Source Control Management - Guidelines

iInterchange System Pvt. Ltd.

Version 1.0

iInterchange™

Your Indian Technology Partner

REVISION HISTORY

VERSION NO	AUTHOR	RELEASE DATE	Change Details
1.0	Tech	02-Dec-2013	Initial Draft

Source Control Management - Guidelines

1. Introduction

Source control is your backup of your code, as well as your change history to track changes.

With the source control (we use TFS); we can share project code and cooperate with other team members. Using it allows us to track changes, compare code and even roll-back if required. Moreover, it keeps our code safe that is the most important.

However, the best use is the blame game. You don't just fix code, you see who broke it, fix it, and then let them know.
Here are some of the reasons:

- Viewing the changes in source control on each individual file.
- We can select different change sets and compare the changes
- We can select different change sets and compare the changes.
- Using annotate is great. It allows us to find the coder who made the breaking changes, to understand his thoughts before deleting/changing his or her code.

2. Workspace

"Map folders in Visual Studio Team Foundation Server to folders on your local computer"

Work space is the local copy of the files and folders in the server.

It maps Server folders to Local folders. This also includes:

- A list of all the files in your workspace.
- The version of each file.
- A list of the pending changes.
- Active and cloaked items.

A Workspace defines all of the rules of how your local operations on those files relate to what is on the server.

3. The Practice

Never allow a situation where a developer can check out code and the code does not compile. This is called "breaking the build" and the punishment in our office is fixing broken links for an hour!

Figure 1: Bad Process

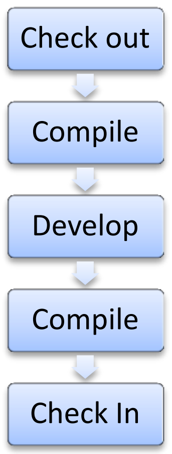
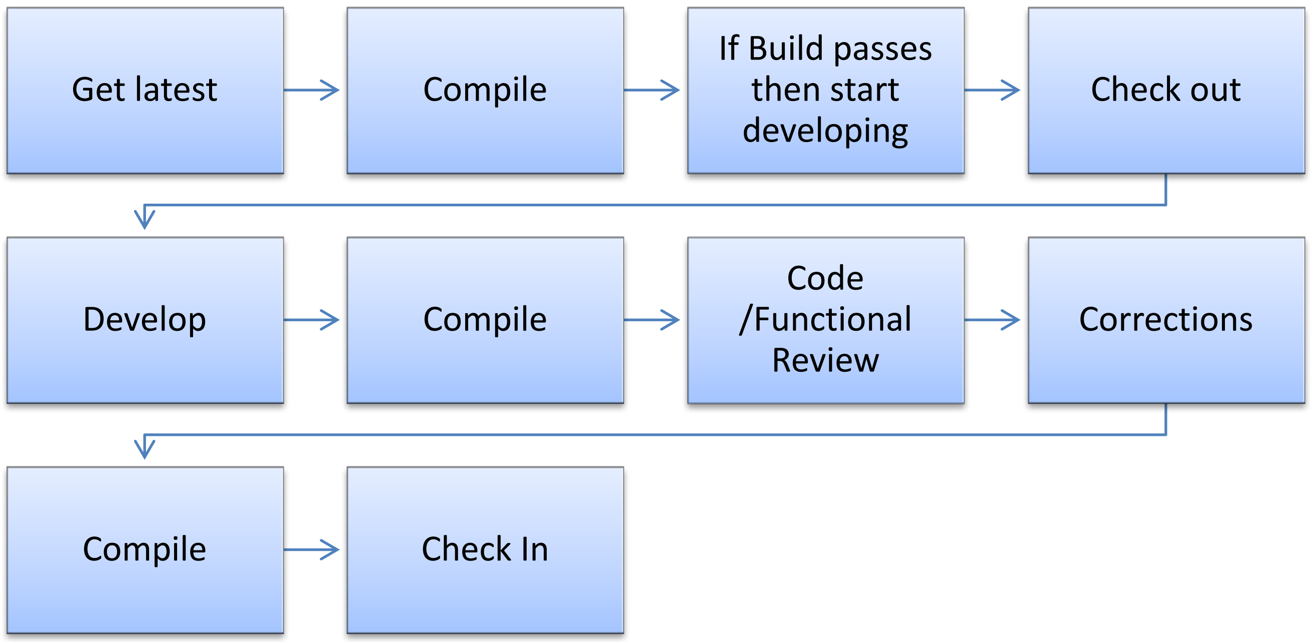


Figure 2: Good Process



4. Check-In

Check in commits pending changes in the current workspace to the server.

Too many people treat Source Control as a networked drive. Don't just check-in when the clock ticks past 6 o'clock. If code doesn't reviewed or won't even compile put your code in a shelve set.

Frequently developers work on long or difficult features/bugs and leave code checked out for days or worse still, weeks.

Source code should be checked in regularly. We recommend a check-in:

- Immediately after completing a piece of functionality, where the code compiles and passes the review.
- Before leaving your workstation for an extended period of time

If the changes would break the build or are in a state that cannot be put into the main trunk, then this code should be put into a shelve set in source control.

Another good reason to check-in regularly is that it makes it easier to merge your changes with other developers. If all developers' check-in lots of changes in one go, you will spend a lot of your time resolving conflicts instead of doing work.

When working on a task spanning multiple files, do not check-in only one or two of the files, this leads to the problem of partial check-ins where references to new classes or methods are unavailable because they are in the files that haven't been checked in. So either, check-in all the files you are working on or none at all if you aren't finished working on the task.

5. Comments with check-ins

Without comments, some of the other built in features like History become redundant without comments. No Comments against the check-ins we don't know what changes were made in each revision.
To enforce this behavior, you will need to:

1. Install Team Foundation Server Power Tools using the Extension Manager in Visual Studio

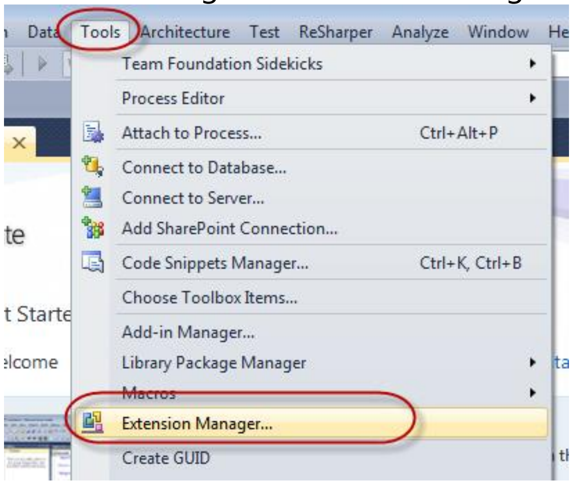
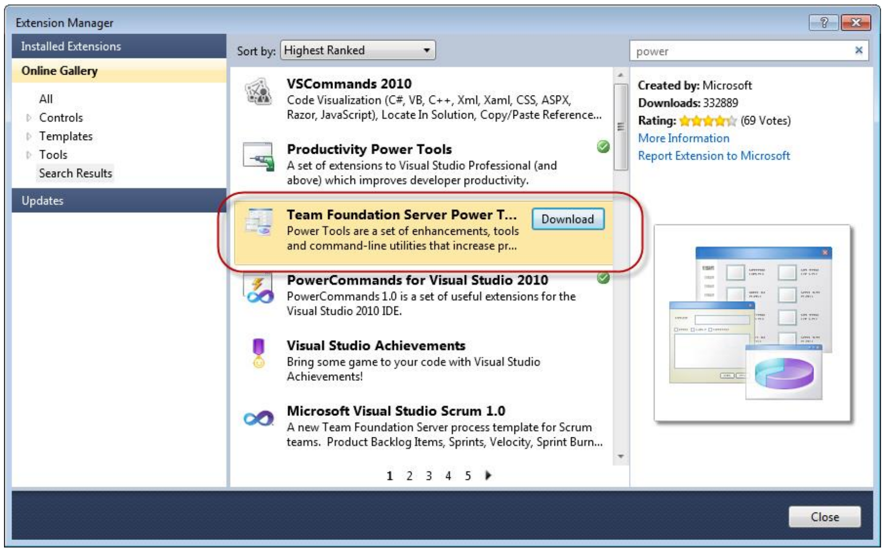


Figure: Opening the Extension Manager in Visual Studio



VSCommands 2010
Code Visualization (C#, VB, C++, Xml, Xaml, CSS, ASPX, Razor, JavaScript), Locate In Solution, Copy/Paste Reference...

Productivity Power Tools
A set of extensions to Visual Studio Professional (and above) which improves developer productivity.

Team Foundation Server Power T...
Power Tools are a set of enhancements, tools and command-line utilities that increase pr...

PowerCommands for Visual Studio 2010
PowerCommands 1.0 is a set of useful extensions for the Visual Studio 2010 IDE.

Visual Studio Achievements
Bring some game to your code with Visual Studio Achievements!

Microsoft Visual Studio Scrum 1.0
A new Team Foundation Server process template for Scrum teams. Product Backlog Items, Sprints, Velocity, Sprint Burn...

Created by: Microsoft
Downloads: 332889
Rating: ★★★★☆ (69 Votes)
More Information
Report Extension to Microsoft

Figure: Installing TFS Power Tools from Visual Studio's Extension Manager

2. Right click the Team Project in **Team Explorer** > **Team Project Settings** > **Source Control**
3. Select the Check-in Policy tab
4. Click Add
5. Select the Change set Comments Policy

6. Comment convention

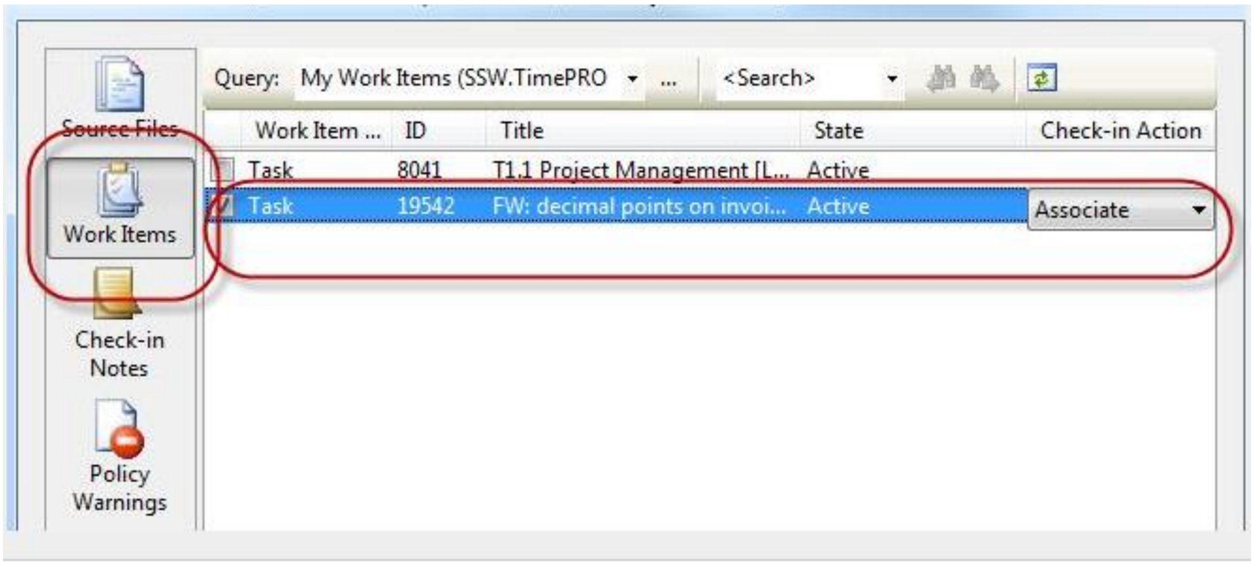
New, Bug or Refactor should be the prefix.
Here are some examples:

- New P112: Added a new control for Date of Birth.
- Bug P113: Fixed validation to now allow US dates.
- Refactor: Moved the email regex from inline to a resource file.

7. Work Item association

One of the big advantage of using TFS is end to end traceability, however this requires the developer to do one extra step to link their code (change set) with requirements (work items). Code is the body of software, while user requirement is the spirit. Work Item association feature helps us to link the spirit and body of software together. This is especially useful when you trying to identify the impact of a bug in term of user requirements.

In order to achieve this, developers need to choose the Work Item tab when check-in and "associate" code with a related work item.



	Work Item ...	ID	Title	State	Check-in Action
☐	Task	8041	T1.1 Project Management [L...	Active	
☑	Task	19542	FW: decimal points on invoi...	Active	Associate

8. Project/Product Version conventions

Creating a solution and having it maintainable over time is an art and not a science.

8.1 Branching

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 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, at least every day if possible if active changes are taking place.
- Ensure you build and run sufficient BVT's to measure the stability of the branch.
- For stabilization changes, make your changes, build and pass all the reviews before merging (RI) the change and supporting changes to the parent.
- A merge (RI) to MAIN is treated like 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 release all DEV branches can begin next version work.

8.2 Why Branch?

- Mange Concurrent /parallel work.
- Isolate risk from concurrent changes.
- Take snap shots for separate support

8.3 Avoiding Branching

- Save or share development changes.
- Work in the "Next version" branch mentioned above if the changes are approved for next version release.
- Is the branch needed now, or can it rather be created at a later date, perhaps from a change set version.

8.4 Permissions

From Team Foundation Server you have two permissions related to branching and merging

- Manage Branch
- Merge

These two permissions allow teams to designate certain individuals to be responsible for creating new branches, while others can be responsible for merging code between branches, while most developers will be restricted to working only on certain branches.

8.4.1 Branch Permissions

Users require Manage Branch permission on a given path in order to:

- 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)

8.4.2 Merge Permission

Users require Merge permission in order to do 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 a particular branch or folder from being merged to another path. Merge permission is not limited to branches – it can be applied to folders and branches under a given path.

8.5 Label your versions and releases in Source Control

TFS takes labeling to a new level unlike VSS which was a point in time label. TFS labels each file based on their change set version. You can then get code as it was when you labeled the source.
Labeling a release is a good way to go back to a version and generate a compiled version. If you wanted to develop an older version then you would create a branch instead (of course you can create a branch off a label)

Table 1 : Branching Vocabulary

Term	Description
Development Branch	Changes for next version work
Forward Integrate (FI)	Merges from parent to child branches
Hot Fix	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 where ship stopping bug fixes are made before major product release. After product release this branch may become 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

8.6 Branch Types

There are three general branch types, DEV, MAIN and RELEASE. Regardless of branch type, you should apply the following considerations to all branches

- Build daily to give the team a daily cadence to work toward.
- Implement continuous integration builds to immediately identify quality issues
- Code flow, the movement of changes between child and parent branches, is a concept all team members must consider and understand.

The following table lists additional considerations and comments for each type.

Table 2 : Branch Type Considerations

Attributes	Comments
MAIN • Every branch you intend to merge with should have a natural merge (RI) path back to MAIN (i.e. no baseless merges). • Breaks in MAIN need to be fixed immediately. • No direct check-ins to MAIN branch; only build and BVT fixes are a good practice. • Successful MAIN build indicates child DEVELOPMENT branches should merge (FI) from MAIN. • QA team should be able to pick up any MAIN build for testing.	MAIN is the junction between DEVELOPMENT and RELEASE branches. Changes in MAIN will probably merge (FI) into DEVELOPMENT, so it is critical that builds from MAIN remain high quality. At minimum this means MAIN must remain buildable and pass all BVT's. As the number of DEVELOPMENT branches increase, the need to merge (FI) following each successful MAIN build increases.
DEVELOPMENT Before creating a DEVELOPMENT branch, make sure you can do the following: • Select a parent branch – if your changes are focused on your next product release then the parent branch should be MAIN. • Branch from a recent known good state of the parent – start your branch from the latest successful build of the parent branch. On day one of your DEVELOPMENT branch it should be in the same state as its parent (i.e. build and pass BVTs successfully). • Merge (FI) frequently – your goal should be to be no more than 1-2 days out of sync with MAIN. This will help reduce the complexity of "big bang" type merges. Ideally you should merge (FI) every time the parent branch builds and passes BVTs. • Merge (RI) from child to parent based on quality. Minimum merge (RI) requirements are • Be in sync with parent branch • Build successfully	A self-contained branch effectively enables each development activity to proceed at its own pace, without taking any dependency on another. It follows that these branches are allowed to diverge their respective sources along the particular development activity they are involved with – fixing a bug, implementing a feature, or stabilizing a breaking change.

RELEASE A successful release branch strategy enables the following scenarios. 1. Developers only need to check in once based on which release vehicle the change is for (i.e. Hotfixes go into the product HOTFIX branch). 2. No need for baseless merges. Create a natural merge path back to MAIN by creating a hierarchal branch structure based on your release vehicles.	Your release branch plan should be built around your software release vehicles. A release vehicle is how your software is delivered to your customer. The most common release vehicles are the major release, Hotfix and service pack. In a software plus services scenario the names may be different however and the release may be more frequent.

8.7 Branching Plans

8.7.1 Basic Branch Plan

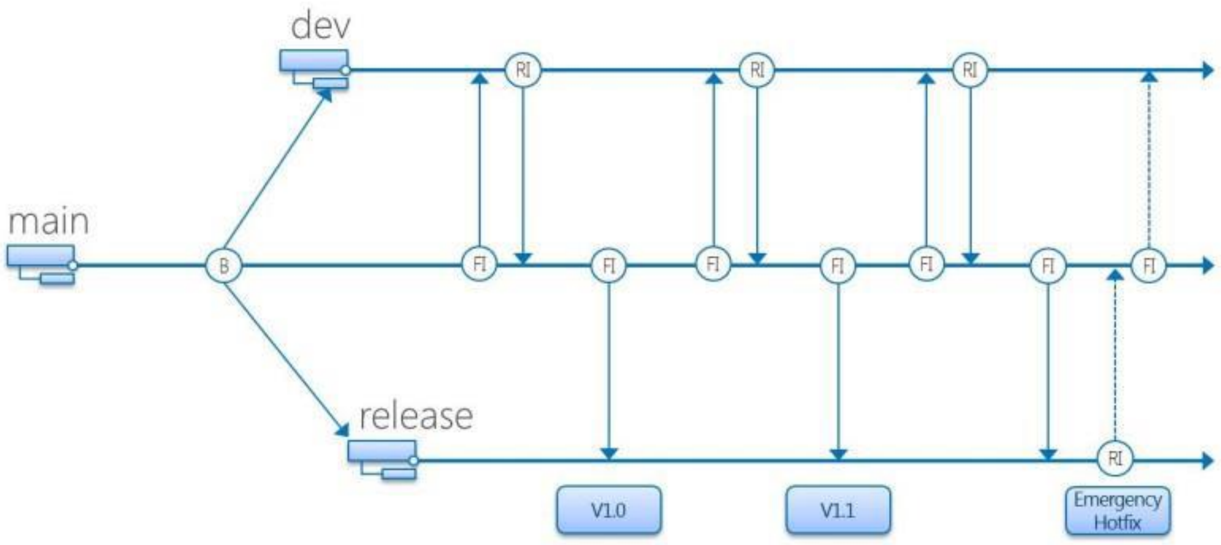
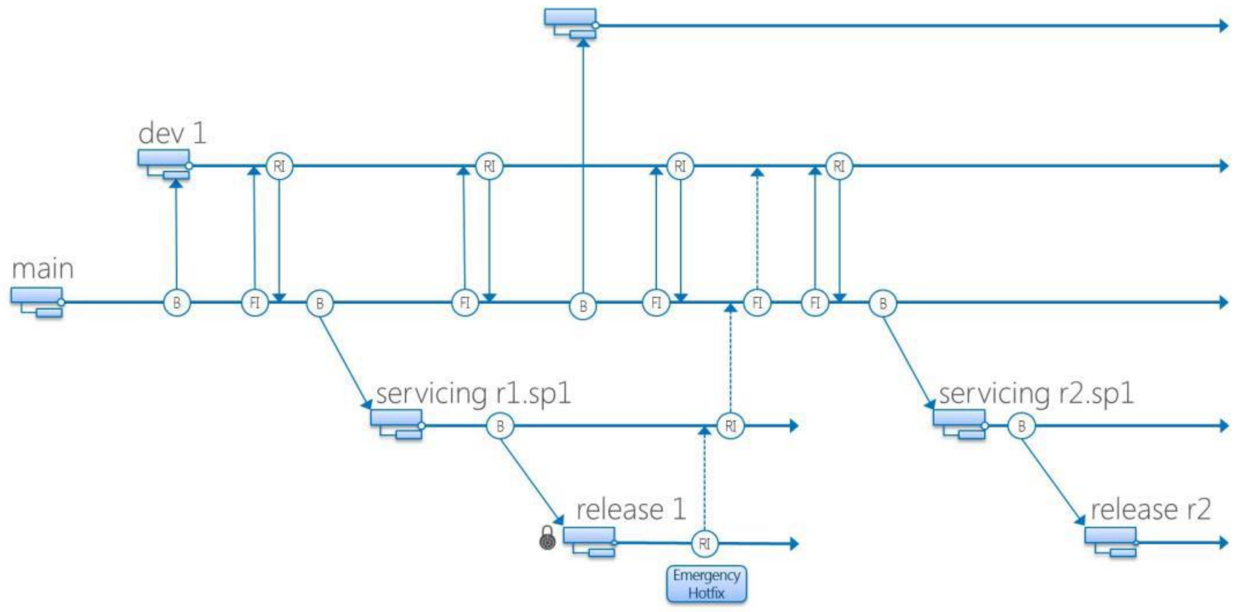


Table 3 : Basic Branch Plan Summary

Usage Scenarios	Considerations
1. You have a single major release (i.e. a single release vehicle) that is shipped to customers.	DEVELOPMENT (DEV) branches for next version work. a. Work in DEV branches can be isolated by feature, organization, or temporary collaboration. b. Each DEV branch should be a full branch of MAIN.

2. Your servicing model is to have customers upgrade to the next major release. a. You are not supporting multiple versions in production at once. 3. Any fixes shipped from the release branch will include all previous fixes from that branch.	c. DEV branches should build and run Build Verification Tests (BVTs) the same way as MAIN. d. Forward Integrate frequently from MAIN to DEV if changes are happening directly on MAIN. e. Reverse Integrate from DEV to MAIN based on some objective team criteria (e.g. internal quality gates, end of sprint, etc.). RELEASE branch where you ship your major release from. a. RELEASE is a child branch of MAIN. b. Your major product releases from the RELEASE branch and then RELEASE branch access permissions may be set to read only to prevent changes from happening directly on the branch. c. Changes from the RELEASE branch merge (RI) to MAIN. This merge is one way. Once the release branch is created, MAIN may be taking changes for next version work not approved for the release branch d. Create new RELEASE branches for subsequent major releases if you require that level of isolation.

8.7.2 Standard Branch Plan

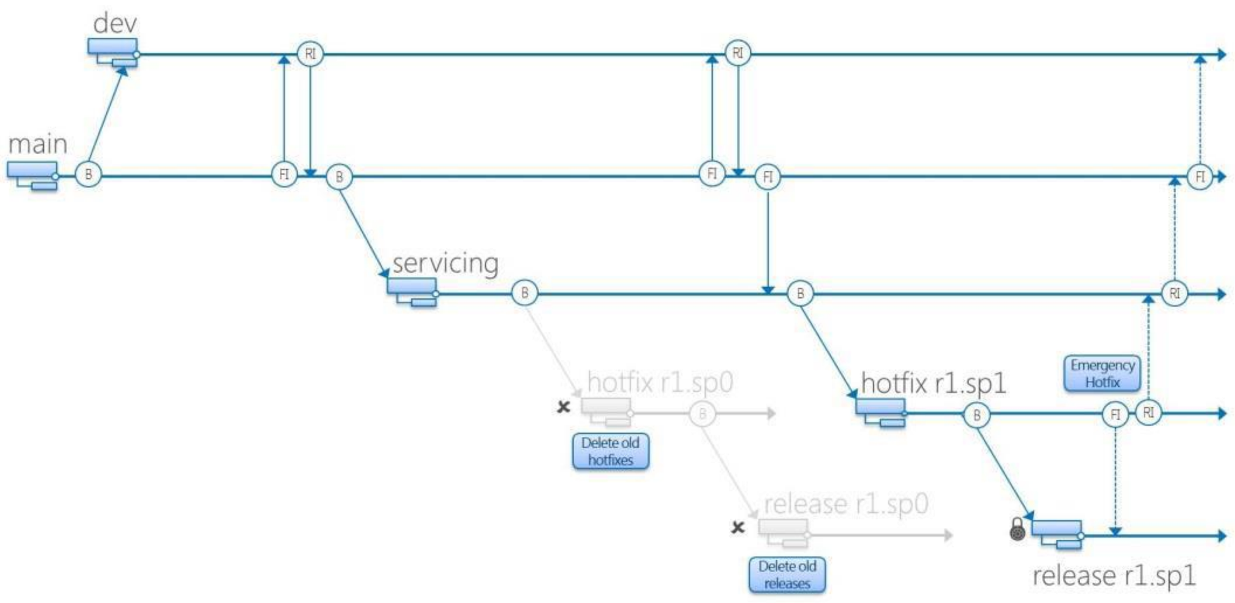


All of the guidance from the Basic plan applies to the Standard plan.

Table 4 : Standard Branch Plan Summary

Usage Scenarios	Considerations
1. You have multiple ship vehicles (e.g. major release and additional service packs for that release). 2. You want to enable concurrent development of service pack and next version products. 3. You have any compliance requirements that require you to have an accurate snapshot of your sources at release time.	1. RELEASE branches for release safekeeping and Service Pack work 2. RELEASE tree (i.e. SP and RELEASE) are branched from MAIN at the same time to create MAIN・SP・RELEASE parent/child relationship. 3. Product releases from the RELEASE branch and then that branch can be changed to read only. 4. Servicing changes are checked into the Service Pack (SP) branch. 5. Changes in SP branches merge one-way to MAIN (SP・MAIN). 6. Ship stopping bug fixes checked into the release branch should RI merge back to MAIN through the SP branch (SP・MAIN). 7. Duplicate RELEASE tree plan for subsequent major releases if you require that type of isolation.

8.7.3 Advanced Branch Plan



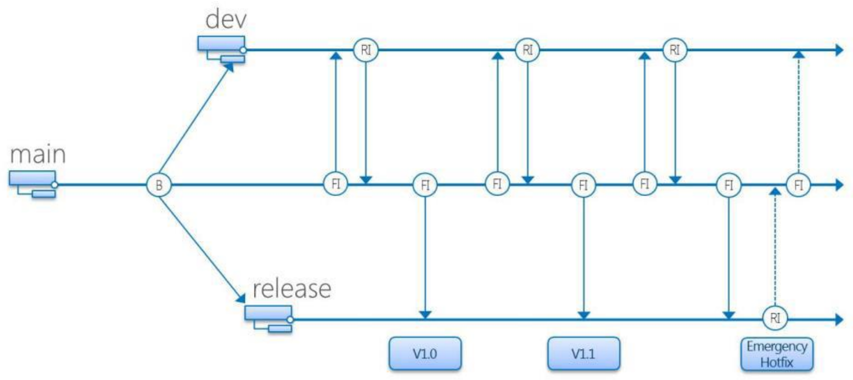
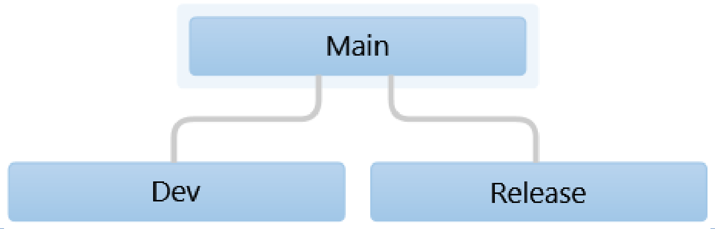
All of the guidance from the Basic and Standard plans applies to the advanced plan. Additional Considerations are,

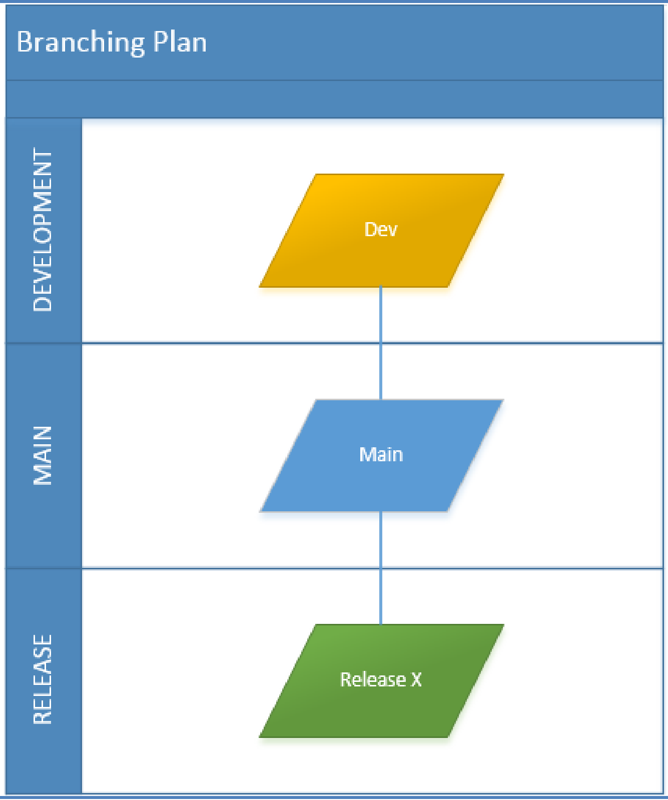
1. RELEASE branches for release safekeeping, HOTFIX and Service Pack work
2. RELEASE tree (i.e. SP, HOTFIX, and RELEASE) are branched from MAIN at the same time to create MAIN・SP・HOTFIX・RELEASE parent/child relationship.
3. After creating the RELEASE tree make any final ship stopping bug fixes in the RELEASE branch. Produce the build you will release from the RELEASE branch. After product release update the access permission in the RELEASE branch to read only if desired. This will ensure that you have the precise set of sources that made the shipping version of your product, and helps stop accidental changes from happening on a release branch.
4. Check-in based on which release the change applies to (e.g. Hotfixes are checked into the HOTFIX branch). Maintenance of additional branches is expense so try to consolidate hot fixes for a single release in the HOTFIX branch associated with a particular release. If a customer specific hot fix is required then branch just that single component into a customer specific HOTFIX branch. A separate customer specific HOTFIX branch should be very unusual.

5. Changes in HOTFIX and SP branches RI merge one-way through intermediate branches to MAIN (HOTFIX • SP • MAIN).
6. Since hot fixes merge from the HOTFIX branch to the SP branch any release from SP would include all release hot fixes + any changes made directly in the SP branch. Duplicate RELEASE branch plan for subsequent major releases.

8.8 Visualizing Branch Plans

Many a whiteboard session has been had in teams where the same branch plan has been drawn in a different manner. It is helpful if you standardize on visualization within your team so that you can all clearly interpret the Current areas of code churn and delivery vehicles. Here are three common visualizations showing the same plan.

Name	Diagram
Horizontal Timeline This is the most common visualization and provides a clear view of structure and milestones. It can often become cluttered though and some people do not always see the code flow paths.	 dev main release B RI FI V1.0 V1.1 Emergency Hotfix
Vertical Hierarchy This is the view presented by Team Foundation Server. It is a fairly basic generation and provides a clear view of code flow but little milestone information.	

Swim lanes Swim lanes provide a clear view of milestones and code flow paths. They are particularly beneficial for advanced and feature branch plans.	 Branching Plan DEVELOPMENT MAIN RELEASE Dev Main Release X

9. Who Does What?

Item	Developers	Team Leader	Build Initiator	Tech
User Configuration				✓
Solution Configuration				✓
Branching				
Main				✓
Development				✓
Release				✓
Others		✓	✓	✓
Merging				
Main		✓		✓
Development	✓	✓	✓	✓
Release		✓	✓	✓
Others	✓	✓	✓	✓
Check In				
Main		✓		✓
Development	✓	✓	✓	✓
Release		✓	✓	✓
Others	✓	✓	✓	✓
Check Out				
Main		✓		✓
Development	✓	✓	✓	✓
Release		✓	✓	✓
Others	✓	✓	✓	✓
Label				
Main				✓
Development		✓		✓
Release		✓	✓	✓