crew branching.



## End-to-End Branching Implementation:

For our scenario imagine we have a company that has only one product in its portfolio. This is what the physical view would look like:

### Profile

**Company Name:** AdventureWorks

**Business:** Insurance company specializing in adventure, filming and sport industries

**Developers:** 130

**Applications:** 10

**Team Foundation Server Instances:** 1

**Location:** Globally Distributed

**Configuration Manager:** John Smith

**Notes:** AdventureWorks has three development teams who are responsible for 10 of the applications used throughout the company. One of the applications is a secure Web site that is customized for each of the sectors they do business with.

Creating good custom branching structures is one of the most important tasks within the scope of work of a configuration manager. Just as an architect looks to a building to stand the test of time, the configuration manager needs to look at the branch structure to stand the test of multiple releases, parallel development, and product expansion. By doing the critical thinking and thorough research ahead of time,

you not only ensure that the right structure is in place, but also that your team's effectiveness and productivity raise to new levels.

When preparing for this whitepaper, we surveyed individuals throughout Microsoft, enterprises, and smaller organizations. From this, we learned that most of the time the persons responsible for creating the branch structures lacked guidance on which series of steps are necessary in order to obtain all the data needed to reach an educated decision. To this end, we compiled the following checklist:

- ✓ Decide on team project structure
- ✓ Understand the culture and requirements for each of the teams
- ✓ Map requirements to our logical and physical guidance and create first draft of the logical
- ✓ Make necessary modifications to the structure (baseless merge where appropriate)
- ✓ Finalize logical and physical diagrams
- ✓ Follow release scenarios on these diagrams
- ✓ Create Team Project and physical version control structure (import any existing code)
- ✓ Create best practices and provide training
- ✓ Learn from experiences, customize the branch structure, and make it yours

## **Step1 – Deciding on a Team Project structure**

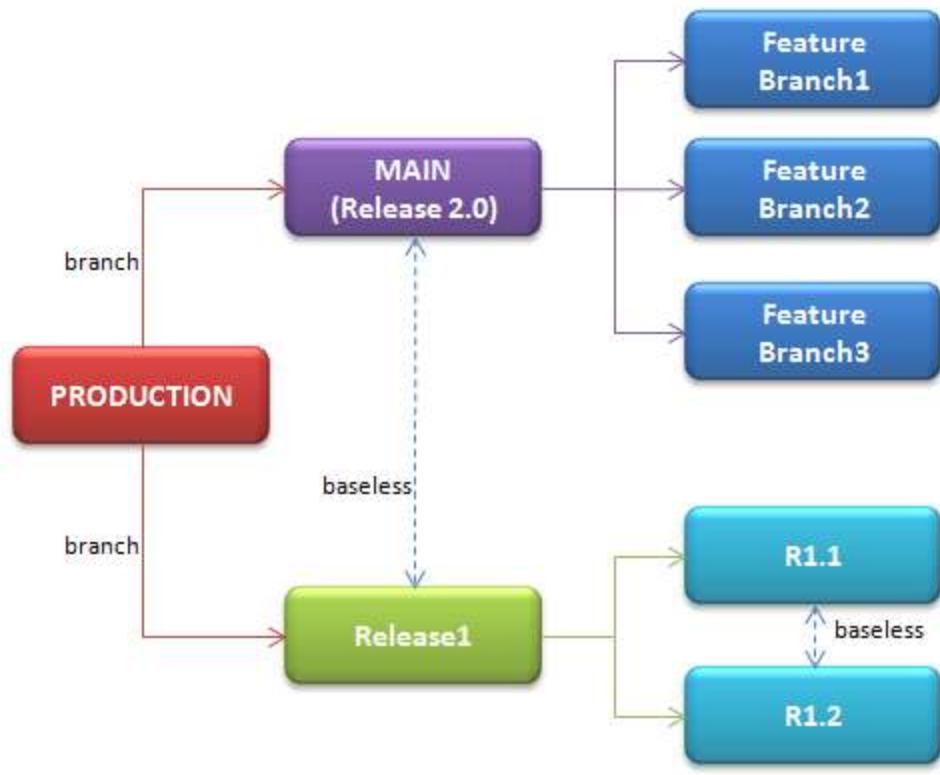
John's first step is to decide how to position these teams and projects within Team Foundation Server. For that, the first question is how to break it down into team projects. Since this paper is about branching and not team projects, we recommend that you read the Team Project Guidance whitepaper (to be released shortly).

John read the paper and decided to break down the company's product portfolio per buckets of applications (Client Insurance, Internal Analytics, and Web site). The main reason for his decision is that the breakdown matches nicely with the process employed to ship each application and the geography of the developers assigned to build each component. In this case, all of the Client Insurance applications are employing a Scrum process while the Internal Analytics teams have a custom process that is significantly different, so having them in the same team project would have been challenging.

## **Step2 – Deciding on a branch structure**

With this decision made, John's next step is to document and understand the development culture of the organization, including their process. This will give him the necessary list of requirements that will need to be checked against the feature crew branch structure guidance we are recommending here. After analyzing the teams, he determines that all but one team will be comfortable with separate branches for their features.

John creates the following logical diagrams for their branch structure:



In the diagram above, John has modified his original approach by creating a relationship between Main and the various Feature Branches. Let's analyze a few of the elements presented here in order to understand John's rationale when making these decisions.

1. The company today already employs a promotion model, so changing the promotion model to incorporate feature branches is not a big change.
2. Developers usually work on two releases at a time (Release 1.1 and Release 1.2). Since the release schedule is about every 6 months, the development portion of each release is about 3 months or longer. Branching from the Release1 folder to Release 1.1 1 and Release 1.2 creates the necessary isolation.
3. The team will establish a merge relationship between Release 1.1 and Release 1.2 using the baseless merge command in order to merge changes between those branches. Since fixes made in Release 1.1 are also needed in Release 1.2, we don't want to wait until that code has made it into the Release 1 branch. In order to merge it back to Release 1.2, a relationship needs to be established.
4. Often the fixes made in the intermediate versions will need to make their way into the feature branches in Release 2. In order to accomplish that and avoid having to make the code fixes manually on all of the branches, a merge relationship is established using the baseless merge command. Once that is established, fixes made in Release 1.1 can easily travel back to Release 2.

From this analysis, let's examine why we need to use the baseless merge feature.

Going back to our scenario, we need a way to move code between the Release 1 and Release 2 branches in order to expedite the code fixes and not duplicate code. But we did not have a merge relationship between these branches. The way to solve that problem is to perform a baseless merge between the two branches. This must be done once, so that a direct merge relationship is established.

Another caveat is that even though a relationship is now created, the Team Explorer UI does not show the relationship so consequent merges between these branches will need to be executed from the command line. In addition, the first baseless merge that you perform will create a lot of conflicts (one for each file that has the same name), so you want to allocate enough time to resolve all of those conflicts.

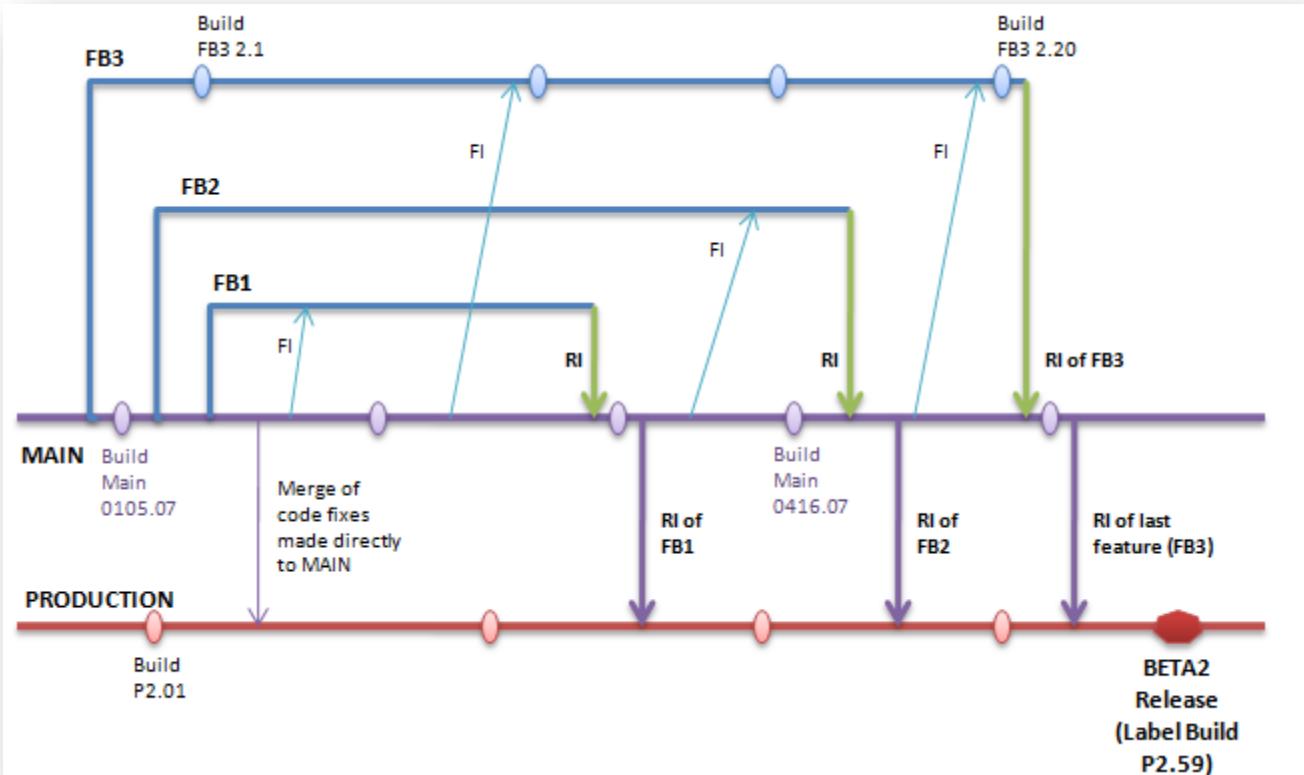
### Step3- Following a release on paper

With the logical view and branch structure complete the next step in the process is to follow on paper and pencil how a release will utilize these branches so we can verify that nothing has been missed. We call this the **"promotion and merge pipeline."**

In order to illustrate the pipeline, the best thing is to illustrate a series of scenarios. Here are a series of diagrams demonstrating how the promotion and merge pipeline could evolve.

#### Scenario 1: Releasing a Beta for Release 2

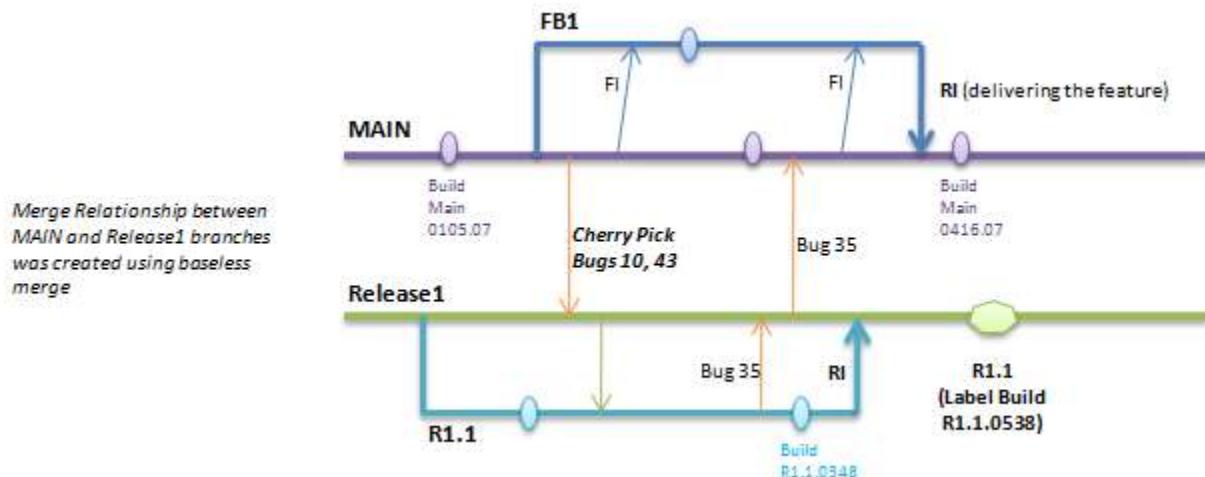
In this scenario, we will demonstrate the concepts we explained under the Defining Your Code Promotion Model section, especially emphasizing the Forward/Reverse Integration operations which are critical for pushing a series of feature development endeavors staged in the Feature Branches through the pipeline. The diagram below captures the operations leading up to the release of a Beta for Release 2.



- Release 2 is stabilized in MAIN, while several Feature Branches – FB1, FB2, FB3 – host the development activities for the implementation and stabilization of features marked for the Beta2 release.
- Based on the criteria defined in the Code Promotion Best Practices section, each of these feature branches periodically sync up to the latest changes in MAIN via a series of Forward Integration (FI) operations , represented by the upward arrow into the respective branches.
- At some point in time when FB1 reaches its feature completion, its changes are Reverse Integrated (RI) into MAIN after satisfying the quality gates criteria as well as FI/RI best practice guidance.
- Note that there are two significant FI operations following FB1's RI. Both operations enable the FB2 and FB3 branches to catch up to the FB1's latest features, so that when they RI back into MAIN, their changes contain their feature development work as well as FB1's changes. As we emphasized in the Code Promotion Best Practices section, periodically forward integrating from MAIN enables Feature Branches to integrate changes from Feature Branches with fewer merge conflicts and stabilize integration issues sooner and in smaller, manageable iterations. Scheduling continuous integration builds alongside these operations further enhances the success of the FI operations.
- On MAIN, we also see similar FI activities needed to push the results of the incoming RI's from the Feature Branches into the PRODUCTION branch. Once again, doing so enables the PRODUCTION branch to stabilize integration issues in smaller and more manageable iterations. Hence you see a series of FI's into PRODUCTION from MAIN (represented by the down arrows titles RI of FB1, RI of FB2, RI of RB3) on the heels of each RI from the Feature Branches into MAIN.

- We schedule Release Builds on both MAIN and PRODUCTION each time these branches accept an RI – such builds are tested end-to-end by Quality Assurance (QA) teams to reach the Beta2 milestone readiness criteria for feature completeness, performance and security considerations.
- At some point after the merge labeled “The RI of the last feature (FB3)” we reach the milestone readiness criteria for Beta2 with build P2.59.

### Scenario 2: Releasing 1.1 and merging some code fixes to Release 2



In this scenario, we are highlighting the use of cherry picking fixes. Oftentimes during a lifecycle, it becomes necessary to fast-track a fix from one branch to another. An example of such a fix could be a critical bug fix that has been made in one version of the product but not in another.

As you can remember, we created the relationship between the branches so we can move fixes that we perform in 1.1 to MAIN (Release 2). Without the relationship established by MAIN we would have had to use the following merge path, which is not ideal:

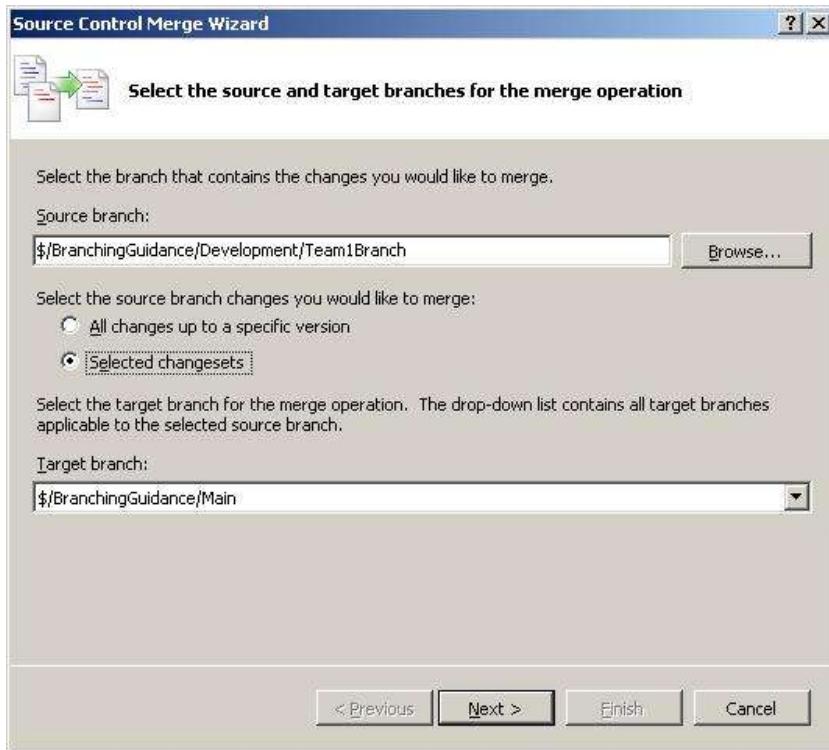
1.1 → Release 1 → PRODUCTION → MAIN

Instead, the fix follows a more natural path:

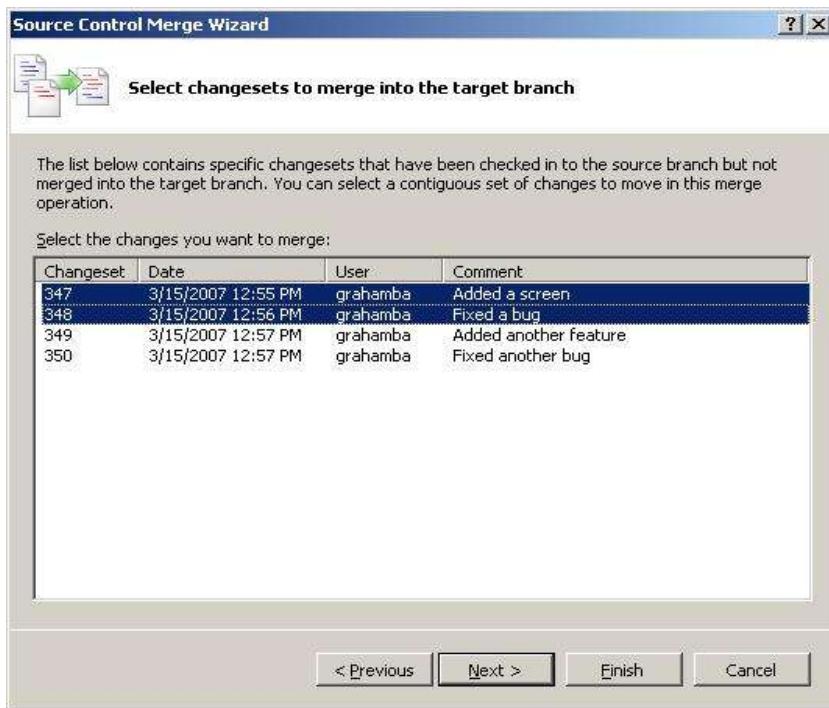
1.1 → Release 1 → MAIN

Team Foundation Server provides the ability to cherry pick these changes by specifying in the Merge Wizard the changeset or consecutive range of changesets you want to merge.

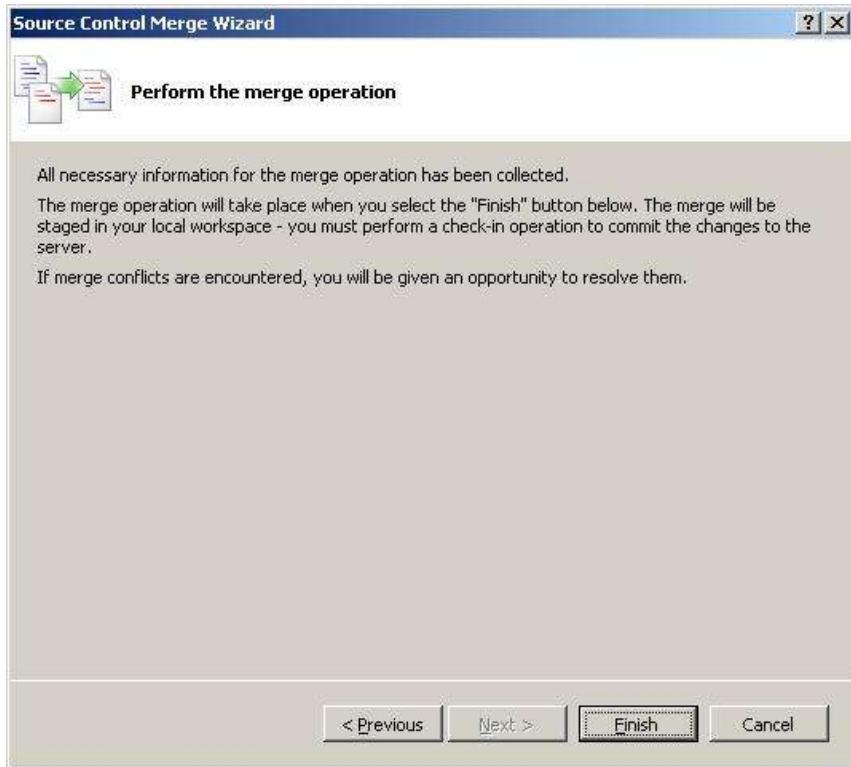
Here's an example of using the Merge wizard to cherry pick changesets to merge:



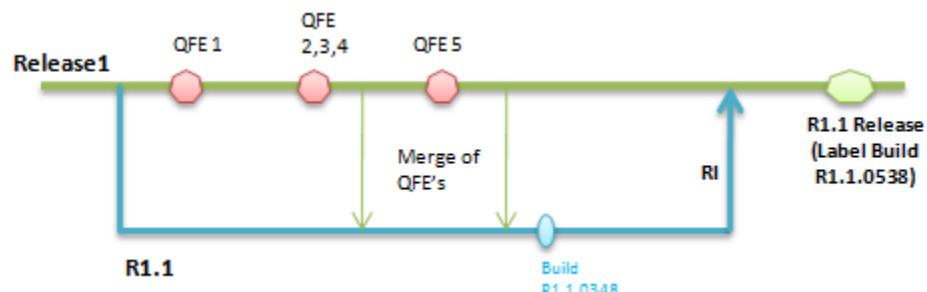
Select the desired changeset(s):



Complete the merge:



### Scenario 3: Sustained Engineering: Maintenance Activities on Releases



In this scenario, we are visualizing how John's team will handle servicing on the 1.1 and 1.2 releases. Sustained Engineering and Quick Fix Engineering (QFE) both refer to operations to support the maintenance of releases in production. Both are informed by the need for providing fixes to correct:

1. Bugs which were undetected at the time of the initial release.
  2. Bugs introduced by other software which was placed into production.
  3. Bugs postponed in the prior release cycle, but now deemed critical to fix.
  4. Bugs introduced due to hardware changes.

In our guidance, we are recommending that teams perform their QFE fixes directly into the Release 1 (Sustained Engineering) branch and have the isolation at the workspace level instead of at the branch level. These QFE's can then be labeled when the necessary testing and verification has been made and merged to the 1.X releases.

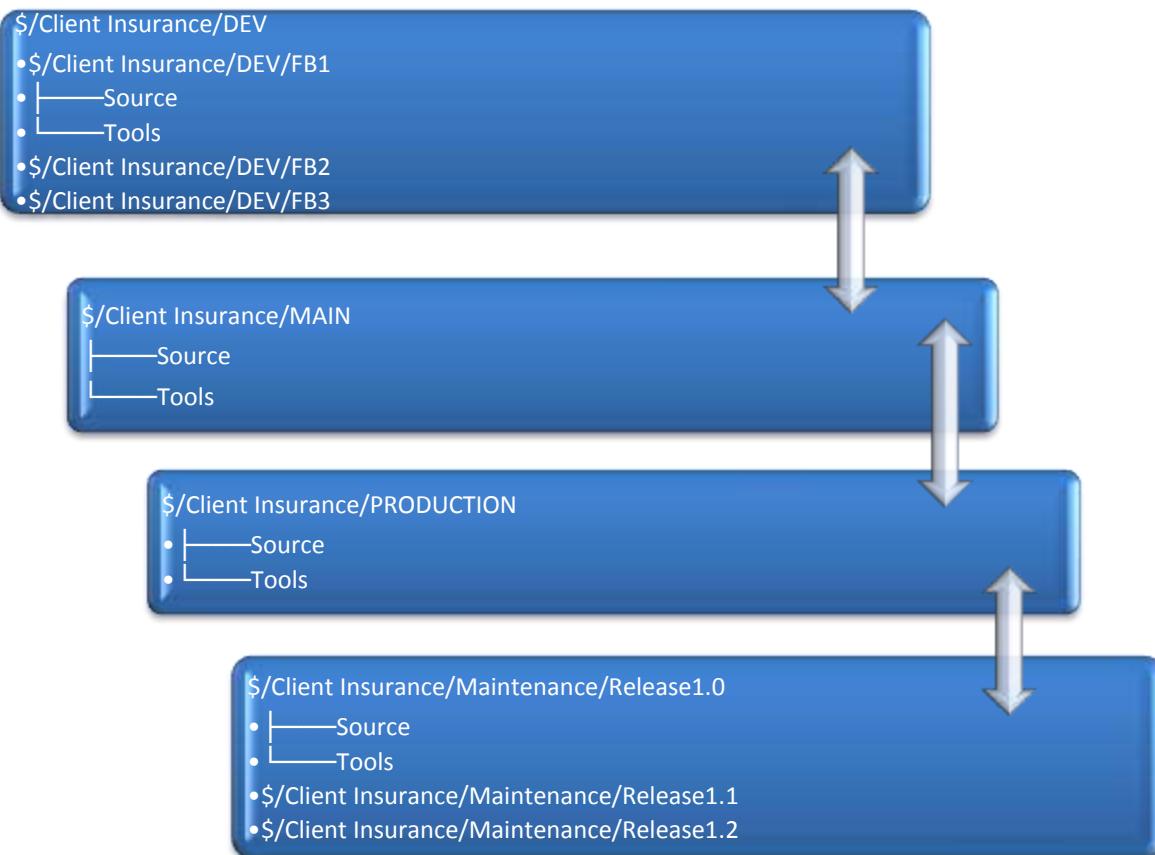
Note that our branching structure for this example allows for sustained engineering support for the most recent release. In cases where there are several versions of legacy releases to support, the branching structure will be different - a Release 2 branch will be present, but the same principles would apply.

## Step 4- Creating the physical structure

Now that the hard work is behind him, John now needs to put everything he has on paper into Team Foundation Server. The first step is to create the team projects.

John decided to apply the theories in the Team Project Creation whitepaper and utilize both per application and per team approaches. This produced three team projects: Client Insurance, Internal Analytics, Web site.

Since John's company has a significant number of developers working on each application, he decided to go with the feature branch model for their branch structure. John's branch structure resembled the following:



## **Appendix**

In this section, we'll address some additional topics as supplemental reading that will help you understand the recommendations in this whitepaper.

### **Software Configuration Management**

Software Configuration Management (SCM) is "The process of identifying and defining the configuration items in a software system, controlling the release, versioning and change of these items throughout the software system life cycle, recording and reporting the status of configuration items and change requests, and verifying the completeness and correctness of configuration items." (Source: Infosys)

SCM practices provide the foundation that helps all members of a development team collaborate effectively. No one doubts the importance of having a solid foundation to support a high-rise building, but few seem to understand that SCM plays a similar role in software development.

We frequently hear teams talk about their frustrations and failures, and more often than not discover that the root cause was poor SCM practices such as testing the wrong configuration, incorrect merging of the code, deploying a faulty build, and so on.

The benefits of good SCM practices include the following:

- Safeguarding intellectual property—the software assets
- Helping to improve communication among team members
- Providing a way to establish clear responsibilities and accountabilities
- Providing traceability and reproducibility
- Facilitating reusability
- Providing for consistency, reliability, and integrity

There is one thing that SCM affects the most and that is Version Control. The typical goal of Version Control is to provide a self-contained, version-controlled repository of the needed sources, tools, external dependencies and process automation such that, when the process automation is invoked on similar, hardware, OS, and network configurations, identical software is produced. The branching and merging features of Team Foundation Server address these critical areas of Software Configuration Management.

### **Branching and Merging Anti-Patterns**

Earlier, we had referred to a whitepaper published on MSDN by Chris Birmele

[http://msdn2.microsoft.com/en-us/library/aa730834\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/aa730834(VS.80).aspx).

Borrowing from the paper outlined above we will now provide a list of anti-patterns which may additionally help in establishing a conversation about branching and merging for your organization. If you experience one or more of the following symptoms in your development environment it may be cause for concern:

- Merge Paranoia—avoiding merging at all cost, usually because of a fear of the consequences.
- Merge Mania—spending too much time merging software assets instead of developing them.

- Big Bang Merge—deferring branch merging to the end of the development effort and attempting to merge all branches simultaneously.
- Never-Ending Merge—continuous merging activity because there is always more to merge.
- Wrong-Way Merge—merging a software asset version with an earlier version.
- Branch Mania—creating many branches for no apparent reason.
- Cascading Branches—branching but never merging back to the main line.
- Temporary Branches—branching for changing reasons, so the branch becomes a permanent temporary workspace.
- Volatile Branches—branching with unstable software assets shared by other branches or merged into another branch.
- Note Branches are volatile most of the time while they exist as independent branches. That is the point of having them. The difference is that you should not share or merge branches while they are in an unstable state.
- Development Freeze—stopping all development activities while branching, merging, and building new base lines.
- Dividing Wall—using branches to divide the development team members, instead of dividing based upon the features people are working on.

If you catch yourself or your team in any of these scenarios we recommend that you take some time to understand the root cause and identify a remedy.

## Branching in Team Foundation Server vs. Sharing in Visual SourceSafe

Branching in Team Foundation Server is different from Visual SourceSafe's sharing feature in that branching allows the divergence of files and folders, and changes between the source and target files and folders are not automatically synchronized (i.e. changes to the source do not update the target, and vice-versa) until an explicit merge operation is performed. In Visual SourceSafe, changes are merged automatically. Sharing is not supported in Team Foundation Server since we promote the use of branching as a more robust way of managing changes.

## Labeling vs. Branching

The **label** command in Team Foundation Server, like in Visual SourceSafe, allows you to save a snapshot of your source code based on a given state – workspace version, time stamp – using a unique label that can be stored and retrieved by users. Typical scenarios in which you would use labels is to denote the snapshot of source code that was successfully built on a given day (Team Build does this automatically for each build). Depending on the permissions granted to specific users, labels can be modified – files can be changed, added, removed from the label. While powerful in its own respect, we'd like to caution against the over-use of labels given that:

- Team Foundation Server does not retain a history of changes made to the label.
- Given certain permissions, labels might be deleted or otherwise invalidated by changes, and there is no way of auditing those changes.

- There can be contention for a given label if more than one person wants to use and modify the label or the files contained in the label.

As such, we recommend using labels only in cases where you need a snapshot of sources for a short duration of time or if you can guarantee via permissions that the label won't be changed.

For any other development activity that needs to use a snapshot of sources over a longer duration of time, combined with the need for history and auditing of changes, and especially where several users might need to access the snapshot and possibly change the files and their contents, we strongly recommend the use of branching. Branching by definition allows files and folders to diverge from their original state (while also preserving the original state), include history, and allow a rich set of file- and folder-level permissions to control changes.

## Branching and Compliance

Branching in Team Foundation Server, along with advanced security features, very easily lends itself to enabling compliance for regulations such as Sarbanes-Oxley Act of 2002 (SOX). This is precisely the reason why we recommend safekeeping read-only branches matching the various versions of your releases. In addition to addressing the requirements for auditing and inviolability of source code matching your released software, they provide an easier mechanism for debugging prior releases.

We do not recommend using labels when compliance and auditing is a requirement because the label can be modified and does not retain history, therefore an audit trail of changes to a label is not possible. Branching for safekeeping provides all of these capabilities and is therefore the recommended solution for most compliance requirements.

## Identifying Environments

In addition to version control isolation, we need to be able to deploy the software to corresponding hardware environments for previewing the state of the software at various stages and for supporting the software in production. An environment can be as simple as a single desktop computer, or as complex as a large set of servers for a distributed application. The motivation for creating separate environments is the same as the motivation for creating branches: to provide isolation from changes to the software in a controlled fashion. Most enterprises typically have one or more of the following environments:

- ➔ **Development** – Environment where all of the development takes place. This is the most volatile environment since all changes take place in this environment. The development environment should reflect the latest version of the software that is checked into MAIN.
- ➔ **Test** – All testing occurs in this environment (often this environment is comprised of virtual servers). Usually only stable "drops" (often once weekly or biweekly) are delivered to the Test environment which allows the development team to focus on reaching a testable state before dropping to the Test environment.
- ➔ **User Acceptance Testing (UAT)** – The UAT environment is more stable than Dev or Test. It is often used for conducting demos and is the environment that project stakeholders use to test the software to reach sign off.

- ➔ **Pre-Production** – Not all enterprises use this environment because the cost associated with this environment can be high. The Pre-Production environment should have hardware which is identical or very similar to the Production environment so that issues that are only manifested in Production can be identified early (such as database clustering issues, Web farm issues, etc.)
- ➔ **Production** – The environment that supports the production use of the application.
- ➔ **Maintenance** – Once the application is released and the development team continues building the next major version of the application or product, the Maintenance environment is used to test bug fixes and changes made against the maintenance code base. .

The table below depicts which branches are typically used to produce releases for each of the environments mentioned above.

Environment	Branch	Comment
<b>Development</b>	DEVELOPMENT	Usually the DEVELOPMENT environment will reflect the latest good build from one of the DEVELOPMENT lines.
<b>Test</b>	Usually MAIN, sometimes a DEV branch	Usually the Test environment will contain a build of the software from the MAIN line. The Test environment at times can also contain a build from a DEVELOPMENT line.
<b>User Acceptance Testing</b>	Usually MAIN, sometimes PRODUCTION when previewing a servicing release that needs to pass User Acceptance Testing.	In most cases the UAT environment will be running a high quality build from the MAIN line.
<b>Pre-Production</b>	PRODUCTION	Pre-Production will almost always have a build from the PRODUCTION line.
<b>Production</b>	PRODUCTION	PRODUCTION always contains a build from the PRODUCTION line.

## Virtual Build Labs

Very large teams at Microsoft that get to a size where Feature Branches no longer provide the necessary level of isolation can adopt the next level of isolation - Virtual Build Labs (VBLs). The main difference between the Feature Branch Model and the Virtual Build Lab model is the scope of isolation – with VBLs providing team isolation. Otherwise, all of our best practice considerations with respect to branching and code promotion still apply. Also note that as we start branching at a higher level – the team organization level as opposed to the feature level – we increase the complexity of our code promotion paths and results in more complex merge resolutions as well as latency between branches. The latter is primarily the

result of having to satisfy several quality gate criteria as we attempt to promote code into our MAIN pipeline. One good way to avoid this kind of complexity and latency of merges is to opt for a flatter branching structure as opposed to a deeply nested one.

Virtual Build Labs (VBLs) use branching heavily and provide the necessary isolation for organizations and teams with 800 or more developers. For more information about VBLs you can visit Brian Harry's blog for a good description: <http://blogs.msdn.com/bharry/archive/2005/11/12/492198.aspx>.