Software Configuration Management

Process & Procedures

Version 1.1

July 2014

Team D Solution Products are trademarks of Team D. Some product names mentioned in this guide may be trademarks of their respective companies and are hereby acknowledged.

This document is being delivered to you AS IS and Team D® Solutions, Inc. makes no warranty as to its accuracy or use. Any use of the technical documentation or the information contained therein is at the risk of the user. Documentation may include technical or other inaccuracies or typographical errors. Team D, Inc. reserves the right to make changes without prior notice. No part of this document may be reproduced, transcribed, or sold without prior express written consent of Team D, Inc.

© Copyright 2013 Team D, Inc. All Rights Reserved.

Team D, Inc.

Contents

TOC \o 2-3 \t "Heading, 4"

Revision History PAGEREF \_Toc \h 7

Introduction PAGEREF \_Toc1 \h 1

Software Configuration Management Responsibilities PAGEREF \_Toc2 \h 1

Definitions PAGEREF \_Toc3 \h 1

Team Foundation Server (TFS) PAGEREF \_Toc4 \h 2

Roles and Members PAGEREF \_Toc5 \h 2

Branching Plan PAGEREF \_Toc6 \h 2

Main PAGEREF \_Toc7 \h 3

Description PAGEREF \_Toc8 \h 3

Roles and responsibilities PAGEREF \_Toc9 \h 3

Builds PAGEREF \_Toc10 \h 3

Testing Procedures PAGEREF \_Toc11 \h 3

Merging Procedures PAGEREF \_Toc12 \h 4

Labels PAGEREF \_Toc13 \h 4

Retention PAGEREF \_Toc14 \h 4

Development PAGEREF \_Toc15 \h 5

Description PAGEREF \_Toc16 \h 5

Types of Branches PAGEREF \_Toc17 \h 5

Roles and responsibilities PAGEREF \_Toc18 \h 5

Builds PAGEREF \_Toc19 \h 6

Testing Procedures PAGEREF \_Toc20 \h 6

Merging Procedures PAGEREF \_Toc21 \h 6

Labels PAGEREF \_Toc22 \h 7

Retention PAGEREF \_Toc23 \h 7

Release PAGEREF \_Toc24 \h 8

Description PAGEREF \_Toc25 \h 8

Types of Branches PAGEREF \_Toc26 \h 8

Roles and responsibilities PAGEREF \_Toc27 \h 8

Builds PAGEREF \_Toc28 \h 9

Testing Procedures PAGEREF \_Toc29 \h 9

Merging Procedures PAGEREF \_Toc30 \h 9

Labels PAGEREF \_Toc31 \h 9

Retention PAGEREF \_Toc32 \h 10

Merges PAGEREF \_Toc33 \h 10

Workflow PAGEREF \_Toc34 \h 10

Step 1 – Integrate your workspace code with the Development branch PAGEREF \_Toc35 \h 10

Step 2 – Create a clean workspace for merging PAGEREF \_Toc36 \h 10

Step 3 – Update your MERGE workspace branches PAGEREF \_Toc37 \h 11

Step 4 – Merge Main branch to Development branch PAGEREF \_Toc38 \h 11

Step 5 – Merge Development branch to Main branch PAGEREF \_Toc39 \h 11

Frequency PAGEREF \_Toc40 \h 12

Labels PAGEREF \_Toc41 \h 12

Notifications PAGEREF \_Toc42 \h 13

Source Code PAGEREF \_Toc43 \h 14

Architecture PAGEREF \_Toc44 \h 14

Check-In Policies PAGEREF \_Toc45 \h 15

Merge Only in MAIN PAGEREF \_Toc46 \h 15

Changeset Comments Policy PAGEREF \_Toc47 \h 18

Code Review PAGEREF \_Toc48 \h 18

Shelving PAGEREF \_Toc49 \h 18

Build Verification Tests (BVTs) PAGEREF \_Toc50 \h 19

Version Management PAGEREF \_Toc51 \h 19

Version Format PAGEREF \_Toc52 \h 19

Identifiers and Categories PAGEREF \_Toc53 \h 19

Version Changes PAGEREF \_Toc54 \h 19

Determining Significance PAGEREF \_Toc55 \h 19

Usage PAGEREF \_Toc56 \h 19

TFS Architecture PAGEREF \_Toc57 \h 21

Source Code PAGEREF \_Toc58 \h 21

Version Tracking PAGEREF \_Toc59 \h 22

Build Version Management PAGEREF \_Toc60 \h 22

Text File Version Management PAGEREF \_Toc61 \h 22

Development PAGEREF \_Toc62 \h 23

Main PAGEREF \_Toc63 \h 24

Release PAGEREF \_Toc64 \h 24

Removal PAGEREF \_Toc65 \h 24

Staging Folder PAGEREF \_Toc66 \h 24

Change log PAGEREF \_Toc67 \h 24

Final Releases PAGEREF \_Toc68 \h 24

Build Architecture PAGEREF \_Toc69 \h 25

Build Definition PAGEREF \_Toc70 \h 26

Typical Configurable Options PAGEREF \_Toc71 \h 26

Templates PAGEREF \_Toc72 \h 27

Names PAGEREF \_Toc73 \h 27

Process File PAGEREF \_Toc74 \h 27

Retention Policy PAGEREF \_Toc75 \h 28

How to Clone a Build Definition PAGEREF \_Toc76 \h 28

InstallShield Standards PAGEREF \_Toc77 \h 28

CM Tools PAGEREF \_Toc78 \h 29

Assembly Binding Log Viewer PAGEREF \_Toc79 \h 29

Release Checklist PAGEREF \_Toc80 \h 31

New Application Checklist PAGEREF \_Toc81 \h 31

Supplementary Information PAGEREF \_Toc82 \h 1

Full Branching Plan Image PAGEREF \_Toc83 \h 1

Development Build Definition Screenshots PAGEREF \_Toc84 \h 2

# Revision History

|  |  |
| --- | --- |
| Date of Release | Revisions |
|  | Draft |
| 7/3/14 | Added Final Releases and updated version notes. |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

# Introduction

## Software Configuration Management Responsibilities

The Software Configuration Management Process attempts to achieve the following:

1. Facilitates Timely Communication
2. Enforces Development Policies
3. Ensures Technical Standards
4. Manages Hand-offs between Environments and Teams

In the end, the Software Configuration Management Process needs to ensure integrity of the software it supports.

It also needs to integrate with all the other standards and processes that surround the software and the business.

# Definitions

|  |  |
| --- | --- |
| **Term** | **Definition** |
| **TFS** | Team Foundation Server is a Microsoft product used by Team D for source control, data collection, reporting, and project tracking, and is intended for collaborative software development projects. |
| **DEVELOPMENT branch** | Changes for next version work. |
| **MAIN branch** | This branch is the junction branch between the development and release branches. This branch represents a stable snapshot of the product that can be shared with QA or external teams. |
| **HOTFIX** | A change to fix a specific customer-blocking bug or service disruption. |
| **RELEASE branch** | A branch where high priority or escalated bug fixes are made before major product release. After new product release the previous release branch becomes read-only. |
| **FORWARD INTEGRATE (FI)** | Code merges from parent to child branches. |
| **REVERSE INTEGRATE (RI)** | Code merges from child to parent branches. |

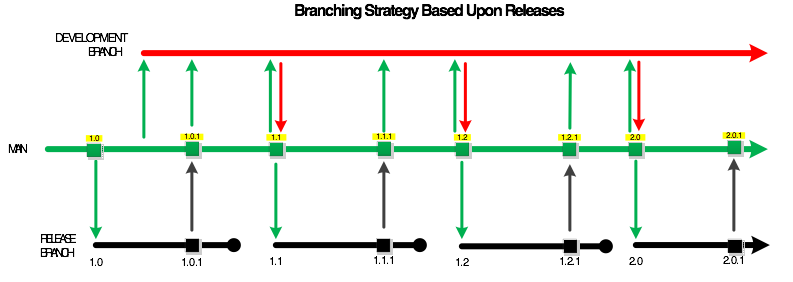
# Team Foundation Server (TFS)

## Roles and Members

|  |  |  |
| --- | --- | --- |
| **Role** | **Description** | **Group** |
| **Team Foundation Administrator** | Can perform all operations for Team Foundation Server. | Software Configuration Manager Engineers |
| **Contributor** | Members of this group can contribute to the project in multiple ways, such as add, modify, and delete code, create and modify work items, and so on. | Development, QA |
| **Reader** | Members of this group can view the project but not modify it. | Consultants, Support |
| **Build Service** | Members of this group have build service permissions for the project. This group should only contain build service accounts and not user accounts or groups. | Build Service Account |

# Branching Plan

Team D utilizes a basic branching plan that enables for concurrent development of new releases, a stable MAIN branch for testing and a release branch for addressing high priority hot fixes.



Full image of Branching Plan



## Main

### Description

Main is primarily for new development QA and which also contains the complete set of all released changes. Once the RELEASE branch is created MAIN and the DEVELOPMENT branches can start taking changes approved for the next product release.

### Roles and responsibilities

#### Developer

Developers merge changes from and to the MAIN branch based upon the type of merge (FI, RI) needed. Changes should only be made the MAIN branch directly in an emergency situation and only by a designated Integration Developer. Integration Developers can also rollback a changeset from the MAIN branch.

#### Software Configuration Management Engineer

Software Configuration Management Engineers can apply labels, make changes directly to an InstallShield project file and create build definitions. Software Configuration Management Engineers are the only members of the team that can branch from MAIN to a DEVELOPMENT or RELEASE branch.

#### QA

QA receives builds from the branch but does not directly edit the branch.

### Builds

Builds are made for the MAIN branch to complete the Build Verification Tests (BVTs) and to create an install for QA to test. This install will contain development (features/fixes) as well as the latest hot fixes. An automated build will be started whenever a change is made to MAIN.

Please refer to the Builds section for more information.

### Testing Procedures

QA will receive a notification providing information on what has changed and where to locate the software to test.

### Merging Procedures

#### From DEVELOPMENT to MAIN (RI)

On each successful build of a DEVELOPMENT branch a Reverse Integration (RI) Merge should be completed into MAIN. Prior to merging into MAIN a Forward Integration (FI) merge should be accomplished to ensure the latest code is being used. This is completed by the designated Integration Developer working in the DEVELOPMENT branch.

#### From MAIN to DEVELOPMENT (FI)

After each merge from RELEASE to MAIN a FI Merge into DEVELOPMENT should be completed to obtain the latest fixes. This is completed by any Developer working in the DEVELOPMENT branch.

#### From RELEASE to MAIN (RI)

On each successful build and release from the RELEASE branch a Forward Integration (FI) Merge should be completed. Only after the notification from the QA Manager that a hotfix version has been released should a RELEASE merge occur. This is completed by the designated Integration Developer working in the RELEASE branch.

Please refer to the Merges section for more information.

### Labels

Labels are to be applied by the Software Configuration Management Engineer prior to each RELEASE branch on MAIN and after each RELEASE merge on a RELEASE branch.

Please refer to Labels section for more information.

### Retention

The MAIN branch is retained indefinitely.



## Development

### Description

Multiple development areas are supported by creating additional development branches from MAIN. These are peers to each other and children of MAIN. Once the release branch is created MAIN and the development branches can start taking changes approved for the next product release.

### Types of Branches

There are three reasons for creating a DEVELOPMENT branch:

1. New version development
2. Feature specific development
3. Proof of Concept (POC)

Each branch will be named after its purpose either by name or version.

### Roles and responsibilities

#### Developer

All new code is applied directly to the DEVELOPMENT branch. Developers RI merge changes to the MAIN branch and obtain updates from the MAIN branch by performing a FI merge.

Please refer to the merging section for more merging information.

#### Software Configuration Management Engineer

Software Configuration Management Engineers create build definitions that conduct BVTs. Software Configuration Management Engineers are the only members of the team that can branch from MAIN to a DEVELOPMENT branch.

#### QA

QA does not actively participate in the DEVELOPMENT branch. All development features/fixes are obtained from MAIN.

### Builds

#### New Version / Feature

Builds are made for the DEVELOPMENT branch to complete the Build Verification Tests (BVTs). An automated build will be started whenever a change is made to DEVELOPMENT and a summary will be provided to the developers on the pass/fail status of the BVTs. This will provide the information needed to know if a FI Merge is appropriate for the MAIN branch.

#### Proof of Concept (POC)

Builds are made for the POC branch to complete the Build Verification Tests (BVTs) and to create an install for QA to test. This install will contain development POC features. An automated build will be started whenever a change is made to the POC branch. An install will be created as it would in MAIN and placed in the QA share appropriately.

Please refer to the Builds section for more information.

### Testing Procedures

#### Development

QA does not actively participate in the DEVELOPMENT branch. All development features/fixes are obtained from MAIN.

#### POC

QA will participate in testing the POC development work as determined by the QA Manager.

### Merging Procedures

#### From DEVELOPMENT to MAIN (RI)

On each successful build of a DEVELOPMENT branch a Reverse Integration (RI) Merge should be completed. Unsuccessful builds should not be merged into MAIN for any reason. Prior to merging into MAIN a Forward Integration (FI) merge should be completed to ensure the latest code is being used. RI Merges should not be completed until a successful build and all BVTs have been passed in the DEVELOPMENT branch. This is completed by the designated Integration Developer working in the DEVELOPMENT branch.

Please refer to the merging section for more merging information.

#### From MAIN to DEVELOPMENT (FI)

After each merge from RELEASE to MAIN a FI Merge into DEVELOPMENT should be completed to obtain the latest fixes. This is completed by any Developer working in the DEVELOPMENT branch.

#### From POC to MAIN (RI)

Only after the Product Owner, Development Manager and QA Manager approve the Proof of Concept work can it be RI merged into the MAIN branch. Unsuccessful builds should not be merged into MAIN for any reason. Prior to merging into MAIN a Forward Integration (FI) merge should be completed to ensure the latest code is being used. RI Merges should not be completed until a successful build and all BVTs have been passed in the POC branch. This is completed by the designated Integration Developer working in the POC branch.

Please refer to the Merges section for more information.

### Labels

Labels are not applied to the DEVELOPMENT branch.

Please refer to Labels section for more information.

### Retention

The DEVELOPMENT branch is retained during the scheduled development cycle. Once the version has been released the DEVELOPMENT branch is deleted.

Feature or version specific DEVELOPMENT branches are removed once the code has been released and branched into a Release branch. This is considered a finite timeline branch.

General DEVELOPMENT branches, commonly used for small projects with few changes, are not deleted after a version has been released. This is considered an infinite timeline branch. Branches of this nature are deleted and recreated as needed. It is common to rebuild at least once a year.



## Release

### Description

Additional releases are supported by creating additional release branches for each product release. Each release branch is a child of MAIN and a peer to each other (e.g. release 2.0 branch is peer to release 3.0 and both are children of MAIN). Once the release branch is created MAIN and the development branches can start taking changes approved for the next product release.

### Types of Branches

There are two reasons for creating a RELEASE branch:

1. Version release
2. Isolated Hotfix (Child of RELEASE)

Each branch will be named after its purpose either by version (1) or name (2).

### Roles and responsibilities

#### Developer

Developers work directly on the release branch, making very selective changes based on priority one issues identified by the business. Once the changes are complete and have passed QA the developer will RI to the Main branch.

#### Software Configuration Management Engineer

Software Configuration Management Engineers can apply labels, make changes directly to an InstallShield project file and create build definitions. Software Configuration Management Engineers are the only members of the team that can branch from MAIN to a RELEASE branch.

#### QA

QA receives builds from the branch but does not directly edit the branch.

### Builds

#### Version

Releases from the RELEASE branch will be a third revision bump from the previously scheduled release.

#### Isolated Hotfix

Isolated hotfix releases will be a third revision bump from the previously scheduled release.

Please refer to the Builds section for more information.

### Testing Procedures

#### Version

QA will receive a notification providing information on what has changed and where to locate the software to test.

#### Isolated Hotfix

QA will receive a notification providing information on what has changed and where to locate the software to test.

### Merging Procedures

#### From RELEASE to MAIN (RI)

After each merge from RELEASE to MAIN a FI Merge into DEVELOPMENT should be completed to obtain the latest fixes. This is completed by any Developer working in the DEVELOPMENT branch.

Please refer to the Merges section for more information.

#### From Isolated Hotfix to Release (RI)

After the hotfix has been verified by QA a RI into RELEASE should be done, with subsequent RIs to propagate the change from RELEASE TO MAIN TO DEVELOPEMENT. Getting the change into MAIN is the responsibility of the developer that has completed the hotfix. Getting the change into DEVELOPEMENT is completed by any Developer working in the DEVELOPMENT branch.

Please refer to the Merges section for more information.

### Labels

Labels are to be applied by the Software Configuration Management Engineer prior to each RELEASE branch on MAIN and after each RELEASE merge on a RELEASE branch.

Please refer to Labels section for more information.

### Retention

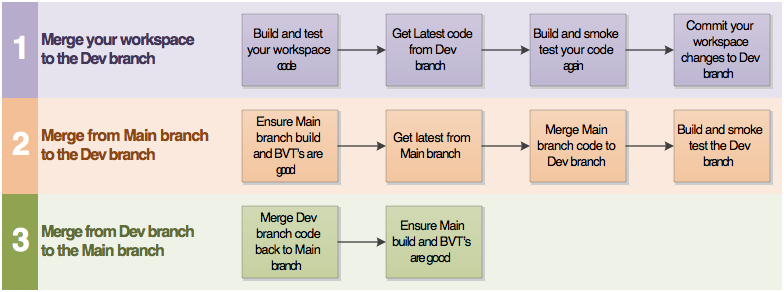
Up to three release branches for a particular application should remain in source control at any time. Once a fourth is created the oldest should be deleted as releasing hotfixes for that version is no longer supported.

# Merges

This section provides a summary of how to integrate workspace source code to development and main branches. Although it focuses on the development branch that has multiple developers committing source code simultaneously, the following steps are also applicable to personal development branch merging as well.

## Workflow

Following is a high level work flow for integrating your workspace code to the main branch



### Step 1 – Integrate your workspace code with the Development branch

After testing your workspace code and unit tests, get the latest code from the Development branch and test your workspace code once more before committing. If the development branch has automated tests and BVT’s, ensure none of those fail after you commit.

### Step 2 – Create a clean workspace for merging

*Note: Skip this step if you are integrating a personal branch which only you have committed code to.*

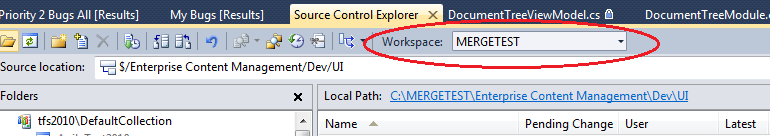
If you are integrating a development branch that has multiple developers committing code simultaneously, it is recommended to set up a separate clean workspace for merging the code to the main branch. This will limit mistakes and unintended check-ins.

[Learn how to create a workspace](http://msdn.microsoft.com/en-us/library/gg490753.aspx)

### Step 3 – Update your MERGE workspace branches

*Note: Skip this step if you are integrating a personal branch which only you have committed code to.*

3.1 Ensure you are working in your “Merge” workspace!



3.2 Ensure the Development and main branch current builds are good. (Check automated build status if available)

3.3 Get the latest code from the Development branch and build and smoke test the application.

3.4 Get the latest code from the Main branch.

### Step 4 – Merge Main branch to Development branch

4.1 Once you have the latest Main branch code, merge the Main branch to the Development branch by right clicking on the Main branch in *Source Control Explorer* and selecting *Branching and Merging Merge*. Follow the directions in the Merge Wizard.

4.2 Commit changes.

4.3 After committing the changes ensure the Development branch code is good. (Smoke Test, check for automated build status, etc.)

### Step 5 – Merge Development branch to Main branch

5.1 Ensure you don’t have any files checked out to avoid unintended commits when accepting the merge changes. Merge the Development branch to the Main branch by right clicking on the Development branch in *Source Control Explorer* and selecting *Branching and Merging Merge*. Follow the directions in the Merge Wizard.

5.2 Before committing the merge changes, pay close attention to the files being merged. For instance, a red flag could be a configuration file. In this case you will need to verify that the changes are intended and not a temporary change to have an application run locally for debugging.

If you are merging a branch with commits from multiple developers, do not merge any files to Main unless you are sure that they are OK to merge. When in doubt, contact the developer responsible for the change set.

5.3 Commit merge changes.

5.4 Ensure the Main build is good (BVT’s and automated build are green)

## Frequency

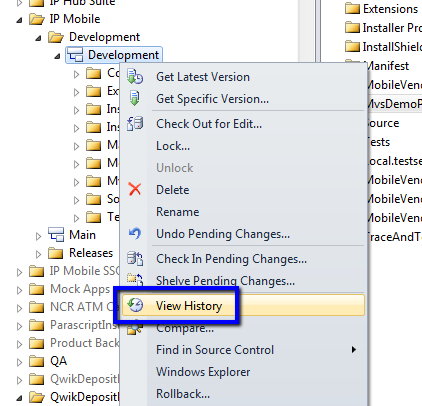
Developers should Forward Integrate (Merge (FI) with each successful build of MAIN. Reverse Integrate (RI) merges occur based on project objective criteria (e.g. feature complete, end of sprint, etc.).

# Labels

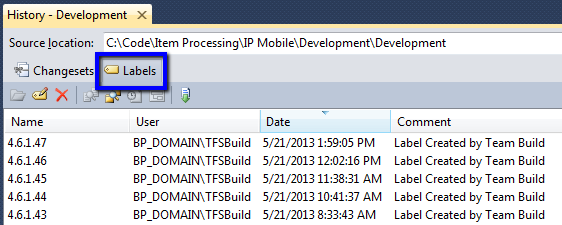
Labels are applied to versions not files. Labels are version-specific in that you can only attach a given label to one version of a file or folder. Files and versions can have multiple labels associated with them. Labels are not pending changes that must be committed with the Check In command. When you issue the Label command, the appropriate updates are immediately reflected in the Team Foundation source control server.

At each build, regardless if it is automated or manual, a label should be applied to the source with the full version name (Major.Minor.Release.Build). A branch can then be created at a later time by label ensuring the correct versions of the files are branched. A

Versions can be retrieved by right clicking on the desired branch and clicking on View History.



Once the tab opens click on the Labels button to display the labels that are associated with that branch. The number of Team Build created labels that are saved is setup in the Retention Policy section of the build definition.

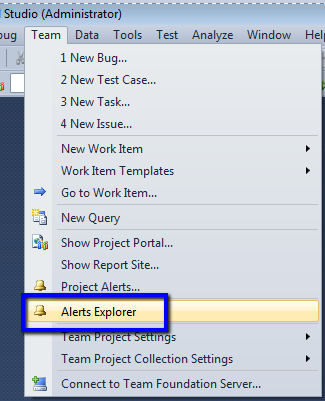


# Notifications

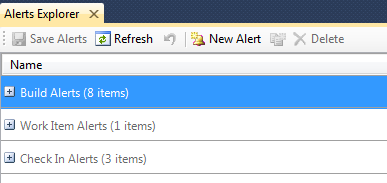
As changes occur to work items, source control files, and builds, you can receive email notifications for alerts that you define. All of R&D and QA should be notified when builds complete so they can take the appropriate action based on the build results. CM team members should be alerted to Check Ins so they can monitor the content going into the product branches.

In TFS 2010 the creator of an alert is the only person that can edit or delete it. In order to avoid having notifications that can’t be edited when a creator is unavailable, CM has setup a TFSAdmin domain user to be used when creating them.

To set up notifications log into a windows machine as TFSAdmin and open Visual Studio 2010 (PowerTools should be installed). Navigate to the Team tab on the ribbon menu and click on Alerts Explorer.



Here you can add alerts for any team member or domain group.

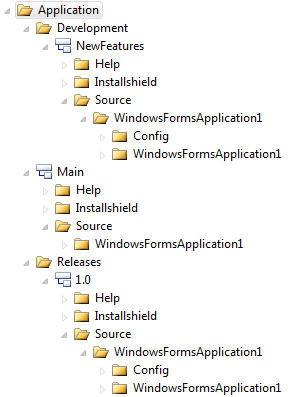


# Source Code

## Architecture

Within each code branch the following directories must be maintained:

|  |  |
| --- | --- |
| **Folder** | **Details** |
| **Help** | Contains the CHM file used in the application and installation. Maintained by the Technical writer. |
| **Installshield** | Contains the InstallShield project file. Maintained by the Software Configuration Management Engineer. |
| **Source** | Contains the source code for the application. Organization of this folder is managed by the Developer. |
| **Source\<Application>\Config** | A master copy of the application configuration file is kept in this folder and maintained by the Developer. The Developer is responsible for keeping the current features and master copy of the configuration in sync. This file is used in the installation and provided to the customer. |

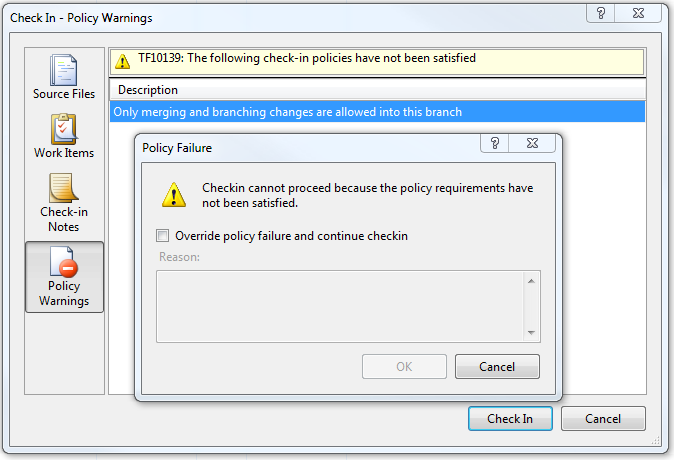


## Check-In Policies

### Merge Only in MAIN

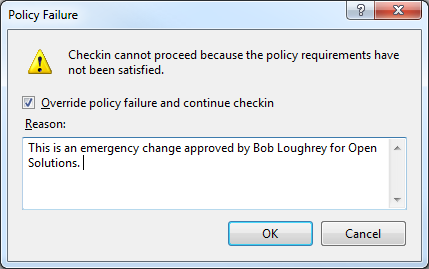
This document is in regards to point #3 (Merge Encouragement).  It is typically safer to merge your code changes rather than editing the branch directly.  In order to encourage our developers to use the merging method a check-in policy is being implemented that will prompt you for more information if a direct edit is attempted.

If you attempt to directly edit any files in the MAIN branch, upon check-in you will receive the following message.



#### Policy Override

It is possible to override this policy but you will need to provide a reason why.  This explanation will be sent to the team and management.  Acceptable reasons are that it is an emergency change approved by Management or there is a technical impediment that cannot be resolved within an acceptable schedule.  To override the policy place checkmark in “Override policy failure and continue checkin”, enter the reason why for the override and press the OK button.



#### Notifications

A notification will be sent out with your check-in comment and the policy override reason.

[**Changeset 34662: Updated version.**](http://tfs2010:8080/tfs/web/cs.aspx?pcguid=33d3f91e-acc9-41b9-8744-fc2ed904235a&cs=34662)

**Summary**

|  |  |
| --- | --- |
| Team Project(s): | Test Team Project |
| Checked in by: | DRAYMOND |
| Checked in on: | 4/22/2013 8:29:07 AM |
| Code Reviewer: | None |
| Performance Reviewer: | None |
| Security Reviewer: | None |
| Comment: | Updated version. |
| Policy Override Reason: | This is an emergency change approved by Bob Loughrey for Open Solutions. |

**Policy Failures**

|  |  |
| --- | --- |
| Type | Description |
| Custom Path Policy | Only merging and branching changes are allowed into this branch |

**Items**

|  |  |  |
| --- | --- | --- |
| Name | Change | Folder |
| [AssemblyInfo.cs](http://tfs2010:8080/tfs/web/diff.aspx?pcguid=33d3f91e-acc9-41b9-8744-fc2ed904235a&opath=%2524%252fTest+Team+Project%252fMain%252fSource%252fConsoleApplication1%252fConsoleApplication1%252fProperties%252fAssemblyInfo.cs&ocs=34661&mpath=%2524%252fTest+Team+Project%252fMain%252fSource%252fConsoleApplication1%252fConsoleApplication1%252fProperties%252fAssemblyInfo.cs&mcs=34662) | edit | $/Test Team Project/Main/Source/ConsoleApplication1/ConsoleApplication1/Properties |

Notes:  
- All dates and times are shown in GMT-07:00:00 Pacific Daylight Time  
- You are receiving this notification because of a subscription created by BP\_DOMAIN\DRAYMOND  
Provided by [Microsoft Visual Studio® Team System 2010](http://go.microsoft.com/fwlink/?LinkID=129550)

### Changeset Comments Policy

Always begin the check-in comment with the user story ID or TFS Bug Number that the work is being done for.

Example: US4407: Changed logic to not show the Optional field & prompt when a container is free form, or it is not a container (i.e. a dosfile/etc).

If you are making the same change to multiple branches (DEV & MAIN) be sure to use the same comment for each branch. Do not use “Merged from DEV” as the DEV branch will be deleted and rebuilt.

### Code Review

Code review policies are defined and enforced by the Development team. It is their responsibility to schedule code reviews when the work warrants it and in a timeframe that makes sense for all developers involved.

## Shelving

Shelving can be used as an effective alternative to checking code into a branch. Some suggested uses are:

**Context Switching**: Saving the work on your current task so you can switch to another high priority task. If priority shifts in the middle of a coding effort, say due to a priority 1 bug, you can shelve your work, fix the bug, then come back and unshelve to work on your changes later.

**Sharing Changesets**: If you want to share a changeset of code without checking it in, you can make it easy for others to access it by shelving it.  This could be especially helpful during the code review process as another developer could review the code prior to it going into a branch.

**Saving your progress**: While you're working on a complex feature, you may find yourself at a 'good point' where you would like to save your progress. This is an ideal time to shelve your code. It is also a backup since the code gets saved in source control rather than on your computer’s hard drive.

If for some reason other members of the team would need to get your changes and work on them they would have accessible through source control.

Perhaps you’re making a change and want to start over but not lose all the code that that you’ve written so far. Having a shelved version of the code would allow you to do so.

# Build Verification Tests (BVTs)

A Build Verification Test (BVT) is a set of tests run on each new build of a [product](http://en.wikipedia.org/wiki/Product_(business)) to verify that the build is testable before the build is released into the hands of the QA team, a failed BVT deems the build untestable. BVTs include unit tests created by developers and automated functional tests created by QA personal. Their execution should be a part of the automated build process.

# Version Management

Software versioning is the process of assigning either unique version names or unique version numbers to unique states of computer software. This document will explain the process and standards for defining versions within Team D.

## Version Format

### Identifiers and Categories

Team D versions its software and files in four-sequence identifiers: [99].[99].[99].[999]. Each of these identifiers applies to a release category: [major].[minor].[hotfix].[build].

Sequence-based identifiers are used to convey the significance of changes between releases. Changes are classified by significance level, and the decision of which sequence to change between releases is based on the significance of the changes from the previous release.

Categories are used to explain to the customers what type of software release is being deployed.

## Version Changes

### Determining Significance

The sequence to increment is determined the significance of the change. The first sequence may be incremented when the code is completely rewritten or a high priority core feature is implemented resulting in a major release. A change to the user interface or the documentation may only warrant a change to the second sequence thus resulting in a minor release.

A change to the third sequence occurs when a bug fix is implemented to a previously released version of the software. The first and second sequences are not changed in this instance resulting in a hotfix release.

The Product Owner, Development Manager and QA Manager influence the decision on the weight of significance.

## Usage

The internal technical teams (Research & Development, Support and Professional Services) utilize all 4 sequences for reference in troubleshooting specific issues or in the SDLC process.

In customer facing situations (Documentation, Marketing, Sales) only the first 3 sequences are used and most likely the Major/Minor release category name is referenced.

Example:

Team D Internal: [major].[minor].[hotfix].[build]

5.4.0.8

Customer: [major].[minor].[hotfix]

5.4.0

## TFS Architecture

### Source Code

The source code is managed and branched according the Team D Branching Strategy. Each release branch is created based upon a Major or Minor release.

 Product Name

 Development

 <Feature Name>

 <POC>

 Development

 Main

 Releases

 1.0

 1.1

 2.0

## Version Tracking

There are two methods to tacking versions. One is to let the build definition determine the version. The second is to store the version number in a text file.

### Build Version Management

In the Advanced section in the default Team D TFS 2013 build template (TfvcTemplate.12.Team D\*.xaml) you configure the build number to match what the version of the application(s) will be. The build number is automatically incremented based upon how many builds are completed. All files that are *compiled* will be versioned the same as the build number.



### Text File Version Management

Each application’s version is tracked individually and separated by branch in text file named Version.

TFS Team Project / Versions

 Product Name

 Development

 <POC>

 Version.txt

 Development

 Version.txt

 Main

 Version.txt

 Release

 1.0

 Version.txt

 1.1

 Version.txt

 2.0

 Version.txt

### Development

#### Features

The <Feature Name> branch version will consist of identifying with the current or scheduled Major release number followed by a decremented number in the second sequence (“88”. “77”, etc.) and “99” for the third sequence. The decremented number in the third sequence should be significant and not duplicated by any other feature branch builds for that specific code branch.

Example: 2.88.99.[build]

Any deployments utilizing this numbering scheme are not susceptible to a service contract and thus should not be generating bug reports. Members of the QA team are also not to be testing software using these versions as they have not been merged into the main code branch and run through all automated tests.

#### Proof of Concept

The <POC> branch may not be merged into the Main code branch until determined by the Product Owner, Development Manager or QA Manager. Thus it is treated like a parallel branch to Main containing a version number, build script and install until it has been decided to be part of the application and then takes on the Main branch version number. Once the decision to merge into the Main branch is made the POC version text file is removed. The POC branch version will consist of all nines in the first three sequence identifiers.

Example: 99.99.99.[build]

Any deployments utilizing this numbering scheme are not susceptible to a service contract and thus should not be generating bug reports.

#### Development

The Development branch version will consist of identifying with the current or scheduled Major release number followed by “99” for the second and “99” for the third sequences.

Example: 2.99.99.[build]

Any deployments utilizing this numbering scheme are not susceptible to a service contract and thus should not be generating bug reports. Members of the QA team are also not to be testing software using these versions as they have not been merged into the main code branch and run through all automated tests.

#### New Development done in Releases

The Releases development branch version will consist of identifying with the current or scheduled Major and Minor release numbers followed by “99” for the third sequence.

Example: 2.5.99.[build]

This should be a rare occasion and only done in emergencies.

### Main

The Main branch contains the version that is currently in a QA phase and the current SDLC. As each Major and Minor release is made the Main version is incremented to accommodate the next scheduled version and the build number is reset.

For example, if the most recent release of the application is 2.0 and the Main version is currently set to 2.0.0.25 then it will be updated to 2.1.0.0 to accommodate the next minor change as per the release calendar schedule.

Note: Prior to QA receiving any builds the Product Owner, Development Manager or QA Manager must provide the release calendar and version number for the next scheduled release.

### Release

As each Major and Minor release is made, a copy of the Main version is placed in a directory named after the Major/Minor version. The third sequence (hotfix) is incremented once and the fourth sequence (build) is reset to 0. As hotfixes go through the SDLC the fourth sequence (build number) will increment but the first three sequences will not change.

The third sequence number will be incremented after each hotfix release and the fourth sequence (build) is reset to 0.

### Removal

As each source code branch is deleted so should the accompanying Version file be deleted.

## Staging Folder

The Staging folder is placed in the R&D network share in a folder named after the 4 digit version.

\\Team D.com\Departments\R&D\QA\<Application>\<Branch>\<4 Digit Version>\<Install Type>

Ex. \\Team D.com\Departments\R&D\QA\IP\IP Admin\MAIN\5.0.7.7\EXE

## Change log

The change log is placed within the 4 digit version number folder within the staging folder.

\\Team D.com\Departments\R&D\QA\<Application>\<Branch>\<4 Digit Version>\< Application >\_Changelog\_<4 Digit Version>.txt

Ex. \\Team D.com\Departments\R&D\QA\IP\IP Admin\MAIN\5.0.7.7\Hub Admin\_Changelog\_5.0.7.7.txt

The changelog contains the changeset details from TFS from the last build.

## Final Releases

The Final Releases directory structure will be configured as outlined in the diagram below. This will allow our Technical services to easily deploy the latest version with the included hotfixes.

* \\Team D.com\Software\Final Releases
  + <Application Name>
    - <Major.Minor Version>
      * BETA
        + <Major.Minor.Hotfix>

Documentation

Software

<Application Major.Minor.Hotfix>

<Application Major.Minor.Hotfix>

Tools

Unchanged applications

<Application Major.Minor.Hotfix>

<Application Major.Minor.Hotfix>

* + - * Documentation
      * Software
        + <Application Major.Minor>
        + <Application Major.Minor>
      * Tools
      * Hotfixes
        + <Major.Minor.Hotfix>

Documentation

Software

<Application Major.Minor.Hotfix>

<Application Major.Minor.Hotfix>

Tools

Unchanged applications

<Application Major.Minor.Hotfix>

<Application Major.Minor.Hotfix>

# Build Architecture

<INSERT DIAGRAM>

Servers

Naming convention

Responsibility Matrix (RACI)

Types

TFS

Build

Test

File Share (Drop Locations)

Resource Monitoring

Types

Disk Space

CPU, RAM, Bandwidth, etc.

How do we get notifications?

Who gets notifications? (See RACI chart)

Who do we report issues to? (See RACI chart)

# Build Definition

A build definition contains your instructions about which code projects to compile, which additional operations to perform, and how to perform them.

## Typical Configurable Options

Here are the configurable options that are different per branch type. These are guidelines for creating a build definition. Slight changes are allowed but adherence to the standard template is encouraged.

|  |  |  |
| --- | --- | --- |
| **Group** | **Field** | **Value** |
| **General** | Definition Name | <PRODUCT> -- <Branch>  Ex. IP Capture -- POD |
| **Trigger** | Check-in Trigger | * Development: Continuous Integration * Main: Gated Check-in * Release: Gated Check-in |
| **Workspace** | Source Control Folder  $/Item Processing/IP Hub Suite/**Development/POD**/Common | Build Agent Folder  $(SourceDir)\Common |
| **Process** | Delete Build Directory | True |
| **Process** | Clean Workspace | All |
| **Process** | Staging Sub-Folder | <BRANCH>  Ex. POD |
| **Process** | Changelog Source Directory | <TFS Path>  You may want to collect information on just the application or a path that contains multiple potential changes. Ex. Application directory and Common directory. |
| **Process** | VersionFileLocation | <TFS Path>  Do not specify the filename, just the path. |

Development Build Definition Screenshots

## Templates

Templates of the build definitions are currently maintained in the **Item Processing** team project.

* Application-DEV-Feature
* Application-Main
* Application-Release-5.0

## Names

The names of the build definitions should be succinct and its purpose easily understood.

<Application>-<Type>-<Branch>

Ex.

QwikDeposit Home-Main

IP Mobile-Release-4.4

The maximum length of a build number is 64 characters. Characters which are not allowed include '"', '/', ':', '<', '>' , '\', '|', '?', and '\*'. The build definition name is included in the 64 characters.

## Process File

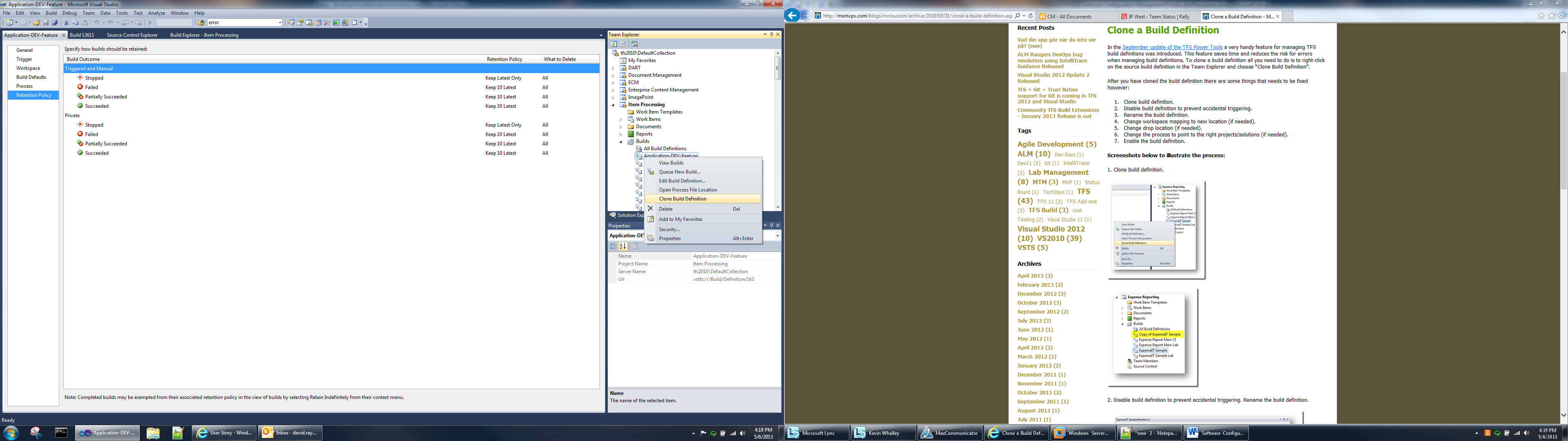
All build definitions should be able to use the same process file as long as the configurable options are properly set.

## Retention Policy

|  |  |  |
| --- | --- | --- |
| **Build Outcome** | **Retention Policy** | **What to Delete** |
| **Stopped** | Keep Latest Only | All |
| **Failed** | Keep 10 Latest | All |
| **Partially Succeeded** | Keep 10 Latest | All |
| **Succeeded** | Keep 10 Latest | All |

## How to Clone a Build Definition

When a new build definition is needed it is recommended that you clone one of the templates rather than creating new. The feature to clone a build definition is included in the [Team Foundation Server Power Tools](http://visualstudiogallery.msdn.microsoft.com/c255a1e4-04ba-4f68-8f4e-cd473d6b971f/). This feature saves time and reduces the risk for errors when managing build definitions. To clone a build definition all you need to do is to right-click on the source build definition in the Team Explorer and choose “Clone Build Definition”.



# InstallShield Standards

Template

Product name standards

Team D – {Application Name}

Application architecture

Windows Service

Web Service

Thick Client Application

Thin Client Application

Deployment Types

EXE

MSI

CAB

Digital Signatures

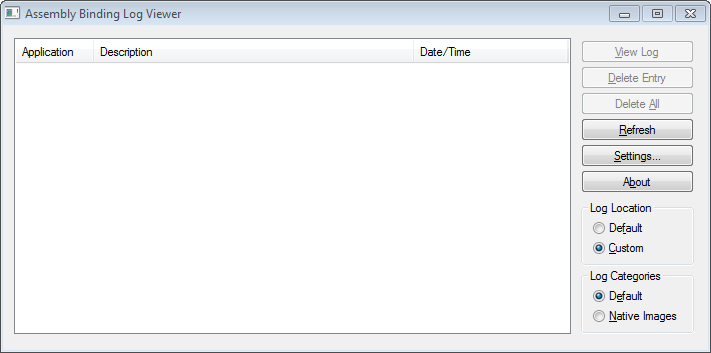
# CM Tools

## Assembly Binding Log Viewer

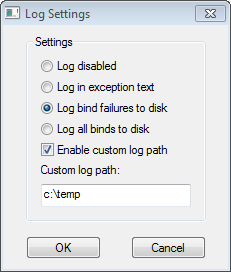
The Assembly Binding Log Viewer displays details for assembly binds. This information helps you diagnose why the .NET Framework cannot locate an assembly at run time. These failures are usually the result of an assembly deployed to the wrong location, a native image that is no longer valid, or a mismatch in version numbers or cultures. The common language runtime's failure to locate an assembly typically shows up as a [TypeLoadException](http://msdn.microsoft.com/en-us/library/system.typeloadexception.aspx) in your application.

From the VS Command Prompt enter the following; fuslogvw (Visual Studio should be closed)

The follow window appears, have the following settings selected. Click on the Settings button.



Select the following.



Launch Visual studio and open your xaml.  Log files with be in your log folder; C:\temp\Default\devenv.exe

# Release Checklist

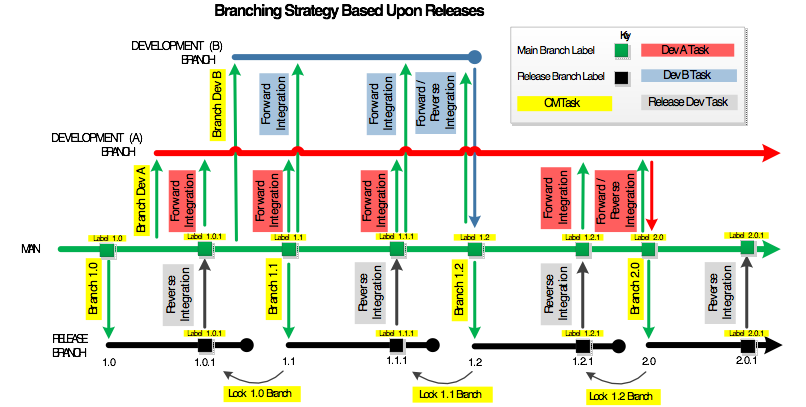
* + Create a label based off the release candidate identified by QA
  + Create a new "Release" branch from the "Main" branch based on that label (or changeset)
* Please see the Release section for more information.
* Create a Version.txt file in $/TFS/Versions under the appropriate application branch folder
* Clone the build definition for the Main branch build
  + Edit the Workspace tab to point to the new Release branch code
  + Edit the Process tab (the Release branch build definition should be able to use the same Build process file as the Main branch)
* Update Projects to build to the Release branch solutions
* Update Version file to look at the new Version.txt file
* Update any other references to the Main branch

# New Application Checklist

* Create a folder under the team project with the name of the application
* Under this new folder create three more:
  + Development
  + Main
  + Releases
* Convert the Main folder into a branch. This is where developers will put their code.
* Once developers have gotten their code into source control, branch from Main into the Development folder, also call the new branch Development.
* Under the BuildProcessTemplates folder under the team project, find an application build that has similar requirements and make a clone of it. Make sure to edit any application specific settings, such as:
  + Staging Area
  + ISMName
  + Which InstallShield release to build
  + Which files to copy to the drop location
* Create a Version.txt file in $/TFS/Versions under the appropriate application branch folder
* Create a new build definition
  + Edit the Workspace tab to point to the code in source control
* Edit the Process tab, set the Build process file to the template created above.
  + Update Projects to build to the new application’s solutions
  + Update Version file to look at the new Version.txt file
* Build, debug, and repeat.
* Once Main is building successfully create a build definition for the Development branch.
  + Clone the Main build definition replacing all reference to Main with references to Development. (the Development branch build definition should be able to use the same Build process file as the Main branch)

# Supplementary Information

## Full Branching Plan Image



## Development Build Definition Screenshots

