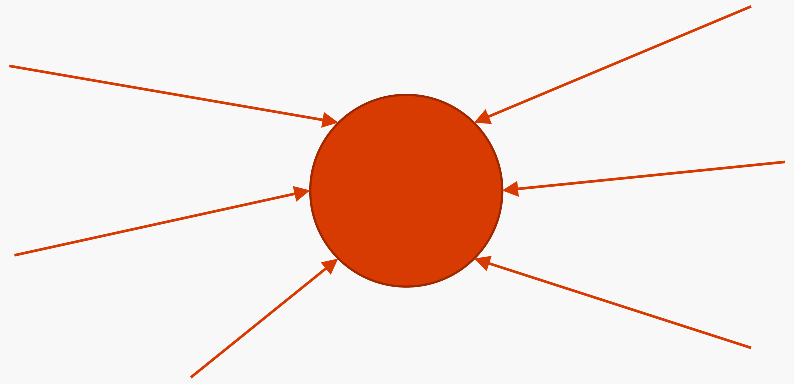
Microsoft

Git

Crystal Tenn
Crystal.Tenn@microsoft.com

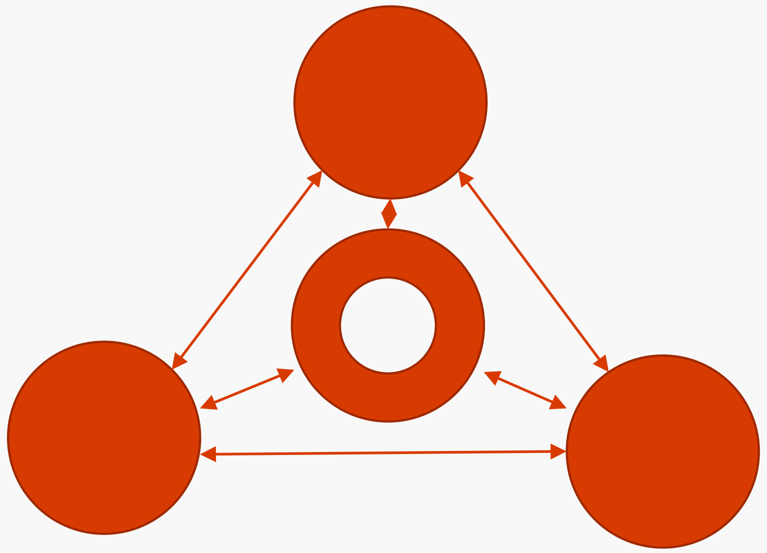
Centralized Version Control System

- Most version control systems are centralized version control systems (CVCS). In a CVCS, a single, shared repository is used to maintain an authoritative copy of the source code, and all changes are made against the central copy. Individuals have local copies of the files which they modify and send back to the server to share with others.
- Team Foundation Version Control (TFVC) is a centralized, server-based version control system.
  - TFVC enables two modes of operating against the mode uses Server Workspaces, where the server must be contacted before any developer can add, edit, rename, delete, or even diff their changes against the original.
  - The second (default) mode uses Local Workspaces, where the additional metadata on the developer's machine tracks changes to files, but the server must still be contacted to update files and submit changes. Local workspaces offer greater flexibility for working with source control while offline, and without good communication can lead to additional merging before changes can be checked in. NOTE: Metadata does NOT include complete file history.
- In addition to TFVC, there numerous other CVCSs – including systems such as Visual SourceSafe5, CVS6 and Subversion7.



Distributed Version Control System

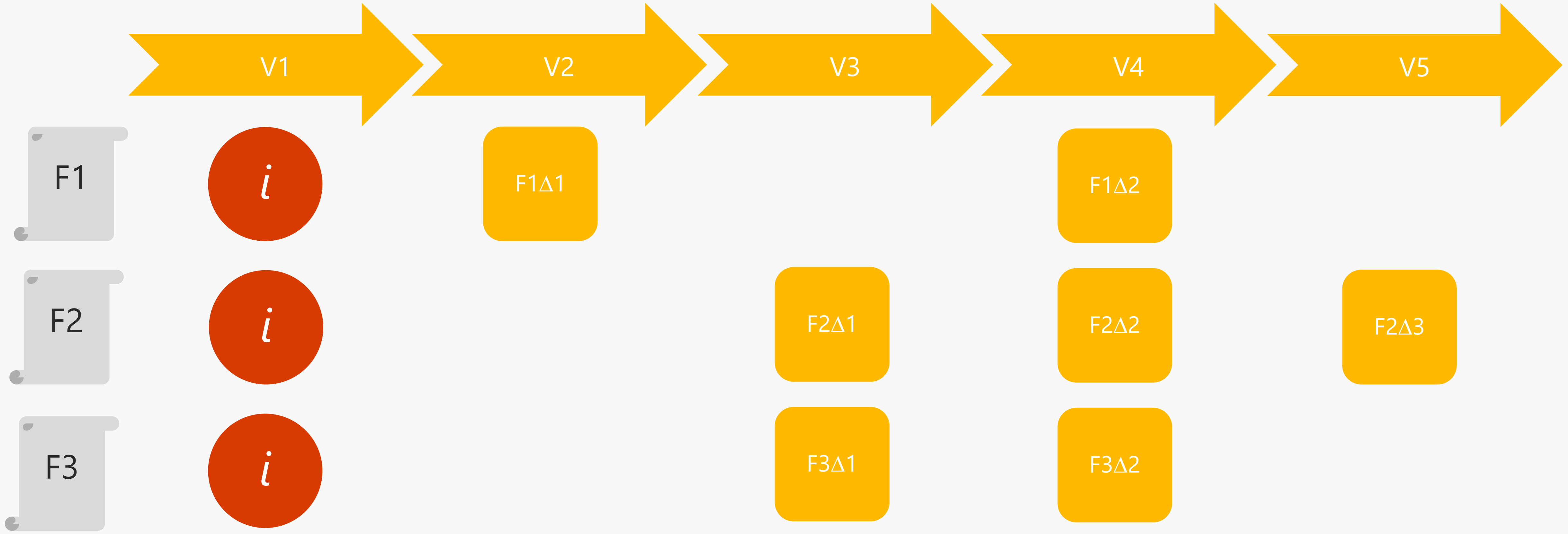
- Git is an open source distributed version control system. Git enables developers to work in two modes
  - **By setting up a central copy and working in a hub and spoke model**
  - **Or, by creating a true peer-to-peer model.**
- The first mode uses a central repository. Once the repo is cloned by developers, the central repo need not be contacted to be able to commit and view full history from their local repo. Developers can work in complete isolation and disconnected from the central repo until they are ready to merge their changes with the central repo.
- In addition to Git, there are numerous other DVCSs – including Monotone and Mercurial.



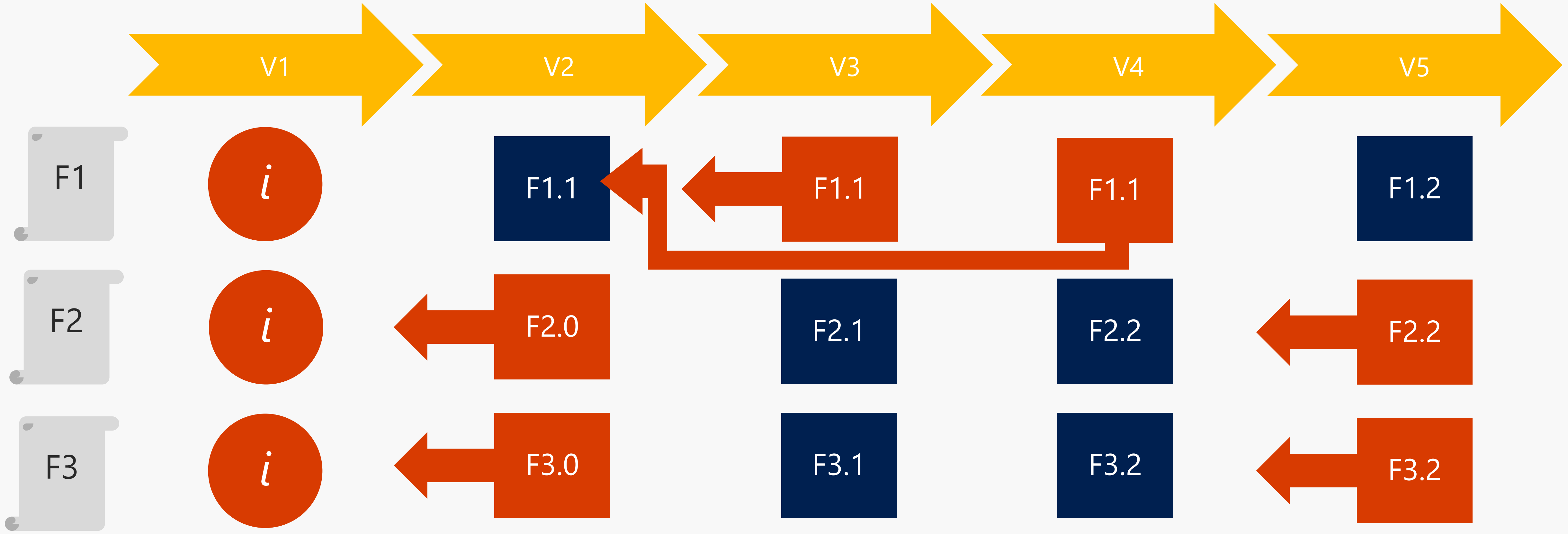
Modern source-control approaches

		Strengths	Best for
Centralized Version Control	Check-in Check-out	• Fine level permission control • Allows usage monitoring	• Large integrated codebases • Control and auditability over source code down to the file level
	Edit Commit	• Offline editing support • Easy to edit files outside Visual Studio or Eclipse	• Medium-sized integrated codebases • A balance of fine-grained control with reduced friction
Distributed Version Control (DVCS)		• Fast offline experience • Complete repository with portable history • Flexible advanced branching model	• Modular codebases • Integrating with open source • Highly distributed teams

Versioning in TFVC (and others)

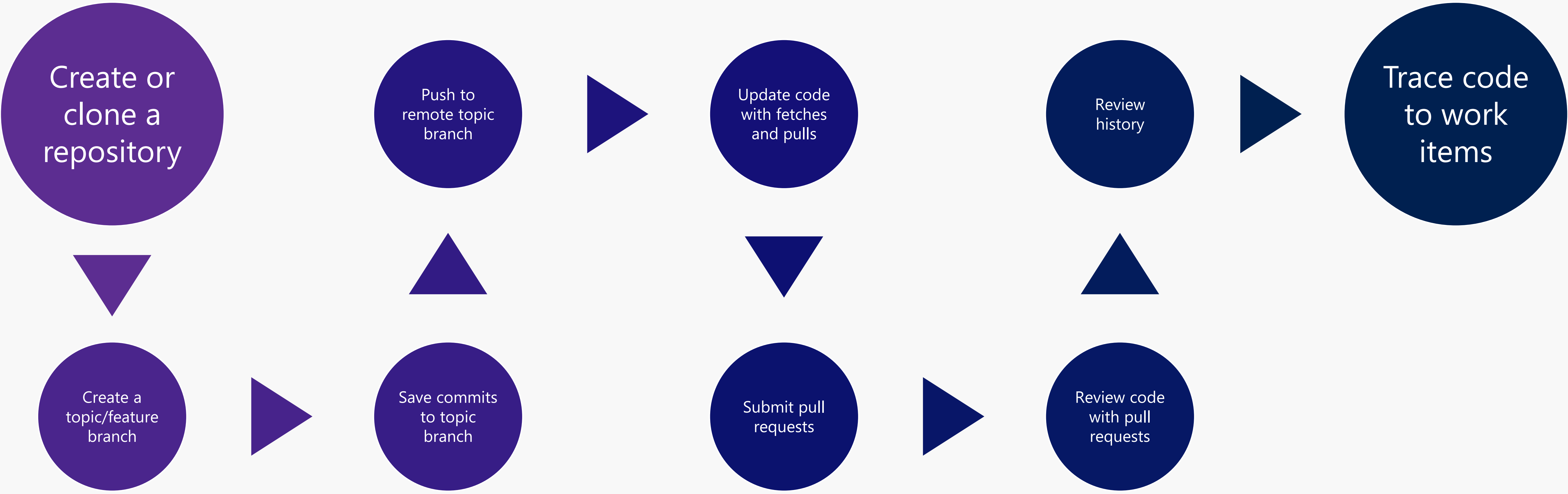


Versioning in Git



Git operations are performed locally

Source Control Collaboration


Create or clone a repository
Create a topic/feature branch
Save commits to topic branch
Push to remote topic branch
Update code with fetches and pulls
Submit pull requests
Review code with pull requests
Review history
Trace code to work items

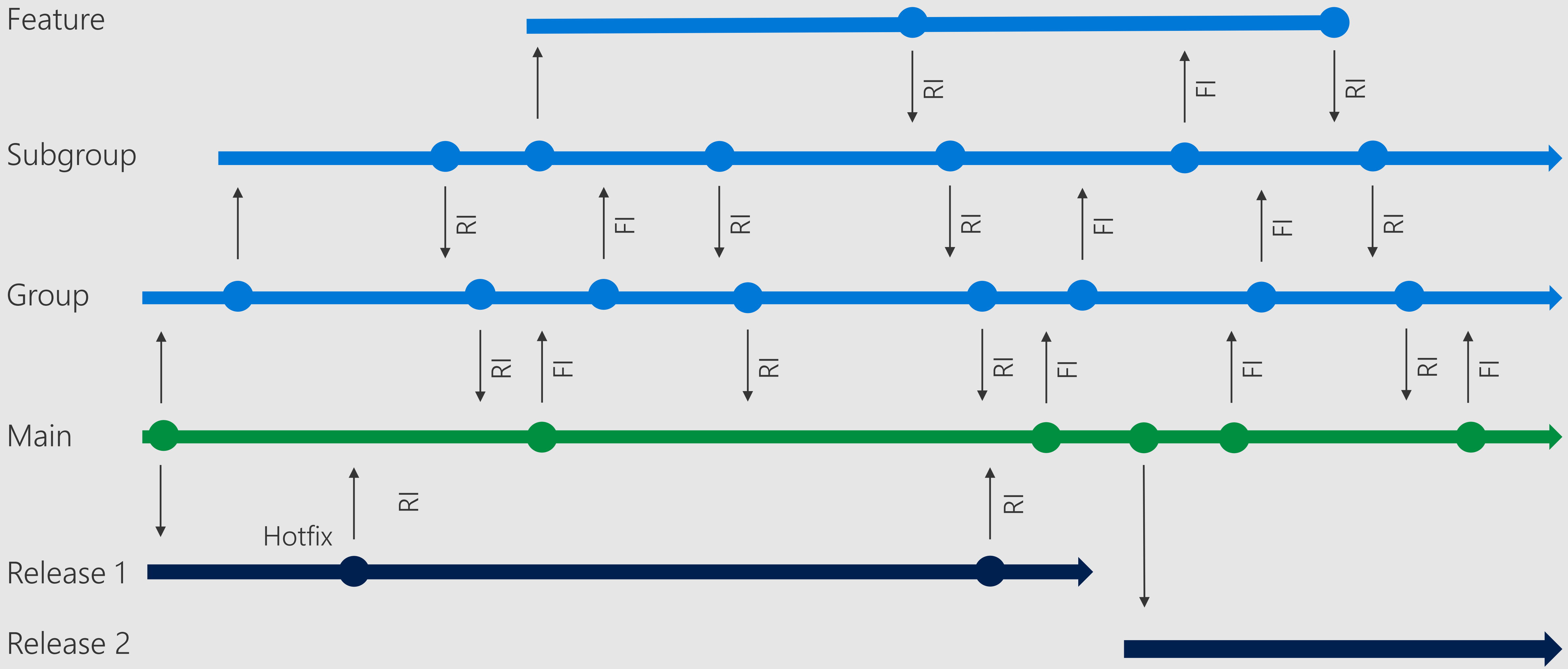
Basic Git with VSTS Demo

Branching Strategies

BRANCHES

Sometimes they appear out of nowhere.

Typical TFVC Branching Structure



"Organizations which design systems... are constrained to produce designs which are copies of the communication structures of these organizations..."

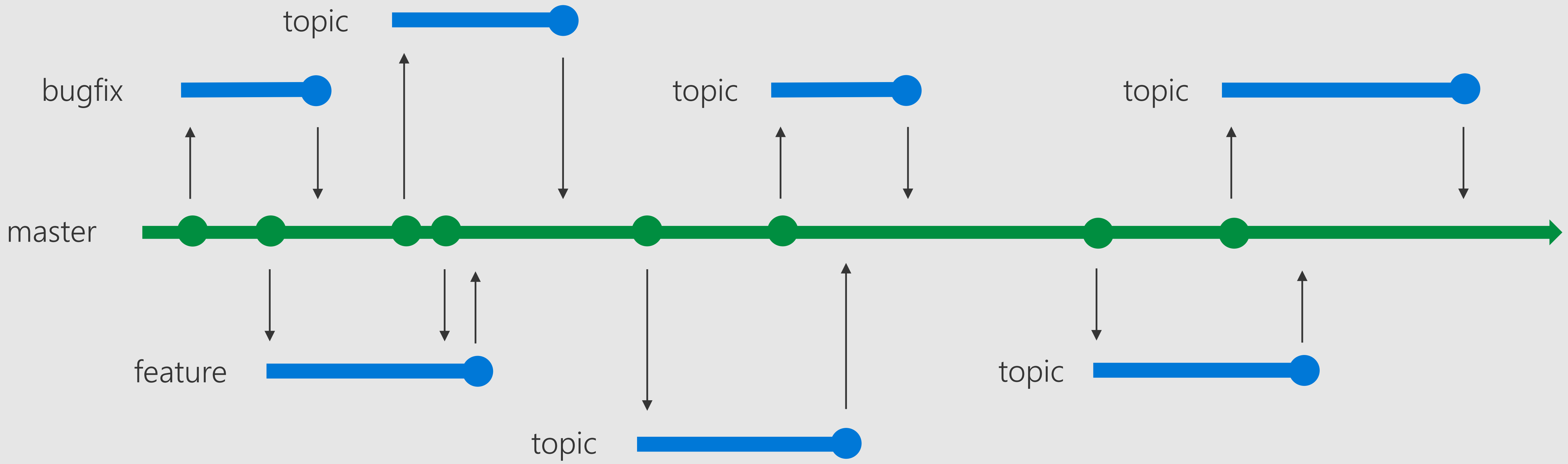
Conway's Law

Organizations tend to produce branching structures that copy the organization chart.

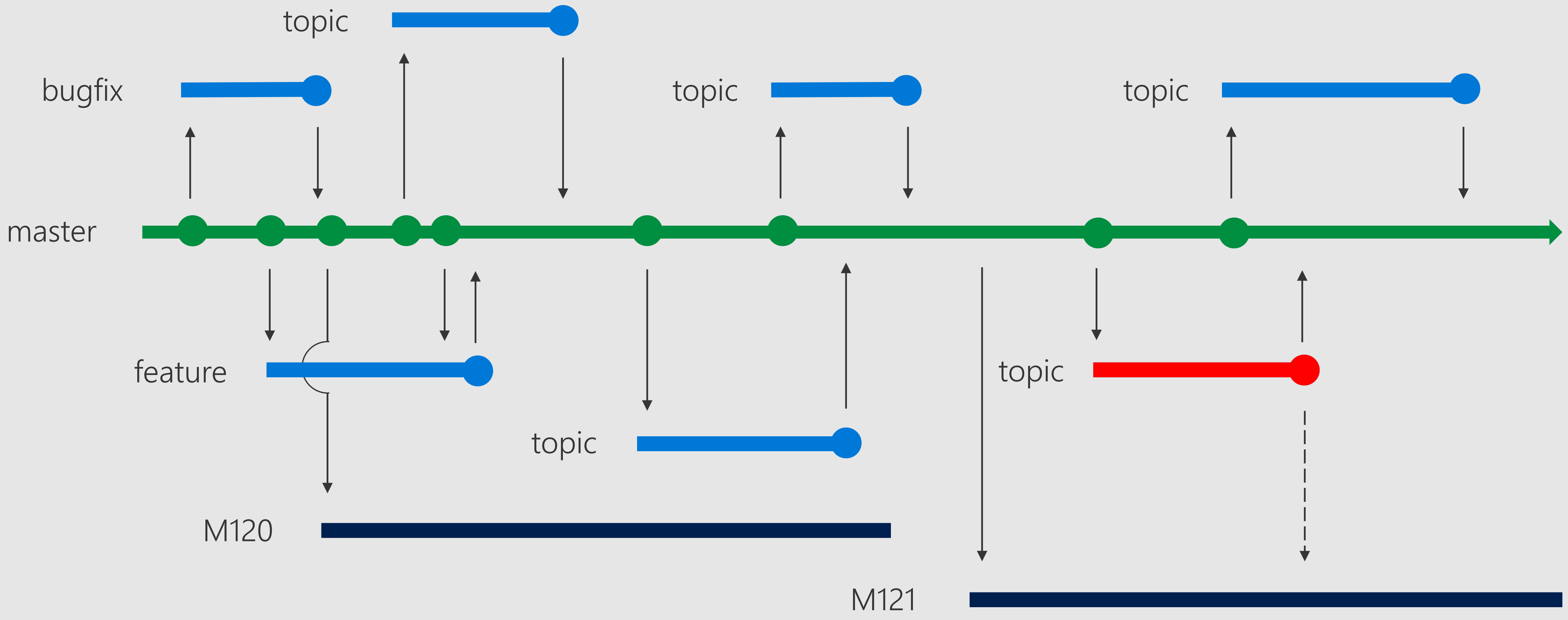
Simplified Branching Structure

- Code close to master
- Small, simple changes
  - Fewer merge conflicts
  - Easy to code review
  - Encourages pull requests
  - Simpler to ship; faster velocity

GitHub Flow Branching Structure



Release Flow Branching Structure



Common Git Commands and Mapping to TFVS

Common Operations

Operation	Command
Create (initialize) a local repository	git init
Clone a remote repository	git clone
Fetch and then pull changes from a remote repository	git fetch, git pull
Manage the set of repositories ("remotes") whose branches you track.	git remote
Stage and then Commit changes	git add, git commit
Undo a committed change	git revert
Branch and merge/rebase	git branch, git merge or git rebase
Push changes to a remote repository	git push

command mapping

Get

Check In

Pending Changes

Shelve

Rollback

TFVC

Git

command mapping

Get → Clone | Fetch | Pull

Check In → Add | Commit | Push

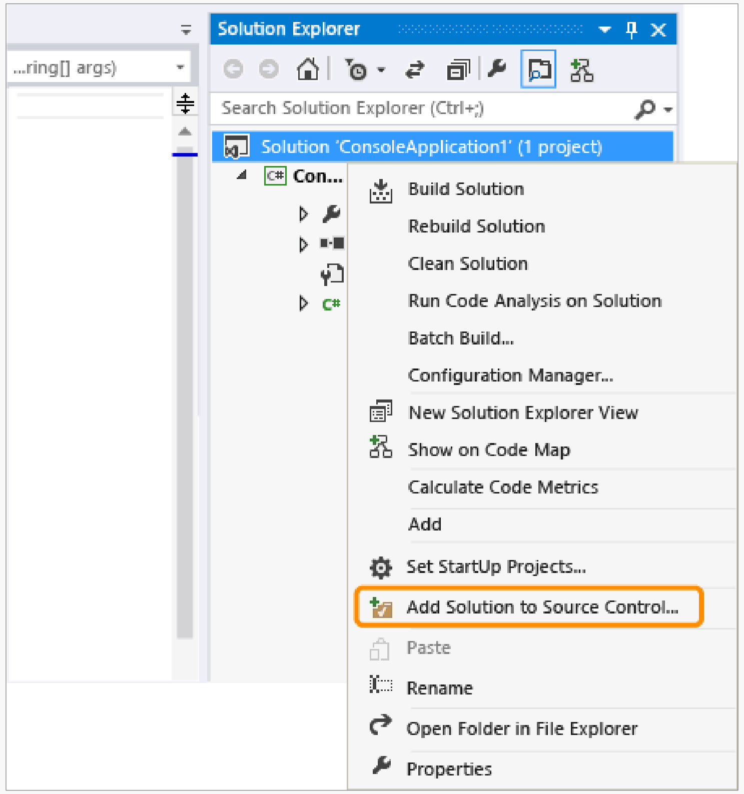
Pending Changes → Status

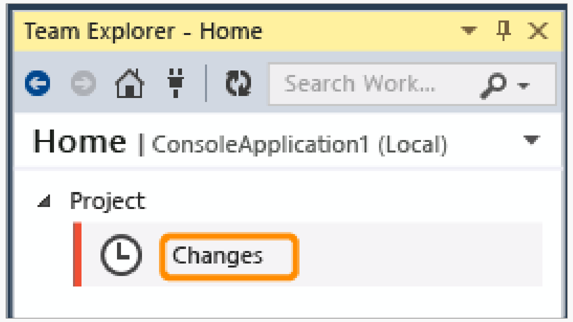
Shelve → Stash | Branch

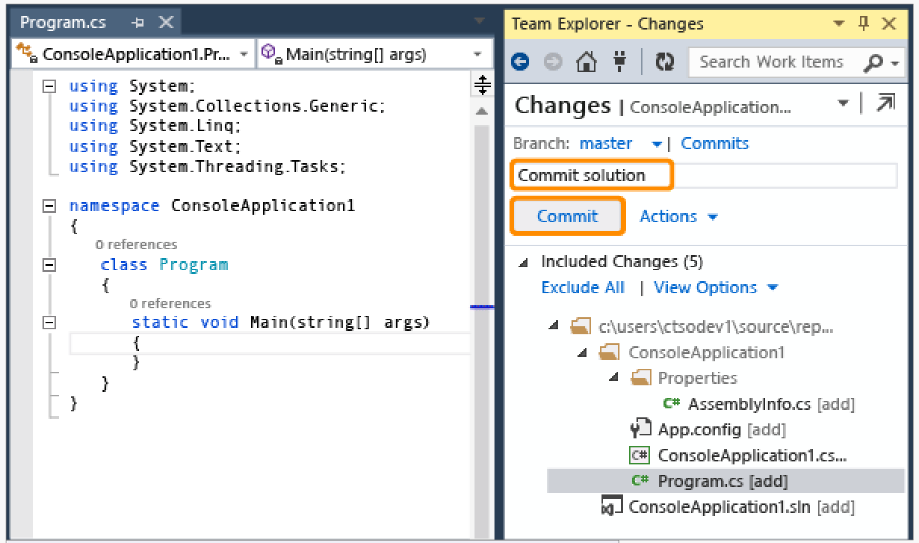
Rollback → Reset

TFVC Git

Add an Existing Solution under Local Git Version Control





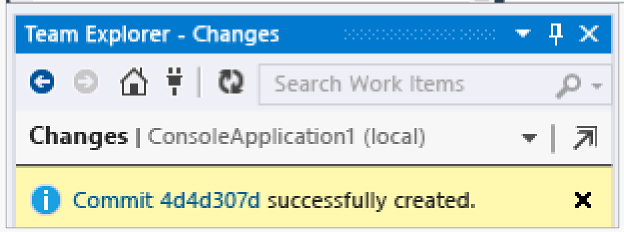


```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApplication1
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
        }
    }
}
```

Commit solution

- c:\users\ctsodev1\source\rep...
  - ConsoleApplication1
    - Properties
      - AssemblyInfo.cs [add]
    - App.config [add]
    - ConsoleApplication1.cs...
    - Program.cs [add]
  - ConsoleApplication1.sln [add]



Commit 4d4d307d successfully created.

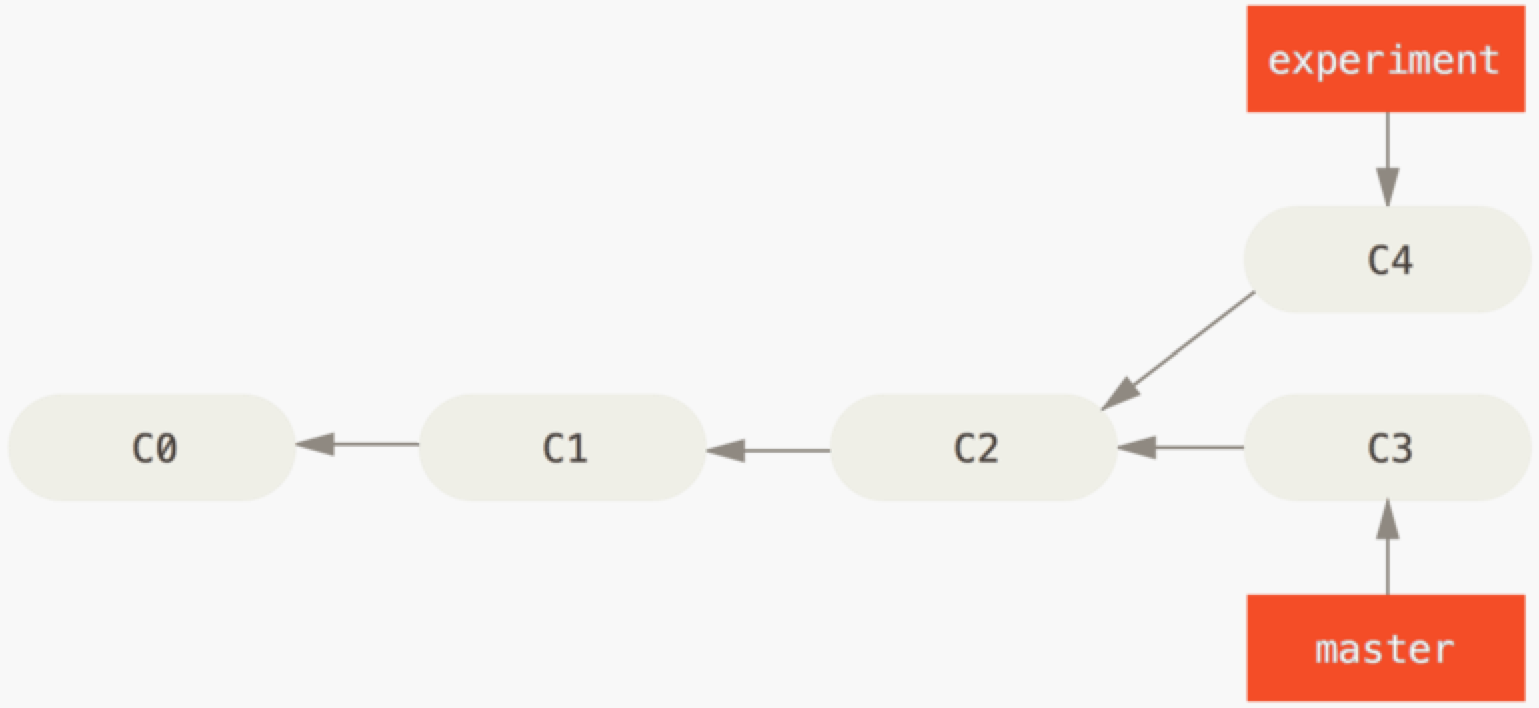
Rebase or Merge?

Rebase or Merge

- In Git, there are two main ways to integrate changes from one branch into another: the merge and the rebase.
  - **Merge** takes all the changes in one branch and merges them into another branch in one commit.
  - **Rebase** is recreating your work of one branch onto another. For every commit that you have on the feature branch and not in master, new commit will be created on top of the master. It preserves the original commits.
  - The project's history then looks as if it had evolved in a single, straight line. No indication remains that it had been split into multiple branches at some point.

Merge

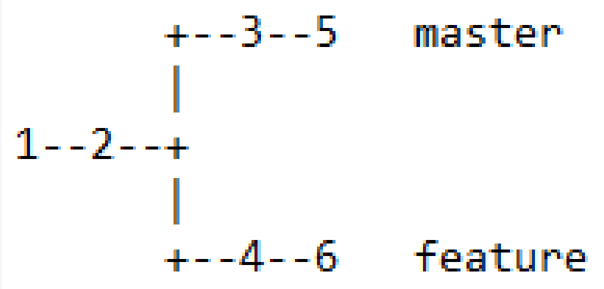
- The easiest way to integrate the branches is the merge command. It performs a three-way merge between the two latest branch snapshots (C3 and C4) and the most recent common ancestor of the two (C2), creating a new snapshot (and commit).



Rebase

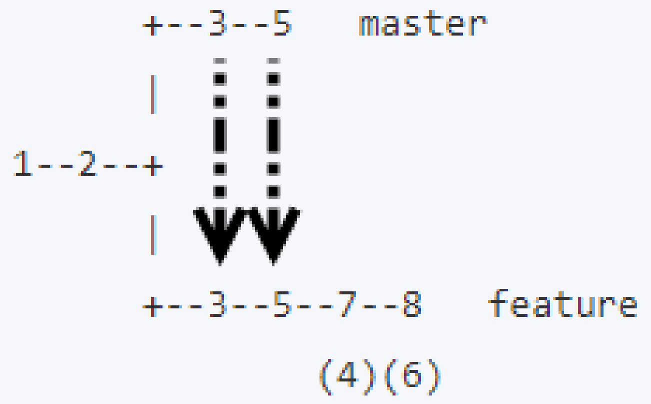
- Say we are on the feature branch. We want to rebase our feature branch work with the current master branch.
- We pick up our work from commits 4 and 6, Git will undo these temporarily and store them for us.
- Our feature branch will take on the current master branch, which has commits 3 and 5.
- Git will now reapply our commits 4 and 6 we temporarily saved, but will give those commits a new ID. Change 4 gets committed as 7. Change 6 gets committed as 8. It’s the same code changes, just a new ID.

Start



```
      +--3--5   master
      |
1--2--+
      |
      +--4--6   feature
```

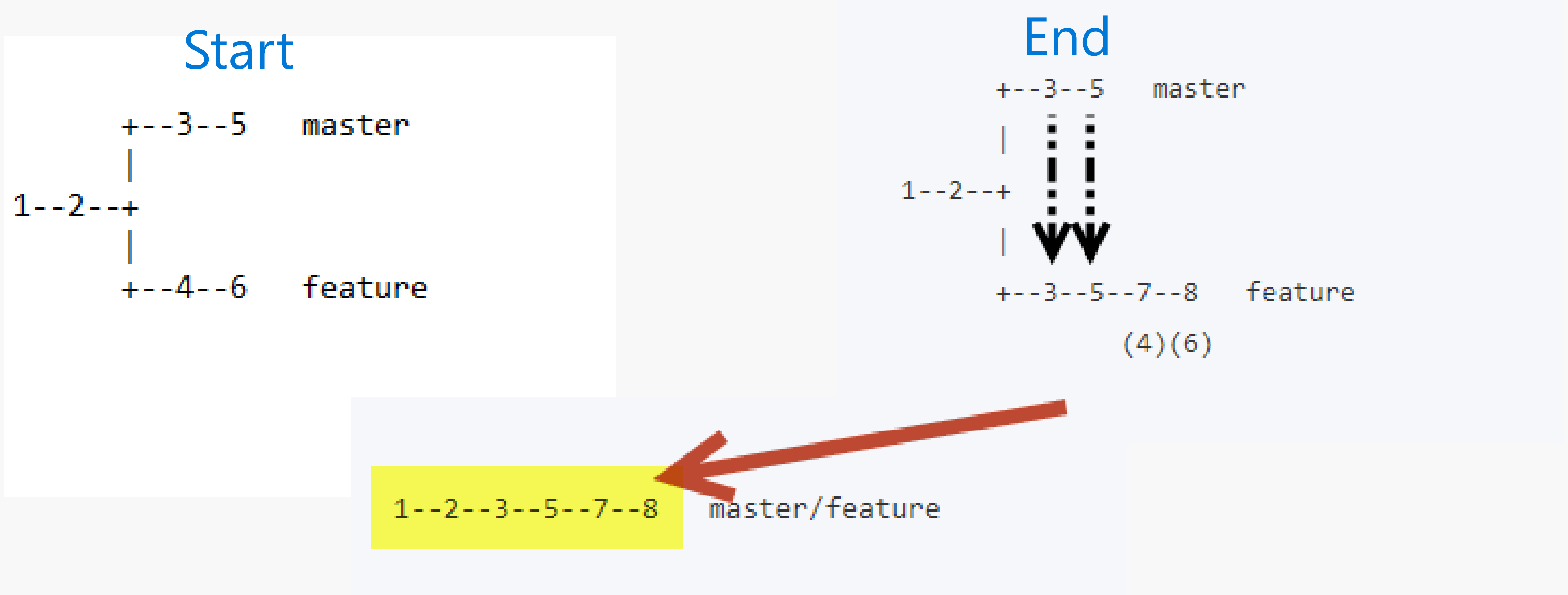
End



```
      +--3--5   master
      |
1--2--+
      |
      +--3--5--7--8   feature
               (4)(6)
```

Fast-Forward Merge

- The only time a merge creates no new commits is the fast-forward merge. It happens in a situation when there are no commits in an another branch.
- After you do fast-forward merge, you will have:



Start

```
      +--3--5   master
      |
1--2--+
      |
      +--4--6   feature
```

End

```
      +--3--5   master
      |
1--2--+
      |
      +--3--5--7--8   feature
           (4)(6)
```

```
1--2--3--5--7--8   master/feature
```

Which one do I want to choose?

- Merge

Let's say you have created a branch for the purpose of developing a single feature. When you want to bring those changes back to master, you probably want merge (you don't care about maintaining all of the interim commits).

- Rebase

A second scenario would be if you started doing some development on your feature branch and then another developer made an unrelated change (that you wanted in your branch). You probably want to pull and then rebase to base your feature branch changes on top of the newer version.

Microsoft