Branching Taxonomy

Brendan Murphy
Microsoft Research
Cambridge, UK
bmurphy@microsoft.com

Jacek Czerwonka
Microsoft
Redmond, USA
jacekcz@microsoft.com

Laurie Williams
NCSU
Raleigh, USA
williams@csc.ncsu.edu

Date: 4th February 2014

Version: 1.0

Abstract

The development of software products are often managed through the use of Version Control Systems (VCS). VSCs allow for versioning of source code and manage the process of merging newly developed code into the product primarily through the use of branches. The branching architecture and the development processes used by the product groups are inter-related and have often evolved over many releases to reflect unique aspects of the product, the development preferences of the team, and the structure of the development organization. Within Microsoft, there is a lot of commonality in the development tools used by the product teams, but there is great diversity in the development processes, and the structure and use of branching. This diversity in the development processes across Microsoft provides a unique learning platform to characterize the effectiveness of specific practices when applied to specific product developments.

Understanding the characteristics of different software development processes is becoming increasingly important as a number of product groups are migrating from developing a box set product to simultaneously developing a product deployed as a service, released as a stand-alone product, and shared as a component with other product groups within Microsoft. To assist the product group in making their decisions regarding the most appropriate branching architecture for their product, this document examines the benefits and drawbacks for different branching architectures. It also delineates the issues product groups often have in managing the simultaneous development and release of multiple products that share the same code base. The document should aid engineering teams in choosing the best branch structure for the software product they are developing and in understanding the requirements for sharing code between products.

Introduction

The process chosen by a product team to develop its next product version is usually heavily influenced by the successes or failures in the development of previous releases. Often, the context and usage of the product does not change significantly between versions. Traditionally major changes to the development process will only occur due to failures of a past development of a product, the group adopting a new development methodology or due to the (lack of) performance of tools used in the development process. Most often if a product group finds that a specific development process has a track record of success, they will evolve this process between releases ensuring it satisfies the needs of the next product development. The process chosen by the product groups will be reflected in their code development and especially in its branch architecture. These decisions are often based on subjective opinions rather than objective data.

Unfortunately, there is scarcity of independent research into which development process and branching architecture is optimum for a specific product type to assist teams with these critical

decisions. This document defines taxonomy of branching architectures used to manage the development of software both within Microsoft and externally in the software industry, addressing all aspects of software development from the creation of new software through to the maintenance of prior releases of software. For a team to choose the appropriate branching architecture for their product development, they have to make two main decisions: (1) how many trunks[1] are required to manage the development; and (2) the depth of the branch tree underneath each trunk. These decisions are independent: a product group may decide to go with a single trunk with no child branches; or to go with a number of trunks and deep branch hierarchy underneath each trunk.

The traditional focus of the development process is to release a single next version of a product while simultaneously maintaining past releases. Product groups often develop complex interactions between the development processes of the current and past products to avoid regressions in the current release. As a result, the maintenance of prior versions often occurs in the same development environment as the current release. The cost of releasing a change in prior versions of box set products is high, mainly driven by the cost of verifying the change, so the cost of maintaining interfaces between the development processes was not a critical factor.

However, significant changes are occurring to Microsoft's, and a number of other companies, product profiles where product groups are moving from focusing on a single box set product to developing in parallel a range of products to satisfy different markets and different release profiles (e.g. Windows Client, Server, Phone and Xbox are re-using the same core operating system). These changes mean that development processes that were successful in the past may no longer be guaranteed to efficiently produce software in the future. Additionally, it is becoming increasingly important to not only consider how to develop a single product but also how to share code with other product groups.

A different set of cost drivers influence service software as the software matures. A majority of service based software developments apply agile methods, focusing on software evolution through incremental steps. The deployment of service products allows the developers to verify changes based on a small subset of their customers. They deploy software to all systems in their service but they can control the percentage of customers that will use the new software. If the software is defective the process can pull the changes. There are a number of examples of successful products that use this process but as the software matures there appears to be common problems that stretch the limit of this process:

1. The software architecture becomes diamond shaped

   As services, the functionality increases at a far greater rate than its user interface and the interface to the underlying process than manages the service. Architecturally the software becomes diamond shaped, with increasing number of components and increasing number of interfaces between components. This leads to versioning issues (discussed later in the document) and complications when trying to develop functionality requiring changes across components.

2. The cost of bugs increase

   The rapid deployment processes of the agile processes are reliant on the high quality of the deployed code and also the low cost of a bug that is deployed to customers. For instance if a features that changes the way results are displays contains a bug which corrupts the display, the customer may just replay their request; the cost of the bug is low. As the services become more successful so their product increases in functionality and a greater proportion of their functionality require managing state (which cannot be fixed through replaying requests).

[1] A trunk is the base of the project development tree, sometimes called the main branch

Additionally the reputation of the product grows in importance and consequently the reliability issues in general and the security and privacy guarantees increase in importance and the cost of failure becomes significant. In these circumstances product groups soften increase the extent of pre-check-in[2] verification for code changes which decreases their code velocity and strains their current development processes and culture.

These changes to the product profiles mean that the product groups will have to evaluate their current development process and branching architecture to identify if it can meet their new set of requirements. Unfortunately, there is little unbiased information to assist the product groups in making informed decisions about which branch architecture would be ideal for their future needs. The information that forms the basis of this paper has been obtained through:

1. Analysis of the development processes of external companies, where published, and other external discussions and papers in this space.
2. Monitoring and analysis of the development processes of all major product groups within Microsoft.
3. Detailed discussions with the product groups about how they developed past projects and their future plans.
4. Woking in partnership with a major product groups in re-architecting their product development and tracking the results of their re-architecture efforts to monitor their impact.

The remaining part of this document is structured in the following way:

- The goals of branching. A major aspect of the branching structure is the ease with which the verification of changes (features and bug fixes) is performed and ensuring that those changes do not regress the rest of the system. This section examines how this verification process has a significant impact on the chosen branch architecture.
- Decision points for developing branch structures. Describing the major factors that influence the choice of branch structure.
- Configuration of trunk branches. A trunk provides a version of the product, continually or at specific versions that can be integrated into other product components or released. A major decision is whether to go for a single trunk per product or for the single product to have multiple trunks (often called componentization).
- Branch tree architectures. A product group may decide to develop software directly in the trunk branch or may have child branches which isolate changes that are later merged into the trunk branch.
- Future requirements for branch structures. Discussing the influence that changes in product profiles will have on future branch architectures
- Summary of results.
- Initial recommendations for an ideal branch system

Further research and analysis is ongoing to better understand the advantages and disadvantages of different branch taxonomies applied to different product development classifications, this research will be reflected in updates to this document.

[2] "Check-in" defined as a code modification visible to other developers as opposed to a strictly local commit.

Goals of Branching

Branches are used for two main purposes:

1. To ensure source code for a released version of software can be isolated from future development for the purpose of enabling post-release fixes. In short, these are the branches used for **product maintenance**.
2. To isolate parallel development on a project requiring multiple contributors. This is the branching structure used for **product development**.

Branch Structures Used for Maintenance

A team often simultaneously maintains multiple versions of a product, for instance the Office team maintains a number of past releases of Office. It is important to ensure these versions are maintained both independently (as the software is at different revisions) and to develop them in lock step (as a security fix has to be released simultaneously across all past releases). In the realm of branching structures used for maintenance, the ideal situation is having one snapshot of code (branch) corresponding to each deployment target.

For services, there often exists only one instance of a running product, i.e. the latest version of the product, and thus a rolling "production" branch easily resolves the issue. Complexities such as internally having multiple instances of the service (staging, pre-production, and production) are resolved by having a branch pipeline. For example, Figure 1 depicts a series of branches one product team uses to move code from development branches into branches that are used in the staging environment (**a1**), limited production (**a2**), and wide production (**a3**). As code graduates from **a1** to **a2** and finally to **a3**, it conforms to increasingly more rigorous verification constraints. This process requires the product team to port (through backward and forward integrations) code changes made in each staging branch to the other staging branches and to the trunk branch. The code in each stage is a fork of the code in the previous stage.

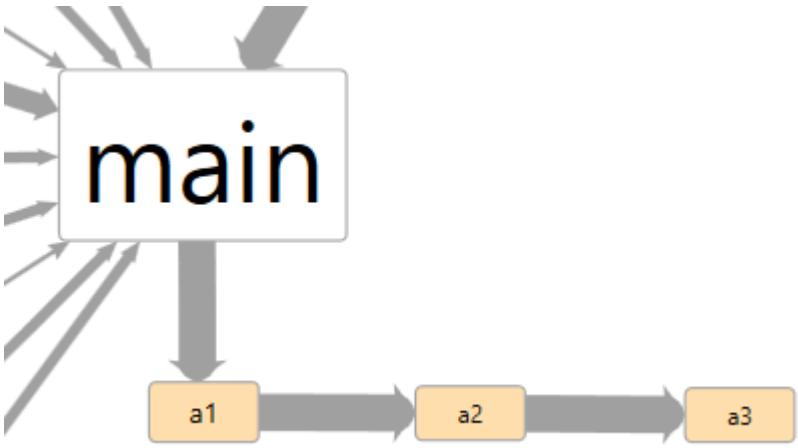

Figure 1. Branch pipelining for maintenance of services with multiple deployment targets

For products which have more than a handful of deployment targets, which is essentially any product that is not a service, the situation is more complex. Windows client, with its 1.3+ billion users represents the extreme. In those cases, the attempt is to create a branch structure that groups the targets into a relatively small number of classes. A class is a group of deployment targets that share the same version of the code, for Windows: one class is all PCs running the stock version of Windows 8 with only security fixes on it, another could be Windows 7 + Service Pack 1 and yet another is corporate clients that have maintenance contracts with Microsoft running code with fixes specific to

their deployment. Each class corresponds to many—in some cases hundreds of millions—of deployment targets (PCs), each of which is approximated by the same snapshot of the Windows code.

Creating deployment target classes is an attempt to minimize the cost of the branches, including the costs of the implementation and testing of the same fix across many branches, against the risk of unnecessarily introducing changes to deployment targets that do not otherwise need them. For instance, for the last several years, each release of Windows maintains a separate branch structure for each of its Service Packs further subdivided into a branch containing broadly applicable fixes (the General Distribution Release or GDR) used by general public and limited scope fixes (the Limited Distribution Release or LDR) targeting the corporate deployments, see Figure 2.

Most other teams have maintenance branch structures in between the two extremes. There could be a discussion about specifics however the philosophy of trying to approximate a "branch per target" method within the cost constraints is the most prevalent situation today. Branch structures used for maintenance are not a subject of further discussion in this document.

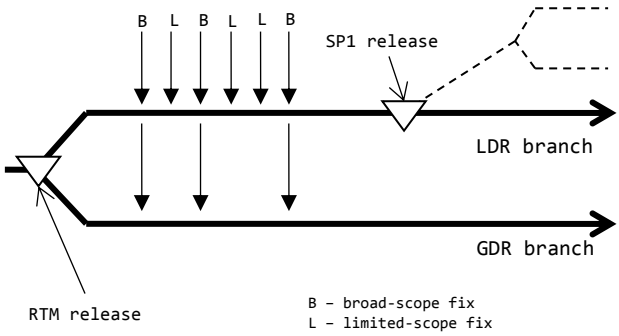

Figure 2.Windows branch structure used for maintenance of a single release

Branch Structures for Development

The objective of the development processes is to allow multiple engineers to simultaneously develop software (features or bug fixes) that are efficiently integrated together to form a software product which can then be released at a specific level of quality. Unlike during maintenance where changes are often sparse, relatively small and infrequent, the development period deals with large amounts of often very disruptive churn.

During product development individual engineers or a team of engineers separately implement features or bug fixes. These changes are verified through a number of different processes such as code reviews, unit tests, static analysis, component verification, system verification etc. The objective of the verification process is to ensure the feature meets quality goals by:

1. Satisfying its functional requirements.
2. Satisfying any constraints placed by the system.

The constraints that a feature has to satisfy vary based on the product that the feature is being integrated into. The same feature developed for Bing UI and for the Windows kernel would have to comply with completely different sets of constraints. The same levels of constraints do not universally apply within the same product. For instance, the verification requirements are not the same for an equivalent change in the memory management system of Windows as compared to its UI. Typical constraints found in today's software relate to compatibility, security, standards and regulatory compliance, privacy, accessibility, UI guidelines, backward compatibility, physical limitations, power consumption and performance.

As the number and complexity of the constraints increase, the more difficult it becomes for an individual or group of engineers to verify that new software meets all constraints without some form of process (to either assist the engineer during development or to verify the product after the change has been integrated into it).

In addition, previously, an engineer had to ensure that a software change met the constraints of a single product, but in the future, software may have to meet the constraints of multiple products. For instance, changes to the memory management in Windows are required to meet the physical constraints of the entire family of Windows products (Client, Server, Phone, Azure and Xbox) as well as backward compatibility, security and power constraints. It is unlikely that every engineer would deeply understand implications and consistently apply verification steps for all of these constraints.

The complexity of the constraints and the cost of a defect will define the minimum level of verification required for the software. The product team may decide that the constraints can be met by the engineering team on its own (e.g. UI changes). But if the team requires an automated verification process, then the automated verification will need to be incorporated into the development process which may subsequently impact the branch architecture.

Decision Points for Development Branching Structures

In designing the branch architecture for a product, the development group has to make decisions around two independent aspects, specifically:

The configuration of the trunk branches

A trunk, within Microsoft, is the central branch where the code of a product or component is maintained at an acceptable level of quality. It is often also a release point (or at least a source of a release branch) for the finished product. A trunk branch can either be at a consistent level of quality where it is built and tested frequently or where a period of development is followed by a stabilization period.

A team may use a single trunk per product, which is the technique used by, among others, the Windows group. Alternatively, a project may develop software across a number of trunks where the total product is built through combining the software components from multiple trunks, which is the method used by the Bing and Skype teams.

The branch structure depth

A product group may develop their code directly in the trunk branch, or develop code in child branches which subsequently merge into the trunk. Development beneath the trunk can be achieved through the creating of a branch tree that contains multiple levels of development or integration branches.

The secondary consideration is whether child branches are long-lived or short-lived. This decision is largely driven by the complexity and cost of creating and maintaining new branches. Especially in situations where creating a new branch carries with it infrastructure costs of establishing new build and test lines, the tendency is to create branches on a semi-permanent basis (often assigned organizationally to teams). One has to be rigorous about cleaning up such branch structures periodically, although it is often a rare activity undertaken when infrastructure costs exceed a certain threshold. On the other hand, short-lived branches, created to complete a feature or architectural change offer isolation and limit infrastructure costs.

Both decisions are independent of each other and the method the product groups used to manage these branches. The following sections describe these categories in detail.

Configuration of Trunk Branches

There is a general acceptance that a product during development is required to be at a known state at frequent intervals. The acceptable minimum known state is that the software is capable of being built; more often there are extra requirements e.g. that the software passes a set of product-specific verification gates.

There are two ways a product may configure the trunk branches, and this is determined by the development methodology or the tools available to the product team. The choices are to either maintain the full product in a single trunk or to split the product into a number of independent modules and to develop these in parallel.

Single Trunk Branch

In this configuration, all of the software that forms the product is maintained within a single branch. Any change submission, build, verification or deployment tools applied to the software are applied to the current version of the software in the single branch. The current version is the latest version of all software in that branch. Additionally when the software under development is dependent on other software, using a single branch ensures that all dependencies are referencing software at the same version. This is the process used by product groups such as Windows, where the product is maintained in a branch called ***winmain***. If the product groups wishes to make a product release, such as Beta releases or Release to Manufacturing (RTM), they take a fork of the current version of the trunk branch.

A single trunk branch simplifies the management of the software but does place restrictions on the developers as they must ensure their development methodology complies with the entire group's change submission process. As the product scales, the total process becomes more dependent on the speed of the tools and processes (especially build and test) and as a project grows, the tools and processes often become bottlenecks.

When products are self-contained, a single trunk branch simplifies the product management. Where interactions exist between products, then a single trunk branch can complicate these interactions. If a product requires code/component from another product, they may prefer to use a specific version of that component not necessarily the latest version of the component, as exists in the trunk. The structure of the trunk branch complicates this componentization management. One method is to create a release branch for each component. In this scenario the component code from the trunk is sync'd to the component branch. Tests are ran on the code in the component branch to ensure the component complies with the constraints of the other product (which may differ from the main product). These tests can be viewed as a quality sign off for the component. This method provides flexibility as if the component fails these tests the code can be forked on the component branch to fix the bug. Thereby minimizing the delay in releasing the component at the required quality level. To ensure the fork between the main product and the component is manageable it is important that the forked code is fed back to its owners in the main product who can then propagate the fix into the main product at a later date. The forked code can then be removed from the component branch. Product groups within Microsoft use this hybrid process of sharing components between product groups.

Multiple Trunk Branches

Multiple trunk branches configuration is not just an extension of the single trunk branch model, but it carries with it profound implications for the rest of the process. In this model, the software product is split into a number of independent components interacting through strictly controlled interfaces. The development of each component is managed separately in its own trunk branch. This enforces architectural control over interfaces. This method is less demanding on the tools as there is typically less software needed to be built and verified in each branch, which is partially why it might be a choice for team's struggling with tools' scalability. Additionally it allows the different components to

be developed using separate methodologies which are optimized to the specific component or based on the preferences of the team.

In this development methodology if a component has a dependency on another component, it is managed through versioning interfaces rather than sharing code. Typically, each of the trunk branches periodically produces a set of executables comprising a version of the component and all dependent components consume the functionality through the interface; sometimes this dependency is for a specific version of a component.

The issue of versioning is sometimes managed by requiring all components to always reference the latest version of all other components. This methodology has been successfully applied by the service products such as Bing (see Figure 3), Google Search, and Facebook, i.e. products which do not have the same legacy issues as boxed products. Specifically, where there is only one deployed instance of the product, it is easier (but still by no means a simple task) to enforce that components be compiled and run against only the latest version of all their dependencies.

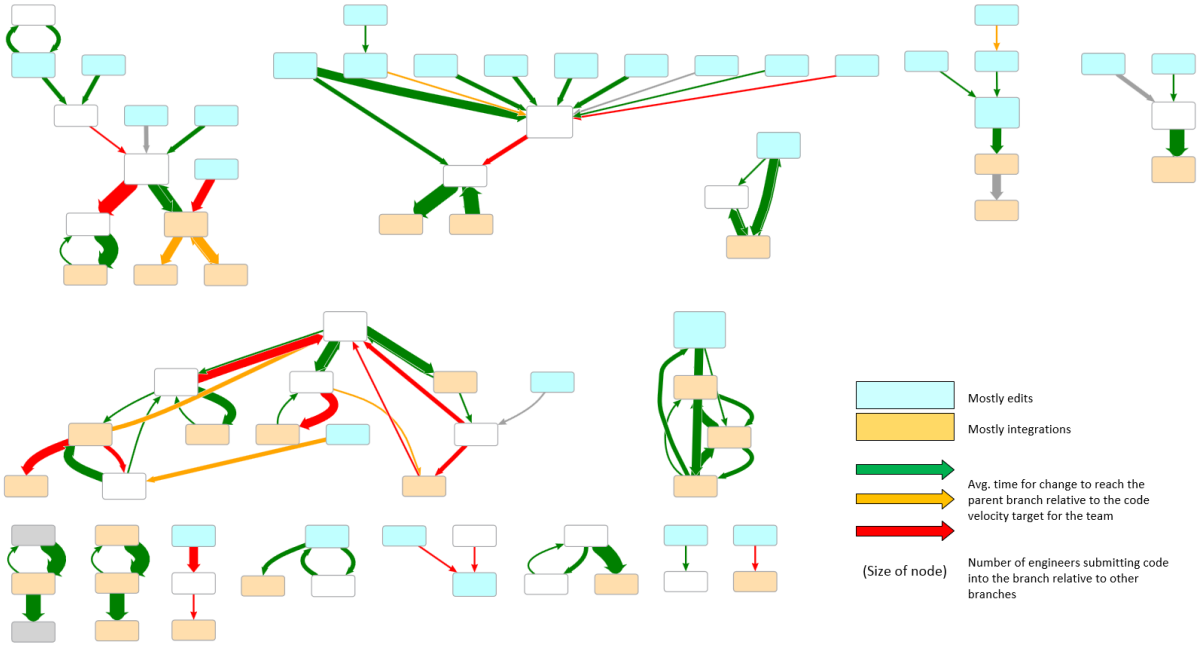

Figure 3.Multiple trunk branches structure in Bing

This methodology of "always using the latest" is not being applied universally and, for multiple reasons, it may be preferable to reference specific component versions rather than to always depend on the latest version. Referencing the latest version may result in failure in the dependent code as changes to the latest version of the dependency conflict with the dependent component. In these circumstances, versioning can become an issue. In Figure 4, Project B is composed from A and C, which directly or indirectly reference two different versions of Project D, resulting in a conflict.

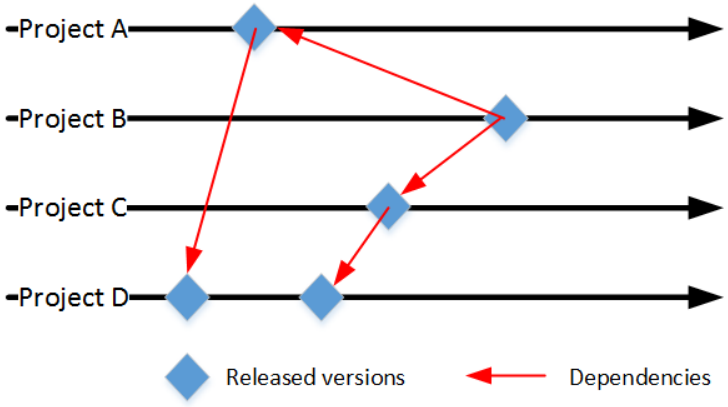

Figure 4. Component versioning issues

There are package management tools that can identify these versioning conflicts but they nearly always require engineering effort to resolve them.

Considerations have to be taken regarding the size of the component that forms the trunk branch. Smaller component will be developed and maintained by individual engineers, whose availability may not always be guaranteed. Additionally a small component is more likely to become 'stale' and may not be updated to take account of new versions of other dependent components which can subsequently increase versioning issues in other components that are dependent upon the 'stale' component. Alternatively large components may result in the time between the publishing of stable versions of the component being longer, which may result in the team having to maintain published versions by providing bug fixes. It can also lead to complications where another component may have a dependency on features which are yet to be released which may result in multiple forks of previously released versions.

Where a team has to maintain past versions of the product, having multiple trunk branches complicates the process as a snapshot has to be taken of the software that formed the complete product. To achieve this, the process has to ensure it captures the software that formed each version of each component. The product has to determine the specific versions that formed the total product and identify the time that the package was created as it can then take a snapshot of the trunk branch at the time of the package creation. This differs from using a single trunk branch as you only require taking a snapshot of the branch at the time of the product release.

Summary

The following provides an overview of the advantages and disadvantages of the different trunk branch configurations.

	Advantages
Single Trunk Branch	Easier version management Simpler product management Easier to snapshot and support released versions
Multiple Trunk Branches	Tools scaling less of a problem Enables variation in development processes to product area Enforces modular architecture Easier sharing of components with other products

Table 1: Configuration of Trunk Branches

Depth of Branching Structure

For each trunk branch a product group has to decide whether to check code directly into it or to check code into development branches that subsequently merge changes into the trunk. A major factor that influences this decision is the method used to verify the changes. The most common practices are pre- and post-check-in verification.

With pre-check-in verification, the goal is to maintain stability by only allowing those changes into the trunk branch that meet a minimum quality bar. A pre-check-in verification process, can either be a stand-alone process (used in shallow branch trees) or a part of a quality gate that is run at the time of integration from a child branch (implying a deep branch hierarchy). The process applied to both is similar: in a separate area away from the trunk branch, the latest code from the trunk branch is merged with the new change. A series of tests are run and, if successful, the change is merged into the trunk branch.

The verification process may be optimistic whereby the tests run per change are limited but after a number of changes have been applied, a more complete set of tests is run and any failures are corrected at that point.

The choice of which type of tests to apply is based on the quality of the new code and also the time required applying the tests. For the verification process to be successful, the changes must be applied to the trunk branches serially. Therefore the longer it takes to run the verification process, the slower it is to integrate each change into the trunk branch.

With post-check-in verification, a change may undergo some lightweight verification (e.g. code review) prior to check-in but no full-scale testing occurs prior to integrating code into the trunk branch. This method can be viewed as pure agile methodology. In this scenario, a check-in may subsequently be found to be incorrect when all the software is finally built and tested. At this stage, the product group may revert the change (i.e. remove the full change from the trunk branch) or apply fixes directly into the trunk branch until the software stabilizes again. If the change is trivial or the constraints on the software are small then this methodology promotes code ownership and code quality as all code failure is public.

Product groups who prefer to check changes directly into the trunk branch without system verification tend to prefer not to develop code in child branches. Product groups who do apply pre-check-in verification processes can apply both types of branch architecture.

Deep Branch Trees

For large, complex, non-modular products changes made to the core or kernel of the product can have impact on other parts of the product in complex ways which are difficult for any individual engineer to comprehend. This is especially true for a product like the operating system, where a new feature must not only meet the functionality requirements, but also comply with various constraints such as compatibility, security, privacy, performance of multiple end products (Client, Server, Phone, etc.). Additionally, its impact on public APIs must be understood due to legal requirements. With such a **large set of constraints**, an individual engineer is unlikely to be able to ensure the change meets all of them. In such circumstances, a product group develops processes to enable consistent verification of the changes.

Another important factor when designing the branch structure is the **number of engineers** involved in the development of code. If the code is checked directly into a trunk branch then the process of checking and verifying code directly into this branch becomes a bottleneck. Using a deep branching tree, individual features can be checked into child branches, then verified in isolation, and only then merged with other features and verified again. The set of verified features are then checked into the trunk branch in a single merge which increases overall productivity vs. the alternatives.

There are a number of methods to manage branch trees but they inevitably involve a mixture of process and engineering management. The largest product that uses deep branch trees are those of Windows and Windows Phone; the tree structure for the Windows Client development is partially depicted in Figure 5.

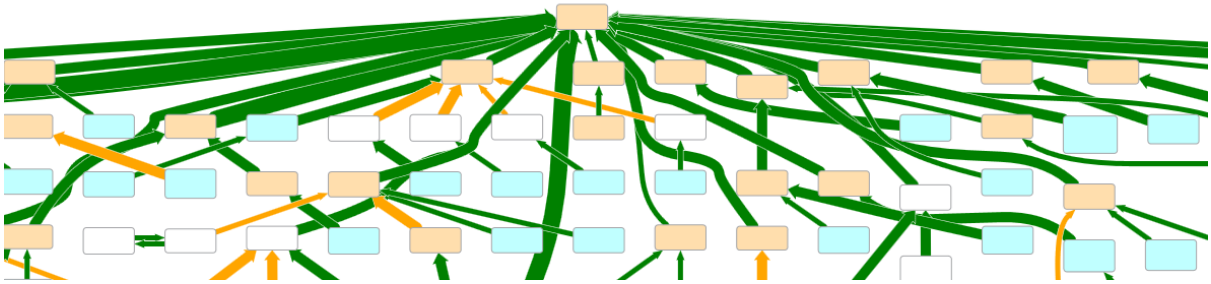

Figure 5: Deep branch tree structure in Windows

In the Windows development process, a feature is checked into a child branch designated as a feature branch (designated by the colour blue in Figure 5). Some groups use pre-check-in verification to ensure the change meets a certain level of quality. The feature moves through the branching structure through integration branches (orange coloured branches in Figure 5), which are controlled by a designated team. As the change moves through the structure the feature is continually verified and merged with other features developed by larger and larger circle of neighbouring groups. The process continually ensures that features work with the current trunk versions of the product through integrating changes which already made in into the trunk branch with all child branches.

When the feature reaches the trunk branch, it is automatically synchronized with all other branches so that all other code in the child branches or passing through the development process does not conflict with code that resides in the trunk branch. This places the onus on the developers to ensure that their product does not conflict with code that exists in the trunk branch. If two code changes have a conflict and both exist in different areas of the tree, the conflict will be identified when the first of the changes reaches the trunk branch. Soon after the first change reaches the trunk branch, the conflict will occur on all branches that contain the second change. The developers of the second feature will have to resolve the conflict before the integrations can continue.

There are a number of benefits of this process. First, it allows complex and time consuming testing to be applied to the changes as they progress through the branching structure. This also allows the **process to ensure the changes meet 'system' constraints** such as security and privacy. The development process ensures that no code checked into the trunk branch conflicts with code that already exists in the trunk branch. All engineers have access to the latest copy of all other code that is used within the product which promotes code reuse.

Large architectural changes are performed on a branch or set of branches created specifically to develop the architectural change and all other code that is perceived to be impacted by the change. During development, the branch will continually sync with the trunk branch, identifying early if changes by others conflict with the architectural change. Once the architectural change is complete it is then integrated into the trunk branch and in a short space of time replicated across all branches ensuring that all future development is automatically verified against this new architecture.

A major disadvantage of this process is **code velocity**, which is the time between having an engineer check the code into a feature branch and that code being integrated into the trunk branch. There are multiple side-effects of the slow code velocity.

First, the delay complicates the process of correcting defects. The delay between the code being written and the defect being found requires the engineer to context switch. The engineer may have moved onto other work while their older code had been migrating through the process.

The second disadvantage is the costs of fixing shallow defects like merge conflicts, where a change in the trunk branch breaks changes being migrated through the branch tree. The slower the velocity of the process then the greater the probability that conflicts will occur.

Lastly, this process complicates **ownership of quality**. If a failure occurs in an integration branch then this may be due to an issue with the branch and verification process or it could be due to a combination of features that have been checked into the branch and at that point it is unclear which engineer is responsible for correcting the defect.

It is worth pointing out that using sub-branches does not necessarily mean the code velocity will be very slow. The slowness is often due to a combination of inadequate tools and rigid policies. For example, the common cause of slow velocity is having builds or tests taking hours to complete. An example of an inflexible process is applying the same verification steps to all changes, irrespective of how complex or simple the code change is, resulting in trivial changes being verified through the same process as that of the most complex changes.

An important consideration in architecting the branching structure is the size of the feature branches (specifically the amount of code and engineers working on the branch). The analysis of the changes in branch architecture for a major product development indicated that the smaller the size of the feature branches the greater the number of merge conflicts that occur as the change migrates through the branch tree. In the initial product development each team had its own branch, which provides a greater level of control to the team. The result is that conflicts with code from other teams in the group are identified only in integration branches, documented through by formally filed bug, probably discussed by the entire triad resulting in an additional amount of process friction. Alternatively, in the development following re-architecture teams shared feature branches which resulted in an increase in continuous integration and the assumption that conflicts were identified and fixed in the feature branch itself way before the issue was noticed at a "system" level on an integration branch.

Another issue that product groups have to consider is the depth of branching between the feature branch and the trunk (i.e. how many integration branches are required). If the focus is on quality then there is a temptation to go to a deep branch structure with strict quality gates managing the movement of code. It is important that these quality gates add value, repeating the same sets of tests at each branch level may guarantee compliance but will also decrease code velocity and increase the probability of 'merge conflicts'[3] occurring. If a quality gate never finds any bugs and the tests from the gates are repeated in branches nearer the trunk then these gates add no values. Analysis of the development process of product groups have identified whole integration branches which added no value to the development process. These branches can be collapsed, increasing code velocity while not impact quality. This emphasises the point that product teams should continually assess whether integration branches are still beneficial and aggressively prune those which are not.

Shallow Branch Trees

An alternative development method is to check code directly into the trunk branch and not use branch trees. This process is viewed, by some, as an essential aspect of the agile development methodology. For existing Microsoft projects, it is still rare to have a shallow branch hierarchy however examples

[3] Merge conflicts is when the code being developed from one set of developers conflicts with code being developed from another set of developers. The conflict is identified when the code collides in a branch or one set of code is integrated into the trunk.

exist. An example of a product which has a single main branch for its current development and has multiple branches for maintaining past releases of the product (see Figure 6).

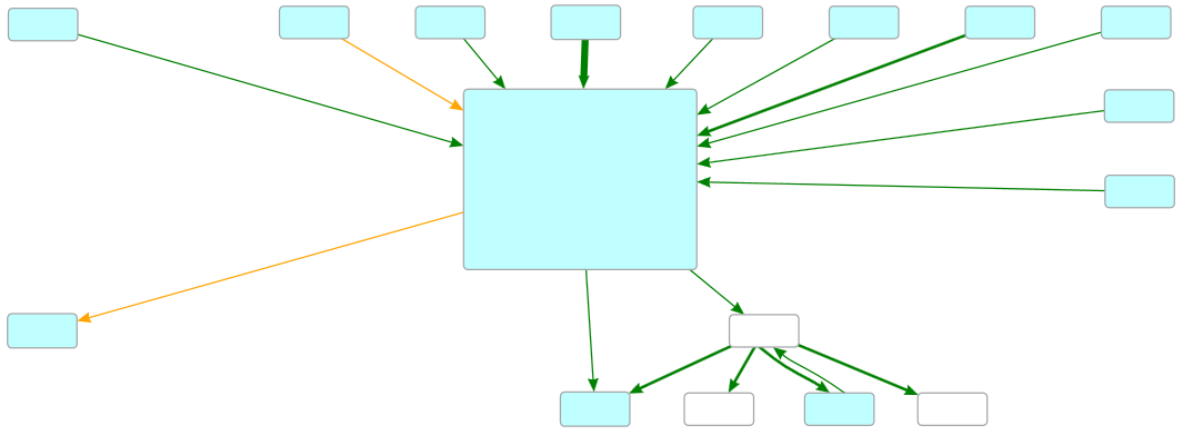

Figure 6. Single trunk branch model used in a product (all other branches are for product maintenence)

There are three main methods of verifying code in the single branch structure:

1. **Pre-check-in verification outside of the branch tree** where all changes are synchronized with the code in the trunk branch, a series of system tests are run on the changes and, if successful, the changes are checked into the trunk branch.
2. **Post check-in verification in the trunk branch** where a change is merged directly into the trunk branch and it is then verified. If the change fails the verification, it is then checked out of the trunk branch.
3. **Periodic verification of the trunk branch** where a number of changes are applied to the trunk branch and then the product is built and verified. This method is the most efficient if the quality of the code being checked into the trunk branch is high. The time between verification is managed by the group and may be based on the number of changes or specific time periods such as a day or a week. If the subsequent verification process fails this process complicates identifying which change is responsible for the failure.

The defining factor impacting the choice of the check-in verification process is **the rate, applicability, impact and thus cost of defects making it into the trunk branch**. The quality of the changes is not only dependent upon the ability of the engineers to complete the feature; it is also impacted by the product-specific constraints. Where the complex constraints exist and where the cost of correcting escaped defects is high, then the greater the verification required prior to the change entering into the trunk branch. Otherwise, the accumulating technical debt will eventually lead to at least requiring designated stabilization periods before the release; a highly undesirable situation in an environment where we are striving for having high release time flexibility.

The advantage of not using a branch tree is that there is **clear ownership** of the change as the engineers submit the changes in the trunk branch. Any issues with the change are identified quickly which increases both the code velocity and decreases the cost of finding shallow defects. This advantage is not necessarily still valid where a team uses periodic verification of the trunk branch.

The disadvantages of this process is that **deep defects**, specifically those discovered a period of time after the change containing the problem is submitted, are more difficult to diagnose. Also, while the process shortens the code velocity, it does not necessarily increase overall productivity. This is because while an individual change is being checked into the trunk branch, the branch itself is locked, specifically no other changes can be checked into the trunk branch as that would undermine the

verification process. Therefore, productivity is determined by the time to verify each change, whereas in using a branch tree any number of changes can be checked into the trunk branch in a single check-in.

The last issue is the ability to make **architectural changes**. Using a single trunk branch architectural changes occur incrementally rather than in step functions. For a change to the UI, this may be preferable but for changes to the architectural core of systems this may not be ideal.

Summary

The following provides an overview of the advantages and disadvantages of the different branch tree architectures.

	Advantages
Deep Branch Tree	Isolation while developing a feature; easier to introduce architectural changes Scales better with number of people Allows for finding deep bugs before code reaches trunk branch
Shallow Branch Tree	Higher code velocity Strong ownership of churn and accountability for fixing issue

Future Requirements for Branch Structures

The previous sections focused on the branch structures applicable to single products. This is the traditional focus of the development whereby the development process is focused on the next major release of the product. All products provide hotfixes and service packs for older versions of the product which are often managed in branches that exist in the same source code version control system as the main product. The product groups ensure bugs fixed in prior releases are also resolved in the current release, to avoid perceived regressions in functionality, but the code flow process between the service branches and the branches of the main development process are often not well optimized.

Changes to the product profile means that the traditional development process and accompanying branching structures may no longer be viable as an effective mechanism. With few exceptions, product groups will simultaneously be involved in at least two of the following:

1. Developing and maintaining a box set product.
2. Developing and maintaining a service version of the product
3. Sharing (and maintaining) components with other product groups within Microsoft.

For instance, as the Windows team develops the next version in both client and server flavours, it maintains all legacy products from Windows XP onwards, they produce Azure and they share code with both Windows Phone and also possibly Xbox.

Simultaneously, developing a number of products, with common functionality but with different target markets and different cadence significantly complicates the development environment. This will bring a new set of issues to the development process, specifically:

Managing the development process through the use of metrics

Unlike the design or implementation of a feature itself, the integration and deployment of completed features or bug fixes into a product is reminiscent of a manufacturing process. As the time between product release decreases and the products are deployed in multiple versions (box set, services, etc.) their development process becomes more complex. Complex manufacturing processes can become unstable and inefficient if they are not continually managed. They can also become unstable if they

are optimized for one aspect of its process to the detriment of all other attributes (reliability over velocity etc.). The traditional way manufacturing processes are managed is through the use of metrics.

Over the last couple of years the CodeMine[4] team has worked with a number of product groups to develop a set of metrics to measure development attributes such as code velocity, deployment velocity, branch performance, organization performance and quality. These metrics can be used by the development team, and their customers, to track the ongoing status of the component development, ensuring they are meeting the goals of the overall project.

Different development and verification methodologies

Product groups may differ in their methodologies but will dependent upon one another's code. Different verification process between groups complicate identifying which group is responsible for defining the quality of shared code.

Variations in branch structures

Even teams working within the same product family may use different branching philosophies. Where code is transferred between different branch systems, each group may require an agreement in regard to timing, frequency and quality of the transferred code.

Features meeting constraints of multiple products

In traditional product development, the verification process will ensure all features meet the constraints of the product. But with parallel development, a feature may go into multiple products, each of which can have different constraints. One solution may be to verify all changes against all constraints of all potential products but that will slow down the development process as most features do not need to meet all constraints.

Product support

Products in the service space do not need to manage legacy products as historical defects are corrected in the current version. Whereas the majority of Microsoft products do need to support past product releases, if they incorporate code from products in the service space then this complicates the question of which groups owns the maintenance of the joint software. The different product groups will require SLAs over product support for shared code, this is especially important for security defects.

Code re-use

There are a lot of advantages for code re-use, but it can cause conflicts where engineers change code and are not aware that other groups depend on it. These conflicts occur within a single product and are caught through their verification process. Where code is shared between products, then conflicts may only become apparent when changes are shared. This would be especially problematic where conflicts occur in software that is of mission-critical nature.

Deployment

The traditional deployment of a product is releasing the product at the end of its development (RTM) or to deploy software as services (Bing). But where parallel, inter-related product developments occur then each product group will be deploying their software to other product group at various cadences. The deployment process is the period between the provider group deciding the version of the software that to be transferred and the receiving product group deploying that software within their product development

The future plans for the product groups will require greater dependencies between them than has existed previously within Microsoft. It is therefore important that a product group does not optimize

[4] http://research.microsoft.com/apps/pubs/default.aspx?id=180138

their product development to the detriment of other product developments that have dependencies on them. Additionally, it is important that overloading the product developments with verification processes does not overburden development with bureaucracy and subsequently delaying the whole software development process.

Ideal Branch System

As different product groups apply different development processes and branching architectures to address their specific product needs and tool and process constraints, it is hoped that the learnings gained will enable a more verified set of recommendations for suitable branching architectures. Such recommendation should eventually take a form of a decision tree that one could apply to a product and a team. We are not yet ready to create a precise recommendation, however we could start from describing an "ideal" branching system; one that would be possible if costs of tools (especially build and test) and processes (creating and deleting a branch) were relatively small.

In this section, we attempt to describe such an idealized branch structure and provide a list of practical considerations for each major decision point as a starting point to a discussion on the future of branch structures.

Identify the objectives of the branching architecture

A software development process should have a clear set of measureable objectives, preferably related to the development of past product releases. These objectives should be based around the goals for quality, release cadence, engineering efficiency, development flexibility, and process efficiency.

There are a number of metrics developed to measure software development process and product groups can choose the most appropriate metrics that best able to characterize their product and environment in which they operate. The product groups also need to ensure they have the framework to allow them to collect these metrics and also a way to interpret the metrics to exclude dirty data and identify gaming. There are no gold standards for metric values therefore each development should compare their performance against past product developments.

It is unlikely that any software development would be able to improve all metrics, as compared with a past release, but rather they should target improvement of those metrics that best reflect the objectives of the next product release, but they need to continually track all metrics to ensure that they are developing their product to plan and not optimizing for one aspect of the process sacrificing the others.

Number of trunk development branches:

The ideal system would conform to the following:

1. One trunk branch per releasable component of the product. Enables variation in processes and release cadence.
2. Components take dependency on other components' interfaces. Enables sharing of code within a product and between products.
3. Ability to point at a version of a dependency and re-point to another when ready.
4. If needed, ability to federate source code and build a tree consisting of both the client code and its dependencies.
5. Each trunk branch forms its own branch hierarchy (development and maintenance) around it.

Practical considerations:

1. Number of components (roughly: trunk branches) depends on the product architecture and components' release cycle.
2. Tools and systems to help manage versioning issues and version conflicts.

3. Ensure development and verification tools scale.
4. Tools and systems for maintaining components products must ensure serviceability of each released version independently.
5. The size of the trunk. The smaller the trunk, the more dependent it is on individual engineers; conversely, the larger the trunk, the greater the risk of forking.

Depth of tree under a trunk branch

The idealized system would conform to the following:

1. All check-ins made directly into the trunk branch by default, *unless* working on a disruptive feature.
2. Child branches created on demand with build and test support for features which are disruptive to the rest of the product. Branches taken down at the end of feature development.
3. Frequent integrations from the trunk branch down to sub-branches assure compatibility with the newest code while ensuring isolation of development for the disruptive feature.

Practical considerations:

1. High rate of failures locking the trunk branch might force creation of temporary or, for very large teams, permanent "level 2" branches where code would be required to meet quality checks before entering the trunk branch.
2. Needs guidelines for when creating a child branch (created on demand for disruptive changes) is beneficial.
3. Needs a decision whether pre-check-in or post-check-in verification process is the default.
4. Pre-check-in verification must be cheap, otherwise developers will combine changes into bigger chunks.
5. Post-check-in verification (with undo if a change is faulty) is optimistic and works only if the rate of failures is relatively low vs. the size of the team which is affected by the blockage.

Maintenance branches:

1. Maintenance branches are children of trunk development branches. Each trunk branch needs at least one maintenance branch, more likely several.
2. One branch per deployment target class, where a class is running the same product version and having the same profile of maintenance issues.
3. If the tree requires more than one branch for maintenance, they are likely connected to each other in a way that either:
    a) Allows for code to meet progressively more stringent constraints (a pipeline with content moving through) or
    b) Ensures that churn in a child branch is a strict subset of the content in the parent to differentiate scope of changes applied to different deployment classes.

Practical considerations:

1. Optimize the tradeoff between the cost and risk of having to check-in (and verify) the same change in multiple branches and the risk of applying the same code changes to multiple classes of deployment targets where at least one of these classes receives churn it does not need.