Perforce Branching

Moving Fast from Theory into Practical Application

Presented by
C. Thomas Tyler
The Go To Group, Inc.

Document Version 1.4.

Table of Contents

1 Introduction 1
2 Branch Model Basics 1
2.1 The Mainline Model 1
2.2 Planned vs. Organic Release Processes 1
2.3 Back to the Mainline 2
2.4 Integration Types 3
2.4.1 A Refresh: 3
2.4.2 A Promotion: 4
2.4.3 Selective Integration: 4
2.5 Planned and Organic Release Processes 4
2.6 More than one Main? 4
3 Directory Structure Considerations with Perforce 5
3.1 How many Depots? 5
3.2 Product Families 6
3.3 Products and Projects 7
3.4 Branch Container Directories 7
4 Release Process Classification 8
5 Sample Case Studies 11
5.1 Case Study #1: Licensed Software, Large Global Development Team 11
5.1.1 Release Process Characteristics 11
5.1.2 Directory Structure 11
5.1.3 Notes 12
5.2 Case Study #2: Classic Large Scale Engineering 13
5.2.1 Release Process Characteristics 13
5.2.2 Directory Structure 13
5.2.3 Notes 14
5.3 Case Study Sample 2 – Organic Release Process 14
5.3.1 Release Process Characteristics 14
5.3.2 Directory Structure 14
5.3.3 Notes 14
5.4 Sample 3– Consulting Model with Customization 14
5.4.1 Release Process Characteristics 14
5.4.2 Directory Structure 14
5.4.3 Notes 15

Copyright © 2007 The Go To Group, Inc. All rights reserved.

1 Introduction

This document provides information helpful when defining a branching structure and corresponding directory structure in Perforce. Included is general information such as directory structure layout considerations and common branching strategy notes. With Perforce's Inter-File™ branching mechanism, the directory structure and branch model are related. A well-defined directory structure helps convey branch structure and software life cycle information, making it intuitive to use.

This document contains generalizations and is not intended to replace a specific assessment for any given environment. It is hoped that the information herein will provide some initial planning information that will help define an initial branching structure.

2 Branch Strategy Basics

Branching strategies are generally intended to meet some combination of the following objectives (some of which are mutually exclusive):

- Provide a promotion path for software changes as they evolve from development, through testing and Quality Assurance, and into Production.
- Allow for a variety of different types of changes to be made concurrently, such as urgent "hot fixes" made to Production code, separated from batches of new development changes.
- Support multiple versions of delivered/released software, delivery patches and updates to released code.
- Allow for a set of planned, structured releases, where new development activities are segregated into distinct efforts. For example, concurrently working on a 2.1 and a 3.0 after a 2.0 has been released.
- Allow for an organically evolving system, where changes are delivered on a very granular level, with each small change being promoted and delivered, perhaps daily.

2.1 The Mainline Model

The so-called Mainline Model is a well established standard branching structure. The Mainline is akin to the origin in geometry; it is the theoretical starting point for branches. It takes into account the objectives identified above, and allows for variations of the code base as it evolves over time. The key concept of the Mainline Model is that variations of the code base must be *justified* and *temporary*. For example, the desire to segregate code in early development from changes in Production justifies maintaining an extra branch of a codeline. However, changes made in Development and Production codelines are encouraged to eventually return to the Mainline, thus reducing variations of the code base to only those needed to support the mission.

2.2 Planned vs. Organic Release Processes

There are many variations of the Mainline Model and how it is employed, and many factors which determine exactly what variations make the most sense for a given software

Copyright © 2007 The Go To Group, Inc. All rights reserved.

product line. For example, the strategy for managing C++ code for a complex and mission-critical application, perhaps with a deep hierarchy of dependencies, to be burned onto chips as firmware would probably follow a planned release model. This would help promote a rigorous test and release cycle that focuses on promoting only fully tested and approved configurations. ASP and HTML changes for a dynamic and fast changing web application, where speed of delivery is paramount to success in the business environment, would be more likely to follow an organic release process. In an organic release process, changes tend to be released in smaller, more granular chunks, resulting in a constant flow of smaller changes to software running in hosted applications in a data center. Organic systems evolve constantly, perhaps hourly in extreme cases. There is no "release" of a 2.2 of the application, instead the large applications many components are modified independently as each change goes through a microcosm of larger development life cycle.

Planned and organic release processes can be combined, such that small changes are delivered in an organic fashion, while larger architectural overhauls are handled as planned releases.

2.3 Back to the Mainline

In each of those situations, the Mainline Model encourages changes to "return to the Mainline", eliminating unnecessary divergence of the code base and helping keep overall software development costs down. Figure 1 below illustrates one simple example of a Mainline Model, with the Mainline (labeled 'MAIN') running through the middle, Dev branches indicating new development efforts (Feature Sets 3.1 and 3.2) below MAIN, and '#.#-R' branches indicating support for released software above MAIN. Changes made on each of the diverging codelines are propagated toward the Mainline. Fixes made in support of released software are typically merged quickly back to the Mainline to make them available for integration into new development efforts. Changes from new development efforts are pushed to and through the Mainline on their way to Production.

Copyright © 2007 The Go To Group, Inc. All rights reserved.

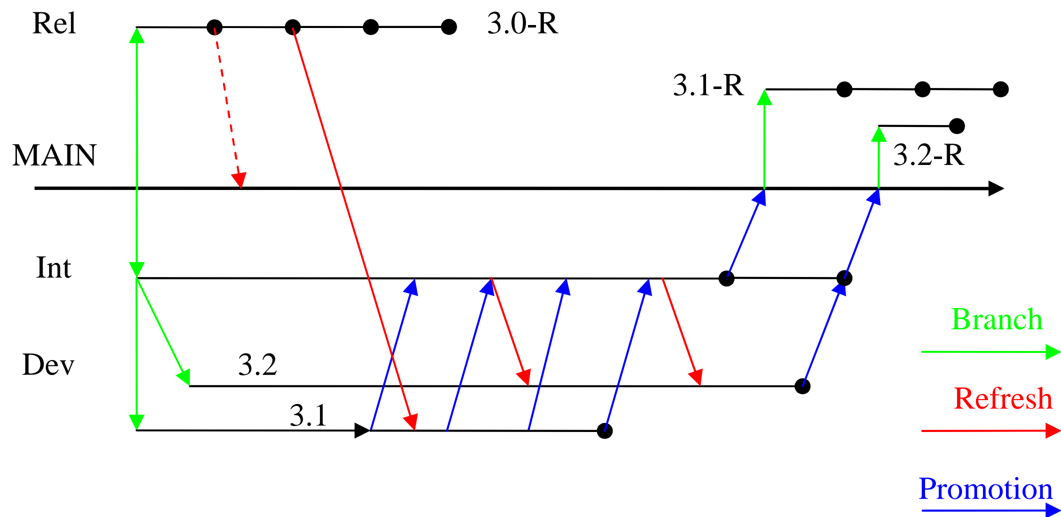


Figure 1: A Sample Mainline Model Branch Diagram

2.4 Integration Types

When using branching mechanisms to support your release process, it is helpful to classify integrations based on the intent of integration along a particular path. Following are some helpful classifications:

2.4.1 A Refresh:

- is intended to integrate changes in one codeline with changes made in other codelines. For example, a refresh might combine patches to a supported product with new development changes.
- is an integration from a more stable to a less stable codeline, e.g. from MAIN to a Dev branch.
- requires potentially complex merge work, and may require manual resolution of conflicts. This is usually started as a '`p4 resolve -am`', causing Perforce to make its best guess at the merge result.
- can introduce instability in the target codeline. It is presumed that the less stable codeline can accept the instability.
- is best performed by someone familiar with the software, requirements, and ideally some insight to the history of changes.

Copyright © 2007 The Go To Group, Inc. All rights reserved.

- is often done as a retail operation, e.g. by subsystem or areas of subject matter expertise.

2.4.2 A Promotion:

- is intended to promote exact copies of tested, trusted software to the next step in the release process, one step closer to Production.
- is an integration from a less stable to a more stable codeline, e.g. from a Dev branch to an Int branch, or an Int branch to MAIN.
- does not require resolution of changes with others, because the files are promoted as they are, verbatim.
- is resolved with '`p4 resolve -at'`, then a submit, followed by a *diff driven merge*, which forces the source and target codelines to match.
- can be performed as a wholesale operation by a centralized Configuration Management or Release Engineering team by people unfamiliar with the software
- promotes the entire codeline from a known state as it meets ever-increasing quality bars for each level of promotion. For example:
  - a promotion from a development branch to an integration branch might require that code compile and pass unit tests.
  - a promotion from an integration branch to the Mainline might require successful completion of directed functional tests.
  - a promotion from Main to a release branch might require that all tests available (regression, performance, stress, etc.) be run.

2.4.3 Selective Integration:

- is intended to "cherry pick" selected changes from a codeline, such as extracting a generic bug fix from a codeline normally used for custom development.

2.5 Planned and Organic Release Processes

Within the context of the Mainline Model theme, there are a great variety of possible release processes, such "Planned Release" and "Organic" processes. First, you want to be sure you're on the most appropriate model for your mission. These classifications will help determine an optimum release structure for any given application.

2.6 More than one Main?

An enterprise may have a single Mainline, or many instances of the Mainline concept. You want to have one at least one Mainline for each type of release process within your organization, based on the release process classifications discussed below. If all developed software products in your organization follow the same release process, they could theoretically all fit under a single Mainline tree, regardless of whether the products share code.

Products and Product Families that follow a similar release process and also share code might live under the same Main directory. Products which have different release processes should have separate Main directories. For example, an enterprise might have set of legacy products nearing end of life, but for which multiple old versions must be

Copyright © 2007 The Go To Group, Inc. All rights reserved.

supported. At the same time a set of active new development efforts for new products, including formal planned releases to licensed software, and also a set of hosted products running in a data center. For such an enterprise, it might have a separate Mainline for the legacy products, one Mainline for the planned release model supporting the licensed software, and yet another Mainline to support organic release process for the hosted applications.

Products within a single product family, or even group of related product families, should share the same Mainline, especially if they share a common code base.

3 Directory Structure Considerations with Perforce

For purposes of this document, we divide the directory structure into the high-level and low-level parts of the directory structure. The lower-level parts of the directory structure, those nearest the root of the tree, convey generic project management information. They help make it clear what a codeline is used for.

For example, if you see the directory `//Gizmos/PROD/BluGizmo`, you might guess (correctly) that production quality code sits in that structure.
If you see `//Gizmos/Dev/BLUGIZMO-3.5/BluGizmo`, you might guess (correctly) that this directory is where you make new development changes for not-yet-released BLUGIZMO-3.5 project. Seeing a few directories in a well organized structure will start to imply a release process, even without any training or discussion.

High level parts of the directory structure vary greatly based on the nature of the software being developed and are beyond the scope of this document.

3.1 How many Depots?

The directory in a Perforce directory structure at the root of the hierarchy is called a depot. This has certain implications for physical storage for the Perforce administrator, but is like any other directory to users. There is no reason to confine development to a single depot, once administrators have installed proper backup procedures to account for the possibility of multiple depots. There are no limitations on code sharing or branching of files in one depot to or from another. Allowing multiple depots allows the top level directory to be meaningful in the directory hierarchy – if everything is under //depot, then the "//depot" directory level isn't helping organize your code base.

Creation of a new depot does require Perforce administration involvement to ensure that new depots are properly created in a manner consistent with the backup procedures, and also that avoid assigning physical storage for a depot to the same area as the Perforce databases. So there are some reasons to discourage rampant proliferation of many depots. But having several depots is typical even in a small organization. Large enterprises typically to have a few dozen.

The number of files involved plays some part in planning. For example, say an organization maintains eighteen web sites, where of which are large complex web

Copyright © 2007 The Go To Group, Inc. All rights reserved.

applications, and fifteen of which are simple microsites. In that case you might have four depots, one for each major application, and one for all the microsites.
Access controls also play a part. If an organization has a need for a particularly secret subset of code, that code might be put into its own depot, to simplify access controls, thus enhancing security.

We recommend against using the default depot in Perforce, named `//depot`, for anything. Use of this depot can cause problems in common corporate merger and acquisition scenarios where two companies desire to combine independent instances of Perforce into a single system. Moreover, it doesn't give the impression of a polished and well thought out directory structure!

The initial set of depots might include the following:

- //3rdParty – Contains 3rd Party/COTS software and tools used by your organization. May also contain branching structures for software delivered in source form and modified locally, making optimal use of Perforce to integrate vendor updates with local modifications.
- //OpenSource – Contains all open source software, either used as tools or built into your product. It is a good idea to segregate open source code into a separate depot, partly to promote re-use, and also to simplify "black duck" analysis (analysis of potential legal liabilities introduced by inappropriate use of open source software).
- //*Gizmo* – Source Code for Gizmo product family
- //*Gizmo*-Build – Build area, populated only by fully automated build processes (no humans allowed). Contains variations in build configurations, such as 32/64 bit, debug/optimized, or Windows/Mac/Linux/Solaris.
- //Giz-Release – Contains as-released software, suitable for distribution to runtime environments, burning to CDs or firmware, or otherwise delivered. This includes files from //Giz-Build, plus various config files, such as DB connection strings, XML files defining app server settings, etc.
- //G2G – Contains Go2Group-deployed and maintained scripts and utilities.

Note that there are sets of related depots. Each product or product family might have a set of depots for source code, builds, and as-deployed files, e.g. //Gizmo, //Gizmo-Build, and //Gizmo-Deploy. A single //Config depot might contain various small config files for all products.

3.2 Product Families

If there is a single product that might eventually grow into a family of related products, the directory structure should account for that by inserting a *ProductFamily* directory level in the directory structure, perhaps using *ProductFamily* as the depot name. For example, you might have:

```
//Gizmos/MAIN/BluGizmo
```

Copyright © 2007 The Go To Group, Inc. All rights reserved.

Where Gizmos is a Product Family and BluGizmo is a software Product within that family.

Any given Product Family can be assigned its own depot. If different Product Families have different release processes, and thus deserve their own instance of a Mainline, those Product Families probably also deserve their own depot.

3.3 Products and Projects

For purposes of this document, a Product is considered to be a long-lived entity. A Project is a temporary concept, with a clearly defined beginning and end. If BluGizmo lived for years or decades, it would be your product, while the BlueGizmo-3.3 and BlueGizmo-3.4 projects came and went.

3.4 Branch Container Directories

We recommend establishing a set of *container directories* to hold branch directories of a certain type. Using container directories helps convey the release process visually through the directory structure.

For example, consider the following structure:

```
//Gizmos/Custom/ACME-C/BluGizmo/…
//Gizmos/Rel/BLUGIZMO-2.0-R/BluGizmo/…
//Gizmos/MAIN/BluGizmo/…
//Gizmos/Int/BLUGIZMO-3.0-Int/BluGizmo/…
//Gizmos/Dev/BLUGIZMO-3.0-FSA/BluGizmo/…
//Gizmos/Dev/BLUGIZMO-3.0-FSB/BluGizmo/…
//Gizmos/PD/ttyler/BLUGIZMO-3.0-FSB/BluGizmo/…
```

This structure seems to imply a promotion scheme. Just looking at the structure, you might make a few educated guesses about the release process:

- The BluGizmo product in the Gizmos product family is undergoing active development.
- A 2.0 version of BluGizmo product has been released. Any changes intended to be released as patches (e.g. BluGizmo 2.0.1, 2.0.2, …) should be made in the BLUGIZMO-2.0-R directory tree.
- There are two concurrent new development efforts, a Feature Set A (FSA) and a Feature Set B (FSB), that at some point will be integrated into the upcoming 3.0 release. (It might be hoped, but not required, that both Feature Sets make it into the release).
- A team assigned to work on Feature Set B within the Gizmo-3.0 project, and one member of that team, ttyler, has embarked on a solo effort within Feature Set B in his personal development branch.
- Software changes needs to pass some hurdles, e.g. unit testing to make it into the **Int** (Integration) area from **Dev**.

Copyright © 2007 The Go To Group, Inc. All rights reserved.

- Software changes need to pass more rigorous testing to go from **Int** to MAIN, the Main integration area. This might require formal QA using targeted functional tests.
- Software changes need to pass the most rigorous testing to go from MAIN to a **Rel** codeline. This might require formal QA doing comprehensive regression tests and formal sign off.
- Regular "new development" software changes start life in some development branch under a **Dev** container directory.
- Maintenance changes start life somewhere under the **Rel** directory.

Using container directories makes it easier for administrators to apply policies across all codelines of a certain type. For example, it is common to require that all changes to released code require some sort of ticket or bug number from an issue tracking system, and that policy could be applied to `//Gizmos/Rel/`….

4 Release Process Classification

Following are a series of questions to ask about your release process. As the questions are answered, a potential directory structure and implied branching structure will evolve.

1. What best describes the primary development/release cycle?
   - Planned Releases – Formal releases are planned, developed, and delivered. Planned releases are further characterized by cycle times for the release, categorized as:
     - Hyper: One release per month or faster.
     - Short: About 3-5 releases per year.
     - Nominal: Roughly one release every 6-18 months.
     - Long: Typically 18+ months per release
   - Organic - An ongoing series of updates or patches rather than formal planned releases. The cycle of Dev->QA->Production still applies, but changes are tested and delivered at a very granular level. A higher number of changes, each a relatively small scope, are delivered to the Production environment more frequently.

Within the **Dev** structure, a *Project* directory level is needed to support concurrent development of Projects in Planned Release model. With the organic approach, all changes originate in a singe **Dev** directory, and are promoted quickly upon completion.

2. Classify your Maintenance Requirements
   - Simple: Minimal maintenance of released products; the product structure isn't expected to change appreciably in maintenance
   - Complex: Extensive, large scale development effort is focused on support of released products, which could take years.

Copyright © 2007 The Go To Group, Inc. All rights reserved.

Rewriting Products in maintenance branches requires a more elaborate Release structure, possibly with equivalents of **Dev** and **Int** branches under the **Rel** tree. This structure and corresponding business practices around it are to be discouraged if practical.

3. What best describes the deployment model of your product:
   - Hosted: No need to support old releases – your clients run whatever software versions are running in the data center.
   - Licensed Software Product: You need to support customers on multiple releases of your software.
   - Burn & Ship: Major releases are shipped (e.g. burned into firmware or CDs). Patches may be required to shipped software.

The hosted business model is very common, in part because it eliminates the need to support customers on old releases of your software product. This works great if your product lives completely within the digital world. If your product is shipped on CDs or burned into firmware, the hosted model is not applicable.

In the Hosted Model, there is typically a single codeline named PROD that represents the software deployed in the data center (or the source code from which software running in the data center was compiled).

With the Licensed and Burn & Ship models, there are typically a **Rel** container directory, and a set of branches that represent supported old releases.

4. Are all changes generic, or is there any need to support customizations?
   - Yes
   - No

This determines whether a **Custom** container directory is required. Typically files in the **Custom** codeline are initially branched from some version of the generic product under the **Rel** container directory.

5. If customization is required, can it be assumed that any given customer will be on exactly one version of whichever products they are using? Or are there customer installations so large that there's a need to support multiple releases of products for the same customer?
   - Simple: Yes, any given customer will have exactly one version.
   - No: We need to account for the possibility that a specific customer might use different versions of our product simultaneously (e.g. one version in their Production environment, another in their Training environment, yet another in an Evaluation environment, etc.).

The answer to this will influence the level of sophistication needed in the **Custom** structure.

Copyright © 2007 The Go To Group, Inc. All rights reserved.

6. How many developers/contributors are involved? How many geographic sites are involved? Is there (or are you trying to form) a formal QA organization?

An **Int** (Integration) container directory may be called for in large development efforts, or for efforts which have formal QA policies and procedures, and/or if the product in question is particularly complex. Note that the Mainline itself is a form of Integration branch, and is sometimes called the Main Integration branch. Adding an **Int** container directory between **Dev** and MAIN is simply an extension of the concept, adding an extra layer of branching. Adding an integration branch provides more opportunities to test, but typically results in a somewhat slower, but more controlled, release process.

Sophistication is justified Complexity. Keep in mind when establishing a branching strategy that each level of branching requires more overhead and labor, yet provides more flexibility and release process control. The challenge is to get just the right degree of complexity/sophistication to achieve your mission.

7. Do users want Personal Development Branches (aka Sandboxes)?
   - Yes
   - No

With Perforce, each user typically has one or more workspaces in which to do their work. A workspace does *not* imply a separate branch. Users each have their own set of files in their workspace, and Perforce supports concurrent development of teams of users on the same files in the same directory in the branch structure. If two or more people modify the same file, they are forced to resolve their changes immediately upon submit. This helps keep development activities well integrated.

However, there are times when a developer may need to embark on a solo effort for a time, perhaps to implement a far-reaching architectural change. In this case, a personal development branch may be called for. A **PD** container directory would contain personal development branches, which are generally per-user, per-project (or simply per-user in an organic model, where there are no Project branches.)

A common policy recommendation for personal development branches is to allow for their use, but not to dictate their use. That way your users will naturally balance the clutter vs. usefulness line, accepting the extra overhead of a personal development branch only when justified by the type of work their doing. A given user might even work both on a personal sandbox branch (to isolate complex architectural changes) and on the regular development branch (on easier enhancements) at the same time.

8. Do we want Per-Bug branches?
   - Yes
   - No
   - Maybe?

Copyright © 2007 The Go To Group, Inc. All rights reserved.

You'll want to balance the value of the per-bug branch vs. the clutter factor. This approach leads to a higher number of branches. If you couple this approach with a sparse branching methodology that keeps branches small, or if you simply don't have very many files, this should work fine. You'll just have many small branches. But if you have say 20,000 files to branch, and a full population (non-sparse) branch policy, and you create many per-bug branches, you'll soon be overwhelmed in clutter. Perforce is particularly efficient at handling branching operations, but it can still be overloaded with excess clutter.

5 Sample Case Studies

Following are some sample case studies of environments, with resulting directory structures, based on various release process classifications. In all the samples below, only the source code depots is shown, since the branching strategies discussed here apply primarily to source code.

5.1 *Case Study #1: Licensed Software, Large Global Development Team*

The fictional company Acme, Inc. develops one flagship product, Giz, and distributes it as licensed software on their web site. They hope to add another product, Gyro, in the next year. They have a large formal QA organization. They plan to deliver only bug fixes on released software, focusing development activity forward on new releases. They recently released 1.0, and are currently working concurrently on 2.0 and 2.1.

5.1.1 Release Process Characteristics

- Planned Releases
- Nominal Release Cycles
- Large, Multi-Site Teams
- Simple Maintenance
- No Customization Support – Generic Product Only
- Sandboxes used sparingly

5.1.2 Directory Structure

//Eng/
  Rel/<PROJECT>-R/[<ProductFamily>]/<Product>/...
  MAIN/[<ProductFamily>]/<Product>/...
  Int/<PROJECT>-INT/[<ProductFamily>]/<Product>/...
  Dev/<PROJECT>/[<ProductFamily>]/<Product>/...
  PD/<User>/<PROJECT>/[<ProductFamily>]/<Product>/...

5.1.3 Directory Structure Diagram

Copyright © 2007 The Go To Group, Inc. All rights reserved.

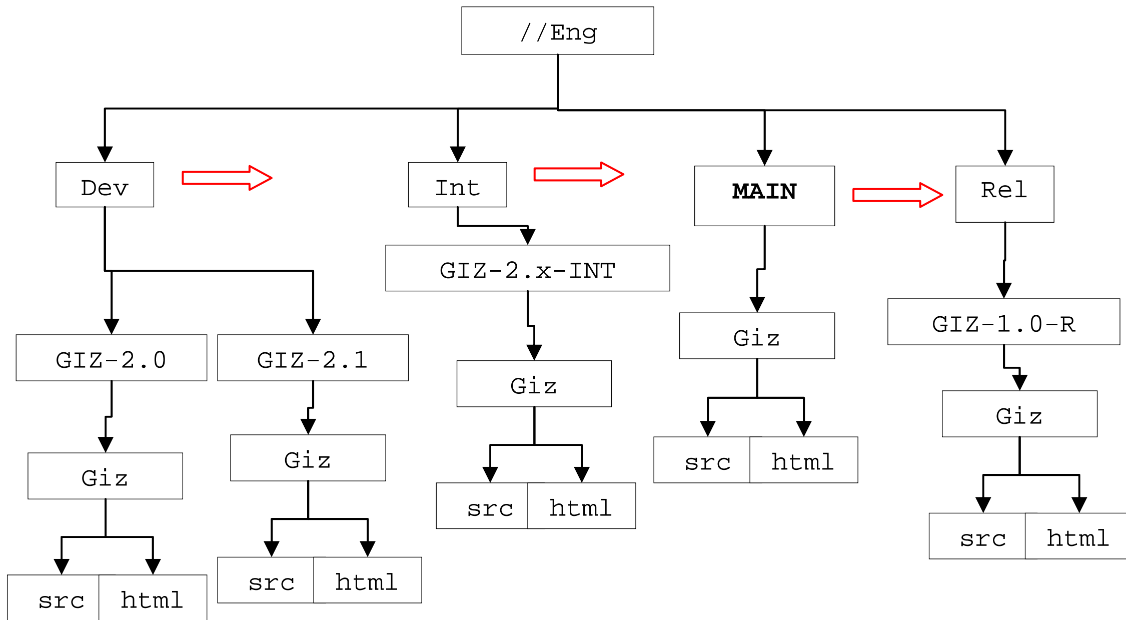


Figure 2: Planned Release Structure

5.1.4 Notes

- Typical structure. Files are branched at and below the UPPERCASED dirs.
- Red arrows indicate promotion within the SDLC.
- The '-R' and '-INT' seem to be redundant, but are helpful when those branch directories are used in other contexts.
- In the **Rel** structure, Project names look like GIZ-1.0-R
- In the **Int** structure, Project names look like GIZ-1.1-Int
- The Sandbox container directory is not shown.
- Sometimes the best practice is to select project names that *do not* imply a release order. In that case, concurrent development projects are given some identifier that is associated with a set of features/functionality to be delivered, rather than the intended order of release. This advanced approach adds project management flexibility at the cost of some extra complexity.
- The diagram depicts a directory structure that aids in comprehending the underlying branching structure, even though the branching structure isn't directly depicted.
- This model would have the following branch specs defined:

Branch Type	Branch Spec	Source	Target
Personal Dev	PD.juser.GIZ-2.0.B	//Eng/Dev/GIZ-2.0/Giz/…	//Eng/PD/juser/GIZ-2.0/Giz/…
Dev	GIZ-2.0.B	//Eng/Int/GIZ-2.x-INT/Giz/…	//Eng/Dev/GIZ-2.0/Giz/…
Dev	GIZ-2.1.B	//Eng/Int/GIZ-2.x-INT/Giz/…	//Eng/Dev/GIZ-2.1/Giz/…
Int	GIZ-2.x-Int.B	//Eng/MAIN/Giz/…	//Eng/Int/GIZ-2.0-INT/Giz/…

Copyright © 2007 The Go To Group, Inc. All rights reserved.

Rel	GIZ-1.0-R.B	//Eng/Rel/GIZ-1.0-R/Giz/…	//Eng/MAIN/Giz/...

Table 1: Branch Specs for Case Study #1

5.2 *Case Study #2*: *Embedded Systems Engineering*

The fictional company Acme2, Inc. develops one flagship product, Bali, which is an embedded system. They have no formal QA organization and no immediate plans to form one. They plan to deliver only bug fixes on released software, focusing development activity forward on new releases. They recently released 1.0, and are currently working concurrently on 2.0 and 2.1.

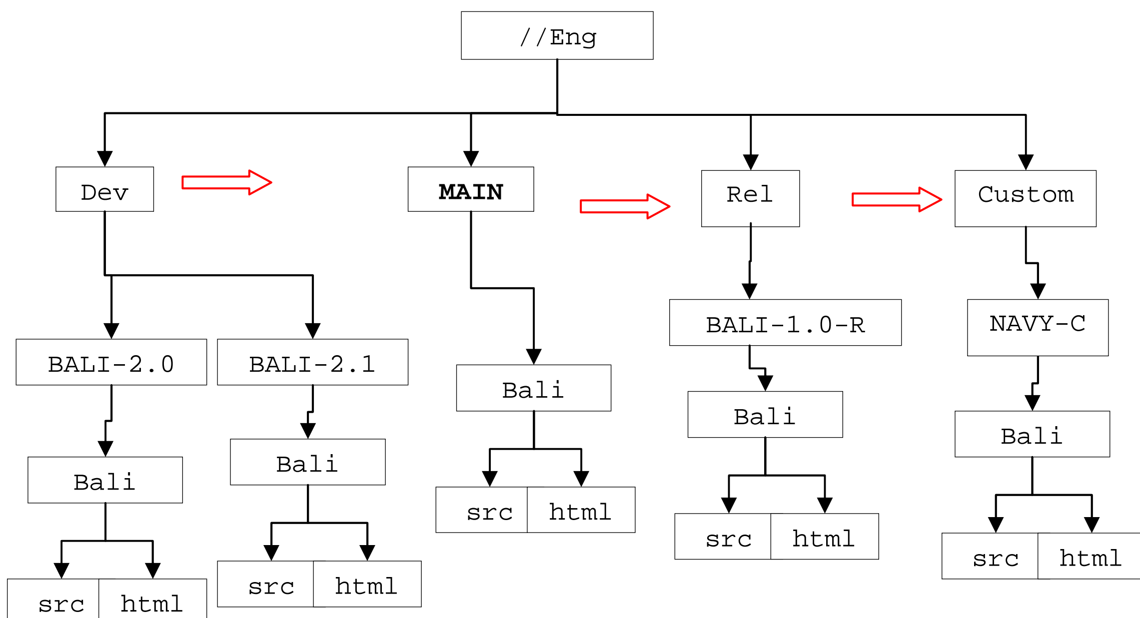
5.2.1 Release Process Characteristics

- Planned Releases
- Long Release Cycles
- Simple Maintenance
- Simple Customization Support

5.2.2 Directory Structure

//Eng/
  Custom/<CUSTOMER>-C/[<ProductFamily>]/<Product>/...
  Rel/<PROJECT>-R/[<ProductFamily>]/<Product>/...
  MAIN/[<ProductFamily>]/<Product>/...
  Dev/<PROJECT>/[<ProductFamily>]/<Product>/...

5.2.3 Directory Structure Diagram



Copyright © 2007 The Go To Group, Inc. All rights reserved.

5.2.4 Notes

- Typical structure.
- Project Names may be something like 3.0 if that applies to everything under that Mainline, or <Product>-<Version>, e.g. Bali-3.0.
- In the **Rel** structure, Project names look like BALI-1.0-R
- Sometimes the best practice is to select project names that *do not* imply a release order.

5.3 Case Study #3: Hosted Model with Organic Release Process

5.3.1 Release Process Characteristics

- Organic Release Process
- Products organized into Product Families
- Small Development Team
- No Customization Support
- No Concurrent Development
- No Maintenance of old releases

5.3.2 Directory Structure

//Eng/
  PROD/<ProductFamily>/<Product>/...
  MAIN/<ProductFamily>/<Product>/...
  DEV/<ProductFamily>/<Product>/...

5.3.3 Notes

- No <Project> level – new development consists entirely of small-scale changes.
- Note: It is possible to mix Planned and Organic releases processes for a single product, though this requires disciplined classification of changes prior to commencing implementation.

5.4 Case Study #4: Consulting Model

5.4.1 Release Process Characteristics

- Organic Release Process
- No "Production" environment for generic product; all final deliveries are Custom
- No formal QA
- Small Development Team

5.4.2 Directory Structure

//Eng
  Custom/<CUSTOMER>-C/<ProductFamily>/<Product>/...
  MAIN/<ProductFamily>/<Product>/...
  DEV/<ProductFamily>/<Product>/...

Copyright © 2007 The Go To Group, Inc. All rights reserved.

5.4.3 Notes

- No formal QA, thus no **Int** or **Rel** structures.
- Fast path from **Dev** to **Custom**, since products tend to be simple (e.g. Perl scripts).
- Generic product is thought of as seed code, delivered to Systems Integration organization for delivery to customers.
- Generic changes that happen to start as **Custom** should be merged ...
    - Directly to Main if both the change AND merge are trivial
    - to **DEV** if either change OR the merge isn't trivial

Copyright © 2007 The Go To Group, Inc. All rights reserved.