



# TFS Version Control Part 1 **Branching Strategies**

Visual Studio ALM Rangers



Microsoft



Visual Studio

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, you should not interpret this to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This document is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

Microsoft grants you a license to this document under the terms of the Creative Commons Attribution 3.0 License. All other rights are reserved.

*© 2014 Microsoft Corporation.*

Microsoft, Active Directory, Excel, Internet Explorer, SQL Server, Visual Studio, and Windows are trademarks of the Microsoft group of companies.

All other trademarks are property of their respective owners.

# Table of Contents

Foreword .....	4
Introduction .....	5
What's New .....	6
Concepts.....	9
Vocabulary .....	9
Branching Concepts.....	9
Branch Types.....	12
Branching Strategies.....	13
Main Only .....	13
Development Isolation.....	14
Release Isolation.....	14
Development and Release Isolation.....	15
Servicing and Release Isolation.....	15
Servicing, Hotfix, and Release Isolation.....	17
Code Promotion .....	18
Feature Isolation.....	18
Alternative Strategies.....	20
Adapt your branching process for inevitable 'blips' .....	20
Feature Toggling.....	21
Continuous Delivery.....	23
Walkthroughs .....	25
From nothing to complexity or not.....	25
Adapt your branching process for inevitable 'blips' .....	28
Real World Scenarios .....	30
Delivering software at intervals ranging from days to months .....	30
FAQ .....	33
Hands-on Lab (HOL) – From Simple to Complex or not? .....	36
Exercise 1: Environment Setup .....	36
Exercise 2: MAIN Only – Simplicity Rules.....	42
Exercise 3: Development Isolation ... welcome branching.....	47
Exercise 4: Feature Isolation ... a special! .....	54
Exercise 5: Release Isolation ... audit alarm .....	62
Exercise 6: Servicing & Release Isolation .....	65
In Conclusion .....	68

# Foreword

Since the first writing of the TFS branching guide, a lot has changed in the world of version control. Hosted version control solutions are everywhere, and many of them include integration with build, project tracking, and other services. Distributed Version Control is no longer a niche, and has changed the way that many developers think about what it means to version their code. More developers using version control than ever before – and that is a great thing for the billions of end users of those software development projects.

More developers using version control also means that, now more than ever, the industry needs solid, practical, and easy-to-digest guidance that is industry proven. This guide, and those that came before it, strive to do just that – provide the version control guidance that development teams need to be effective while being approachable and flexible. In this latest edition, we have streamlined the guidance and simplified concepts with the goal of helping teams of all shapes and sizes to develop strategies that enable the flexibility and agility that modern development teams demand.

I also need to mention that this guide would not be in its current form without the readers. Thanks to all of you whom have contributed your feedback and ideas that have helped shape this guide over the years. As is in the past, if you see something in this guide that you would like to see changed or improved, please let us know!

Happy versioning!

**Matthew Mittrik** – Program Manager, Cloud Dev Services

# Introduction

This guide aims to provide insightful and practical guidance around branching strategies with Team Foundation Server. Branching of software is a vast topic and it is imperative to realize that every organization is different, and there is 'no one size fits all' branching strategy.

These guidelines serve as a starting point for your planning and you may find that deviating in certain circumstances adds value to your environment. In this guidance, the focus is primarily on practical and real world scenarios of branching that you can apply immediately. We avoid discussing the mechanics of branching but rather focus on best practices and real world examples of patterns that work for many of the business cases we have seen. This guide is one of a set of companion guides as outlined in chapter **What's New**, on page 6.

## Intended audience

In this guide, we primarily target **Garry**, the development lead, **Doris**, the developer, and **Dave**, the TFS administrator. See [ALM Rangers Personas and Customer Profiles](#)<sup>1</sup> for more information on these and other personas.

## Visual Studio ALM Rangers

The Visual Studio ALM Rangers provide professional guidance, practical experience and gap-filling solutions to the ALM community. They are a special group with members from the Visual Studio Product group, Microsoft Services, Microsoft Most Valuable Professionals (MVP) and Visual Studio Community Leads. Membership information is available [online](#)<sup>2</sup>.

## Contributors

Matthew Mitrik, [Michael Fourie](#), [Micheal Learned](#) and [Willy-Peter Schaub](#).

A special thank you to the ALM Ranger teams who laid the foundation with v1 and v2: Anil Chandr Lingam, Bijan Javidi, Bill Heys, Bob Jacobs, Brian Minisi, Clementino de Mendonca, Daniel Manson, Jahangeer Mohammed, James Pickell, Jansson Lennart, Jelle Druyts, Jens Suessmeyer, Krithika Sambamoorthy, Lennart Jansson, Mathias Olausson, Matt Velloso, Matthew Mitrik, Michael Fourie, Micheal Learned, Neno Loje, Oliver Hilgers, Sin Min Lee, Stefan Mieth, Taavi Koosaar, Tony Whitter, Willy-Peter Schaub, and the ALM Community.

## Using the sample source code, Errata and support

All source code in and revisions of this guide are available for download via the [Version Control Guide](#)<sup>3</sup> (formerly the Branching and Merging Guide) site. You can contact the team using the CodePlex discussion forum.

## Additional ALM Rangers and other Resources

[Understanding the ALM Rangers](#)<sup>4</sup>

[Visual Studio ALM Ranger Solutions](#)<sup>5</sup>

[Branching Taxonomy, by MS Research](#)<sup>6</sup>

[The Effect of Branching Strategies](#)<sup>7</sup>

---

<sup>1</sup> <http://vsarguidance.codeplex.com/releases/view/88001>

<sup>2</sup> <http://aka.ms/vsarindex>

<sup>3</sup> <http://aka.ms/treasure18>

<sup>4</sup> <http://aka.ms/vsarunderstand>

<sup>5</sup> <http://aka.ms/vsarsolutions>

<sup>6</sup> <http://research.microsoft.com/apps/pubs/?id=209683>

<sup>7</sup> <http://research.microsoft.com/apps/pubs/default.aspx?id=163833>

# What's New

This guide delivers a new crisper, more compact style, which is easier to consume on multiple devices without sacrificing any content. The authors have updated the content, and aligned it with the latest Visual Studio technologies. This latest guidance incorporates key feedback from the readers of the previous guidance.

NOTE

**Branching** is cheap ... **Merging** is expensive!

This latest guidance promotes the notion of starting with no branching and merging strategy. Instead of selecting the most applicable strategy and evolving your team processes and skills to embrace the strategy, you should adopt and evolve one or more strategies as and when needed.

## Strategy Changes

The strategy names simple, basic and advanced used in previous versions of the branching and merging guide have been replaced with more contextual and meaningful names as summarized in the table below. Additionally two new strategies are included in this guide to address new emerging best practices: **feature toggling** and **continuous delivery**.

New strategy name {Old name in braces}	Visual	Page
<b>Main Only</b> {No Branch Plan}		13
<b>Release Isolation</b> {Release Branching – Basic (Single Branch)}		14
<b>Development Isolation</b> {part of Release Branching – Basic (two branch)}		14
<b>Development and Release Isolation</b> {Release Branching – Basic (two branch)}		15
<b>Servicing and Release Isolation</b> {Release Branching – Standard}		15

## Branching Strategies – What's New

New strategy name {Old name in braces}	Visual	Page
<b>Servicing, Hotfix and Release Isolation</b> {Release Branching – Advanced}		17
<b>Feature Isolation</b> {Feature Branching}		18

**Table 1 – Branching strategies at a glance**

## Walkthroughs



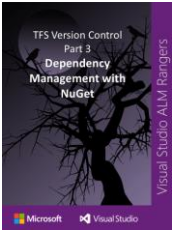

To complement the companion hands-on labs and the branching strategies, new walkthroughs are included to guide you through common scenarios such as:

- Starting with no strategy and adopting one or more strategies as needed.
- Adapting your strategy for 'blips' in your standard process, e.g. if you need to temporarily branch from a specific changeset.

The walkthroughs are intended as practical checklists that guide you through the recommended process and outlines the pros/cons.

### Companion guides

We have split this guidance into the four topics as summarized below. This will allow you to pick the “world” you are interested in while minimizing the noise and distractions.

Guide	Context
 <p>TFS Version Control Part 1 Branching Strategies Visual Studio ALM Rangers Microsoft Visual Studio</p>	<p>Branching Strategies → this guide.</p> <p>Practical guidance on a number of (not all) common Branching Strategies and usage thereof.</p> <p>Core sections:</p> <ul style="list-style-type: none"> <li>• Branching concepts</li> <li>• Branching strategies</li> <li>• Walkthroughs</li> </ul>
 <p>TFS Version Control Part 2 TFVC Gems Visual Studio ALM Rangers Microsoft Visual Studio</p>	<p>Team Foundation Version Control (TFVC)</p> <p>Practical guidance on how to use Team Foundation Version Control (TFVC) features.</p> <p>Core sections:</p> <ul style="list-style-type: none"> <li>• Workspaces</li> <li>• Merging</li> <li>• New features, i.e. Code Lens</li> <li>• Walkthroughs</li> </ul>
 <p>TFS Version Control Part 3 Dependency Management with NuGet Visual Studio ALM Rangers Microsoft Visual Studio</p>	<p>Dependency Management with NuGet</p> <p>Practical guidance on dependency management, using NuGet with Visual Studio.</p> <p>Core sections:</p> <ul style="list-style-type: none"> <li>• Managing shared resources</li> <li>• Dependency management</li> <li>• Walkthroughs</li> </ul>
 <p>TFS Version Control Part 4 Git for TFVC Users Visual Studio ALM Rangers Microsoft Visual Studio</p>	<p>Git for TFVC Users</p> <p>Practical guidance on Git from a TFS perspective.</p> <p>Core sections:</p> <ul style="list-style-type: none"> <li>• Guidance</li> <li>• Walkthroughs</li> <li>• Terminology/Concepts map</li> </ul>

**Table 2 – Version Control guides**



# Concepts

1. **Start** with **no** branching or the Main Only branching strategy
2. **Branch** only when needed and only after analyzing and understanding the value-add and maintenance costs
3. The strategies in this guide are not the only valid branching strategies. Other strategies may work better for your team and situation.
4. If you are **unsure** whether you really need a branching strategy, start with step 1.



## Vocabulary

*"One forgets words as one forgets names. One's vocabulary needs constant fertilizing or it will die" - Evelyn Waugh*

Terminologies may vary in your organization so some translation may be required to the terminology used throughout this guide.

Term	Description
Development Branch	Changes for next version work
Forward Integrate (FI)	Merges from parent to child branches
Hotfix	A change to fix a specific customer-blocking bug or service disruption
Main Branch	This branch is the junction branch between the development and release branches. This branch should represent a stable snapshot of the product that can be shared with QA or external teams
Release Branch	A branch to isolate code in preparation for a release. Make ship-stopping fixes before a major product release. After product release this branch may set to read-only based on your process
Release Vehicle	How your product gets to your customer (e.g. major release, hotfixes and/or service packs).
Reverse Integrate (RI)	Merges from child to parent branches
Service Pack (SP)	A collection of hotfixes and features targeting a previous product release

**Table 3 – TFVC branching vocabulary**

## Branching Concepts

*"Your branch distance from main is equal to your level of insanity" – anonymous*



**Figure 1 – (Heavy) Branching obfuscates simplicity and we recommend the simpler “main only” bamboo strategy ☺**

Branching enables parallel development by providing each development activity a self-contained snapshot of needed sources, tools, external dependencies and process automation. Having more branches increases complexity and merge costs. Nevertheless, there are several scenarios where you will need to consider multiple branches to maintain a required development velocity. Whether you branch to temporarily isolate or stabilize a

breaking change, develop a feature, or jumpstart development on a future version, you should consider the following:

- Branch from a parent with the latest changes (usually MAIN or another DEV branch).
- Branch everything you require to develop in parallel, typically this may mean branching the entire parent.
- Merge from the parent branch (FI) frequently. Always FI before RI. FI as often as it makes sense. Weekly is a good goal.
- Ensure you build and run sufficient Build Verification Tests (BVT) to measure the stability of the branch.
- For stabilization changes, take your changes, forward integrate (FI), build and pass all BVTs before merging (RI) the change and supporting changes to the parent.
- A merge (RI) to MAIN is comparable to a “feature release” to peer development teams. Once the feature branch merges (RI), other teams will use that version of the feature until the next merge (RI).
- When MAIN branches for release (current version), servicing of final ship-stopping bug fixes should be made in the RELEASE branch. The feature branch is just for next version work. In this example when MAIN branches for an upcoming release, all DEV branches can begin next version work.

### Avoiding Branching

#### NOTE

The fewer branches you have the better. Instead of selecting and pre-creating a branching strategy, start with nothing and branch as needed. See **From nothing to complexity or not**, page 25, for a practical walkthrough.

The bar for creating a branch should be set high to avoid the inevitable associated costs. Consider the following before creating a branch for temporary development work.

- **Can you use a shelveset?** A shelveset allows developers to save and share changes in TFS, but not commit them to the branch.
- **Is there another work item you should be working on?** Sometimes other work is not visible to the individual engineer.
- **Are you able to work in the “next version” branch mentioned above?** This may be feasible if the stakeholders approve the change for the next version release.
- **Do you “really” need the branch now?** Alternatively, can we create it later when we need the branch? At that time, perhaps we can create the new branch by branching from a changeset version.

Always make sure you can adequately describe the value a branch is giving your organization; otherwise perhaps the branch is not needed.

### Branching with Labels

Organizations that need a good deal of stability should consider creating branches to capture specific versions of code that went into a release. Utilizing this type of branching strategy makes it relatively easy for a developer to be able to pull down a specific version of code and perform maintenance on that version. Creating a branch from specific versions of source allows the creation of branches for maintenance as needed instead of as a side effect of every release. When branching by version using TFVC, it is important to recognize the value that labels provide.

Labels in TFS are a powerful tool that allow a development team to identify quickly files at specific version levels. Utilizing labeling as part of a branching strategy allows developers to be able to isolate code from subsequent changes that may have happened in the source control system. For example, if a specific version requires maintenance, we can create a branch at any time in the future based on a label. This may lead to a cleaner folder structure in version control as we only create new folders as needed.

When using labels, keep in mind the following caveats:

- Labels are editable and can be deleted (requires permission). These changes are not auditable.
- There can be contention for a given label if more than more person wants to use and modify the label or the files contained in the label.

- Build labels cannot be relied upon because they might be deleted automatically when build retention policy is applied.

As such, only use labels in cases where we need a snapshot of sources and if it is possible to guarantee, via permissions, that we will not change the contents of the label.

### Building Branches

It can be helpful to name builds for branches to easily identify the builds and output associated thereof, e.g.

- **DEV**\_ProductName\_versioninfo
- **REL**\_ProductName\_versioninfo
- **FEA**\_ProductName\_FeatureName\_versioninfo

Automate the update of version information by the build procedure. For example, setting assembly information attribute of an assembly to directly link the artefact to a specific build

This traceability can be helpful if you have multiple branches and build agents. To make it easier to identify binaries, add full build name information into the file version info and/or assembly info. This will help you to identify binaries by knowing to which branch they belong.

### Permissions

TFS provides two permissions related to branching and merging. One allows teams to designate certain individuals to be responsible for creating new branches. The other can be responsible for merging code between branches, while most developers will be restricted to working only on certain branches.

- Manage Branch
- Merge

#### Manage Branch Permissions

The Manage Branch permission enables the following actions:

- Convert folders to branches
- Convert branches back to folders
- Update metadata for a branch, such as owner, description, etc.
- Create new child branches from a given parent branch
- Change the relationships between branches with merge relationships (i.e. re-parenting branches)

The **Manage Branch** permission only applies to branches (not branched folders or files). Denying the Manage Branch permission does not prevent users from branching ordinary folders for which you have denied this permission. TFS scopes Manage Branch permission to a specific path. Like ordinary folders, we can organize branches into folders. This can be a useful way to organize and apply permissions to groups of branches.

For example, a team could deny Contributors Manage Branch permission for all branches organized under the paths: \$/<TeamProject>/Main and \$/<TeamProject>/Releases, but allow Contributors the Manage Branch permission for feature branches organized under the path \$/<TeamProject>/Development. With these permissions, members of the TFS Contributors group can create child branches for development (from existing feature branches), but cannot create new branches from existing release branches or from the Main branch.

#### Merge Permission

NOTE

This “Branching Strategies” guide does not cover merging. Refer to [MSDN](http://msdn.microsoft.com)<sup>8</sup> and the companion “Team Foundation Version Control (TFVC)” guide for more information on merging.

---

<sup>8</sup> <http://msdn.microsoft.com>

The **Merge** permission is required in order to pend merge operations on branches, folders, and files under a specified path. Merge permission is required for the *target* path of a merge operation. There is no permission to prevent you from *merging* a particular branch or folder *to* another path. Merge permission is not limited to branches; you can also apply merge permission to folders and branches under a given path.

## Branch Types

*“The tree was evidently aged, from the size of its stem. It was about six feet high, the branches came out from the stem in a regular and symmetrical manner, and it had all the appearance of a tree in miniature” - Robert Fortune*

You can organize branches into three categories: MAIN, DEVELOPMENT and RELEASE. Regardless of branch type, consider the following:

- Building regularly (continuously) encourages a regular cadence and immediately identify quality issues.
- The movement of changes between child and parent branches is a concept all team members must consider and understand.

### NOTE

If you are looking for a continuous build and deployment pipeline, we recommend that you peruse the [ALM Rangers DevOps](#) <sup>9</sup> guidance and the [Building a Release Pipeline with Team Foundation Server](#) <sup>10</sup> guide.

The following table lists additional considerations and attributes for the aforementioned branch types.

### Main

- **Junction** between DEVELOPMENT and RELEASE, with a natural merge path back to main.
- Branch of “truth”, which must be buildable, meet minimum quality bar and is the trusted source by QA teams.
- Except in a “Main Only” strategy, avoid making changes to MAIN.
- As the number of DEVELOPMENT branches increase, the cost and need to merge (FI) following each successful MAIN build increases.

### Dev (Development)

- All branches are self-contained (isolated) area that enables each development activity to proceed at its own pace, without taking any dependency on another.
- We must base a DEV branch on a parent branch with a known good state, typically MAIN.
- Merge (FI) frequently to reduce the complexity of “big bang” type merges.
- Optionally you can merge (FI) every time the parent branch builds and passes BVTs, but this comes with potential overhead and disruption.

### Rel (Release)

- Release branches should **support** your **release vehicle**.
- Most common release vehicles are major release, hotfix and service pack.
- As with branching in general, less is better!
- The parent/child branch relationship between MAIN→SP→ and HOTFIX branches support merging of changes into future releases (i.e. Hotfixes merge into the SP branch on their way to MAIN) reducing risk of bug regressions in future releases.

<sup>9</sup> <http://aka.ms/treasure54>

<sup>10</sup> <http://aka.ms/treasure53>

# Branching Strategies

*"Save your creativity for your product... not the branch plan" – Anonymous*

The elements of the branching strategies introduced in this section are typically additive, starting with the simple **Main Only** strategy and adopting other strategies to evolve and mix into a more complex environment **when** and **as needed**.

## GOAL

The goal of this section is to introduce the basic strategies we have encountered and used effectively in the field. Please post your scenarios not covered to the community discussion forum on our CodePlex [team site](#) <sup>11</sup>, so that we can consider and leverage alternatives in future updates to this guide.

## Main Only

The **Main Only** strategy can be folder based or with the main folder promoted to a branch to support version control features that provide extra visibility.

In the illustration below, the branch icon on the left indicates that we have promoted the main folder to a branch, which is an optional step. We recommend this approach only if you intend to evolve into other branching strategies at a later stage.

See [Branch Folders and Files](#) <sup>12</sup> and the FAQ for more information on branch folders. Note that examples (V1.0, V1.1, and V2.0) illustrate how we can apply labels.



Figure 2 – Main Only branching strategy

## Usage scenarios

This strategy is the **recommended** starting point for any development team that has no need for code isolation using branches and no issue with mutable labels used to indicate milestones, in the one and only branch.

## NOTE

ALM Ranger projects start their life with this strategy, created by the [TFS Branch Tool](#) <sup>13</sup>, and evolve to other branching strategies when and as needed, reducing merge complexity and costs.

## Considerations

- Without branching, we need labels to mark development and release milestones. The mutability and lack of history for labels adds risk of change control.
- Peruse [Team Foundation Server Permissions](#) <sup>14</sup> and consider protecting the administration of labels.

<sup>11</sup> <http://aka.ms/treasure18>

<sup>12</sup> <http://msdn.microsoft.com/en-us/library/ms181425.aspx>

<sup>13</sup> <http://aka.ms/treasure35>

<sup>14</sup> [http://msdn.microsoft.com/en-us/library/ms252587\(v=vs.100\).aspx](http://msdn.microsoft.com/en-us/library/ms252587(v=vs.100).aspx)

## Development Isolation

The Development Isolation strategy introduces one or more development branches from main, which enables concurrent development of the next release, experimentation, or bug fixes in **isolated** development branches.

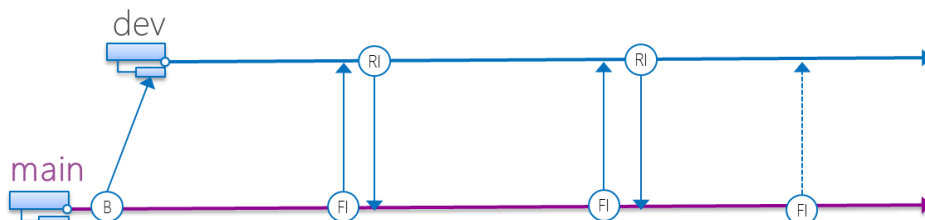


Figure 3 – Development isolation branching strategy

### Usage scenarios

- You need **isolation** and **concurrent** development, protecting the stable main branch.
- You need only **one major release**, supported and shipped to customer using the main branch.
- You need a servicing model for customers to upgrade to the **next major release**, i.e. v1 → v2.

### Considerations

- Each development branch should be a full child branch of the main branch. Avoid creating partial branches.
- We can isolate work in development branches by feature, organization, or temporary collaboration.
- Development branches should build and run Build Verification Tests (BVTs) the same way as main.
- Forward Integrate (FI) frequently from main to development if changes are happening directly on main.
- Forward Integrate (FI) and then Reverse Integrate (RI) from development to main based on some objective team criteria (e.g. internal quality gates, end of sprint, etc.).

## Release Isolation

The Release Isolation strategy introduces one or more release branches from MAIN, which enables concurrent release management.

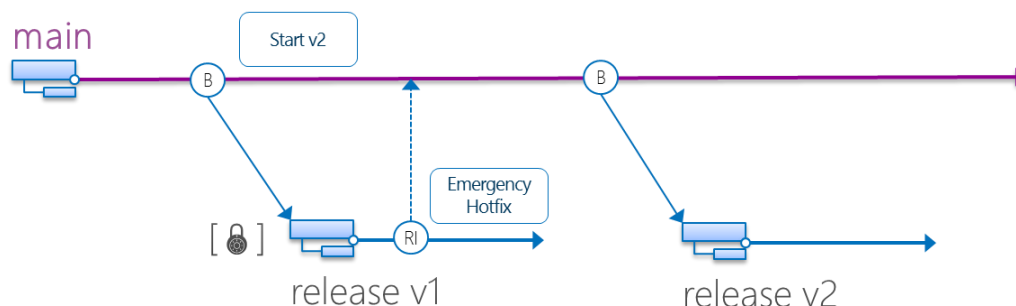


Figure 4 – Release isolation branching strategy

#### NOTE

There will be scenarios that are not black and white, which is why we continue to show emergency hotfixes done on release branches which, in theory, should be immutable.

See the **Real World Scenarios** on page 30, for real-world example using the branching guidance, but deviating (evolving) from the black & white guidelines without affecting delivery or solution quality. Do **NOT** view this guidance as immutable and cast-in-stone. Instead, evolve continuously to match your environment and requirements.

## Usage scenarios

- You need **isolation** and **concurrent** releases, protecting the stable Main branch.
- You have **multiple, parallel major releases**, supported and shipped to customer using the appropriate release branch.
- You need a servicing model for customers to upgrade to the **next major release**, i.e. v1 → v2.
- You have **compliance requirements** that require accurate snapshots of sources at release time.

## Considerations

- Each release branch should be a full child branch of the main branch.
- Your major product releases from the release branch.
- Lock (read-only) the release branch using access permissions to prevent modifications to a release.
- Changes from the release branch may merge (RI) to main. You should not FI from the Main branch into Release branches.
- Create new release branches for subsequent major releases if you require that level of isolation.
- Any fixes shipped from the release branch can include patches from that branch. Patches can be cumulative or non-cumulative regardless of branch plan.

## Development and Release Isolation

The Development and Release Isolation strategy combines the Development Isolation and Release Isolation strategies, embracing both their usage scenarios and considerations.

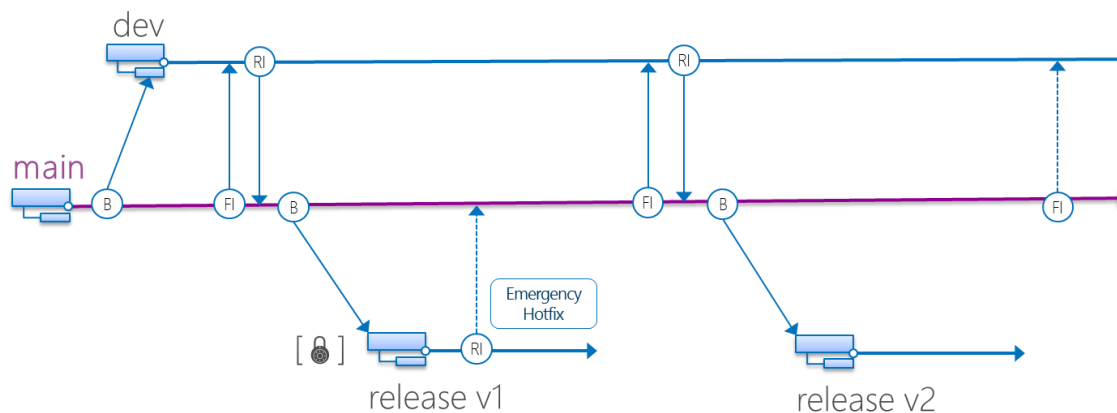


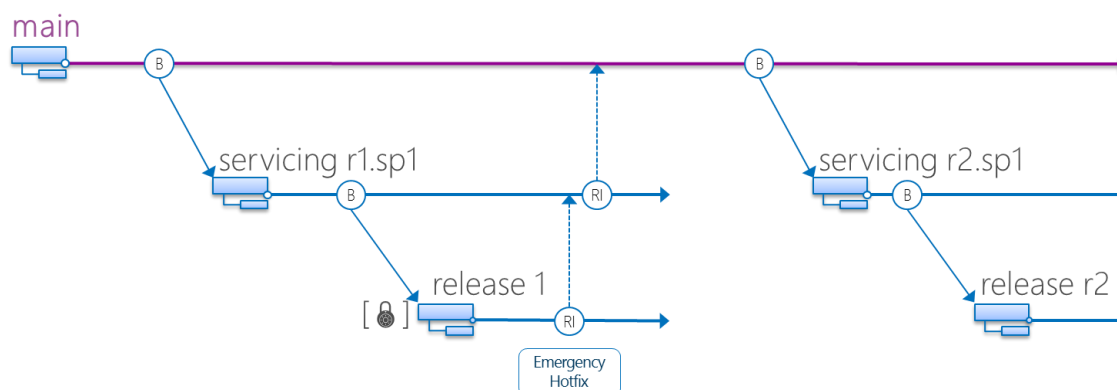
Figure 5 – Development and release isolation branching strategy

## Servicing and Release Isolation

**WARNING**

This is an advanced isolation strategy, which introduces complexity, maintenance and cost in terms of tracking, merging and service management. We recommend that you evolve to this strategy as needed and do not start a new project with this strategy or derivations thereof as a blueprint.

The Servicing and Release Isolation strategy introduces servicing branches, which enables concurrent servicing management of bug fixes and service packs. This strategy evolves from the Release Isolation strategy, or more typically from the Development and Release Isolation strategy.



**Figure 6 – Servicing and release isolation branching strategy**

**NOTE**

Previous versions of the Branching and Merging guidance referred to this strategy as the **Standard Branch Plan**.

### Usage scenarios

- You need **isolation** and **concurrent** releases, protecting the stable main branch.
- You have **multiple major releases**, supported and shipped to customer using the appropriate release branch.
- You need a servicing model for customers to upgrade to the **next major release**, i.e. v1 → v2.
- You need a servicing model for customers to upgrade to **additional service packs** per release, i.e. v1 SP1 → v1 SP2, which is the main differentiator of this strategy to the release isolation strategy.
- You have **compliance requirements** that require accurate snapshots of sources at service and release time.

### Considerations

- The service and release branches are branched from main at the same time to create a main → servicing → release parent-child relationship.
- Servicing branch
  - Each servicing branch should be a full child branch of the main branch.
  - Your service pack releases from the servicing branch.
  - Changes from the servicing branch merge (RI) to main. You should not FI from the Main branch into Release branches.
  - Lock (read-only) the servicing branch using access permissions to prevent modifications to a release.
  - Any fixes shipped from the servicing branch can include all previous fixes from that branch. Patches can be cumulative or non-cumulative regardless of branch plan.
- Release branch
  - Each release branch should be a full child branch of the servicing branch.
  - Your major product releases from the release branch.
  - Changes from the release branch merge (RI) to service. You should not FI from the Main branch into Release branches.
  - Lock (read-only) the release branch using access permissions to prevent modifications to a release.
  - Any fixes shipped from the release branch can include all previous fixes from that branch. Patches can be cumulative or non-cumulative regardless of branch plan.
- Create new service and release branches for subsequent major releases if you require that level of isolation.



The diagram illustrates the release process for a new production version (r1.sp1) using a branching strategy. The main branch (purple) leads to a 'servicing' branch (blue). From 'servicing', a 'hotfix r1.sp0' is created, which leads to a 'release r1.sp0' branch. The 'release r1.sp0' branch is used to 'Delete old releases' and then to create 'release r1.sp1'. The 'release r1.sp1' branch is then merged back into the 'servicing' branch and the 'main' branch. The 'hotfix r1.sp0' branch is also used to 'Delete old hotfixes'.

## NOTE



## Code Promotion

The code promotion plan promotes versions of the code base to new levels as they become more stable. Other version control systems have implemented similar techniques with the use of file attributes that indicate what promotion level of a file. As code in main becomes stable, you promote it to the test branch (often referred to the QA branch), where full regression tests take place and additional bug fixes may occur. Once the code is ready for release in the testing branch, it is merged into the production branch, where final quality certification and testing be done. Concurrent development can continue in the main branch, which promotes the concept of no “code freeze”.

### NOTE

A Code Promotion strategy feels like a relic from the waterfall development era. We may associate this with long testing cycles and separate development and testing departments. Generally, we no longer recommend this strategy since it is dated.

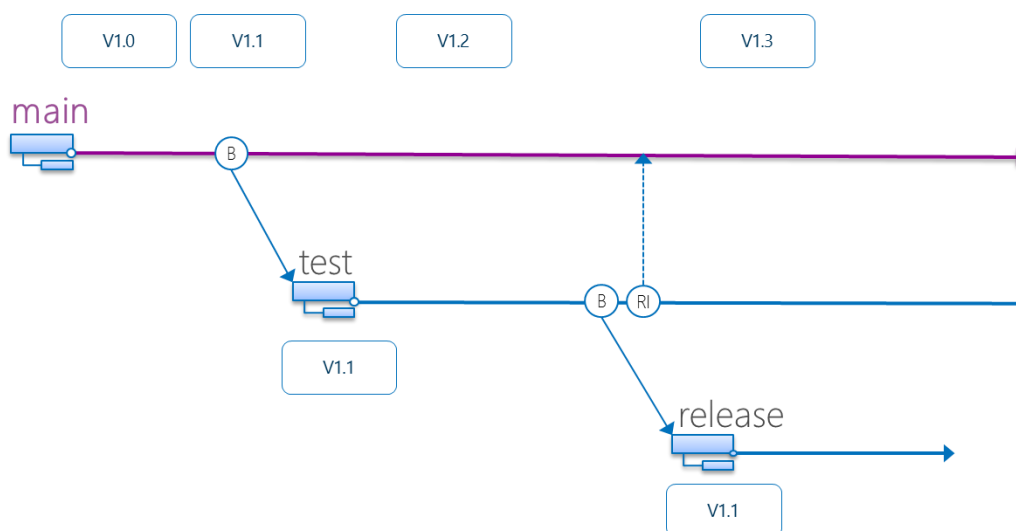


Figure 8 – Code promotion branching strategy

## Usage scenarios

- You have only **one major release**, supported and shipped to customer using the main branch.
- You have **long running testing lifecycles**.
- You need to **avoid code freezes**, and still have isolation for your major release.

## Considerations

- Test branch is a full child branch of the main branch.
- Release branch is a full child branch of the test branch.
- Your major product releases from the release branch.
- Changes from the test branch merge (RI) to main. This merge is one way.
- Restrict permissions to release branch to isolate risk.

## Feature Isolation

The Feature Isolation strategy introduces one or more feature branches from main, enabling concurrent development of clearly defined features for the next release.

### NOTE

The Feature Isolation strategy is a special derivation of the Development Isolation strategy, essentially with two or more development (feature) branches.

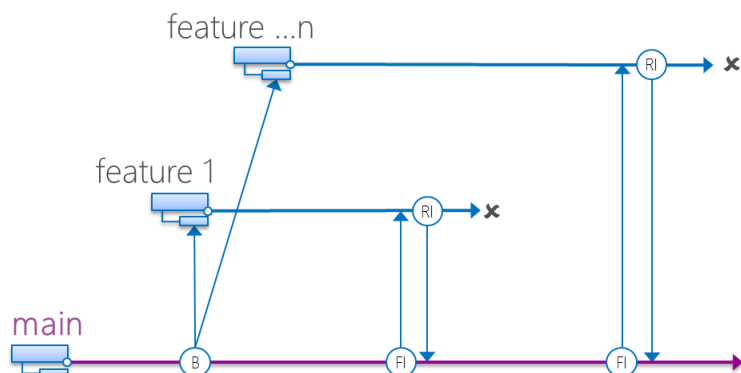


Figure 9 – Feature isolation branching strategy

### NOTE

Consider concept of folders for organizing branches when working with feature branches, for example `$/BranchingScenarios/features/featurePOWER`

With a Feature Isolation strategy, we can isolate each feature in its own branch giving us a lot of flexibility in terms of when we move a release into production. This strategy also helps to prevent situations where a branch needs to be “cherry-picked” for features that are ready for production while other features in the same branch are not ready. Merge features into main as they become stable enough for release, delete and create new feature branches as necessary.

## Usage scenarios

- You need **isolation** and **concurrent** development of **clearly defined features**, protecting the stable main branch.
- You may have **separate dedicated teams** developing various features that all roll into one product.
- You are developing a **large product** with **many features** needing isolation.
- You are developing features in parallel that may not be on the same release cycle.
- You need the ability to easily rollback features from a release. Rolling back features can be costly (and may reset testing) and while possible, should be used with care.

## Considerations

- Each feature branch should be a full child branch of the main branch.
- Keep the life of your feature development short, and merge (RI) with main frequently.
- Feature branches should build and run Build Verification Tests (BVTs) the same way as main.
- Merge (FI) frequently from main to feature branches if changes are happening directly on main.
- Merge (RI) from feature to main based on some objective team criteria, e.g. Definition of Done (DoD).

# Alternative Strategies

## Adapt your branching process for inevitable 'blips'

*Blip - (noun) - an unexpected, minor, and typically temporary deviation from a general trend.*

Even with the best project planning and intentions, the stakeholders may ask your development teams to change course. Be it a temporary change in priorities, failure to meet a deadline or a change in direction, you need to be able to handle these 'blips' as they happen and, *importantly*, deal with their consequences in the longer term.

Remember, all you need to guarantee you can repeat a build is a changeset because it is immutable. Labels and branches are just conveniences. Use them to your advantage to handle blips in your process and to simplify the support and delivery of your software.

Whether you have a simple main branch strategy or an advanced 'main, servicing, hotfix, feature' branching strategy, the process is the same:

### 1. Analyze the new requirement and do not panic

While the team may be panicking over how to deal with the blip, keep a steady head and remember your strategy. Ensure that the person controlling your branching is involved in this analysis. The person who has control over branches in your source control system will likely have a good understanding of which procedures they follow, the exact state of the system, and the release vehicles. They will be able to provide invaluable 'matter of fact' input.

### 2. Branch only if necessary

Once you understand the requirements, you may find that is unnecessary to create another branch, if however, you do need a new branch, then it is important to understand how branching options. Visual Studio allows you to branch by **Changeset**, **Date**, **Label**, **Latest Version** or **Workspace Version**.

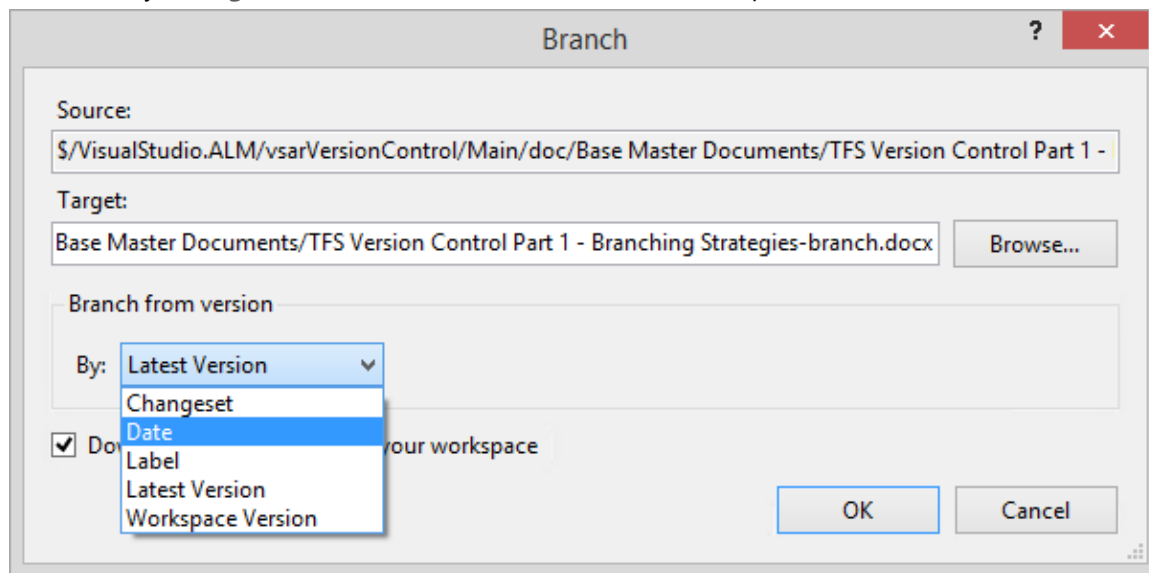


Figure 10 – Branch from version using Date

We generally recommend you get a confident understanding of your requirement and branch by Changeset.

### 3. Remove the branch

If you needed to create a branch, your ultimate goal will likely be to get the code from the branch back into your normal operating branch or branches to mitigate any long-term consequences. Try to be understanding but firm in working towards this goal. Temporary and numerous branches cause confusion and overhead for the team.

See a simple walkthrough on page 28, where we deal with a blip that requires branching to support an 'out-of-band' patch and the subsequent collapsing of that branch.

## Feature Toggling

### NOTE

Please also refer to the [Using Feature Toggles as an Alternative to Development or Feature Branches](#)<sup>15</sup> article, by [Bill Heys](#)<sup>16</sup> for more information on feature toggling.

Feature Toggling is a useful technique for toggling features on and off to determine whether a software feature is available at run time. These may be UI features visible by the end user or simply code itself driving functionality. There are several names and nicknames for this technique that include flags, toggles and switches, and various implementation strategies that include both build and run time. For the purpose of this guidance we will discuss run time feature toggling, which involves the hiding or showing of features during run time. Most of the time the implementation involves a configuration file where the toggles are set. We can combine this with the wrapping of UI elements or classes, and thus control the visibility of the various features. Some implementations leverage a data driven approach where we maintain feature toggle state in a database.

### Usage scenarios

- You need to release features **frequently** and **continuously**.
- You need to be able to **enable** and **disable features** on the fly.
- You need to selectively enable/disable features to a specific **audience** or set of **users**

### Considerations

- Make use of "**Canaries**" to release and evaluate features with a selected audience.
- Ensure you do not leak features inadvertently to the wrong audience or enable features that are not ready.

### Advantages

There are several advantages for implementing feature toggling. One simple advantage is yanking or rolling back a feature becomes easy and done on the fly, and without an entire recompilation and redeployment ceremony.

With the advent of continuous delivery, software releases are happening very frequently, and many features complete at different times, and can span multiple sprints. Often this requires complex branching and merging strategies to coordinate the various feature development activities. This can lead to reduced velocity and increased merge conflicts. The further the various branches drift from each other, the worse the merging exercises will become. With feature toggling, in general, we can expect to use less branches since we can have a single codebase contain both features that are done, as well as features still under development. Everyone can safely integrate to Main continuously and in general without worry of destabilizing the code base.

Another advantage to feature toggling is the ability to do so called "A/B", "canary", and various other scenarios where you may want to test features on only a certain segment of your audience or customers. Many popular online social networking applications leverage these techniques to roll out new features selectively to their audiences. You can gradually open the feature up to more and more of your user base as you get the success you desire during this type of roll out. Of course, you can also roll back features quickly if things do not go so well.

---

<sup>15</sup> <http://aka.ms/vsarfeaturetoggling>

<sup>16</sup> [http://blogs.msdn.com/b/willy-peter\\_schaub/archive/2010/07/23/introducing-the-visual-studio-alm-rangers-bill-heys.aspx](http://blogs.msdn.com/b/willy-peter_schaub/archive/2010/07/23/introducing-the-visual-studio-alm-rangers-bill-heys.aspx)

### Disadvantages

Of course, with any technology choice, there are pros and cons. One disadvantage to feature toggling is you must ensure you do not accidentally forget to wrap the features appropriately, and thus inadvertently make a feature visible before it is ready. Feature toggling also requires upfront planning and some technical overhead on implementation of a strategy.

Inadvertent disclosure is another potential risk. There can be situations where the live code base contains features still under development. Crafty end users may view these feature in plain sight simply by looking at your source. This might give a competitor or a user visibility into what features are coming in your product. This might cause you to implement cryptic naming for your various features if this is an issue for your product.

You also need to ensure you are testing scenarios for the feature being both on and off, which of course involves some overhead.

### Techniques and Practices

You can leverage some common techniques to ensure you optimize your feature toggling approach. One of the key tenets of feature toggling, as well as a typical best practice in building software in general, is to ensure your developers are writing code as modular as possible. For example, you may have method or function where you implement several features simultaneously via nested if/else clauses. This does not line up well with isolating features. Ensure you are compartmentalizing your features as much as possible.

Adopt naming conventions and consistency to ensure the feature toggling techniques are consistent throughout your codebase. Also, if security is an issue, ensure you are naming your features appropriately for obfuscation of the feature. It is very important to keep in mind scenarios where end users are able to view pre-release features or features that you want to hide in source code. Clever users may derive the meaning of hidden features by following loose naming conventions. If this is important to your scenario, you must weigh the risks and determine a strategy.

You will want to ensure you do appropriate smoke testing both with feature toggles on as well as off for your various features. The essential idea is to understand how your application behaves for each of the feature states. Obviously depending on the exact scenario, there may or may not be nuances for specific features.

In some cases you can eventually deprecate the feature flag from your code base, while in other cases you may want to leave the ability to toggle on and off the feature after is it in production a while. This is a preference decision, but in general a cleanup of features implemented long ago can help keep the codebase tidy.

### Tools

There are various open source tools available for implementing feature toggling. Some companies employ their own custom-built implementation. Depending on the type of application you are building, may determine the appropriate technology to leverage. For example in an ASP.NET application, you may wrap UI elements, and drive the toggles via a configuration file.

At Microsoft, the team responsible for implementing Visual Studio Online (also called Team Foundation Service) employs feature toggles. The implementation used in this scenario is a custom-built solution that leverages a data driven approach combined with a REST based service for toggling on and off features. This enables this team to maintain feature toggle state in a database. With this approach, this team achieves the benefits described in this guidance.

#### NOTE

This guidance is not recommending one technology or tool over another at this point. Here is a starter to look at: [Feature Toggle libraries for .NET](http://david.gardiner.net.au/2012/07/feature-toggle-libraries-for-net.html) <sup>17</sup>

---

<sup>17</sup> <http://david.gardiner.net.au/2012/07/feature-toggle-libraries-for-net.html>

## Continuous Delivery

Continuous delivery relies on automation of build, testing and deployment. Change is continuous, frequent and merge operations more challenging as merge conflicts often require manual intervention. It is recommended to avoid branching and rely on other strategies, such as **feature toggling** (page 20), when considering continuous integration.

### Usage scenarios

- You have only **one major release**, supported and shipped to customer using the main branch.
- You have **short** feature development **cycles**.
- You need to release features **frequently** and **continuously**.

### Considerations

- Labels, when used as bookmarks, are mutable (can be changed) and there is no history of changes made to labels.
- Peruse [Team Foundation Server Permissions](#)<sup>18</sup> and consider protecting the administration of labels.
- Avoid **cherry picking** which complicates tracking of changes and subsequent merge operations.

NOTE

If you are looking for a continuous build and deployment pipeline, we recommend that you peruse the [ALM Rangers DevOps](#)<sup>19</sup> guidance and the [Building a Release Pipeline with Team Foundation Server](#)<sup>20</sup> guide.

When using feature toggling, we can adopt the main only branching strategy, enabling and disabling features by toggling them on or off as shown. The main branch becomes the trusted source for the automation pipeline.

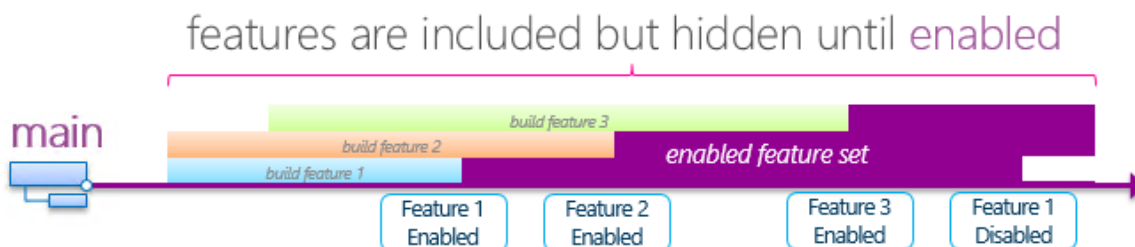


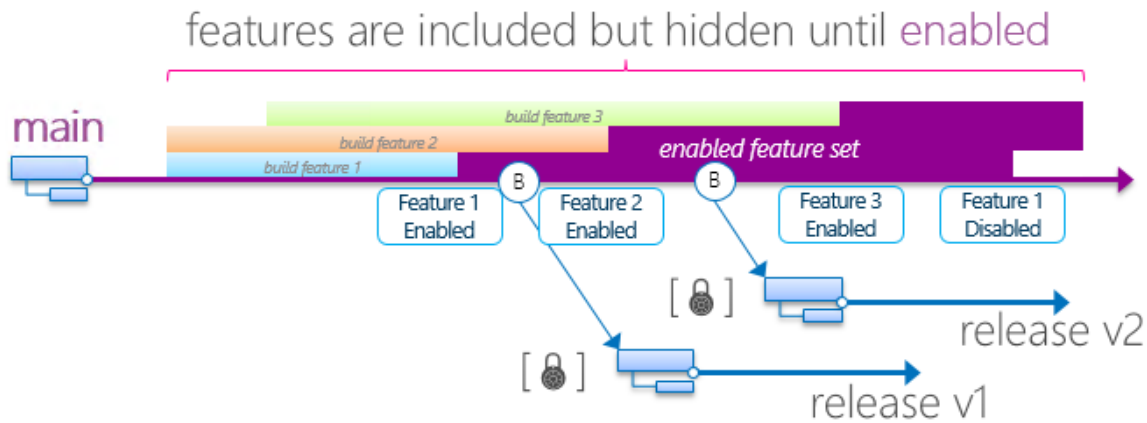
Figure 11 – Continuous integration: Feature toggling

<sup>18</sup> [http://msdn.microsoft.com/en-us/library/ms252587\(v=vs.100\).aspx](http://msdn.microsoft.com/en-us/library/ms252587(v=vs.100).aspx)

<sup>19</sup> <http://aka.ms/treasure54>

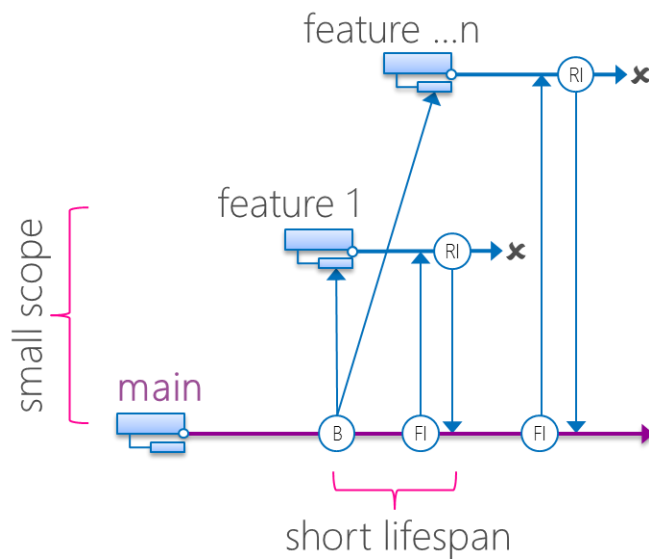
<sup>20</sup> <http://aka.ms/treasure53>

Release isolation is feasible if you use branches as snapshot archives only and lock the branches down appropriately.



**Figure 12 – Continuous integration: Feature toggling with release isolation**

If you opt for the use of feature isolation, it is important to keep the scope of each feature small and the duration short. By making the feature branches short-lived, the merge operations and potential impact are as small as possible. Restrict the lifespan of a feature branch to hours, rather than days, and delete the feature branch as soon you merge (RI) it with the main branch.



**Figure 13 – Continuous integration: Feature isolation**


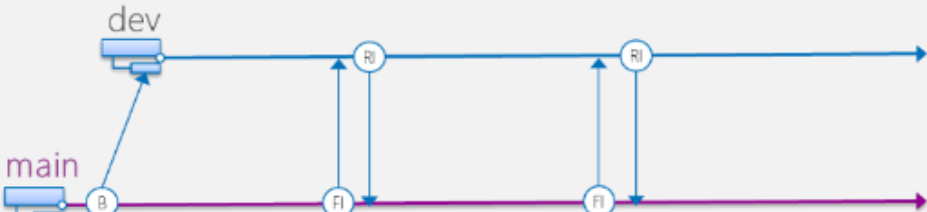


# Walkthroughs

## From nothing to complexity or not

NOTE

The walkthrough relies on the Hands-on Lab (HOL) from “Simple to Complex or not”, page 36. Refer to the HOL for a more detailed and sample solution based walkthrough.

Step	Instructions
1 Starting with <b>Main Only</b> <input type="checkbox"/> - Done	 <p>We <b>recommend</b> that you start with this strategy, avoiding branch and merge complexities.</p> <ul style="list-style-type: none"> <li>Consider when you have <b>no need for isolation</b> (yet) and are comfortable using labels.</li> <li>Optionally convert the main folder to a main branch to enable branch-specific features, such as visualization and the ability to store properties like owner and comment.</li> </ul> <p><b>COMPLEXITY</b> → Low 😊   <b>Isolation:</b> None   <b>Releases:</b> 1</p>
2.1 <b>Development</b> Isolation – Getting Started <input type="checkbox"/> - Done	 <ul style="list-style-type: none"> <li>Consider the <b>Development Isolation</b> strategy if you need <b>isolation of development</b> such as new features, experimentation or bug fixes.</li> <li>Branch from <b>main</b> to <b>dev</b> when you have a need for concurrent support of a major release and development.</li> <li>Before you merge new features from dev to main (<b>Reverse Integration</b>), perform a merge from main to dev (<b>Forward Integration</b>) to pick up changes from the main branch.</li> </ul> <p><b>COMPLEXITY</b> → Moderate 😐   <b>Isolation:</b> Development   <b>Releases:</b> 1</p> <p><b>NOTE</b> Consider creating a second workspace for the dev branch to isolate it from main.</p>
2.2 <b>Development</b> Isolation – Hotfix <input type="checkbox"/> - Done	<ul style="list-style-type: none"> <li>Develop hotfixes in the <b>dev</b> branch if part of the concurrent development stream.</li> <li>Alternatively (not recommended) develop <b>emergency</b> hotfixes in the <b>main</b> branch and merge (FI) changes to the <b>dev</b> branch at the earliest convenience.</li> </ul> <p><b>COMPLEXITY</b> → Moderate 😐   <b>Isolation:</b> Development   <b>Releases:</b> 1</p>

Branching Strategies – Walkthroughs

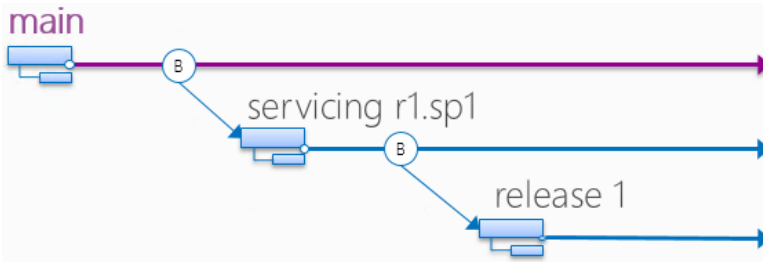
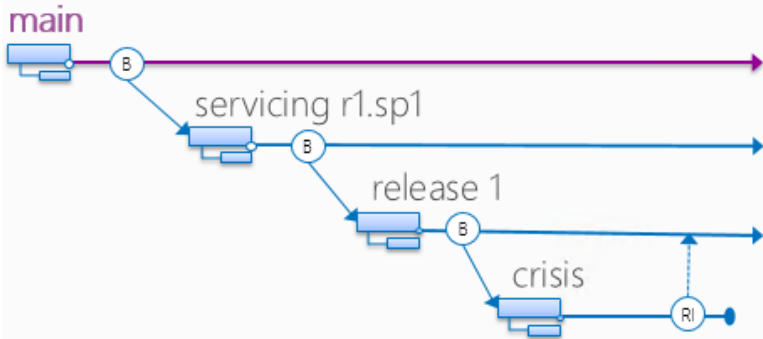
Step	Instructions
3.1 <b>Feature Isolation</b> – Getting Started ☐ - Done	<div></div> <ul style="list-style-type: none"><li>Consider using the <b>Feature Isolation</b> strategy if you have a need to develop <b>clearly defined features</b> concurrently.</li><li>Investigate and decide whether to branch your feature branches off <b>main</b> (as above) or the <b>dev</b> branch (as below).</li></ul> <div></div> <ul style="list-style-type: none"><li>If you find you are frequently developing emergency hotfixes on main, consider using Feature Isolation for <b>hotfixes</b>, branching feature branches off main.</li><li>Before merging new features from feature to parent (main or dev), perform a merge from parent to the feature branch (<b>FI</b>) to pick up latest changes from the parent branch.</li></ul> <p><b>COMPLEXITY</b> → Moderate ☹   <b>Isolation:</b> Development   <b>Releases:</b> 1</p>
3.2 <b>Feature Isolation</b> – Cleanup ☐ - Done	<ul style="list-style-type: none"><li>Optionally <b>delete</b> and/or <b>destroy</b> feature branches when no longer needed to reduce noise.</li><li>Be cautioned that <b>destroy</b> is a non-reversible action and should be used with caution.</li></ul>
4 <b>Release Isolation</b> ☐ - Done	<div></div> <ul style="list-style-type: none"><li>The <b>Release Isolation</b> strategy is useful when you need to snapshot and support multiple main releases, i.e. v1 and v2</li><li>Optionally (not recommended) develop <b>emergency</b> hotfixes in the <b>release</b> branch and merge (<b>FI</b>) changes to the <b>main</b> branch at the earliest convenience.</li></ul> <p><b>COMPLEXITY</b> → Moderate ☹   <b>Isolation:</b> Release   <b>Releases:</b> 2+</p>

## Branching Strategies – Walkthroughs

Step	Instructions
<p>5</p> <p><b>Development &amp; Release Isolation</b></p> <p>☐ - Done</p>	<ul style="list-style-type: none"> <li>You typically evolve from a Development Isolation strategy to a Development &amp; Release Isolation strategy by adding Release Isolation.</li> </ul> <ul style="list-style-type: none"> <li>When you merge changes from a release branch to the main branch, such as due to an emergency hotfix, ensure that you merge (FI) these changes to the development and/or feature branches as soon as possible.</li> </ul> <p><b>COMPLEXITY</b> → Moderate ☺ to High ☹   <b>Isolation:</b> Development &amp; Release   <b>Releases:</b> 2+</p>
<p>6</p> <p><b>Servicing and Release Isolation</b></p> <p>☐ - Done</p>	<ul style="list-style-type: none"> <li>If and only if you require a release strategy with <b>concurrent servicing management</b> of bug fixes and service packs, you can evolve to the <b>Servicing and Release Isolation</b> strategy.</li> <li>Analyze and define your servicing and release <b>naming convention</b>.</li> <li>Rename your <b>release</b> branch to <b>servicing</b>, and branch from <b>servicing</b> to <b>release</b>, as defined by your naming convention.</li> </ul> <ul style="list-style-type: none"> <li>Potentially (<b>not</b> recommended) inject further branches, i.e. hotfix into the hierarchy to create finer grained servicing management.</li> </ul> <p><b>COMPLEXITY</b> → [Very] High ☹   <b>Isolation:</b> [Dev &amp;] Release   <b>Releases:</b> 2+ and <b>Service Packs</b></p>
<p>7</p> <p><b>Other options</b></p>	<ul style="list-style-type: none"> <li>Consider the use of <b>Main Only</b> strategy as your default.</li> <li>Before embracing branching strategies, consider other options such as <b>Feature Toggling</b>.</li> <li><b>AVOID</b> complex branching models that are <b>fascinating</b>, but <b>costly</b> to maintain!</li> </ul>

Table 4 – Walkthrough: From nothing to complexity

## Adapt your branching process for inevitable 'blips'

Step	Instructions
1 Starting point ☐ - Done	<p><b>main</b></p>  <p>We <b>start</b> with a team who are using a <i>servicing and release isolation</i> strategy. There are three teams working on the code base. Some are working on main for the next major release, some are working on servicing to create the next minor release and some are working on the release branch to provide hotfixes.</p> <p><b>NOTE</b> The team providing hotfixes has a release cadence of around 1 hotfix per week, however they provide these hotfixes to the client for additional testing; there will be a delay between providing the fix and its production implementation.</p>
2 Blip – no branch ☐ - Done	<p>The team have provided hotfix 5 to the client and are working on hotfix 6. Hotfix 3 is in production. The client has raised a high priority bug, which needs to go into production as soon as possible. They are not able to take hotfixes 4, 5 or 6. Remember the steps,</p> <p><b>1. Analyze the new requirement and do not panic</b></p> <p>We know the client requires an out-of-band hotfix. After analyzing the bug, we determine that it is in an area of the code that is unchanged in hotfixes 4, 5 and 6. This means we can build the code and provide the fix from the release branch. The important part is that we test this patch, let us call it 3a for simplicity, on an environment that is at hotfix 3 level. <b>Always</b> have an environment or snapshot, which is at the same level as production.</p>
3 Blip – branch ☐ - Done	 <p><b>2. Branch only if necessary</b></p> <p>Let us use the same scenario as #2. However, this time we analyze the bug and discover that hotfix 4, 5 and 6 have all changed code in the same area. We will need to create a branch to meet these requirements, but that leads to more questions: From what branch should we create the new branch? What version should be this branch?</p> <p>Let us first deal with where. Our dev branch is far progressed from the release and the servicing branch is well on its way to providing the next minor release. The best branch to branch from is the release branch.</p> <p>Determining the version to branch depends on what is in production. As production is at hotfix 3, we need to identify the changeset that concluded hotfix 3 and branch by that. We will call this branch 'crisis' rather than 3a for reasons that we will make apparent shortly. We can now create Team Foundation builds for the branch, create the patch, test it and ship it to the client.</p> <p><b>4. Remove the branch</b></p>

# Branching Strategies – Walkthroughs

Step	Instructions
	<p>If we merge the changes back to release, they can be included in hotfix 6 to ensure we do not regress the fix. However, in this situation we find ourselves with a bigger problem. One could argue that we are providing too many hotfixes and the delay between providing the fix and implementing it is unacceptably slow. This may be true, however, this scenario will happen and we need to effectively recover as a priority and then re-evaluate our release cadence to try avoiding the situation happening again.</p> <p>In this scenario, rather than merging back (RI) to release, we need to merge (FI) all the changesets that made up hotfix 4 from release to the crisis branch. We can now use this branch to produce hotfix 4a (this is why we named the branch crisis and not 3a!). We follow the same process to produce hotfix 5a and then we merge (RI) back to release. We delete the branch only after hotfix 6 goes into production. It is potentially a big effort but the process is sound. <b>Always</b> be clear and candid on the impact of a fix; the client may be happy to deprecate a series of fixes and take a cumulative fix if the benefits make sense. It took a bit of effort but consider the crisis averted.</p>

Table 5 – Walkthrough: Adapt your branching process for inevitable ‘blips’

# Real World Scenarios

## Delivering software at intervals ranging from days to months

We base the following scenarios on real world implementations successfully delivering software into production at various intervals ranging from long term -- *6 to 9 months* -- through mid-term -- *1 to 3 months* -- to short term ad hoc deliveries -- *1 to 14 days*. Team sizes range from 20 to 100+ and typically involve a supporting infrastructure of several hundred servers. The code base is typically in the region of 5 million lines. The point is that these are large systems; however, the scenario as a whole could be applicable to much smaller projects.

### NOTE

In this scenario, we will *bend* some of the rules, which we have outlined in this guide. We will be coding directly on the Main and Release branches and illustrating how we use our understanding of the concepts illustrated in this guide to provide a working and successful solution.

## Walkthrough

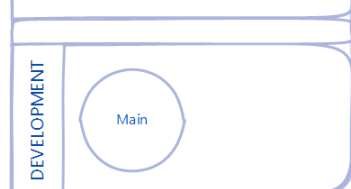
This scenario works through three releases. We will use a three-part numeric sequence with the following makeup r.s.hh, where r = Release, s = Servicing and h = Hotfix. For example, our three releases will be 1.1.0, 1.2.0 and 2.1.0.

The 1.1.0 and 1.2.0 releases will branch off from the Servicing 1.0 branch while 2.1.0 will branch from the Servicing 2.0 branch. When teams are discussing versions / branches, it is clear that a two-part name is a Servicing branch while a three-part name is a release branch. It is also clear from which Servicing branch we took the Release branch.

### Scenario Stages and Detail

**Getting Started** - The team starts with a single main branch. They have a clear set of requirements and a delivery date to meet. At this stage, they do not know what the released version will be, but collectively the team refers to the release as release 1. The Development, Test and Build team work together to develop the solution.

#### Branches

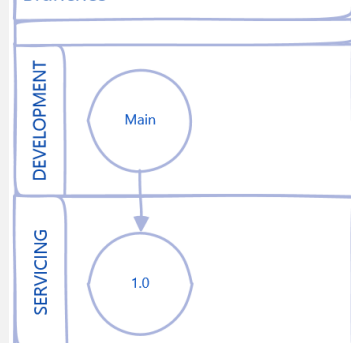


**Branch to Stabilize** - When the code is stable the team decide that they will branch it from Main into a Servicing branch for final stability changes.

Two teams can now operate independently. On Main some of the team can start work on the next major delivery, while on Servicing 1.0 other team members can wrap up final stability fixes.

We would expect a larger test representation of the team working on Servicing to ensure it meets release quality, while a subset start ramping up on new tests for the work on Main. Depending on your level of automation you may see the opposite.

#### Branches



## Branching Strategies – Real World Scenarios

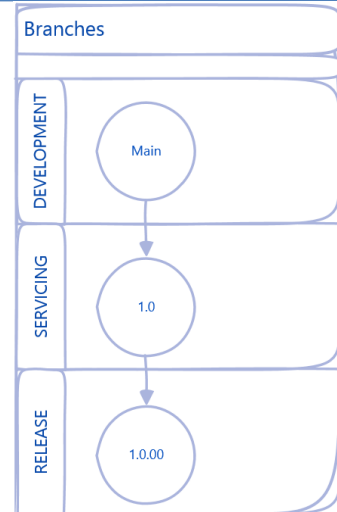
### Scenario Stages and Detail

**Branch to Release** – When you deem the code to be of sufficient quality, the team take a Release branch from Servicing. At this point, it is worth thinking a little about the exact name you give the Release branch. We do not expect to branch more than one or two release branches from Servicing 1.0 so we have single padded the Servicing digit.

NOTE

If you expected to release 10 or more then pad the Servicing number with a zero, e.g. 1.00.00. Why? Simply because in Source Control Explorer the releases will be listed in the correct order.

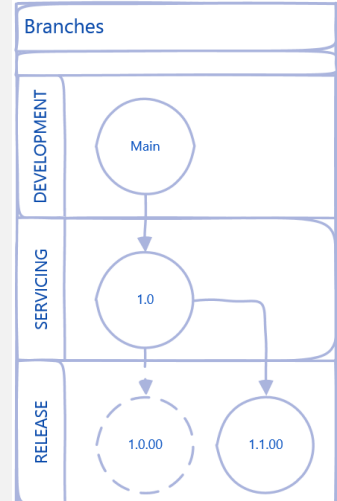
Branch Release 1.0.00 deemed as having sufficient quality, not necessarily production quality. While the release is with the client, create all hotfixes on the Release branch to address any immediate issues. At the same time, a team is working on the Servicing branch to further stabilize the code and reduce the bug count to the point that it reaches production quality. Any hotfixes, e.g., 1.0.01, 1.0.02 are reverse integrated back into the Servicing 1.0 branch and eventually all parent and new child branches.



**Branch to Release (again)** - In our scenario Release 1.0.00 will never actually hit production. We provide this release to the client so that they can perform integration testing with various other partners.

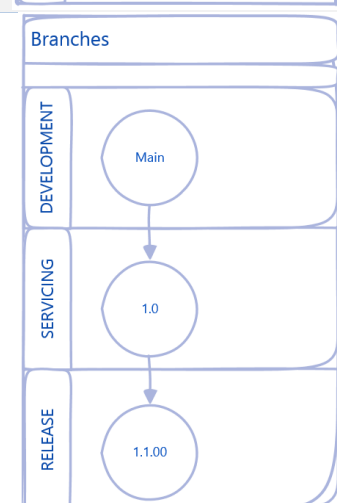
Release 1.0.00 has been issued with several hotfixes which have all been merged back to Servicing 1.0 and the team looking after that branch have deemed that the code is production ready. Release 1.1.00 is branched from Servicing 1.0.

It all comes down to resource, but try to limit the number of branches you are supporting, especially Release branches as taking a fix there means various merges need to take place. Ideally you should try use Servicing as a buffer. If a bug is not of high enough importance, try to convince the client that it will be better to take the fix in the next release. This way you do not have to create and test a hotfix and the change can be made in Servicing which will get more testing time.



**System Live** - Release 1.1.00 is now in production. Any hotfixes are coded on the Release branch, tested and progressed into production as fast as possible. This is to avoid situations where a hotfix is coded on Release and in the time it takes to be signed off, another issue is found which is more important and needs to go in first. In this situation it is often easiest to rollback the first change, implement the critical fix and then redo the first fix. Alternatively you could branch by changeset from the Release branch, but this has additional costs like setting up builds and moving a team over to the new branch.

We also use the Servicing branch for longer term fixes on a release. For example, if a fix is going to take two weeks to implement and you expect to issue hotfixes in that time, then a team can work in Servicing and when they are done the fix is merged into release and the hotfix issued. This assumes that Servicing is not being used to stabilise a new set of code to branch off as another release. If this is the case then you need to branch a Feature branch from the Release branch to provide the isolation needed.



## Branching Strategies – Real World Scenarios

### Scenario Stages and Detail

**Maturity** - Let's talk through the following diagram which shows us supporting eight branches. Production is running Release 1.1.00.

**Main** – here we have a team working on release 3 functionality.

**Servicing 1.0** – this is really just a pass through branch at this stage.

**Release 1.0.00** – is actually deleted and shown here to illustrate how we take multiple releases from Servicing.

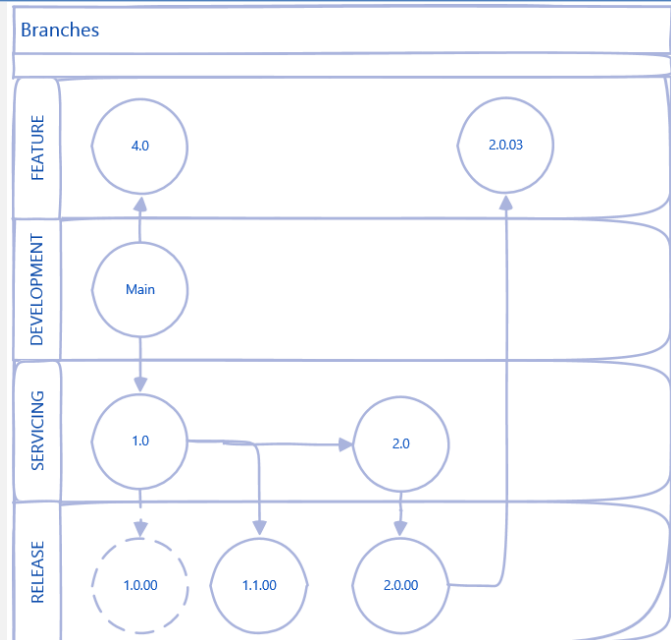
**Release 1.1.00** – this is where we work on production hotfixes

**Servicing 2.0** – has a team working on stabilising the code that will be branched as 2.1.00 and expected to replace 1.1.00 in production.

**Release 2.0.00** – is there to support the release that the client is using for integration testing with other systems.

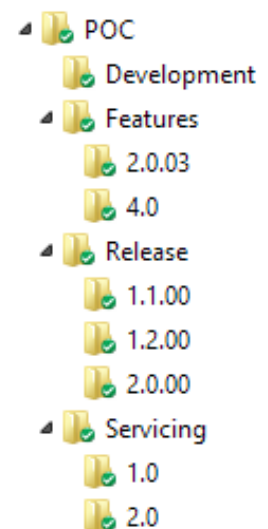
**Feature 2.0.03** – is a hotfix we are working on which we know will take a while to implement and we can't work on Servicing as its stabilizing 2.1.0 code. Once the code is stable we will merge it to 2.0.00, issue the hotfix and probably delete the 2.0.03 branch.

**Feature 4.0** – here work is being done on functionality that is only expected to be ready after the next major release from main. It's important to remember that a Feature branch can also be taken from Main to work on something which may not be signed off for the release; that way Main can easily ship without the feature rather than having to rollback code on Main because the client changed their mind.



### Key Points

- We code on Main for major releases but may use feature branches to offer protection from having features cancelled or postponed.
- We code on Release branches and do not treat them as immutable. We use additional Feature branches and Servicing branches when we expect a fix to take some time. This way we leave the Release branch available to code any high priority hotfixes.
- All you need to guarantee you can repeat a build is a changeset because it is immutable. Labels and branches are just conveniences. Use them to your advantage to simplify the support and delivery of your software.
- We merge code from Release to Servicing as soon as a hotfix ships. We typically merge that hotfix to main and any other later branches so that we do not regress. We merge from Servicing to Main when we branch to Release.
- We leave the Servicing branch in place even though it may prove to simply be a pass through branch for some time. It provides us with a natural place to code a longer term hotfix and may even be used to create another full release if project circumstances change.
- We use a consistent naming system for our branches.
- We try to limit the number of branches that we need to hotfix as hotfixes can be expensive to deliver.
- We arrange our branches into folders for easy navigation as shown here...





# FAQ

## Strategies

### What are the differences between folders and branches?

Starting with TFS 2010 there is a distinction between branches and folders in version control. A branch enables some branch-specific features such as visualization, as well as the ability to store properties like the owner and a comment to each branch. For more information see:

- [Branch Folders and Files](#) <sup>21</sup>
- [View the Branch Hierarchy of a Team Project](#) <sup>22</sup>
- [View Where and When Changesets Have Been Merged](#) <sup>23</sup>
- [Team Foundation Server Permissions](#) <sup>24</sup>

### Why have we changed the strategy names?

We listened to the way engineering and users were talking about branching in casual and technical discussions. Our decision was to introduce strategy names that are meaningful and which you can visualize in your mind. For example, visualize a basic branch plan. Now repeat the exercise, visualize a development, then a release, and finally a development and **release** isolation strategy. Which was easier to visualize?

## Operations

### Are work items a part of the branching/merging model?

No, work items exist at a logical level that is distinct from that of the source control repository. When you perform a branch operation, you will not: duplicate the work items, carry over work item history, or update work item links. For more information see: [Copy a Work Item](#) <sup>25</sup>.

### Can source code be branched across team projects?

Yes, you can but we do not recommend it unless two teams must share source and cannot share a common process.

### How should we handle bugs over branches?

As outlined in **Are work items a part of the branching/merging model?**, page 33, branching of source code has no impact on work items, including bugs. Consider who is responsible for the code and the bugs, whether responsibility and ownership moves (copied) the code or if bugs are resolved as part of the merge process.

For more information, see [Copy a Work Item](#) <sup>26</sup> and [Create or Delete Relationships Between Work Items](#) <sup>27</sup>.

### Is there a preferred "Branch from version" value when creating a branch?

When creating a branch, there are five options available from which to choose the source.

---

<sup>21</sup> [http://msdn.microsoft.com/en-us/library/ms181425\(VS.110\).aspx](http://msdn.microsoft.com/en-us/library/ms181425(VS.110).aspx)

<sup>22</sup> [http://msdn.microsoft.com/en-us/library/dd465202\(VS.110\).aspx](http://msdn.microsoft.com/en-us/library/dd465202(VS.110).aspx)

<sup>23</sup> [http://msdn.microsoft.com/en-us/library/dd405662\(VS.110\).aspx](http://msdn.microsoft.com/en-us/library/dd405662(VS.110).aspx)

<sup>24</sup> [http://msdn.microsoft.com/en-us/library/ms252587\(VS.110\).aspx](http://msdn.microsoft.com/en-us/library/ms252587(VS.110).aspx)

<sup>25</sup> [http://msdn.microsoft.com/en-us/library/ms181321\(VS.110\).aspx](http://msdn.microsoft.com/en-us/library/ms181321(VS.110).aspx)

<sup>26</sup> [http://msdn.microsoft.com/en-us/library/ms181321\(VS.110\).aspx](http://msdn.microsoft.com/en-us/library/ms181321(VS.110).aspx)

<sup>27</sup> [http://msdn.microsoft.com/en-us/library/dd286694\(v=vs.110\).aspx](http://msdn.microsoft.com/en-us/library/dd286694(v=vs.110).aspx)

- **Labels** and **workspaces** do not usually refer a point in time but rather to a collection of files grouped together by the user.
- **Date**, **Latest Version** or **Changeset** usually refer to a point in time and address the needs of larger teams. For example, for bug resolution knowing when the break occurred is a point in time (date) to consider. When moving to another sprint or iteration the Latest Version or Changeset may be the best fit.

### Can you delete branches?

Yes. You need to understand your branching hierarchy and aware of future implications before deleting a branch. By deleting a branch, you are potentially removing a parent or child relationship, introducing the need for baseless merging.

For example, the diagram on the left allows changes to propagate from A -> B, B -> C and vice versa through the branch hierarchy. If you delete the branch B (see image on the right), you can only get changes from A -> C and C -> A using a baseless merge.

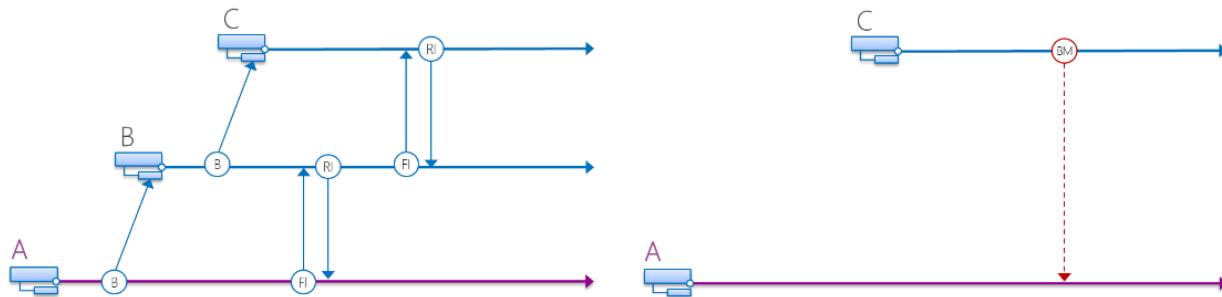


Figure 14 - A -> B -> C branch hierarchy

For more information see [Branch Re-parenting](#)<sup>28</sup> and [Team Foundation Server Permissions](#)<sup>29</sup>

### What is required to consolidate or transition branches?

Before considering, a consolidation or transitioning of branches ask the following questions:

- What are the predominant reasons for considering this action?
- What are the benefits of this action?
- What are the benefits for remaining with the status quo?
- What are the risks of consolidation?
- What resources will be required to complete a consolidation?
- What is the exit plan?
- What effects will this have on future development?
- How will you incorporate any existing best practices?
- Do we need to keep history and associated work items with the source?

Once you have decided on the answers to these questions, you will determine your next steps based on whether you wish to consolidate your branches or transition into a new environment. Transitioning to a new branch may be the easier of the two options. You can essentially lock off your existing branches and have everyone work in the new environment from the designated start point. This is most appropriate when applications that have been created and maintained by teams that were previously separate and do not have anything in common, other than they are now all products of a combined entity.

<sup>28</sup> <http://blogs.msdn.com/b/hakane/archive/2009/05/19/enable-branch-visualization-in-upgraded-team-projects-tfs-2010-beta1.aspx>

<sup>29</sup> [http://msdn.microsoft.com/en-us/library/ms252587\(VS.110\).aspx](http://msdn.microsoft.com/en-us/library/ms252587(VS.110).aspx)

For consolidating branches, you will have to review your existing branches and determine what must be included. Once you have made those decisions, you will need to create a new mainline and proceed to migrate each designated branch. Along each step, you should verify that each migrated branch does not break the new mainline. If possible, do this in a sandbox environment prior to attempting in a production environment.

For more information, see [Branch Folders and Files](#)<sup>30</sup> and [Merge Folders and Files](#)<sup>31</sup>.

### What is the “Re-parent Branch” function and when should you use it?

Re-parent Branch is a feature that you can use to establish a parent-child relationship between baseless merged branches (i.e. branches with no existing merge relationship) as well as to alter existing branches' parent-child relationships in a branch hierarchy.

However, to “reverse” an existing parent-child relationship, you need to disconnect the child branch from the parent branch using the “No parent” option, and then re-parent the former parent branch to the former child branch.

Refer to [Branch Re-parenting](#)<sup>32</sup> for more information.

### When should you use Labels?

We recommend labels in cases where you need a snapshot of source for future reference and or use and where the mutability of labels is not an issue.

## Security

### Can a branch be truly immutable?

Can you lock your house and guarantee that no one can access? The answer is likely to be “no”.

While we can lock-down a branch with permissions, the “truly immutable” depends on your security architecture and enforcement thereof.

### How do I manage permissions across branches for my team?

You should carefully evaluate how you manage access and permissions to source control. Taking the time to evaluate the levels of access that you need across roles and defining a roles and responsibilities matrix can provide you with a consistent and security focused solution. Consider using secure groups, based on Active Directory, since TFS can refresh automatically. Remember to include those people from outside your immediate teams who might also need access, such as IT Ops or Support. You may also need to take into consideration any corporate security or governance policies that apply.

Refer to [Securing Team Foundation Server](#) for more information.

---

<sup>30</sup> [http://msdn.microsoft.com/en-us/library/ms181425\(VS.110\).aspx](http://msdn.microsoft.com/en-us/library/ms181425(VS.110).aspx)

<sup>31</sup> [http://msdn.microsoft.com/en-us/library/ms181428\(VS.110\).aspx](http://msdn.microsoft.com/en-us/library/ms181428(VS.110).aspx)

<sup>32</sup> <http://blogs.msdn.com/b/hakane/archive/2009/05/19/enable-branch-visualization-in-upgraded-team-projects-tfs-2010-beta1.aspx>

# Hands-on Lab (HOL) – From Simple to Complex or not?

## Prerequisites

You require the following to complete this lab:

- The sample solution, documented below
- The latest [Brian Keller VM](#) <sup>33</sup> or an environment with:
  - Visual Studio 2012 Professional or higher
  - Team Foundation Server 2012 or higher

## Our sample solution and context

Our sample solution, which ships together with this HOL on the CodePlex [Version Control \(formerly the Branching and Merging\) Guide](#) <sup>34</sup> site is based on the Managed Extensions Framework (MEF) exploration sample, covered in detail [Managed Extensions Framework \(MEF\) ... simplicity rules](#) <sup>35</sup>.

We start with our favorite calculator solution, which starts with the ability to add and subtract and evolves to include multiplication and division during the course of this lab. The focus is not on MEF, C#, .NET or the exciting world of debugging, but instead on the question “*whether to branch or not to branch*”.

We hope you will enjoy this journey and encourage you to explore the companion guides for details on branching and Team Foundation Version Control features.

## Exercise 1: Environment Setup

### GOAL

In this lab, we assume we start with no pre-configured environment, create a team project, download and check-in our sample solution.

## Task 1: Log on to your environment

### WARNING

Please do **not** use a production system to complete this or other HOLs, as we do not want to create unwanted team projects or branches in production!

Step	Instructions
1 Ligon <input type="checkbox"/> - Done	<ul style="list-style-type: none"> <li>• If you are logging using an instance of BK's VM, i.e. at TechReady login using the <b>administrator</b> P2ssw0rd credentials.</li> <li>• Alternatively, login using your own evaluation environment with credentials that will allow you to create a Team Project.</li> </ul>

Table 6 – Lab 1, Task 1

<sup>33</sup> <http://aka.ms/almvms>

<sup>34</sup> <http://aka.ms/treasure18>

<sup>35</sup> [http://blogs.msdn.com/b/willy-peter\\_schaub/archive/2012/11/23/willy-s-cave-dwelling-notes-10-managed-extensions-framework-mef-simplicity-rules.aspx](http://blogs.msdn.com/b/willy-peter_schaub/archive/2012/11/23/willy-s-cave-dwelling-notes-10-managed-extensions-framework-mef-simplicity-rules.aspx)

## Task 2: Create a Team Project

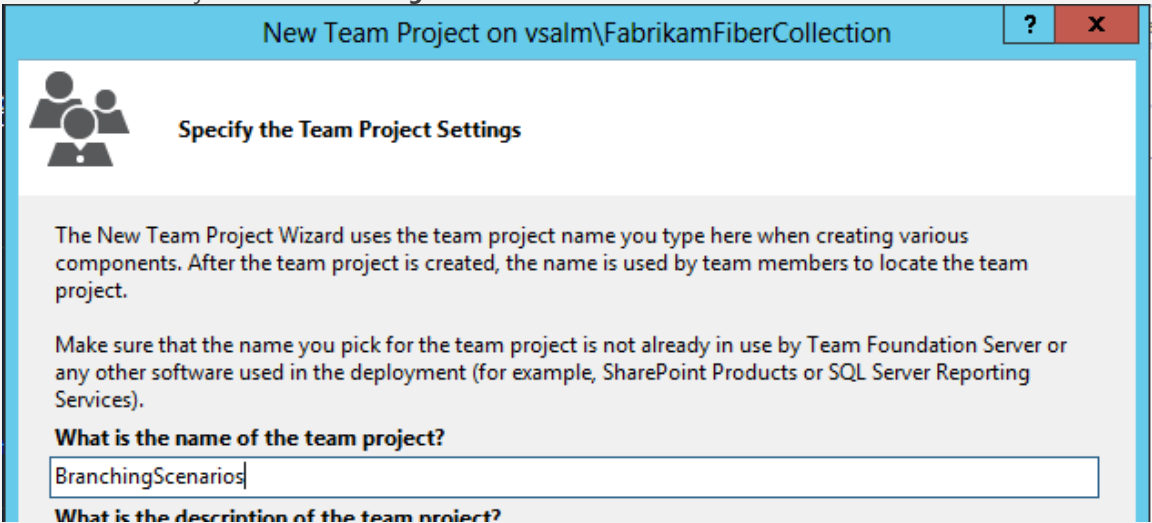
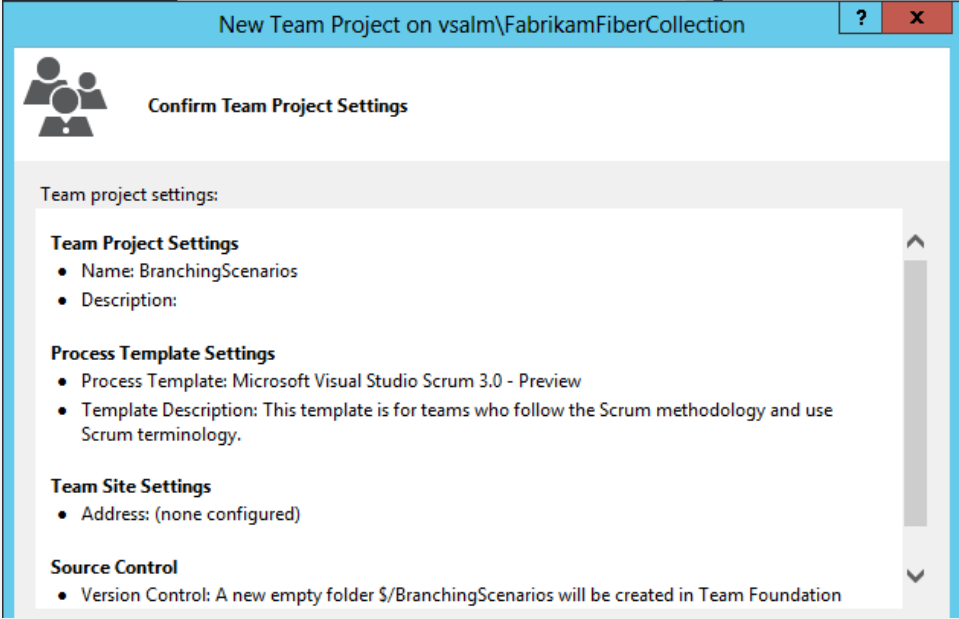
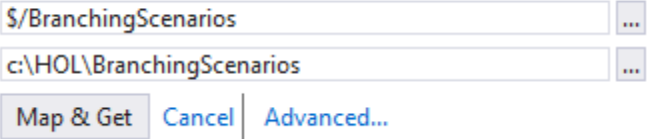
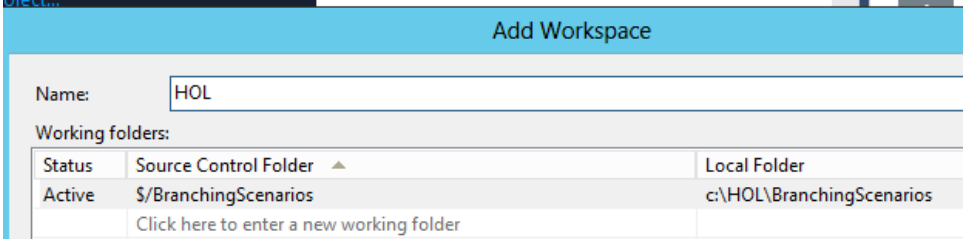
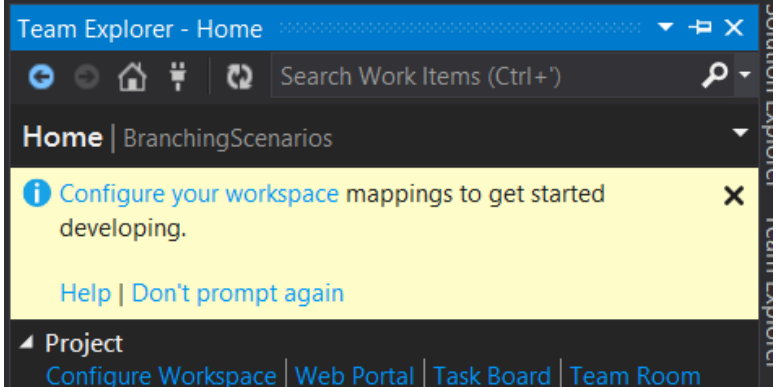
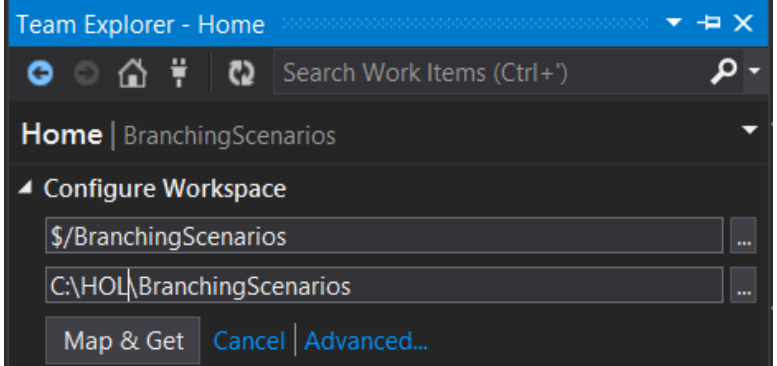
Step	Instructions
1 Start Visual Studio [-] Done	<ul style="list-style-type: none"> <li>Start Visual Studio</li> </ul>
2 Create Team Project (TP) [-] Done	<ul style="list-style-type: none"> <li>Create a Team Project called <b>BranchingScenarios</b></li> </ul>  <ul style="list-style-type: none"> <li>Select the process template of your choice.</li> <li>You do not require a SharePoint portal.</li> <li>Select Team Foundation Version Control for your Source Control.</li> </ul> 

Table 7 – Lab 1, Task 2

## Task 3: Create your sandbox

Step	Instructions
1 Configure workspace mappings ☐ - Done	<p><b>If you are using Visual Studio 2012</b> (see below if using VS 2013)</p> <ul style="list-style-type: none"> <li>Configure your workspace mappings.                     <p><b>Hint:</b> In <b>Source Control Explorer</b>, select <b>Workspaces</b> drop down, <b>Workspaces</b>, your workspace and <b>Edit</b> in the Configure Workspaces dialog.</p> </li> <li>Change the workspace path to <b>c:\HOL\BranchingScenarios</b></li> </ul> <p><b>Configure Workspace</b></p>  <ul style="list-style-type: none"> <li>Select <b>Advanced...</b> and name your workspace <b>HOL</b></li> </ul>  <p><b>Use the process as above, or the new process for Visual Studio 2013</b></p> <ul style="list-style-type: none"> <li>Connect to the new team project called <b>BranchingScenarios</b></li> <li>Select <b>configure your workspace</b> mappings to get started</li> </ul>  <ul style="list-style-type: none"> <li>Map the workspace to the path <b>c:\HOL\BranchingScenarios</b></li> </ul>  <ul style="list-style-type: none"> <li>Select <b>Advanced...</b> and name your workspace <b>HOL</b></li> <li>Select <b>Map &amp; Get</b></li> </ul>
2 Map local path	<ul style="list-style-type: none"> <li>Go to Source Control Explorer</li> </ul>

## Branching Strategies – Hands-on Lab (HOL) – From Simple to Complex or not?

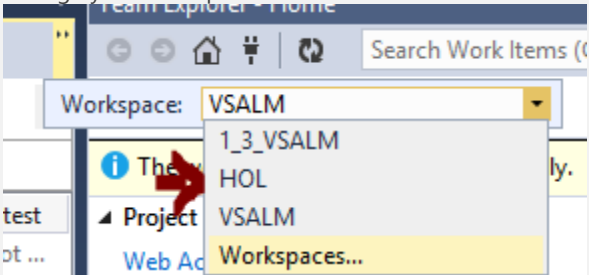
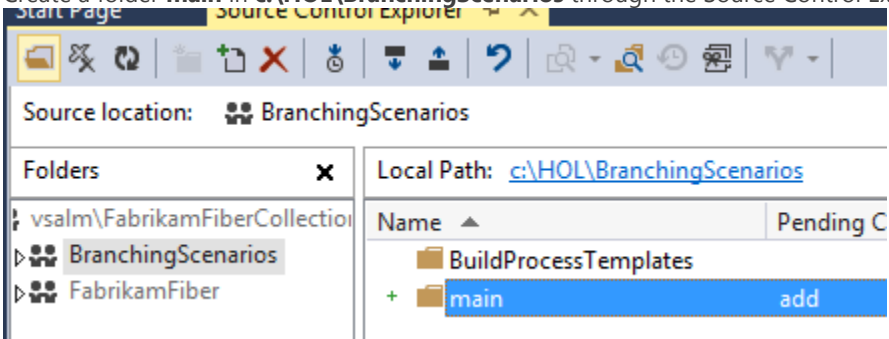
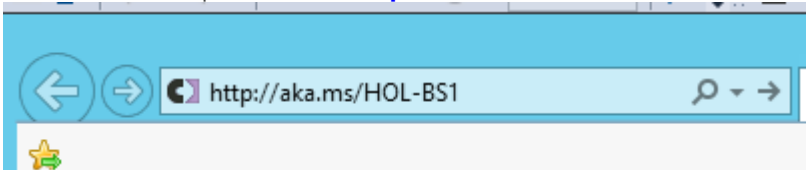
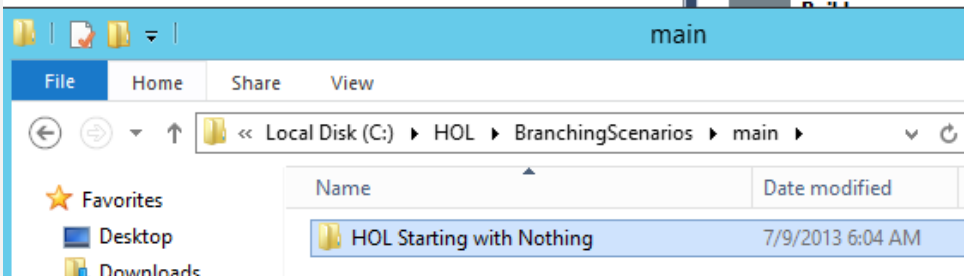
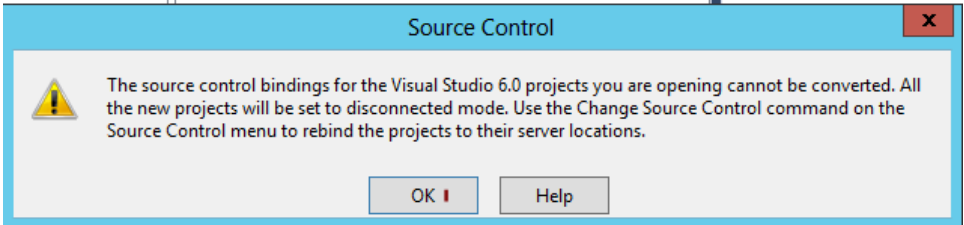
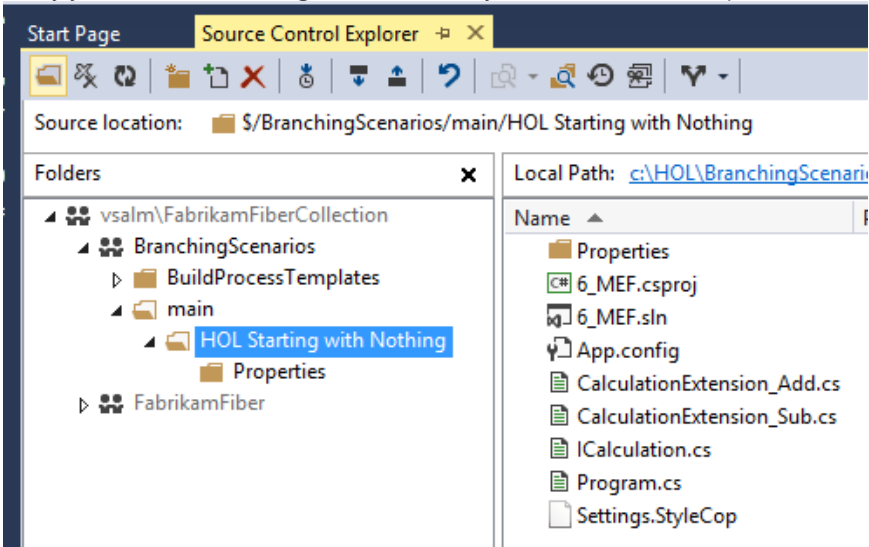
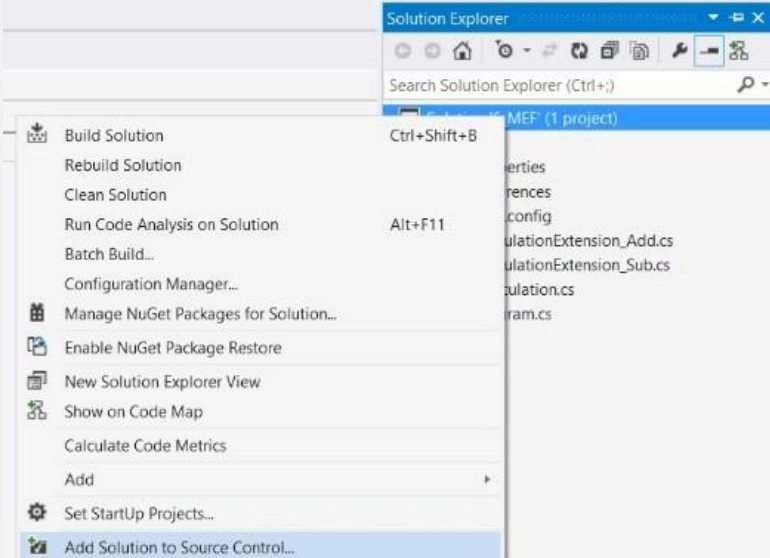
Step	Instructions
<div> <div>☐ - Done</div> </div>	<ul style="list-style-type: none"> <li>Change your workspace to <b>HOL</b></li> </ul> 
<div> <div>3</div> <div>Create main folder</div> <div>☐ - Done</div> </div>	<ul style="list-style-type: none"> <li>Create a folder <b>main</b> in <b>c:\HOL\BranchingScenarios</b> through the Source Control Explorer</li> </ul> 

Table 8 – Lab 1, Task 3

### Task 4: Download and check in sample solution

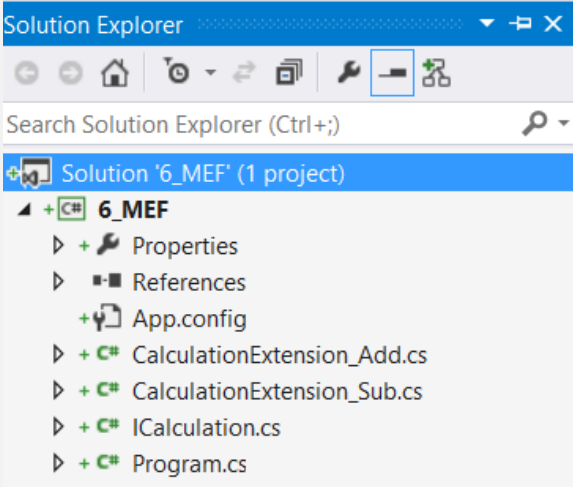

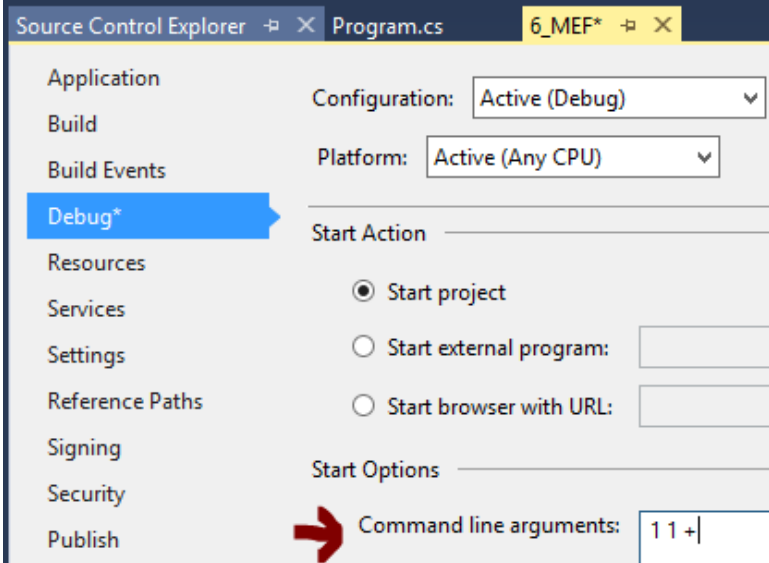
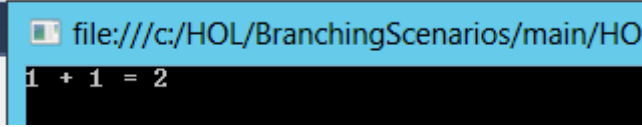

Step	Instructions
<div> <div>1</div> <div>Download Sample Code</div> <div>☐ - Done</div> </div>	<ul style="list-style-type: none"> <li>Download the sample code from <a href="http://aka.ms/hol-bs1">http://aka.ms/hol-bs1</a></li> </ul> 
<div> <div>2</div> <div>Unzip Sample Code</div> <div>☐ - Done</div> </div>	<ul style="list-style-type: none"> <li>Unzip the sample project to the <b>c:\hol\BranchingScenarios\main</b></li> </ul> 
<div> <div>3</div> <div>Explore</div> <div>☐ - Done</div> </div>	<ul style="list-style-type: none"> <li>Open the solution <b>6_MEF</b> in <b>c:\hol\BranchingScenarios\main</b>.</li> <li>Select <b>OK</b> when you get the following Visual Studio 6.0 projects warning</li> </ul>  <ul style="list-style-type: none"> <li>Explore the sample solution.</li> </ul>

## Branching Strategies – Hands-on Lab (HOL) – From Simple to Complex or not?

Step	Instructions
	<ul style="list-style-type: none"> <li>It is a console based sample solution</li> <li>We will use the Debug Command parameter to define test data, i.e. 1 1 + to add 1 and 1.</li> <li>Ensure it builds and runs without any errors</li> <li>Verify you have the following environment in your Source Control Explorer                              </li> </ul>
<p>4</p> <p>Add source to Source Control</p> <p>☐ - Done</p>	<ul style="list-style-type: none"> <li>Right click on the solution in the Solution Explorer, and select <b>Add Solution to Source Control...</b>  </li> </ul>



## Branching Strategies – Hands-on Lab (HOL) – From Simple to Complex or not?

Step	Instructions
	<ul style="list-style-type: none"> <li>Solution, project and associated files are now bound to Source Control.</li> </ul> 
5 Build and Run  - Done	<ul style="list-style-type: none"> <li>Right click on the 6_MEF project in the Solution Explorer and select <b>Properties</b></li> <li>Select <b>Debug</b> tab and define the following command line arguments <b>1 1 +</b></li> </ul>  <ul style="list-style-type: none"> <li>Build and run the program (F5) and verify you see the following result</li> </ul>  <ul style="list-style-type: none"> <li>Save all</li> </ul>
6 Check-in  - Done	<ul style="list-style-type: none"> <li>Right click on the solution in the Solution Explorer, and select <b>Check In</b>.</li> <li>Enter a check-in comment</li> <li>Click <b>Check In</b> button to commit the code to Source Control.</li> </ul>

**Table 9 – Lab 1, Task 4 - Download and check in sample solution**

REVIEW

We have setup our sample environment and are ready to explore the “to branch or not to branch” dilemma.

## Exercise 2: MAIN Only – Simplicity Rules

### GOAL

Explore the **Main Only** branching strategy. We recommend the strategy and the avoidance of branch and associated merge strategy, and are using this strategy for all new ALM Ranger projects.

### Context

Your team needs to implement the DIVIDE feature, has no need for isolation (yet) and is comfortable with using labels to indicate milestones. By using the Main Only strategy you avoid the need for branching and subsequent need for merging, which minimizes maintenance complexity and associated cost.



Figure 15 – Main Only Strategy

### Task 1: Developing on Main

### WARNING

In the real world, you need to implement BVT (Build-Verify-Test) strategies, such as a gated build and unit tests, to protect the quality of the main and other branches. Refer to [Visual Studio Test Tooling Guide](#) <sup>36</sup>, [Team Foundation Build Guidance](#) <sup>37</sup> and [Unit Test Generator](#) <sup>38</sup> for more information on these topics, which are not covered in this walk-through to keep the focus on branching strategies and avoid distractions and delays caused by build and test runs.

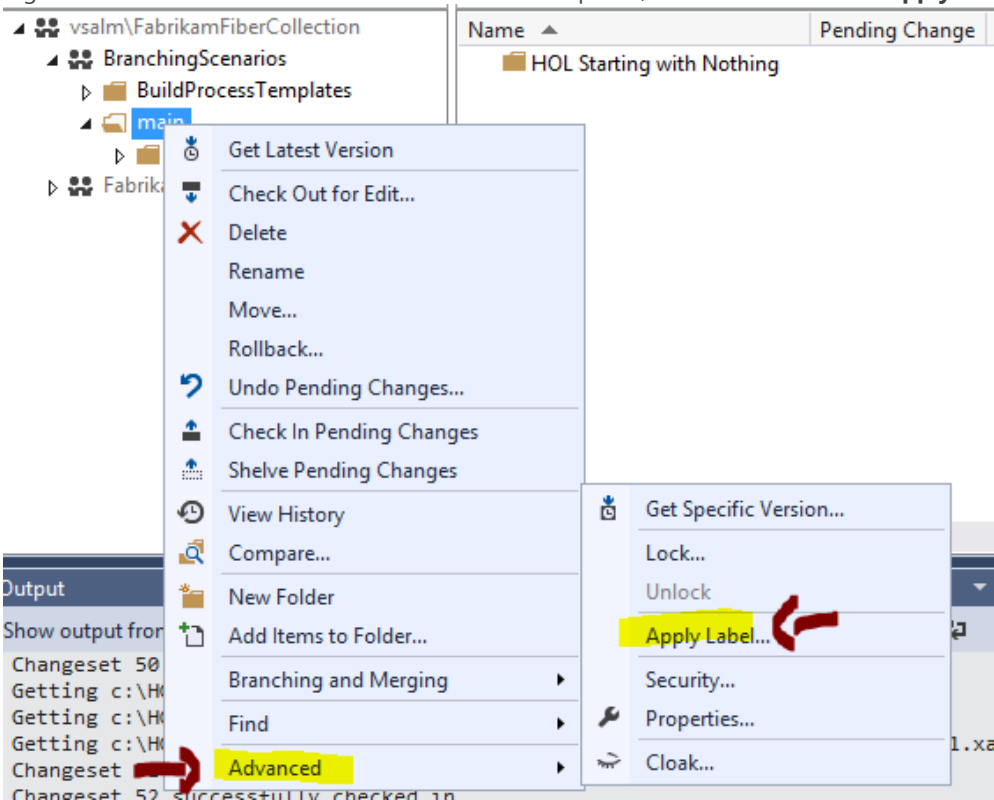
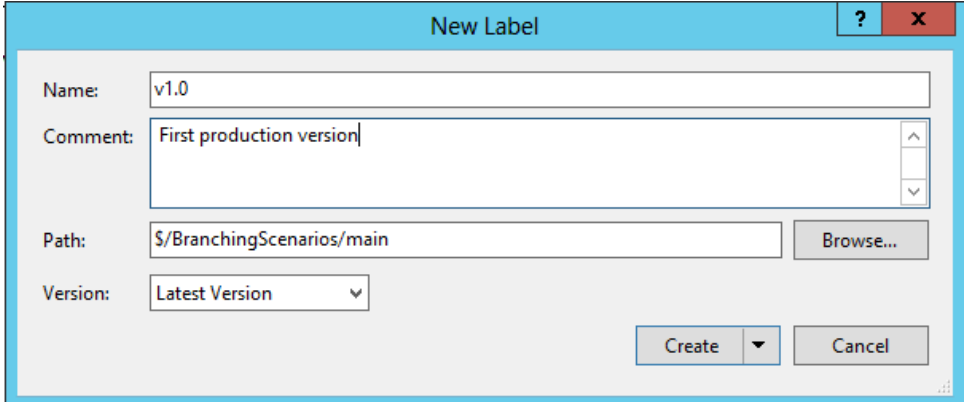
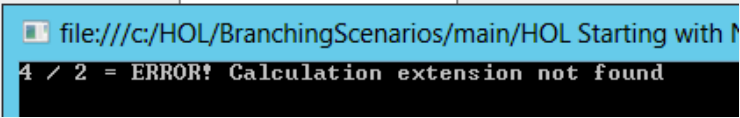
Step	Instructions
1 Label <input type="checkbox"/> - Done	<ul style="list-style-type: none"> <li>To start we label our latest code, we just checked in, as v1.0</li> </ul>

<sup>36</sup> <http://aka.ms/treasure27>

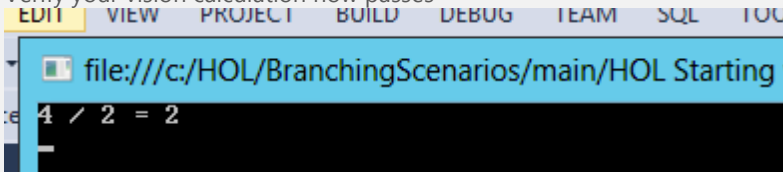
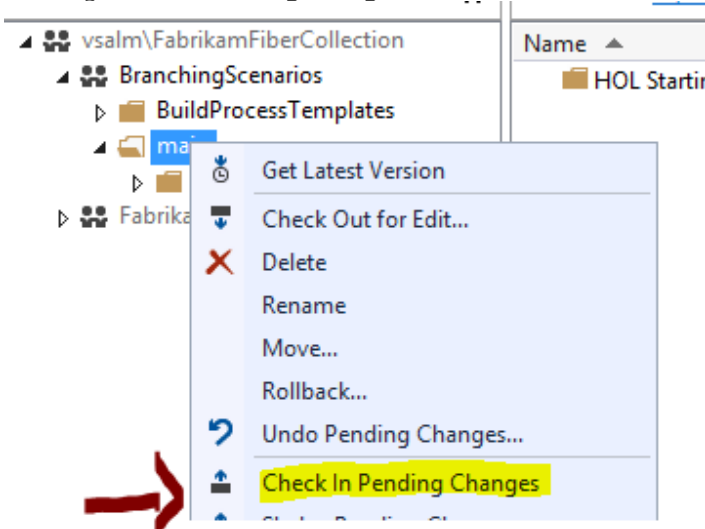
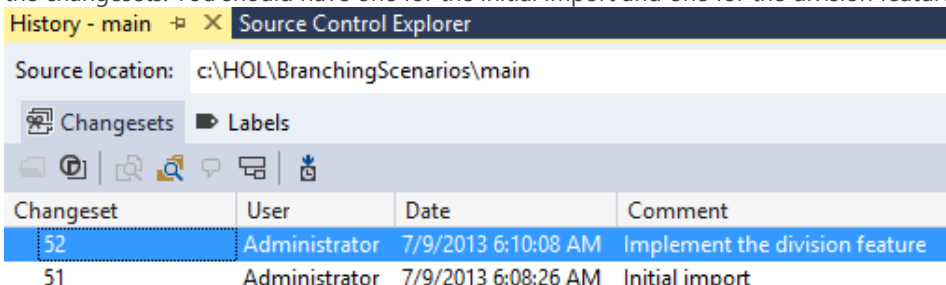
<sup>37</sup> <http://aka.ms/treasure23>

<sup>38</sup> <http://aka.ms/treasure50>

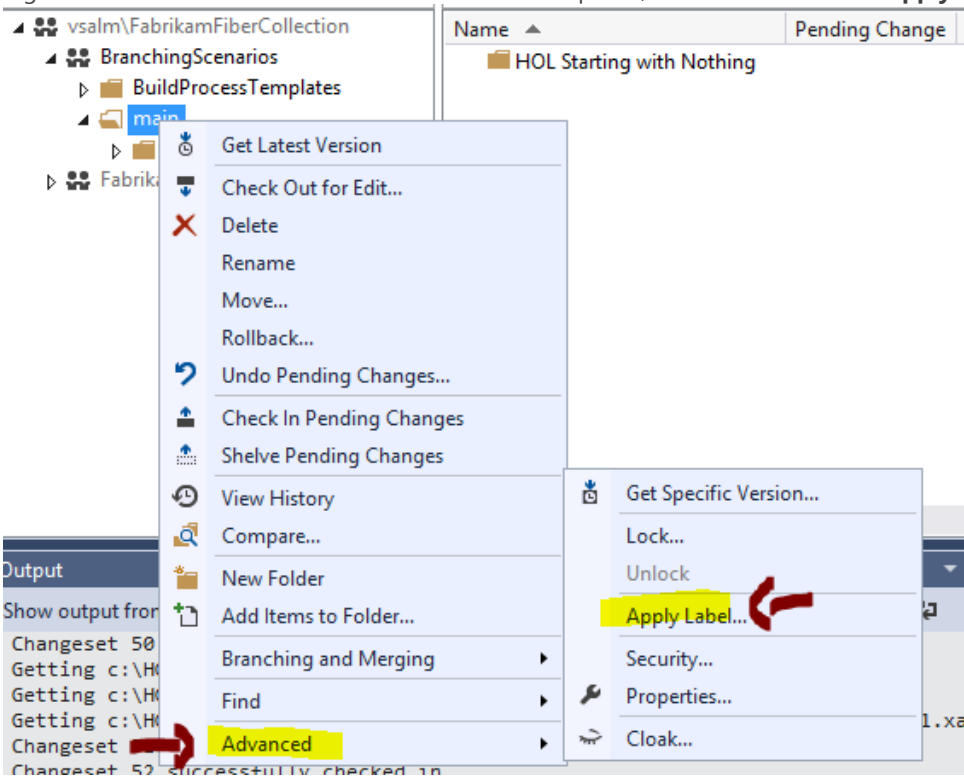
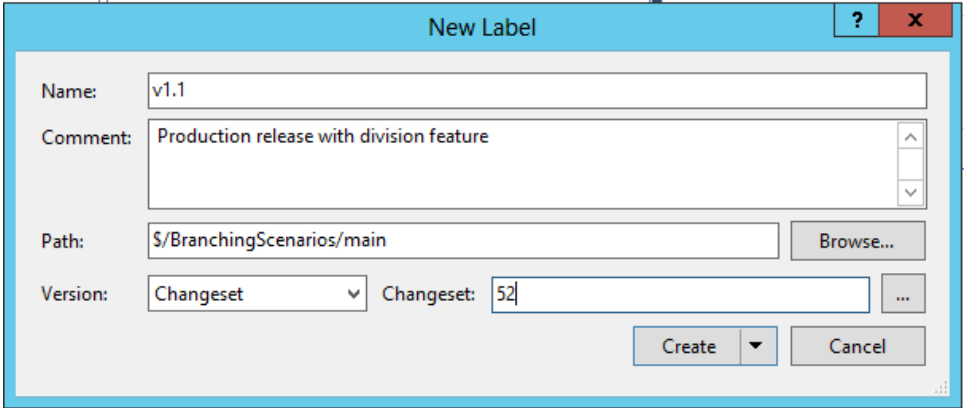
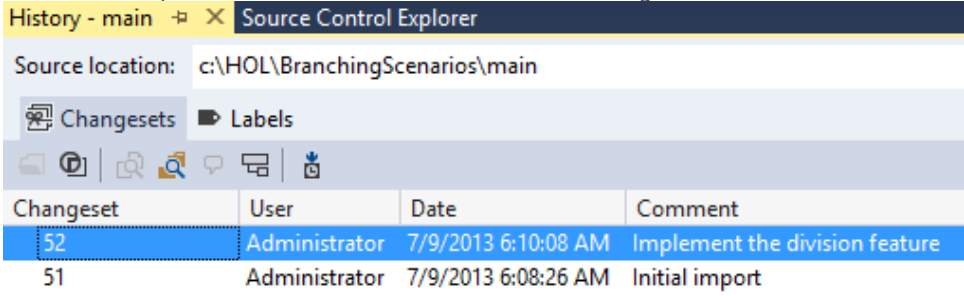
## Branching Strategies – Hands-on Lab (HOL) – From Simple to Complex or not?

Step	Instructions
	<ul style="list-style-type: none"> <li>Right-click on the main folder in the Source Control Explorer, select <b>Advanced</b> and <b>Apply Label</b>  </li> <li>Define a name, comment and select the Latest Version            </li> </ul>
<p>2</p> <p>Run to validate</p> <p>☐ - Done</p>	<ul style="list-style-type: none"> <li>Right click on the 6_MEF project in the Solution Explorer and select <b>Properties</b></li> <li>Select <b>Debug</b> tab and define the following command line arguments <b>4 2 /</b> <div data-bbox="412 1545 781 1650"> <p>Start Options</p> <p>Command line arguments: <input type="text" value="4 2 /"/></p> </div> </li> <li>Build and run the program (F5) and verify that the calculator fails as it lacks this feature            </li> </ul>
<p>3</p> <p>Extend the feature</p>	<ul style="list-style-type: none"> <li>Add class CalculationExtension_Div to the project and the division logic by writing out the class or copy-paste and edit from the *_Add class</li> </ul>

## Branching Strategies – Hands-on Lab (HOL) – From Simple to Complex or not?

Step	Instructions												
<div><div><div></div><div>- Done</div></div></div>	<ul style="list-style-type: none"><li>You must define the ExportMetadata symbol, the class name and the division<pre>[Export(typeof(ICalculation))] [ExportMetadata("Symbol", '/')] 0 references public class CalculationExtension_Div : ICalculation {     4 references     public long Calculation(long valueOne, long valueTwo)     {         return valueOne / valueTwo;     } }</pre></li></ul>												
<div>4</div> <div>Test new feature</div> <div><div><div></div><div>- Done</div></div></div>	<ul style="list-style-type: none"><li>Build and run (F5) the solution</li><li>Verify your vision calculation now passes<div></div></li><li>Feature implemented and tested successfully by developer</li><li>We can now check in the changes without compromising the main folder quality</li></ul>												
<div>5</div> <div>Check in and label</div> <div><div><div></div><div>- Done</div></div></div>	<ul style="list-style-type: none"><li>Check in the code changes by right-clicking on the main folder in the Source Control Explorer and selecting <b>Check In Pending Changes</b><div></div></li></ul> <p>Right-click on the main folder in the Source Control Explorer, select <b>View History</b> and take note of the changesets. You should have one for the initial import and one for the division feature checkin</p> <div><table><tr><th>Changeset</th><th>User</th><th>Date</th><th>Comment</th></tr><tr><td>52</td><td>Administrator</td><td>7/9/2013 6:10:08 AM</td><td>Implement the division feature</td></tr><tr><td>51</td><td>Administrator</td><td>7/9/2013 6:08:26 AM</td><td>Initial import</td></tr></table></div>	Changeset	User	Date	Comment	52	Administrator	7/9/2013 6:10:08 AM	Implement the division feature	51	Administrator	7/9/2013 6:08:26 AM	Initial import
Changeset	User	Date	Comment										
52	Administrator	7/9/2013 6:10:08 AM	Implement the division feature										
51	Administrator	7/9/2013 6:08:26 AM	Initial import										

## Branching Strategies – Hands-on Lab (HOL) – From Simple to Complex or not?

Step	Instructions												
	<ul style="list-style-type: none"> <li>Right-click on the main folder in the Source Control Explorer, select <b>Advanced</b> and <b>Apply Label</b></li> </ul> 												
	<ul style="list-style-type: none"> <li>You can select Latest Changeset as before or define the Changeset number for the check in that included the division feature</li> </ul> 												
	<ul style="list-style-type: none"> <li>Right-click on the main folder in the Source Control Explorer, select <b>View History</b> and then select the <b>Labels</b> tab</li> <li>You now have two labels specifying milestones on our main branch, which you can use for reference, comparisons or checkouts, or the associated changesets as shown</li> </ul>  <table border="1"> <thead> <tr> <th>Changeset</th> <th>User</th> <th>Date</th> <th>Comment</th> </tr> </thead> <tbody> <tr> <td>52</td> <td>Administrator</td> <td>7/9/2013 6:10:08 AM</td> <td>Implement the division feature</td> </tr> <tr> <td>51</td> <td>Administrator</td> <td>7/9/2013 6:08:26 AM</td> <td>Initial import</td> </tr> </tbody> </table>	Changeset	User	Date	Comment	52	Administrator	7/9/2013 6:10:08 AM	Implement the division feature	51	Administrator	7/9/2013 6:08:26 AM	Initial import
Changeset	User	Date	Comment										
52	Administrator	7/9/2013 6:10:08 AM	Implement the division feature										
51	Administrator	7/9/2013 6:08:26 AM	Initial import										

## Branching Strategies – Hands-on Lab (HOL) – From Simple to Complex or not?

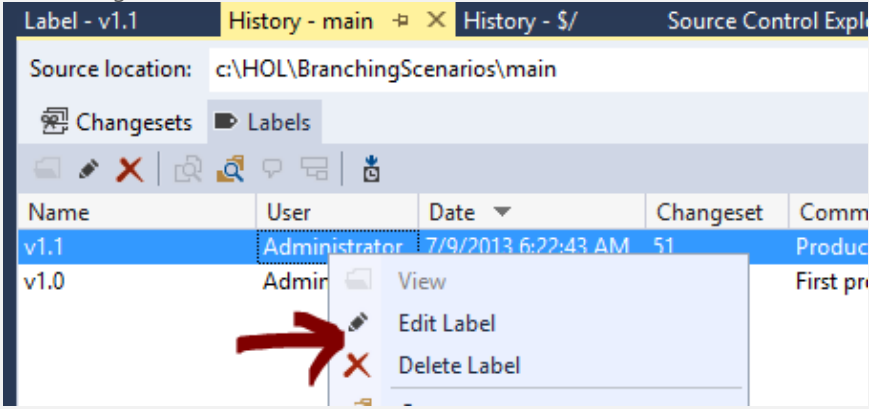
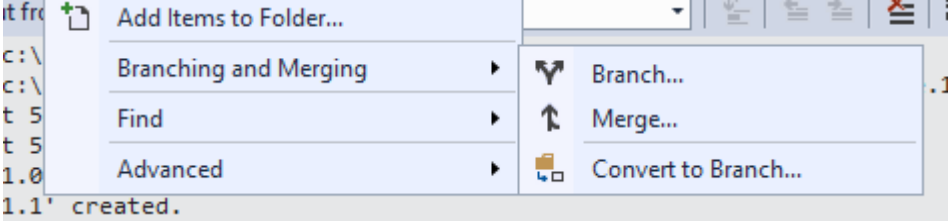
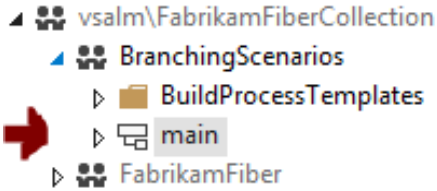
Step	Instructions
6 Spotting the potential Label issue ☐ - Done	<ul style="list-style-type: none"> <li>Right-click on any of the labels</li> <li>Note the edit and delete options for labels, which makes them <b>mutable</b> and therefore not reliable for auditing</li> </ul> 

Table 10 – Lab 2, Task 1

### Task 2: Optionally convert main folder to main branch

Converting the main folder to a branch is an optional step. The branch process automatically performs this when you create branches off main. ALM Rangers frequently use the [TFS Branch Tool](#)<sup>39</sup> utility to implement their version control environment, which automatically performs this step as part of the configuration.

Step	Instructions
1 Check features ☐ - Done before conversion	<ul style="list-style-type: none"> <li>Right-click on the <b>main</b> folder and select <b>Branching and Merging</b></li> </ul> 
2 Convert folder to Branch ☐ - Done	<ul style="list-style-type: none"> <li>Note that we can <b>Branch</b>, <b>Merge</b> and <b>Convert to Branch</b></li> <li>Right-click on the <b>main</b> folder and select <b>Branching and Merging, Convert to Branch</b>.</li> <li>Optionally add a description and then select <b>Convert</b></li> </ul>
3 Check features after conversion ☐ - Done	<ul style="list-style-type: none"> <li>Notice that the folder icon has changed into a branch icon</li> </ul> 

<sup>39</sup> <http://aka.ms/treasure35>

## Branching Strategies – Hands-on Lab (HOL) – From Simple to Complex or not?

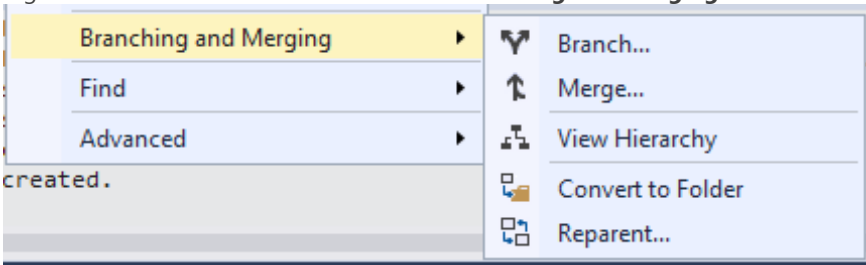
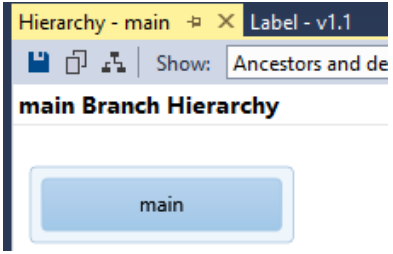
Step	Instructions
	<ul style="list-style-type: none"> <li>Right-click on the <b>main</b> branch and select <b>Branching and Merging</b>  </li> <li>Note that <b>Convert To Branch</b> has vanished, but <b>View Hierarchy</b> and <b>Reparent</b> appeared</li> <li>Select <b>View Hierarchy</b> to view the single (main only) node</li> <li>  </li> </ul>

Table 11 – Lab 2, Task 1

### REVIEW

We explored the Main Only branching strategy and the implications of promoting the main folder to a branch. We worked in only one place, the main folder | branch and used labels to bookmark versioning of the code base.

In this exercise we performed:

- 0 branches
- 0 merges
- 2 labels

## Exercise 3: Development Isolation ... welcome branching

### GOAL

Explore the Development Isolation strategy, which introduces one or more development branches from main, enabling concurrent development of the next release, experimentation or bug fixes in **isolated** development branches.

### Context

Your team needs to maintain a stable main branch for the major release supported in production and invest in concurrent development to add a multiplication feature to the calculator solution.

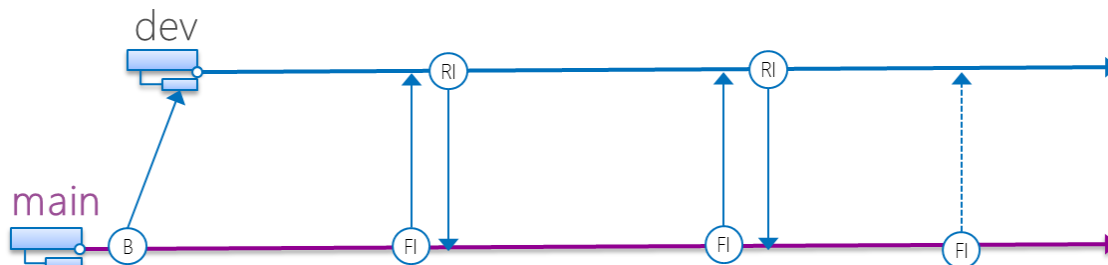
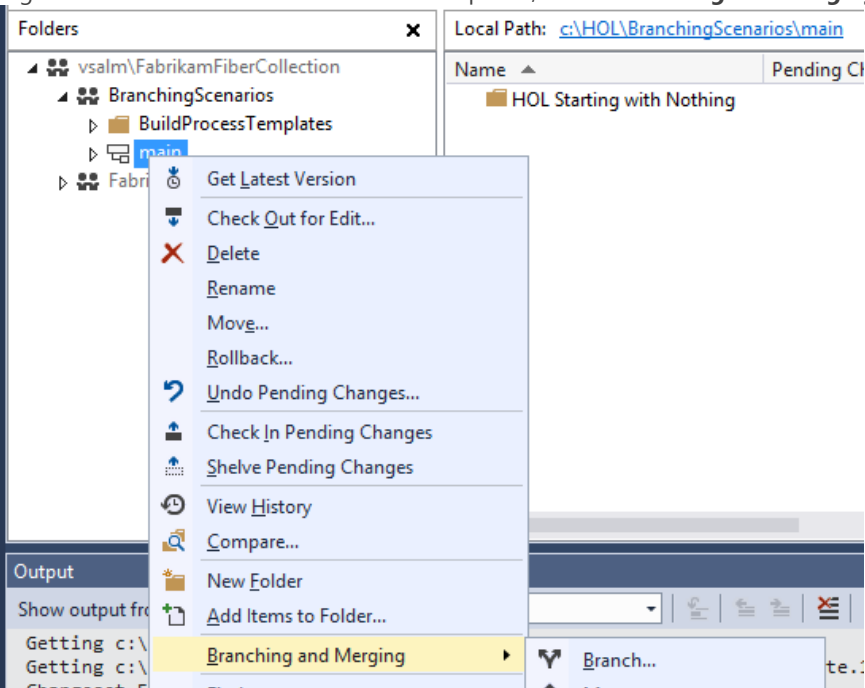
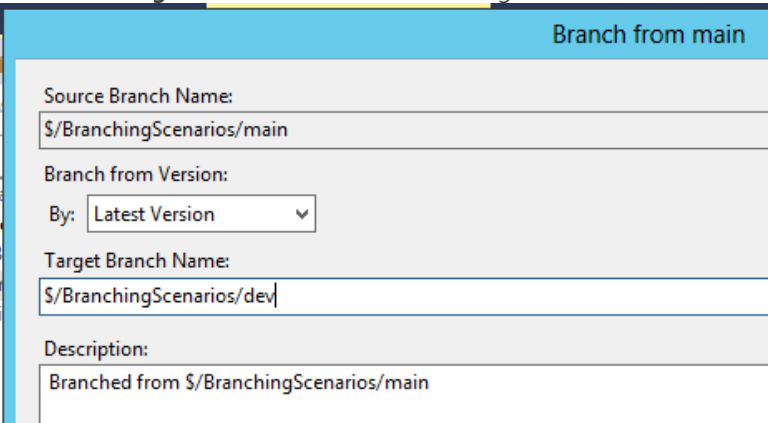
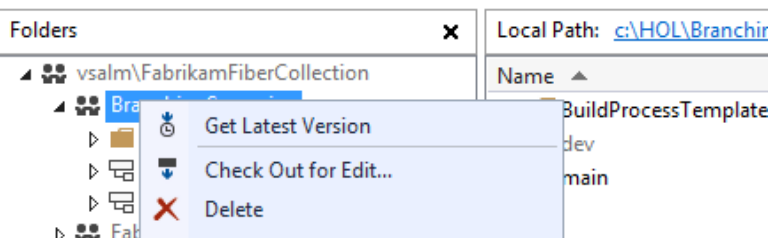



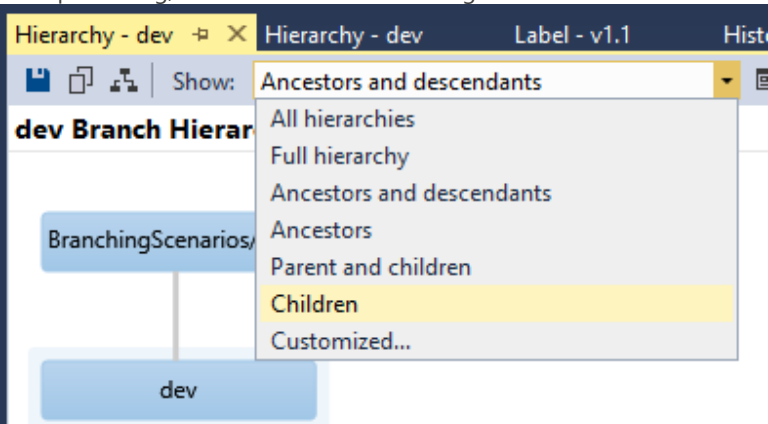
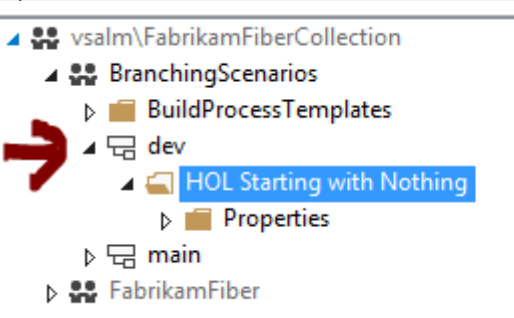
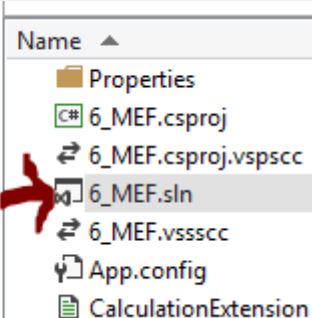
Figure 16 - Development Isolation Strategy

## Task 1: Create the Development Branch to isolate new development

Step	Instructions
1 Create dev branch ☐ - Done	<ul style="list-style-type: none"> <li>Right-click on <b>main</b> in the Source Control Explorer, select <b>Branching and Merging, Branch</b>  </li> <li>Define the <b>Target Branch Name</b> as <code>\$/BranchingScenarios/dev</code>, as shown, and click <b>Branch</b>  </li> <li>You could use any name for the <b>dev</b> branch, i.e. dev, research or development, but strive for consistency to avoid the potential for confusion and error</li> <li>Right-click on <b>BranchingScenarios</b> folder in the Source Control Explorer and select <b>Get Latest Version</b></li> </ul>
	<p><b>NOTE</b> In the real world, you may consider creating a second workspace for the dev branch to isolate it from main.</p> 



## Branching Strategies – Hands-on Lab (HOL) – From Simple to Complex or not?

Step	Instructions
	<ul style="list-style-type: none"> <li>Right-click on <b>dev</b> branch in Source Control Explorer, select <b>Branching and Merging</b> and <b>View Hierarchy</b> to view a visualization of the <b>main</b> and <b>dev</b> branches</li> </ul>  <ul style="list-style-type: none"> <li>Time permitting, view some of the other diagrams</li> </ul> 
<p>2</p> <p>Extend the feature</p> <p>☐ - Done</p>	<ul style="list-style-type: none"> <li>Open the 6_MEF.sln solution in the <b>dev</b> branch</li> </ul>   <ul style="list-style-type: none"> <li>Add class CalculationExtension_Mul and the division logic by writing out the class or copy-paste and edit from the *_Add class</li> <li>You must define the ExportMetadata <b>symbol</b>, the class <b>name</b> and the <b>multiplication</b></li> </ul> <pre data-bbox="406 1533 1242 1816"> [Export(typeof(ICalculation))] [ExportMetadata("Symbol", '*')] public class CalculationExtension_Mul : ICalculation {     public long Calculation(long valueOne, long valueTwo)     {         return valueOne * valueTwo;     } } </pre>
<p>3</p> <p>Test new feature</p>	<ul style="list-style-type: none"> <li>Right click on the 6_MEF project in the Solution Explorer and select <b>Properties</b></li> <li>Select <b>Debug</b> tab and define the following command line arguments: <b>4 2 *</b></li> <li>Build and Run (F5) the solution.</li> </ul>

## Branching Strategies – Hands-on Lab (HOL) – From Simple to Complex or not?

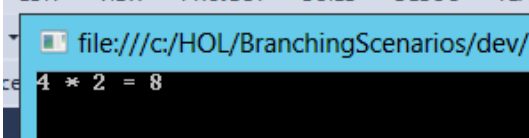
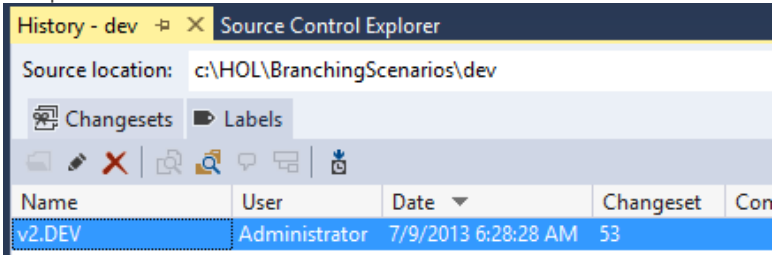
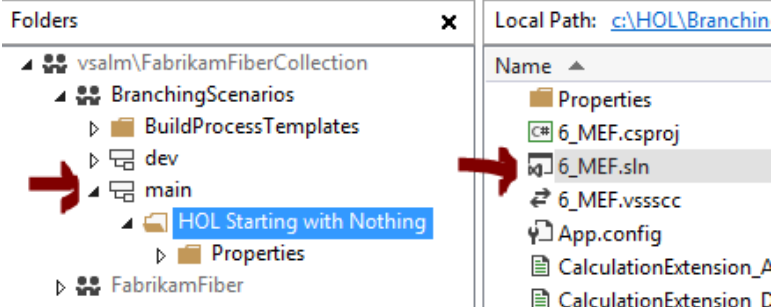
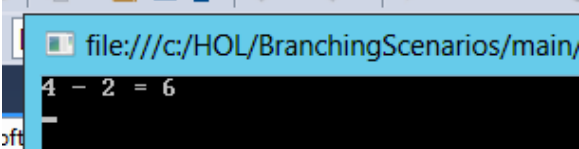
Step	Instructions
☐ - Done	<ul style="list-style-type: none"> <li>Verify that your multiplication feature works as shown  </li> <li>Check in your changes to the <b>dev</b> branch</li> <li>Add a label v2.DEV to this latest feature check-in as a bookmark to indicate that feature testing is complete.  </li> <li>We are feature complete and ready to integrate <b>dev</b> changes to the <b>main</b> branch</li> </ul>

Table 12 – Lab 3, Task 1

### Task 2: Fix a bug in Main as a hotfix

Before we can merge and release the new features, however, we have to investigate and fix a production bug, which has a higher priority.

Step	Instructions
1 Investigate the bug ☐ - Done	<ul style="list-style-type: none"> <li>The users have reported that the subtraction feature in our v1.1 release is broken</li> <li>Switch context to the <b>main</b> branch, which contains the production release</li> <li>Open the 6_MEF.sln solution in the <b>dev</b> branch</li> </ul>  <ul style="list-style-type: none"> <li>Right click on the 6_MEF project in the Solution Explorer and select <b>Properties</b>.</li> <li>Select <b>Debug</b> tab and define the following command line arguments: <b>4 2 -</b> <div>           Command line arguments: <input type="text" value="4 2 -"/> </div> </li> <li>Build and Run (F5) the solution</li> <li>Notice that the user is correct, the subtraction seems to be adding the values  </li> </ul>

## Branching Strategies – Hands-on Lab (HOL) – From Simple to Complex or not?

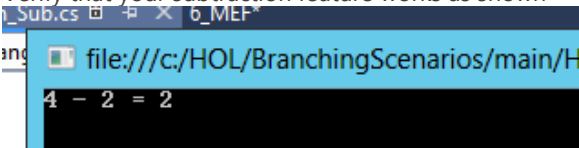
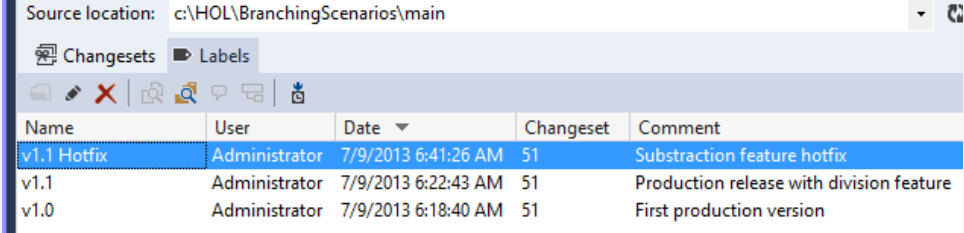
Step	Instructions
2 Fix the bug <input type="checkbox"/> - Done	<ul style="list-style-type: none"> <li>Edit the ...<b>main</b>\CalculationExtension_Sub.cs file and notice the gremlin  <pre>public class CalculationExtension_Sub : ICalculation {     4 references   Administrator   1 change     public long Calculation(long valueOne, long valueTwo)     {         // Intentional bug for HOL should be -         return valueOne + valueTwo;     } }</pre> </li> <li>Change the + to a - to correct the bug and optionally remove the comment  <pre>public class CalculationExtension_Sub : ICalculation {     4 references   Administrator   1 change     public long Calculation(long valueOne, long valueTwo)     {         return valueOne - valueTwo;     } }</pre> </li> </ul>
3 Validate and test <input type="checkbox"/> - Done	<ul style="list-style-type: none"> <li>Build and Run (F5) the solution</li> <li>Verify that your subtraction feature works as shown   </li> </ul>
4 Check-in, label and ship v1.1 Hotfix <input type="checkbox"/> - Done	<ul style="list-style-type: none"> <li>Check-in the changes in the <b>main</b> branch</li> <li>Label the latest changes in the <b>main</b> branch as v1.1 hotfix   </li> </ul>

Table 13 – Lab 3, Task 2

### Context – Intermission

As shown below we have a stable v1.1 Hotfix release in main and a stable v2.DEV release in the dev branch at this point ...

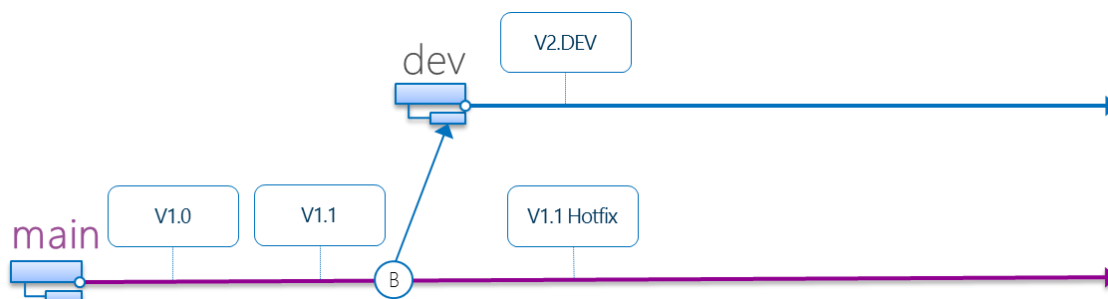
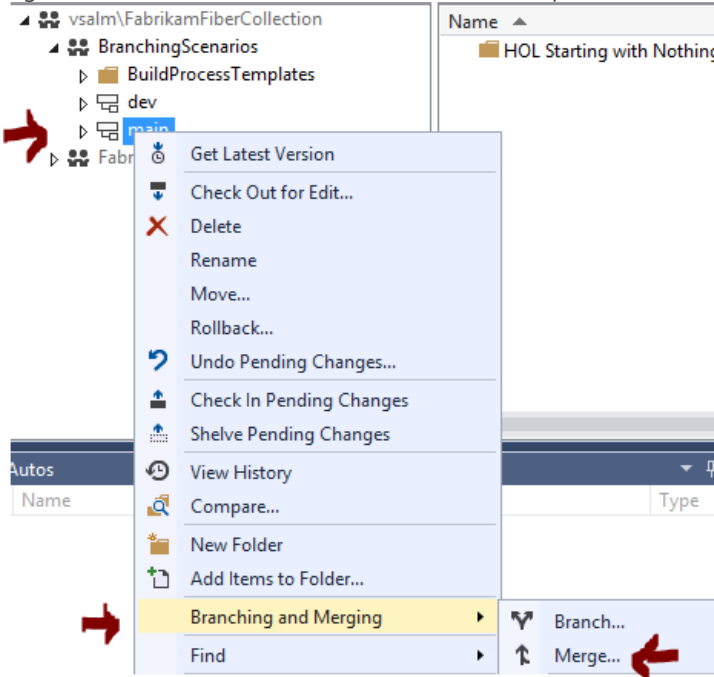
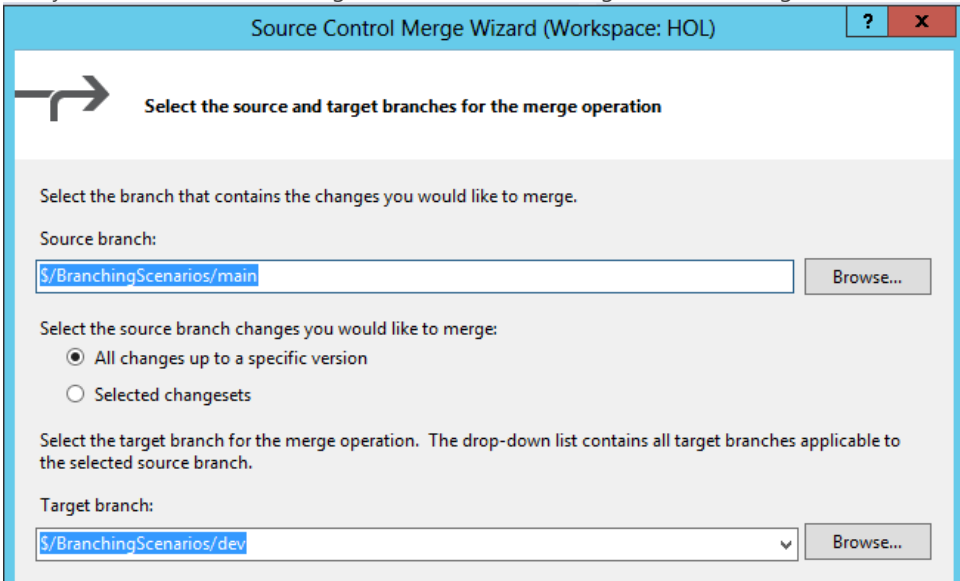


Figure 17 - Development Isolation Strategy Exercise Review

## Branching Strategies – Hands-on Lab (HOL) – From Simple to Complex or not?

What we need to do is share the hotfix with the dev team and then merge all the changes back to our main branch so that we can offer the latest and greatest release, which include + - \* / features with our customers.

### Task 3: Merge all changes and ship a new major release

Step	Instructions
1 Merge hotfix to dev branch ☐ - Done	<ul style="list-style-type: none"><li>Right-click on the <b>main</b> branch in Source Control Explorer, select <b>Branching and Merging</b>, <b>Merge</b> </li><li>Verify that source is <code>\$/BranchingScenarios/main</code> and target is <code>\$/BranchingScenarios/dev</code>  Select <b>Next</b>, set Version type to <b>Latest Version</b> and select <b>Next</b></li></ul>

## Branching Strategies – Hands-on Lab (HOL) – From Simple to Complex or not?

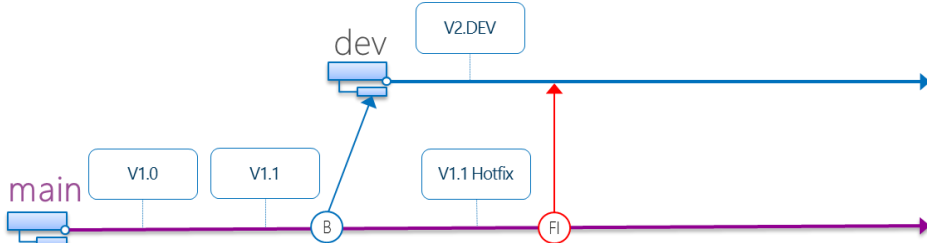
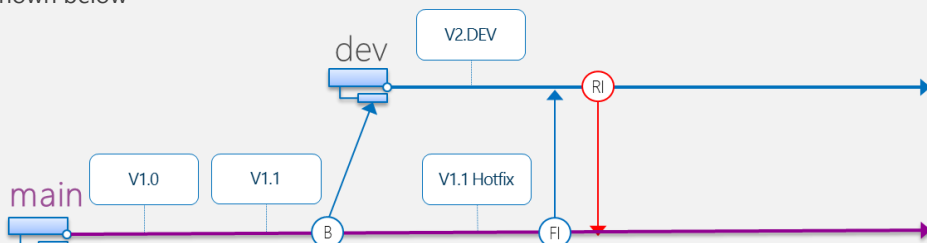
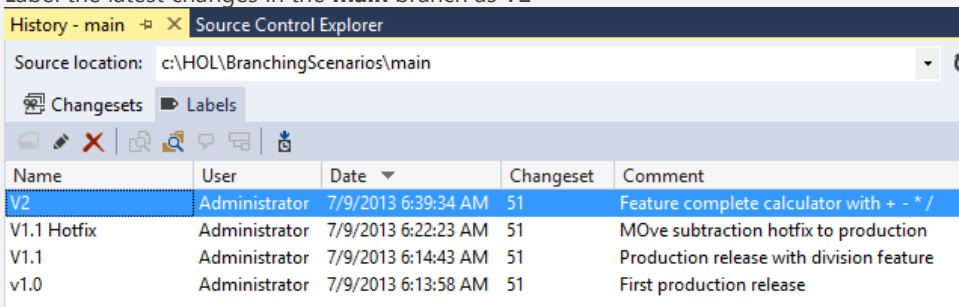
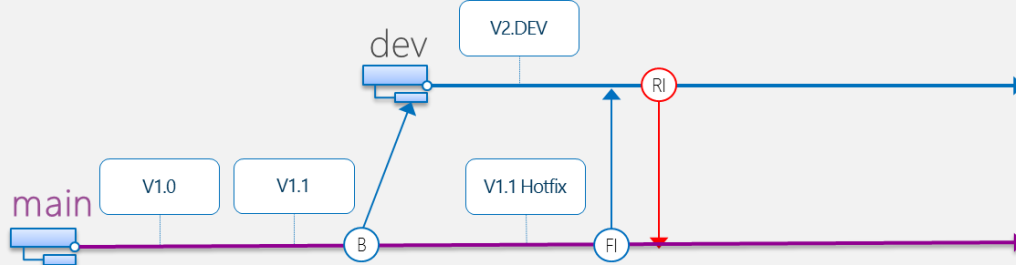
Step	Instructions																									
	<ul style="list-style-type: none"><li>Read the merge operation summary and select <b>Finish</b> to perform a forward integration (FI) merge as shown below:</li></ul> <div></div> <ul style="list-style-type: none"><li>You should experience no merge conflicts as we have not made any conflicting changes</li><li>Check in your changes to the <b>dev</b> branch, comprised of CalculationExtension_Sub</li><li>For safety, re-test the division feature and the subtraction feature</li></ul>																									
2 Merge dev branch to main branch ☐ - Done	<ul style="list-style-type: none"><li>Right-click on the <b>dev</b> branch in Source Control Explorer, select <b>Branching and Merging</b>, and <b>Merge</b>.</li><li>Verify that source is <code>\$/BranchingScenarios/dev</code> and target is <code>\$/BranchingScenario/main</code></li><li>Select <b>Next</b>, set Version type to <b>Latest Version</b> and select <b>Next</b></li><li>Read the merge operation summary and select <b>Finish</b> to perform a reverse integration (RI) merge as shown below</li></ul> <div></div> <ul style="list-style-type: none"><li>Right-click on the <b>main</b> branch in Source Control Explorer and select <b>Check In Pending Changes</b>, which includes 6_MEF.csproj project file (merge, edit) and CalculationExtension_Mul.cs (merge, branch)</li></ul>																									
3 Validate and test ☐ - Done	<ul style="list-style-type: none"><li>Open the solution in the main branch and by setting the command line arguments as before, verify the following calculations:<ul style="list-style-type: none"><li>Test 4 2 +</li><li>Test 4 2 -</li><li>Test 4 2 *</li><li>Test 4 2 /</li></ul></li></ul>																									
4 Label and ship v2.0 ☐ - Done	<ul style="list-style-type: none"><li>Label the latest changes in the <b>main</b> branch as V2</li></ul> <div></div> <table><thead><tr><th>Name</th><th>User</th><th>Date</th><th>Changeset</th><th>Comment</th></tr></thead><tbody><tr><td>V2</td><td>Administrator</td><td>7/9/2013 6:39:34 AM</td><td>51</td><td>Feature complete calculator with + - * /</td></tr><tr><td>V1.1 Hotfix</td><td>Administrator</td><td>7/9/2013 6:22:23 AM</td><td>51</td><td>MOve subtraction hotfix to production</td></tr><tr><td>V1.1</td><td>Administrator</td><td>7/9/2013 6:14:43 AM</td><td>51</td><td>Production release with division feature</td></tr><tr><td>v1.0</td><td>Administrator</td><td>7/9/2013 6:13:58 AM</td><td>51</td><td>First production release</td></tr></tbody></table>	Name	User	Date	Changeset	Comment	V2	Administrator	7/9/2013 6:39:34 AM	51	Feature complete calculator with + - * /	V1.1 Hotfix	Administrator	7/9/2013 6:22:23 AM	51	MOve subtraction hotfix to production	V1.1	Administrator	7/9/2013 6:14:43 AM	51	Production release with division feature	v1.0	Administrator	7/9/2013 6:13:58 AM	51	First production release
Name	User	Date	Changeset	Comment																						
V2	Administrator	7/9/2013 6:39:34 AM	51	Feature complete calculator with + - * /																						
V1.1 Hotfix	Administrator	7/9/2013 6:22:23 AM	51	MOve subtraction hotfix to production																						
V1.1	Administrator	7/9/2013 6:14:43 AM	51	Production release with division feature																						
v1.0	Administrator	7/9/2013 6:13:58 AM	51	First production release																						

Table 14 – Lab 3, Task 3

## Branching Strategies – Hands-on Lab (HOL) – From Simple to Complex or not?

### REVIEW

We explored the development isolation strategy. We demonstrated how to implement this strategy. Finally, we explained why you should use this strategy. We explored how to work on a hotfix and development feature concurrently, merging the innovations back into the stable main branch at the end of the exercise.



In this exercise we performed:

- 1 branch
- 2 merges, one FI and one RI
- 3 labels

## Exercise 4: Feature Isolation ... a special!

### GOAL

Explore the Feature Isolation strategy, which introduces one or more feature branches from main, enabling concurrent development on clearly defined features, which you can merge and release as needed.

### Context

As before, you need isolation, but you have a need to develop clearly defined features Mod and Power, with flexibility in terms of which when you release these features into production. Whether you implement another development branch or named feature branches is your choice, whereby we will implement the following hybrid-model for this exercise as shown below:

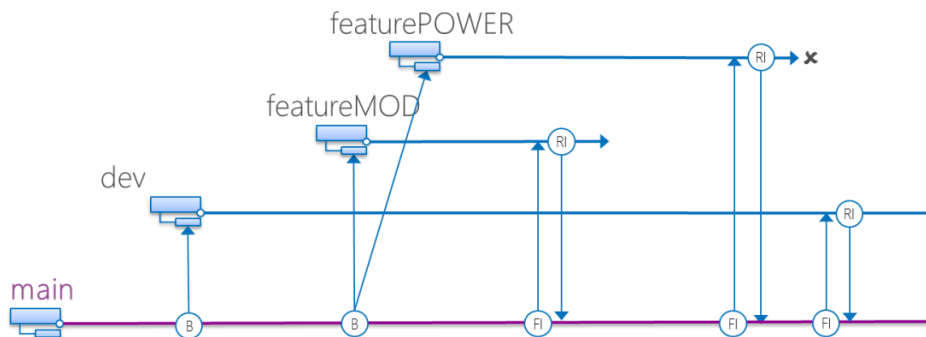


Figure 18 - Hybrid Development and Feature Isolation Strategy Example 1

## Branching Strategies – Hands-on Lab (HOL) – From Simple to Complex or not?

A different strategy could be to branch off the dev branch tying the features and their implementation to the dev branch instead. This strategy would use the development branch as an integration branch, merging all features to the parent.

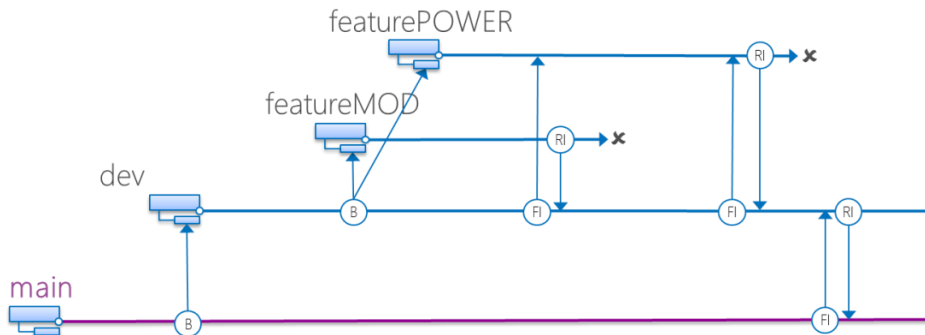


Figure 19 - Hybrid Development and Feature Isolation Strategy Example 2

### Task 1: Create Feature Branches

Step	Instructions
1 Create POWER feature branch ☐ - Done	<ul style="list-style-type: none"> <li>Right-click on <b>main</b> branch in Source Control Explorer, select <b>Branching and Merging, Branch</b></li> <li>Define target as <code>\$/BranchingScenarios/featurePOWER</code></li> </ul> <div style="border: 1px solid #0070c0; padding: 10px; margin: 10px 0;"> <p style="text-align: right; margin: 0;"><b>Branch from main</b></p> <p>Source Branch Name: <code>\$/BranchingScenarios/main</code></p> <p>Branch from Version: By: <span>Latest Version</span></p> <p>Target Branch Name: <code>\$/BranchingScenarios/featurePOWER</code> <span style="color: red; font-size: 2em;">➔</span></p> <p>Description: Branched from <code>\$/BranchingScenarios/main</code></p> </div> <div style="background-color: #ffcc00; padding: 5px; margin: 10px 0; display: flex; align-items: center;"> <div style="writing-mode: vertical-rl; transform: rotate(180deg); font-weight: bold; margin-right: 5px;">NOTE</div> <div>Consider concept of folders for organizing branches when working with feature branches, for example <code>\$/BranchingScenarios/features/featurePOWER</code></div> </div> <ul style="list-style-type: none"> <li>Select <b>Branch</b> and confirm "Continue to branch?" dialog with <b>Yes</b>.</li> </ul>
2 Create MOD feature branch ☐ - Done	<ul style="list-style-type: none"> <li>Right-click on <b>main</b> branch in Source Control Explorer, select <b>Branching and Merging, Branch</b></li> <li>Define target as <code>\$/BranchingScenarios/featureMOD</code> or <code>\$/BranchingScenarios/features/featureMOD</code> if you wish to organize your feature branches further.</li> <li>Select <b>Branch</b> and confirm "Continue to branch?" dialog with <b>Yes</b>.</li> </ul>
3 Get latest and view hierarchy ☐ - Done	<ul style="list-style-type: none"> <li>Right-click on <b>BranchingScenarios</b> folder in Source Control Explorer, select <b>Get Latest Version</b></li> </ul>

## Branching Strategies – Hands-on Lab (HOL) – From Simple to Complex or not?

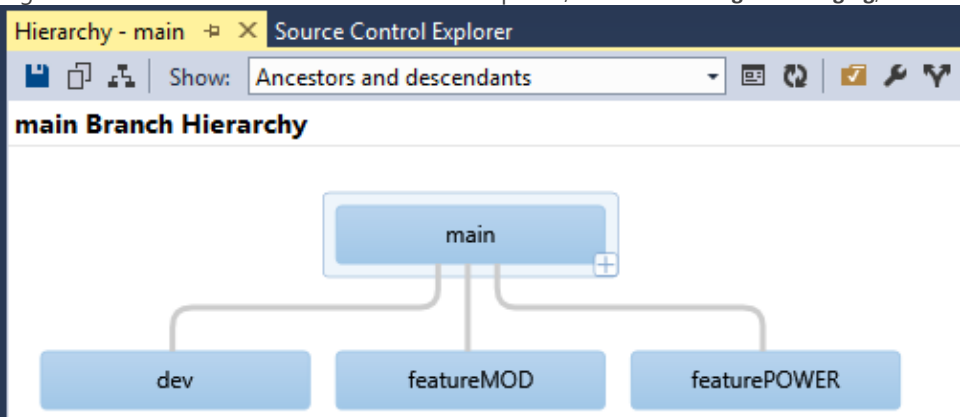
Step	Instructions
	<ul style="list-style-type: none"> <li>Right-click on <b>main</b> branch in Source Control Explorer, select <b>Branching and Merging, View Hierarchy</b></li> </ul>  <pre> graph TD     main[main] --&gt; dev[dev]     main --&gt; featureMOD[featureMOD]     main --&gt; featurePOWER[featurePOWER]     </pre> <ul style="list-style-type: none"> <li>Our hierarchy now matches Figure 18 - Hybrid Development and Feature Isolation Strategy Example 1, on page 54</li> </ul>

Table 15 – Lab 4, Task 1

### Task 2: Create MOD Feature

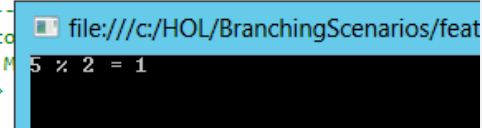
Step	Instructions
1 Implement feature <input type="checkbox"/> - Done	<ul style="list-style-type: none"> <li>Open the 6_MEF.sln solution in the <b>featureMOD</b> branch</li> <li>Add class CalculationExtension_Mod and the Mod logic by writing out the class or copy-paste and edit from the *_Add class</li> <li>You must define the ExportMetadata <b>symbol</b>, the class <b>name</b> and the <b>Mod</b> <pre> [Export(typeof(ICalculation))] [ExportMetadata("Symbol", "%")] 0 references public class CalculationExtension_Mod : ICalculation {     /// &lt;summary&gt; ...     6 references     public long Calculation(long valueOne, long valueTwo)     {         return valueOne % valueTwo;     } } </pre> </li> </ul>
2 Test feature <input type="checkbox"/> - Done	<ul style="list-style-type: none"> <li>Right click on the 6_MEF project in the Solution Explorer and select <b>Properties</b></li> <li>Select <b>Debug</b> tab and define the following command line arguments: <b>5 2 %</b></li> <li>Build and Run (F5) the solution.</li> <li>Verify that your mod feature works as shown</li> </ul>  <pre> file:///c:/HOL/BranchingScenarios/feat 5 x 2 = 1 </pre>
3 Check-in <input type="checkbox"/> - Done	<ul style="list-style-type: none"> <li>Check in your changes to the <b>featureMOD</b> branch</li> </ul>

Table 16 – Lab 4, Task 2



# Task 3: Create POWER Feature

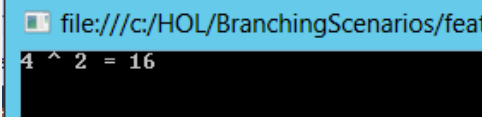
Step	Instructions
1 Implement feature <input type="checkbox"/> - Done	<ul style="list-style-type: none"> <li>Open the 6_MEF.sln solution in the <b>featurePOWER</b> branch</li> <li>Add class CalculationExtension_Pow and the Power logic by writing out the class or copy-paste and edit from the *_Add class</li> <li>You must define the ExportMetadata <b>symbol</b>, the class <b>name</b> and the <b>Power</b> feature <pre> [Export(typeof(ICalculation))] [ExportMetadata("Symbol", '^')] 0 references public class CalculationExtension_Pow : ICalculation {     /// &lt;summary&gt; ...     6 references     public long Calculation(long valueOne, long valueTwo)     {         long valueResult = 0;         if ( 0 != valueTwo)         {             valueResult = valueOne;             for ( long i = --valueTwo; 0 != valueTwo; valueTwo-- )             {                 valueResult *= valueResult;             }         }         return valueResult;     } } </pre> </li> </ul>
2 Test feature <input type="checkbox"/> - Done	<ul style="list-style-type: none"> <li>The code above is not the most optimal and you are welcome to improve the sample ☺</li> <li>Right click on the 6_MEF project in the Solution Explorer and select <b>Properties</b></li> <li>Select <b>Debug</b> tab and define the following command line arguments: <b>4 2 ^</b></li> <li>Build and Run (F5) the solution.</li> <li>Verify that your power feature works as shown  </li> </ul>
3 Check-in <input type="checkbox"/> - Done	<ul style="list-style-type: none"> <li>Check in your changes to the <b>featurePOWER</b> branch</li> </ul>

Table 17 – Lab 4, Task 3

## Context – Intermission

As shown below, we have a stable **main**, a **development** branch with unknown activity and the two new features all ready to go, but isolated.

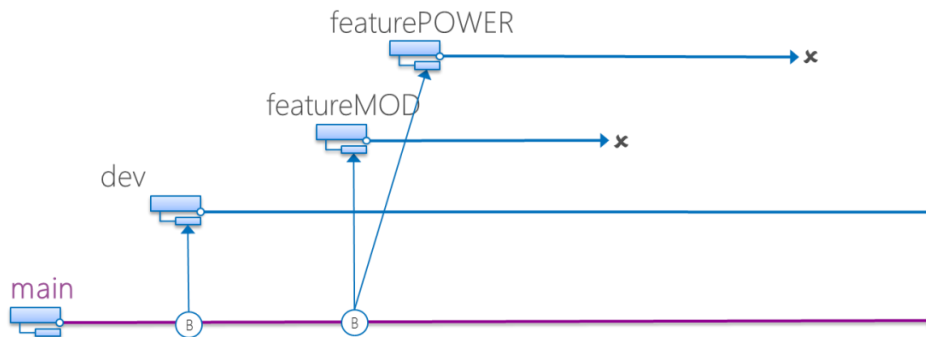
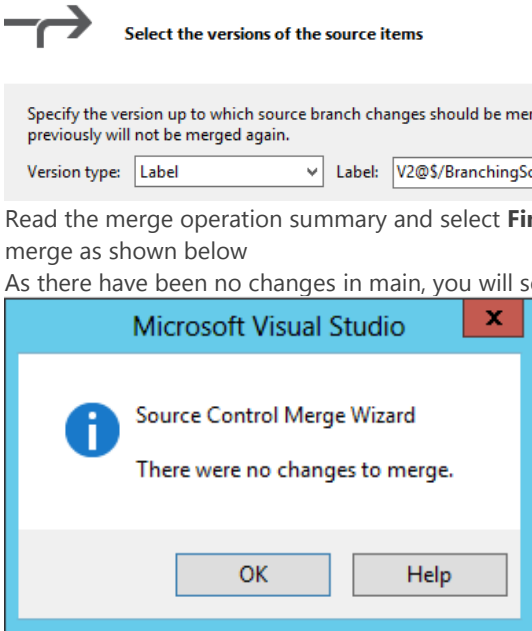


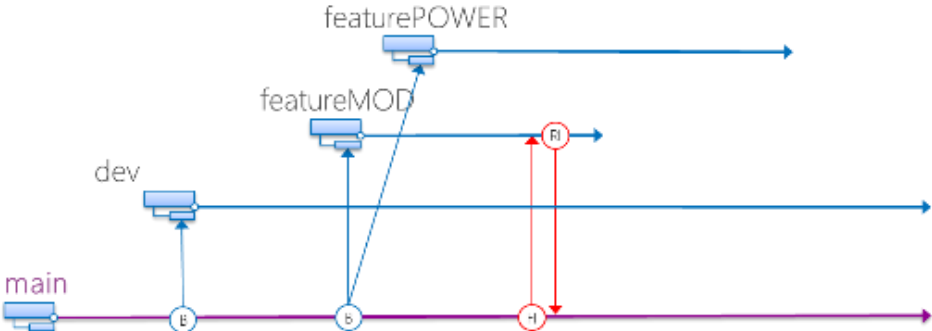
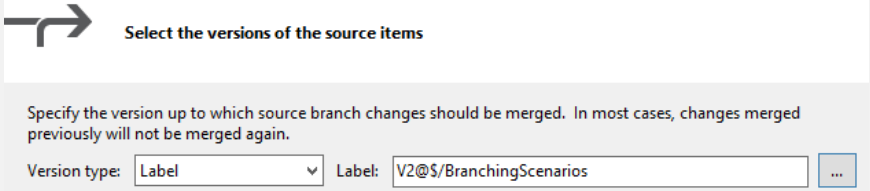
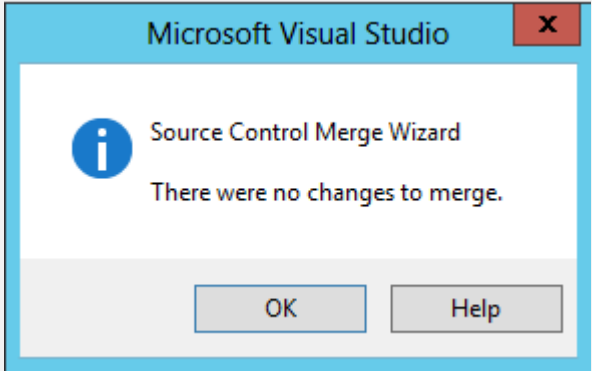
Figure 20 - Feature Isolation Strategy Exercise Review

What we need to do is ensure the feature branches are up to date with any activity that may have occurred during their development phase and then merge the changes back to main, validate and ship v3.

## Task 4: Merge all changes and ship a new major release

Step	Instructions
1 Merge featureMOD <input type="checkbox"/> - Done	<ul style="list-style-type: none"> <li>Forward-Integration (FI) merge                             <ul style="list-style-type: none"> <li>Right-click on the <b>main</b> branch in Source Control Explorer, select <b>Branching and Merging</b>, and <b>Merge</b>.</li> <li>Verify that source is <code>\$/BranchingScenarios/main</code> and target is <code>\$/BranchingScenario/featureMOD</code></li> <li>Select <b>Next</b>, set Version type to <b>All changes up to a specific</b> and select <b>Next</b></li> <li>Select <b>Label</b>, find the <b>V2</b> label as shown and select <b>Next</b></li> </ul> </li> </ul>  <ul style="list-style-type: none"> <li> <ul style="list-style-type: none"> <li>Specify the version up to which source branch changes should be merged. In most cases, changes merged previously will not be merged again.</li> <li>Version type: <span>Label</span> Label: <span>V2@\$/BranchingScenarios</span></li> </ul> </li> <li>Read the merge operation summary and select <b>Finish</b> to perform a reverse integration (RI) merge as shown below</li> <li>As there have been no changes in main, you will see the "no changes to merge" dialog</li> </ul> <ul style="list-style-type: none"> <li>Reverse-Integration (RI) merge                             <ul style="list-style-type: none"> <li>Right-click on the <b>featureMOD</b> branch in Source Control Explorer, select <b>Branching and Merging</b>, and <b>Merge</b>.</li> <li>Verify that source is <code>\$/BranchingScenarios/featureMOD</code> and target is <code>\$/BranchingScenario/main</code></li> </ul> </li> </ul>

## Branching Strategies – Hands-on Lab (HOL) – From Simple to Complex or not?

Step	Instructions
	<ul style="list-style-type: none"> <li>○ Select <b>Next</b>, set Version type to <b>Latest Version</b> and select <b>Next</b></li> <li>○ Read the merge operation summary and select <b>Finish</b> to perform a reverse integration (RI) merge as shown below</li> <li>○ Check in your changes to the <b>main</b> branch</li> <li>• We performed the following FI and RI merges            </li> </ul>
<p>2</p> <p>Merge featurePOWER</p> <p>☐ - Done</p>	<ul style="list-style-type: none"> <li>• Same as previous, but focused on other feature</li> <li>• Forward-Integration (FI) merge to get the latest changes from main, resolve possible conflicts in the feature branch, and stabilize before merging back into main.           <ul style="list-style-type: none"> <li>○ Right-click on the <b>main</b> branch in Source Control Explorer, select <b>Branching and Merging</b>, and <b>Merge</b>.</li> <li>○ Verify that source is <code>\$/BranchingScenarios/main</code> and target is <code>\$/BranchingScenario/featurePOWER</code></li> <li>○ Select <b>Next</b>, set Version type to <b>All changes up to a specific</b> and select <b>Next</b></li> <li>○ Select <b>Label</b>, find the <b>V2</b> label as shown and select <b>Next</b></li> </ul>  <ul style="list-style-type: none"> <li>○ Read the merge operation summary and select <b>Finish</b> to perform a reverse integration (RI) merge as shown below</li> <li>○ As there have been no changes in main, you will see the “no changes to merge” dialog</li> </ul>  </li> <li>• Reverse-Integration (RI) merge           <ul style="list-style-type: none"> <li>○ Right-click on the <b>featurePOWER</b> branch in Source Control Explorer, select <b>Branching and Merging</b>, and <b>Merge</b>.</li> <li>○ Verify that source is <code>\$/BranchingScenarios/featurePOWER</code> and target is <code>\$/BranchingScenario/main</code></li> <li>○ Select <b>Next</b>, set Version type to <b>Latest Version</b> and select <b>Next</b></li> <li>○ Read the merge operation summary and select <b>Finish</b> to perform a reverse integration (RI) merge as shown below</li> </ul> </li> </ul>

## Branching Strategies – Hands-on Lab (HOL) – From Simple to Complex or not?

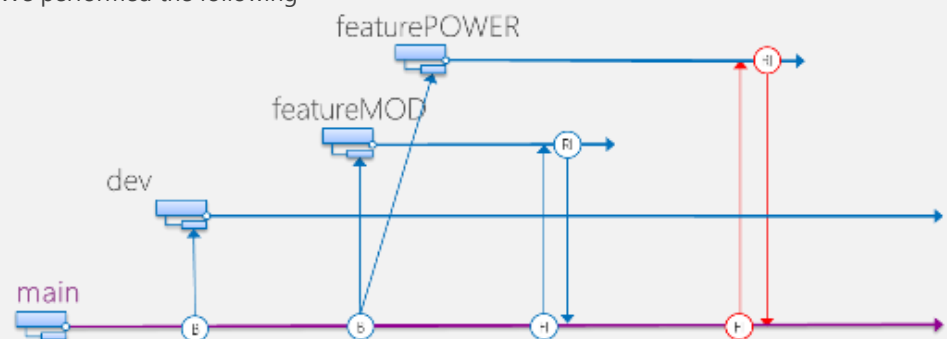
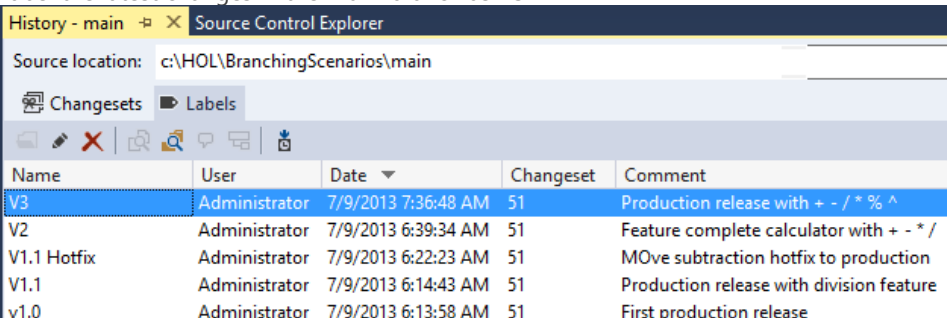
Step	Instructions
	<ul style="list-style-type: none"> <li>We performed the following</li> </ul> 
3 Verify and release <input type="checkbox"/> - Done	<ul style="list-style-type: none"> <li>Open the solution in the main branch and by setting the command line arguments as before, verify the following calculations:               <ul style="list-style-type: none"> <li>Test 4 2 +</li> <li>Test 4 2 -</li> <li>Test 4 2 *</li> <li>Test 4 2 /</li> <li>Test 5 2 %</li> <li>Test 4 2 ^</li> </ul> </li> <li>Label the latest changes in the <b>main</b> branch as V3</li> </ul> 

Table 18 – Lab 4, Task 4

### Task 5: Optionally (not recommended) delete the Feature Branches

#### WARNING

**None of us is crazy about this optional** task, as most teams want the audit trail and traceability. Deleting or worse, destroying everything rather defeats the purpose of Version Control.

We delete the feature branch, folder and associated artifacts, keeping history in version control for auditing and safety reasons. If you are happy to delete permanently the artifacts, you can use the **tf** command line utility to **destroy** permanently. Caution, the destroy operation cannot be reversed.

**tf destroy** → Destroys, or permanently deletes version-controlled files from Team Foundation version control.

## Branching Strategies – Hands-on Lab (HOL) – From Simple to Complex or not?

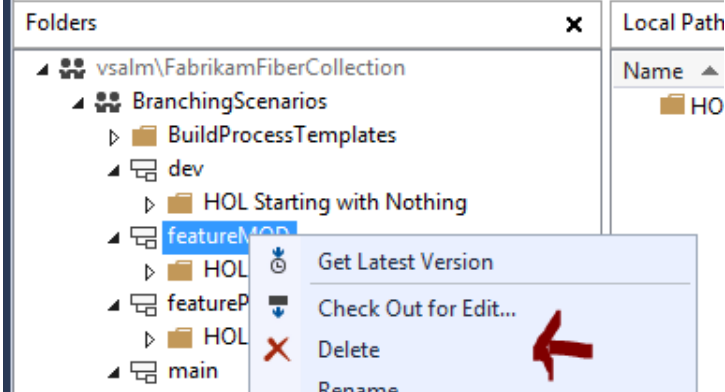
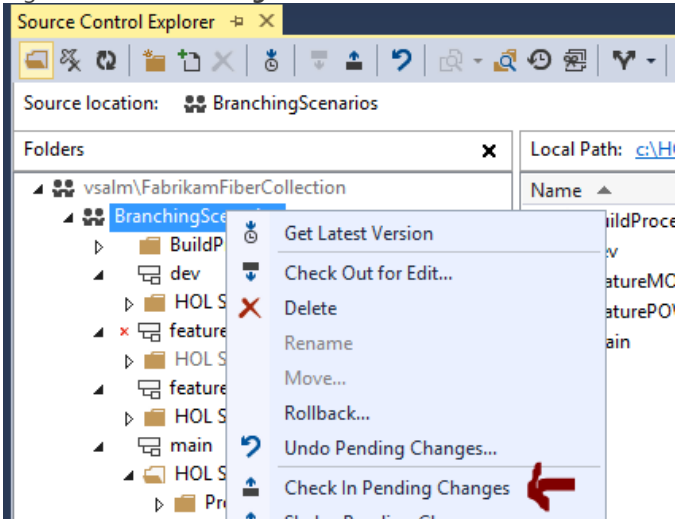
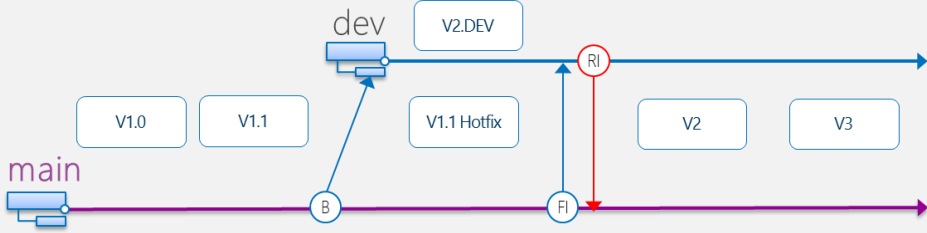
Step	Instructions
1 Delete featureMOD ☐ - Done	<ul style="list-style-type: none"> <li>Right click on <b>featureMOD</b> branch in Source Control Explorer and select <b>Delete</b></li> </ul>  <ul style="list-style-type: none"> <li>Right click on <b>BranchingScenarios</b> folder and select <b>Check in Pending Changes</b></li> </ul>  <ul style="list-style-type: none"> <li>Select <b>Check In</b>, which will delete the <b>featureMOD</b> branch.</li> </ul>
2 Delete featurePOWER ☐ - Done	<ul style="list-style-type: none"> <li>Right click on <b>featurePOWER</b> branch in Source Control Explorer and select <b>Delete</b></li> <li>Right click on <b>BranchingScenarios</b> folder and select <b>Check in Pending Changes</b></li> <li>Select <b>Check In</b>, which will delete the <b>featurePOWER</b> branch.</li> </ul>

Table 19 – Lab 4, Task 5

REVIEW

We explored the feature isolation strategy. We demonstrated how to implement this strategy, and why to use this strategy. We implemented two clearly defined and isolated features and have shipped v3



In this exercise we performed:

- 2 branches
- 4 merges, i.e. two FI's and two RI's
- 1 label

## Exercise 5: Release Isolation ... audit alarm

### GOAL

Explore the Release Isolation Strategy, which introduces one or more release branches from MAIN, which enables concurrent release management. Technically speaking we are introducing the Development & Release isolation strategy in this walkthrough, but will discuss on the release side only in this exercise.

### Context

Management has notified your team that the organization needs to support release v2, v3 and future releases concurrently and that the compliance team has requested that each release has an accurate snapshot of the sources used at release time.

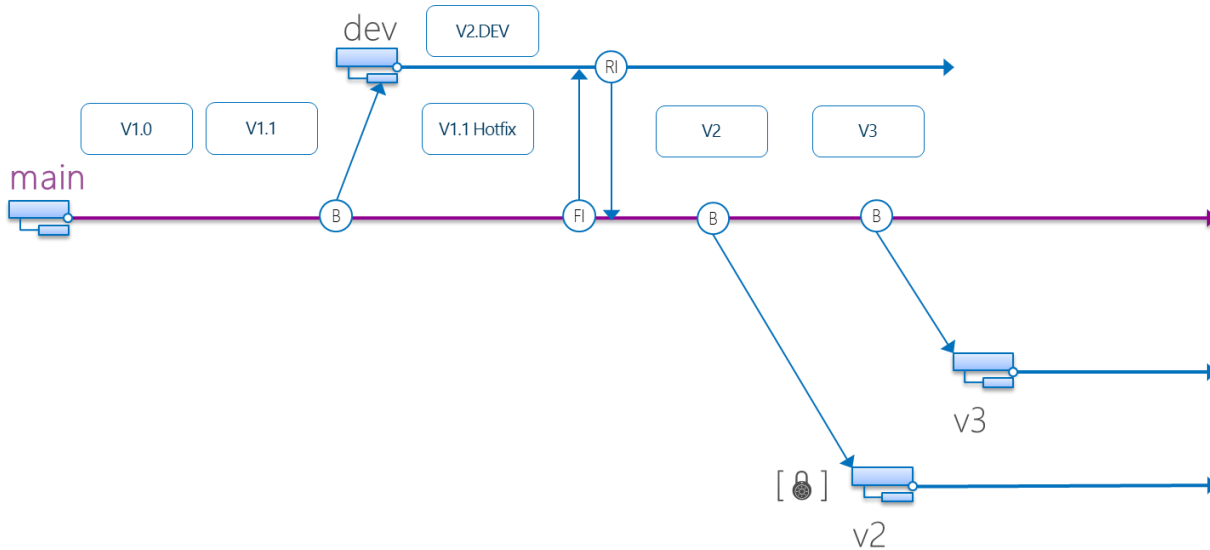
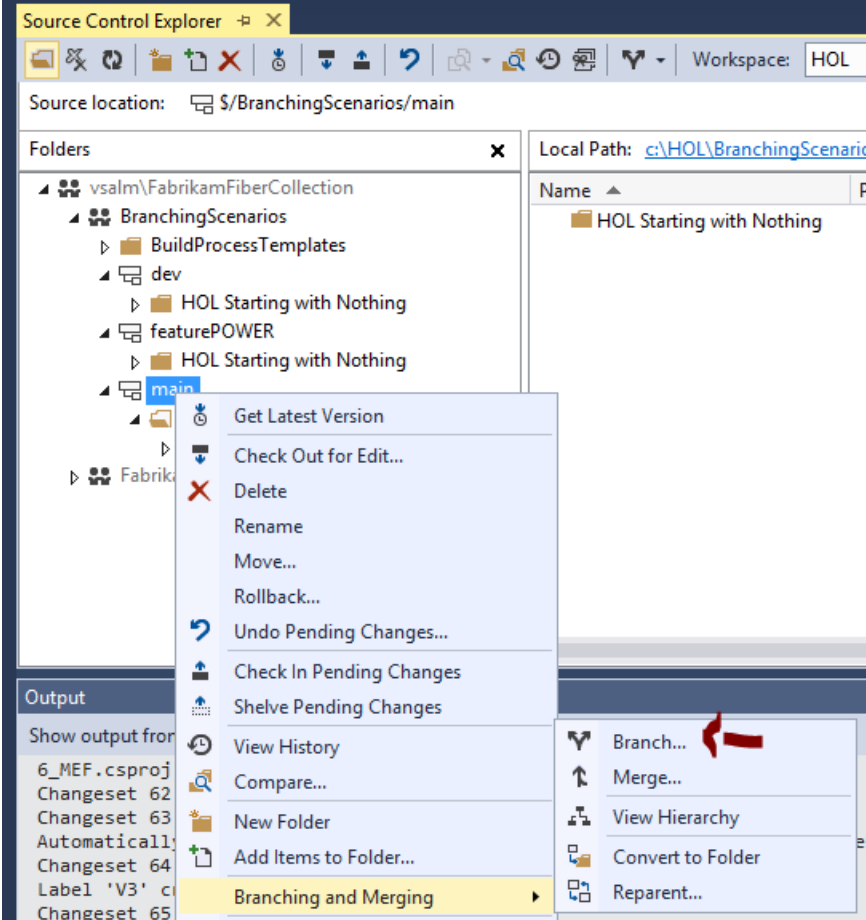
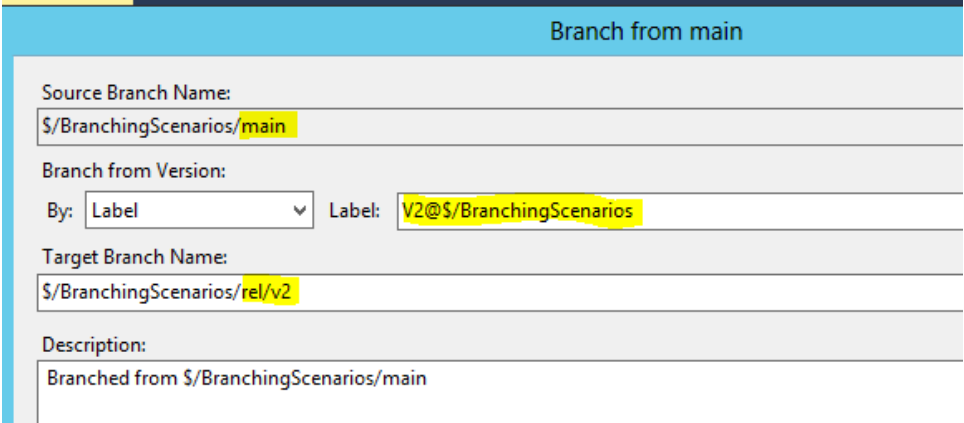


Figure 21 – Release Isolation

## Task 1: Create Release Branches

Step	Instructions
1 Release v2 ☐ - Done	<ul style="list-style-type: none"> <li>Right-click on <b>main</b> in the Source Control Explorer, select <b>Branching and Merging, Branch</b>  </li> <li>Define the <b>Target Branch Name</b> as <code>\$/BranchingScenarios/rel/v2</code>, branch by <b>label V2@\$/BranchingScenarios</b>, as shown, and click <b>Branch</b>  </li> <li>Right-click on <b>BranchingScenarios</b> folder in the Source Control Explorer and select <b>Get Latest Version</b></li> </ul>
2 Release v3 ☐ - Done	<ul style="list-style-type: none"> <li>Right-click on <b>main</b> in the Source Control Explorer, select <b>Branching and Merging, Branch</b></li> <li>Define the <b>Target Branch Name</b> as <code>\$/BranchingScenarios/rel/v3</code> and click <b>Branch</b>.</li> <li>Right-click on <b>BranchingScenarios</b> folder in the Source Control Explorer and select <b>Get Latest Version</b>.</li> </ul>
3	<ul style="list-style-type: none"> <li>It may have seemed like creating two branches from the same point in time.</li> </ul>

## Branching Strategies – Hands-on Lab (HOL) – From Simple to Complex or not?

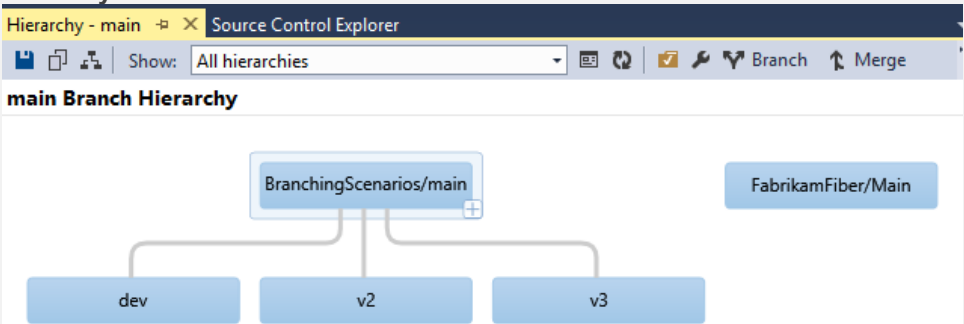
Step	Instructions
Let's recap ☐ - Done	<ul style="list-style-type: none"> <li>Remember that we created <b>v2</b> from the <b>label</b> and then we created <b>v3</b> from <b>latest</b>.</li> <li>We hope this helps to clarify how these branches differ and solidifies why the labeling is important.</li> </ul>
4 View hierarchy ☐ - Done	<ul style="list-style-type: none"> <li>Right-click on <b>main</b> branch in Source Control Explorer, select <b>Branching and Merging</b> and <b>View Hierarchy</b> to view a visualization all the branches</li> </ul> 

Table 20 – Lab 5, Task 1

### What about the lock symbol?

Now that we have isolated branches representing the releases we can optionally lock them down by applying read-only access. The lock symbol shown on some of the guidance diagrams indicate such locked branches. TFS does not yet visualize locked-down status in branching hierarchy models.

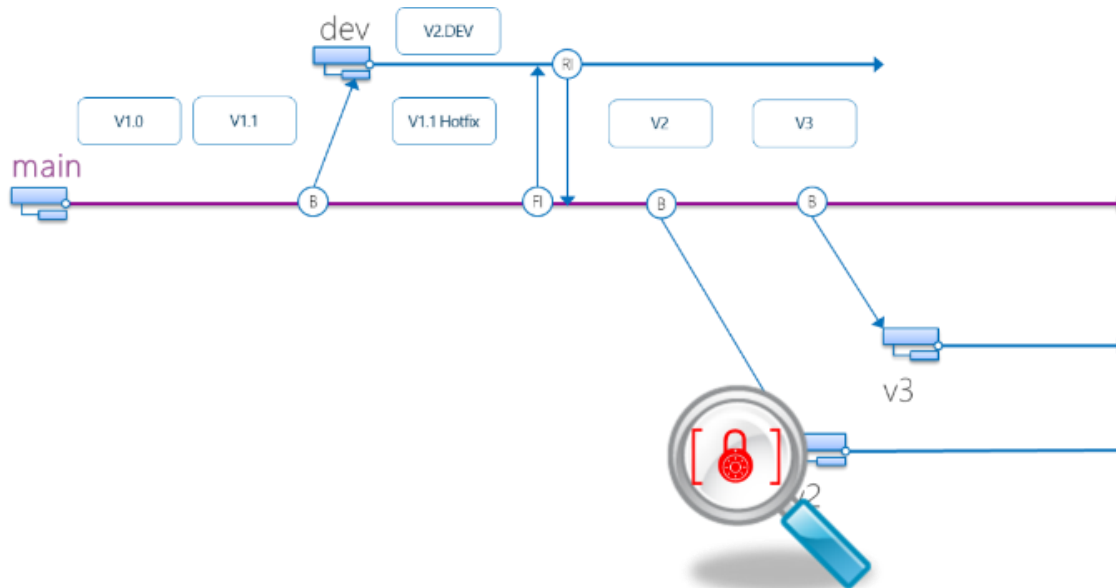


Figure 22 – Release Isolation: Locking down

#### REVIEW

We explored the release isolation strategy. We demonstrated how to implement this strategy. Finally, we explained and why to use release isolation.

We have not promoted the notion of the release branch being secure or having to be immutable. See guidance for discussions around this hot topic ☺



## Exercise 6: Servicing & Release Isolation

### GOAL

Explore the Servicing and Release Isolation strategy introduces servicing branches, which enables concurrent servicing management of bug fixes and service packs. This strategy evolves from the Release Isolation strategy, or more typically from the Development and Release Isolation strategy.

### Context

We will finish this walk-through with an easy and quick exercise. We will focus on demonstrating the concepts, allowing you to expand the concepts to your required level of granularity and complexity as and when needed.

Your team learns that a servicing model enabling customers to upgrade to the **next major release**, i.e. v1 → v2 and, in addition, the servicing model supports **additional service packs** per release, starting with v3.

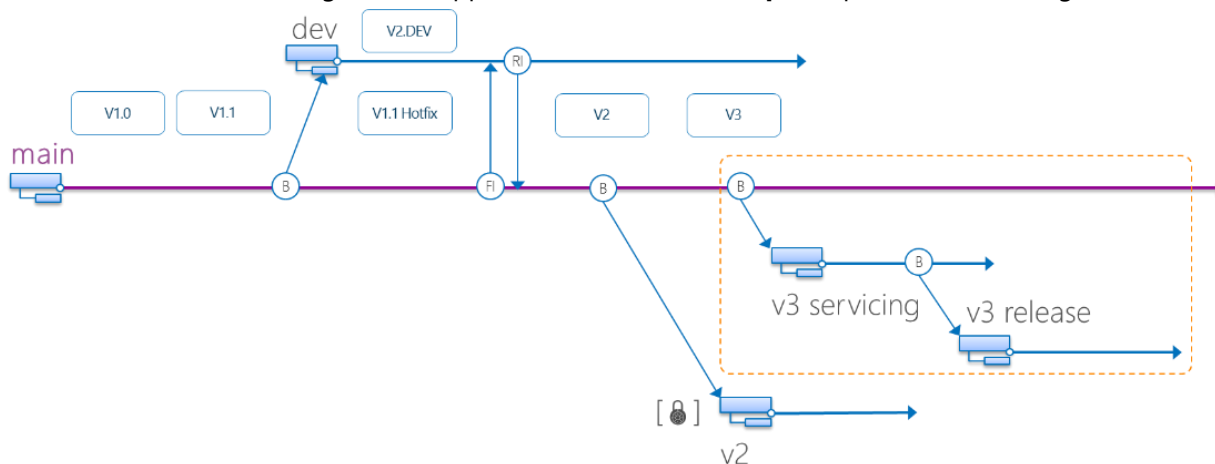


Figure 23 – Servicing & Release Isolation

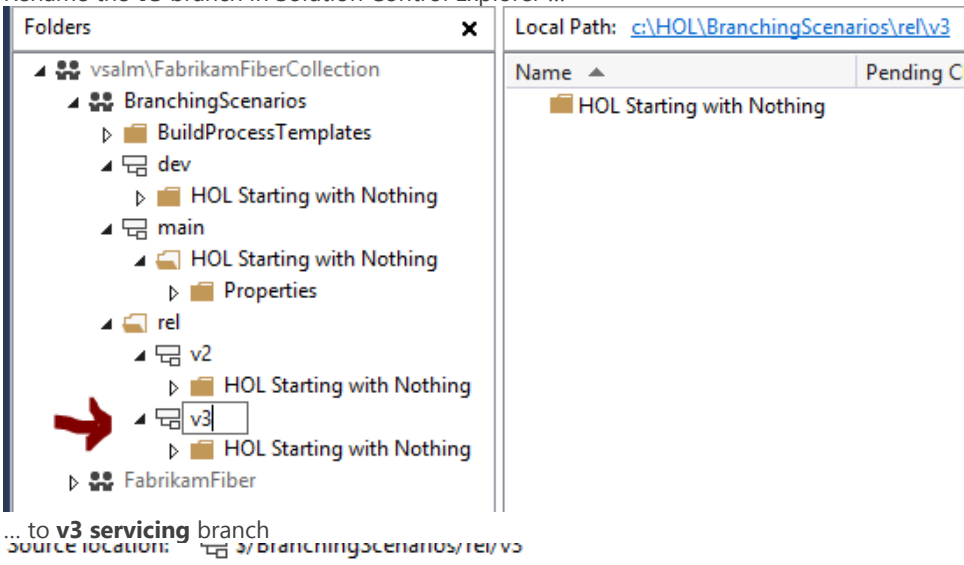
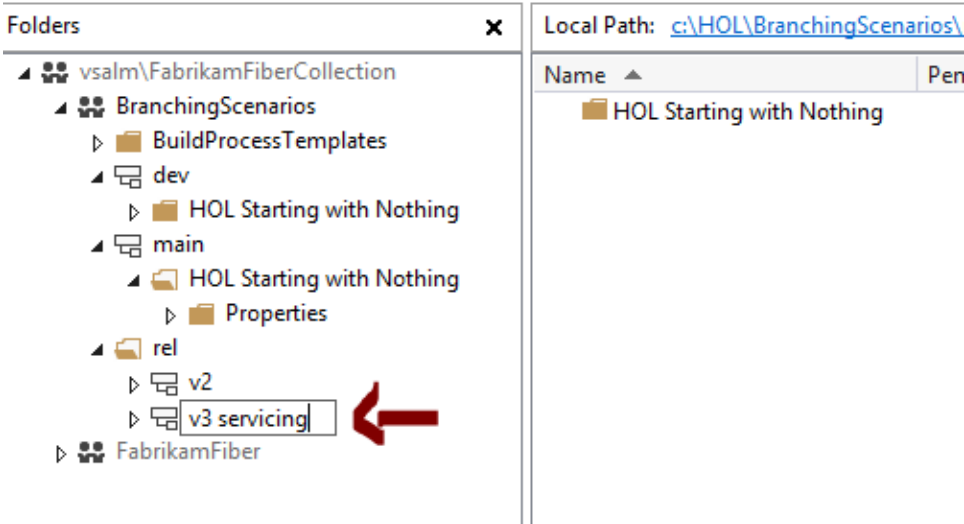
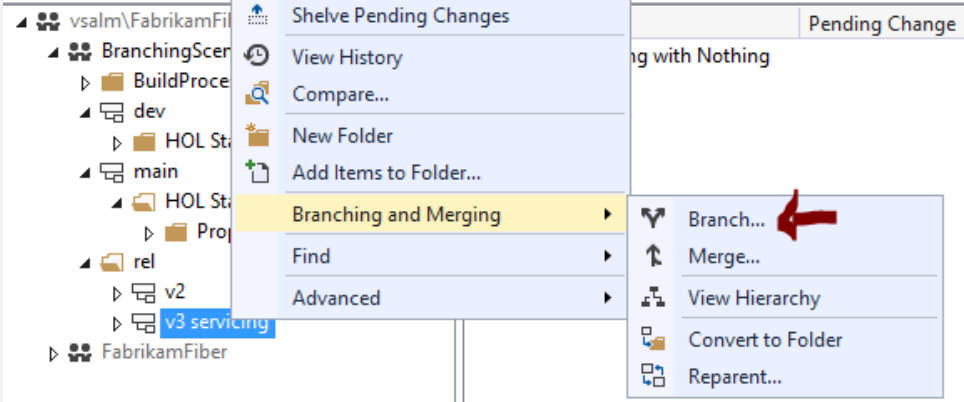
### NOTE

We are going to use an arbitrary naming convention for the branches in this exercise, as we have done before. You need to invest time in defining a consistent naming convention that suits your organization and environment before you perform a branch action. Renaming branches is atypical and as an alternative, you could proactively plan and introduce the servicing branch with v4 instead of injecting it into the v3 hierarchy.

### WARNING

Renaming branches is a technically feasible operation, but can introduce traceability and migration **challenges** in the future if not done consistently. This short and simple exercise demonstrates how easy it is to introduce more branches quickly, which dramatically **increase your merge maintenance and cost**.

## Task 1: Introduce the Servicing Branch

Step	Instructions
1 Rename release to servicing	<ul style="list-style-type: none"> <li>Rename the <b>v3</b> branch in Solution Control Explorer ...</li> </ul>  <p>... to <b>v3 servicing</b> branch Source location: <code>3/BranchingScenarios/rel/v3</code></p>  <ul style="list-style-type: none"> <li>Right-click on <b>servicing</b> branch, select <b>Check In Pending Changes</b> and check in.</li> </ul>
2 Add release branch ☐ - Done	<ul style="list-style-type: none"> <li>Right-click on <b>servicing</b> branch, select <b>Branching and Merging</b> and <b>Branch</b></li> </ul> 

## Branching Strategies – Hands-on Lab (HOL) – From Simple to Complex or not?

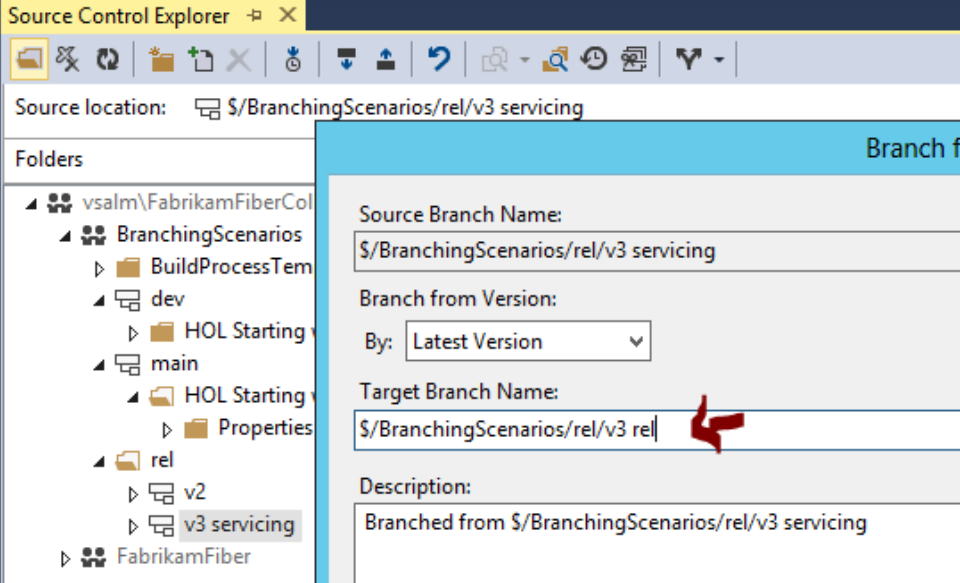
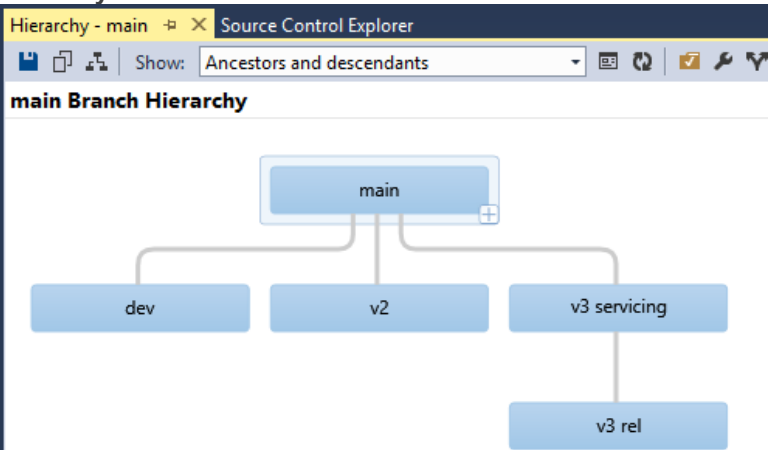
Step	Instructions
	<ul style="list-style-type: none"> <li>Define the Target Branch Name as <code>\$/BranchingScenarios/rel/v3 rel</code></li> </ul>  <ul style="list-style-type: none"> <li>Select <b>Branch</b> and <b>Yes</b> when prompted to continue to branch</li> <li>Right-click on <b>BranchingScenarios</b> folder and perform a <b>Get Latest Version</b>.</li> </ul>
<p>3</p> <p>View hierarchy</p> <p>☐ - Done</p>	<ul style="list-style-type: none"> <li>Right-click on <b>main</b> branch in Solution Control Explorer, select <b>Branching and Merging</b>, and <b>View Hierarchy</b></li> </ul> 

Table 21 – Lab 6, Task 1

### REVIEW

We explored the servicing and release isolation strategy, how to implement, intentionally using a very simple branching scenario. Remember that we discourage the use of branches to isolate servicing, but your requirements may force you to deploy this branching strategy and release a service pack.

# In Conclusion

This concludes our adventure into Branching Strategies. We have touched on basic theory and introduced you to the common strategies and alternatives to branching. We have covered various exercises in the walk-throughs and have complemented this guide with a Hands-on Lab and companion guides.

In the final pages of this guide, you will find our Quick Reference cheat sheets and posters. These are available separately as part of the guidance and you might find it useful to print these and hang them up in the team area. We hope that you have found this guide useful.

Sincerely

**The Microsoft Visual Studio ALM Rangers**



# Branching Strategies

There are no magic potions or silver bullets with branching and merging. You should map the numerous options against your requirements to help you find the best fit or a foundation to build upon. This cheat sheet introduces you to some of the branching plans and strategies covered by the guidance and a matrix that allows you to focus on the most relevant plan. This cheat sheet is a starting point—not an exhaustive guide—for your decision-making on a branch plan.

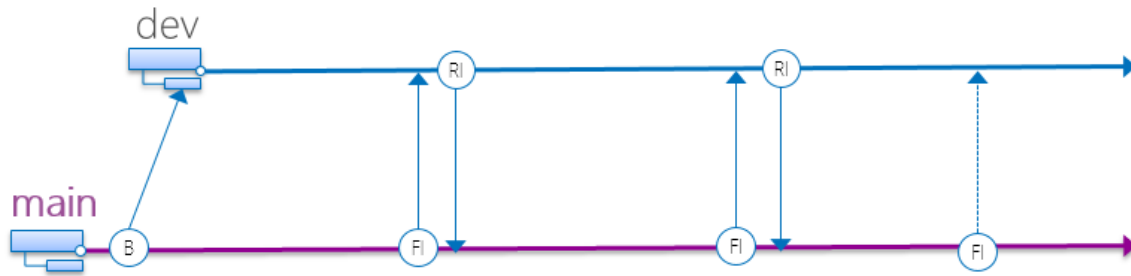
## Why Branch?

- Manage concurrent / parallel work
- Isolate risks from concurrent changes
- Take snapshots for separate support
- Share resources

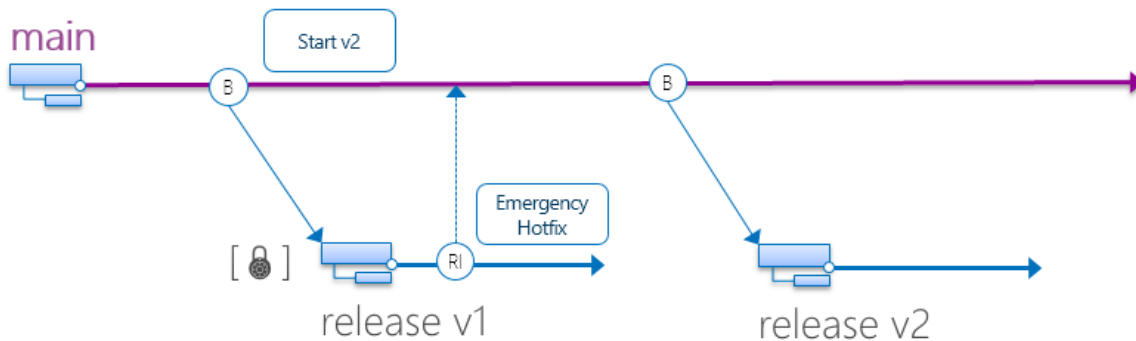
Requirement	Main Only	Development Isolation	Development and Release Isolation	Release Isolation	Servicing and Release Isolation	Servicing, hotfix and Release Isolation	Feature Isolation	Code Promotion	Feature Toggling
Keep it simple & minimize maintenance costs	*	*	*	*			*	*	*
Development isolation		*	*				*		
Feature isolation		*	*				*		*
Release isolation			*	*	*	*			
Long running testing lifecycles			*					*	
Dynamic feature enablement									*
Single release service model				*	*	*			
Multiple release service model				*	*	*			
Multiple release and service pack service model					*	*			
Multiple release, hotfix and service pack service model						*			
Compliance requirement for snapshot of releases				*	*	*			

# Development and Release Isolation Branching

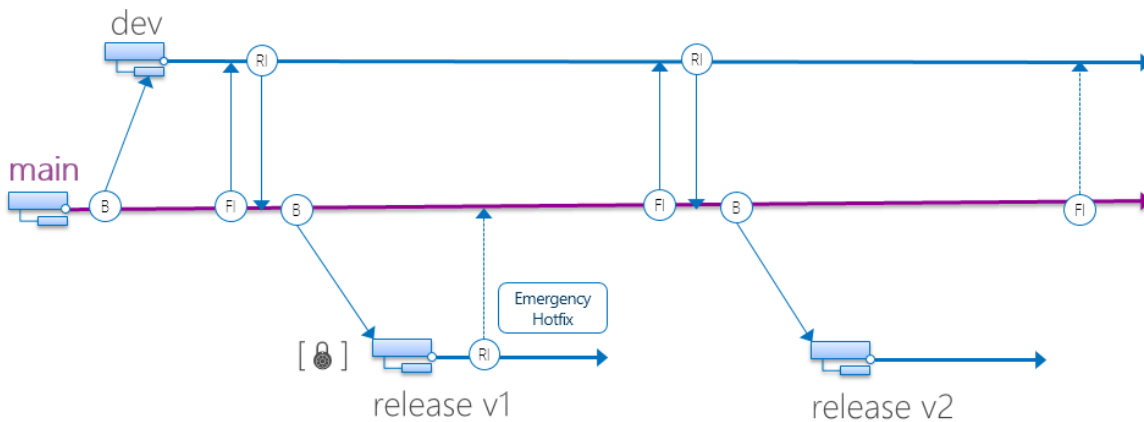
The **Development Isolation** strategy introduces one or more development branches from main, which enables concurrent development of the next release, experimentation or bug fixes in isolated development branches.



The **Release Isolation** strategy introduces one or more release branches from MAIN, which enables concurrent release management.



The **Development and Release Isolation** strategy combines the Development Isolation and Release Isolation strategies, embracing both their usage scenarios and considerations.



Key: **(B)** Branch **(FI)** Forward Integrate **(RI)** Reverse Integrate 🔒 read-only

## When to consider

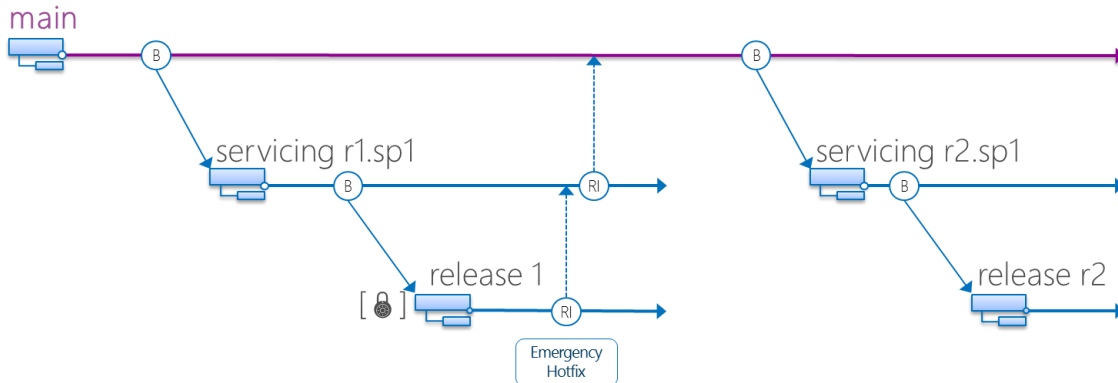
- Isolation and concurrent development
- Multiple major releases
- Compliance requirements

# Service and Release Isolation Branching

The **Servicing and Release** Isolation strategy introduces servicing branches, which enables concurrent servicing management of bug fixes and service packs. This strategy evolves from the Release Isolation strategy, or more typically from the Development and Release Isolation strategy

## When to consider

- Isolation and concurrent releases
- Multiple major releases and service packs
- Compliance requirements



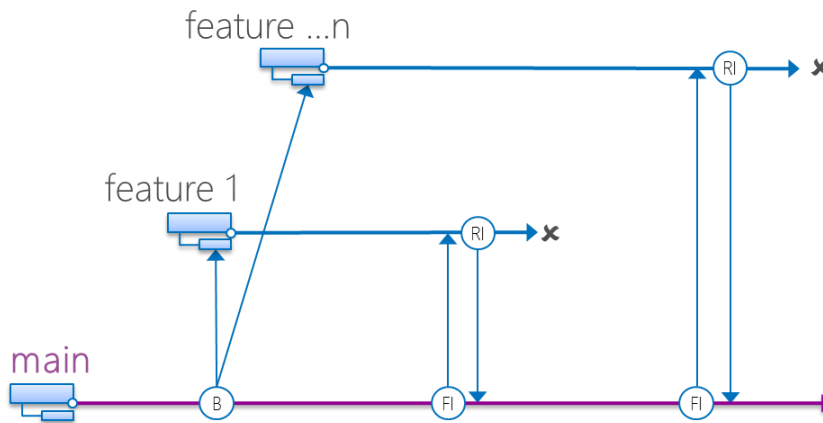
Key: **(B)** Branch **(FI)** Forward Integrate **(RI)** Reverse Integrate 🔒 read-only

# Feature Isolation Branching

On many software projects, you may have clearly defined features you wish to implement in your product. There may be separate dedicated teams that are simultaneously developing various features that all roll into a product. When you have this clear isolation in your process, you may consider **Feature Isolation** branching. Branching by feature can also be a reasonable branch plan for large products that inherently have many features and various teams working on the product.

## When to consider

- Concurrent feature development
- Clearly isolated features in codebase
- Large product with many features
- Features not in same release cycle
- Ability to abandon features



Key: **(B)** Branch **(FI)** Forward Integrate **(RI)** Reverse Integrate **x** Delete

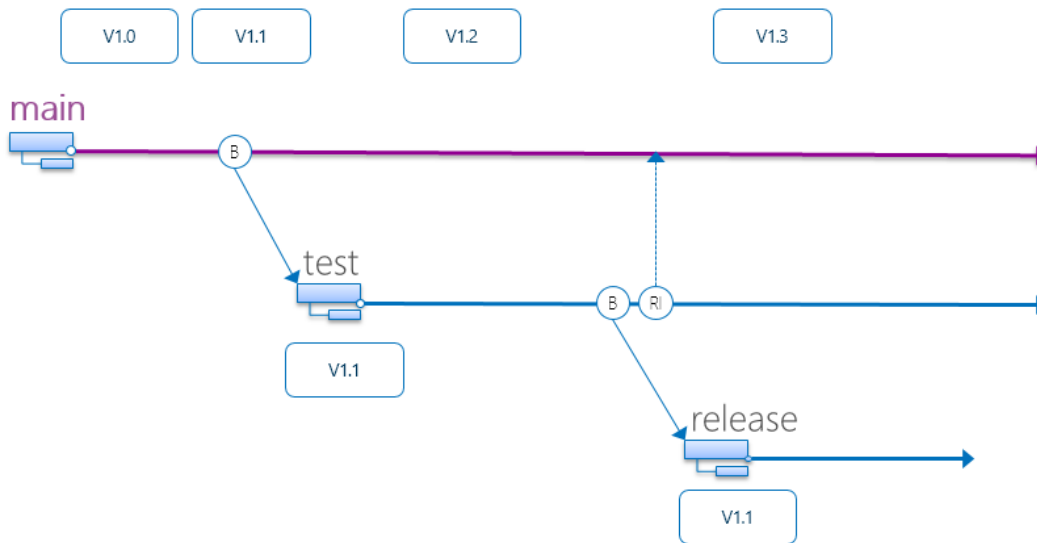


# Code Promotion Branching

**Code Promotion** is a plan that promotes versions of the code base to new levels as they become more stable. Other version control systems have implemented similar techniques with the use of file attributes to indicate a file's promotion level. As the code in main becomes stable, we promote the code base to the testing branch.

## When to consider

- Single major release in production
- Long running testing lifecycles
- Need to avoid code freezes, still having release isolation



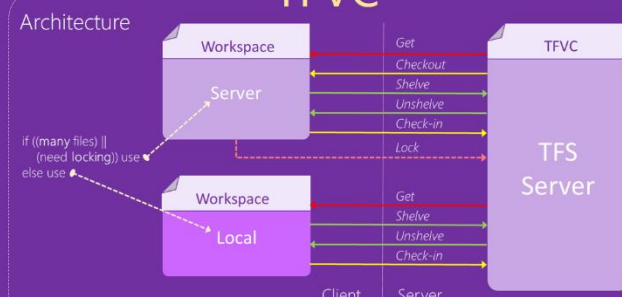
Key: **(B)** Branch **(RI)** Reverse Integrate

# Version Control Cheat Sheet for TFVC and Git

This cheat sheet is available separately in JPG and PDF format as part of the guidance.

## TFVC

**Architecture**



**Concepts**

- Branch** is an isolated copy of item metadata and version control history.
- Changeset** is a logical container for changes of a single check-in.
- Check-in** commits pending changes in workspace to server as a changeset.
- Label** mutable grouping of specific version of a set of source files.
- Shelveset** is a set of pending changes are temporarily saved on the server.
- Workspace** is a local copy of your team's codebase. Create multiple workspaces and switch among them to work on different branches or copies of the codebase.

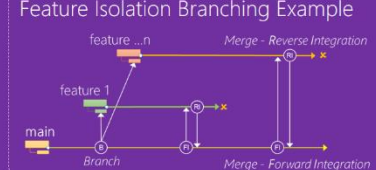
**Commands**

**TF command**  
Team Foundation Version Control (tfvc) command line tool with a variety of commands.

- everyday**
  - add** adds new files and folders from a local file system location
  - get** retrieves a copy of items from TFVC to the workspace
  - checkout** commits pending changes in current workspace to TFVC
  - checkout** makes local file writable and changes status to "edit"
  - delete** removes files and folders from TFVC and disk
  - history** displays the revision history for files and folders
- branching**
  - branch** creates a copy of items, preserving a relationship to the original items
  - merge** applies changes from one branch into another
- shelving**
  - shelve** stores a set of pending changes in TFVC without a commit
  - shelveset** used to save details of a shelveset or view shelvesets belonging to a specific user
  - unshelve** restores shelved changes from TFVC to current workspace
- uncommon**
  - changeset** displays information about a changeset
  - delete** permanently deletes, version-controlled files from TFS. Admin only
  - lock** locks or unlocks to prevent against checkout of items
  - rename** changes the name or the path of items
  - rollback** reverts the changes of one or more changesets
  - undo** restores items that were previously deleted
  - workspace** creates, deletes, displays, or modifies properties and mappings associated with a workspace
- help**
  - Type **TF /?** on the command line for a complete list of commands and arguments.

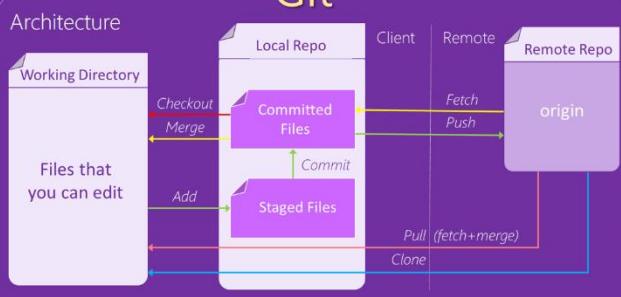
**TFSDeleteProject**  
Command line tool which deletes a team project from a TFS team project collection. It is a non-reversible operation!

**Feature Isolation Branching Example**



## Git


**Architecture**



**Concepts**

- Branch** is a named pointer to a commit history in the repository. Used to isolate changes from each other.
- Commit** (noun) is equivalent (mostly) to Changeset, but can also be an action.
- HEAD** is a pointer to the branch your commits will be associated with.
- Stash** is a set of pending changes are temporarily saved in the repository.
- Tag** is a named pointer to a commit in the repository. Useful for marking a point-in-time in your repository.

**Feature Isolation Branching Example**




**Links**

- [microsoft.github.io](#) - Windows client downloads
- [git-scm.com](#) - documentation
- [github.com](#) - code host
- [bitly/gitsc](#) - Get source control provider VS2008-2012
- [bitly/gitext](#) - Get extensions

**Commands**

**git command**  
git command line tool with a variety of commands.

- everyday**
  - add** adds the current content of existing paths
  - clone** creates a working copy from existing repository
  - commit** (verb) all the staged local changes
  - fetch** fetches the latest changes from origin, not merging
  - pull** fetches the latest changes from origin and merge into working copies
  - push** commits changes to origin
  - rm** removes files from the working tree and from the index
  - status** shows uncommitted changes in the working directory
  - tag** mark a version or milestone
- branching**
  - branch** (noun) creates branch called name based on HEAD
  - branch** (-f) creates branch called name
  - checkout** (-t) switch to the id branch
  - diff** shows changes to tracked files
  - merge** two branches
- uncommon**
  - ls-tree** show who changed what and when
  - ls-tree** -r show who changed what and when
  - grep** search working directory
  - log** show history of changes
  - show** (-p) show a specific file from a specific id
- help**
  - Type **git help** on the command line for a complete list of commands and arguments.



Visual Studio ALM Rangers Solutions – <http://aka.ms/vsarsolutions>

2014-03-20 v3

# Version Control considerations for TFVC and Git

This cheat sheet is available separately in JPG and PDF format as part of the guidance.

