

Perforce Software, Inc.

510.864.7400

consulting@perforce.com

[www.perforce.com](http://www.perforce.com)

Perforce Directory Standard

for

EDITME-YOUR-COMPANY

Table of Contents

[1 Audience 1](#_Toc363603714)

[2 Introduction 1](#_Toc363603715)

[2.1 FGS, The Friendly Greeting System 1](#_Toc363603716)

[2.2 Branching Strategies and the PDS 2](#_Toc363603717)

[2.3 Perforce Streams, Classic Perforce, and the PDS 2](#_Toc363603718)

[3 Review of Branching Strategy Basics 2](#_Toc363603719)

[3.1 Branching Goals 2](#_Toc363603720)

[3.2 The Mainline Model 3](#_Toc363603721)

[3.3 Back to the Mainline 3](#_Toc363603722)

[3.4 Merge Down, Copy Up 6](#_Toc363603723)

[3.4.1 Merge Down 6](#_Toc363603724)

[3.4.2 Copy Up (Promotion) 7](#_Toc363603725)

[3.4.3 Selective Integration 8](#_Toc363603726)

[3.4.4 Planned vs. Organic Development 8](#_Toc363603727)

[3.4.5 Organic Development 8](#_Toc363603728)

[3.4.6 Planned Development 9](#_Toc363603729)

[3.4.7 Combining Planned and Organic Development 10](#_Toc363603730)

[4 Branching Patterns and Release Processes 10](#_Toc363603731)

[5 Terminology 12](#_Toc363603732)

[6 Directory Structure Standard 13](#_Toc363603733)

[6.1 Overview for main 13](#_Toc363603734)

[6.2 Under main 14](#_Toc363603735)

[6.3 Permanent Streams (Reserved Names) 15](#_Toc363603736)

[6.3.1 live 15](#_Toc363603737)

[6.3.2 main 15](#_Toc363603738)

[6.3.3 latest 15](#_Toc363603739)

[6.4 Projects 16](#_Toc363603740)

[6.4.1 Project Branch Names 16](#_Toc363603741)

[6.4.2 Version Identifiers 16](#_Toc363603742)

[6.5 Branch Container Directories 17](#_Toc363603743)

[6.5.1 svcs 17](#_Toc363603744)

[6.5.2 custom 17](#_Toc363603745)

[6.5.3 rel 18](#_Toc363603746)

[6.5.4 int 18](#_Toc363603747)

[6.5.5 dev 19](#_Toc363603748)

[6.5.6 exp 21](#_Toc363603749)

[6.5.7 demo 21](#_Toc363603750)

[6.5.8 sb 22](#_Toc363603751)

[6.6 Sample Structures 22](#_Toc363603752)

[6.6.1 Basic Planned Dev + Advanced Maintenance 22](#_Toc363603753)

[6.6.2 Advanced Planned Dev + Advanced Maintenance 23](#_Toc363603754)

[6.6.3 Advanced Planned Dev + Basic Maintenance + Customization 23](#_Toc363603755)

[6.6.4 Advanced Planned Dev + Basic Maintenance + Sandboxes 23](#_Toc363603756)

[6.7 Import Structures 24](#_Toc363603757)

[7 Version Control “Areas” 24](#_Toc363603758)

[7.1 Source Area 24](#_Toc363603759)

[7.2 Build Area 25](#_Toc363603760)

[7.3 Release Area 25](#_Toc363603761)

[7.4 Third Party Area 26](#_Toc363603762)

[7.4.1 Binary and Source Distributions 26](#_Toc363603763)

[7.4.2 Modifications to Third Party Software 27](#_Toc363603764)

[7.5 Documentation Area 27](#_Toc363603765)

[7.6 User Area 28](#_Toc363603766)

[8 Perforce Depot Names 29](#_Toc363603767)

[8.1 Standard Depots 29](#_Toc363603768)

[8.2 Optional Depots 29](#_Toc363603769)

[8.3 How Many Branch Levels? 29](#_Toc363603770)

# Audience

This document is intended for all EDTIME-YOUR-COMPANY software engineering staff using Perforce, as well as those responsible for the management of software products managed in Perforce.

**THIS DOCUMENT IS NOT A PROPOSED STANDARD FOR ANY PARTICULAR ORGANIZATION.**

Creating a standard for any particular organization requires intimate knowledge of that organization’s business processes, change drivers, constraints, etc. This document is intended to provide a sample standard that can be used as source content by a CM practitioner in an organization to develop a standard for their organization.

Perforce offers experienced consultants to help apply best practices to produce a custom PDS (including document and accompanying presentation) for your organization.

# Introduction

This document defines the Perforce Directory Standard (PDS). The PDS defines a common Perforce directory structure that intuitively conveys the life cycle stage and simplifies parallel development.

Managing the flow of change is essential to optimizing parallel development. For example, a software change that fixes a bug in an old, released version of a software product should be propagated to new development streams, to ensure that bug does not recur. The change should also be propagated to any other actively supported release streams. In practical terms, managing the flow of change means providing pathways for the propagation of software changes from the stream in which they originated to various destination streams. The PDS simplifies parallel development by providing a structure that helps clarify the paths along which changes must be propagated.

The PDS is very flexible, and supports a wide variety of software development processes and branching patterns. Some teams develop and support licensed software products. Other teams develop hosted software solutions operated from a data center. Still other teams might provide professional services, using generic product as seed code from which custom solutions are developed. The PDS provides a common Perforce structure that is flexible enough to meet the diverse parallel development needs of all these teams, while still providing a familiar structure for all.

This document refers to PowerPoint slides in **PDS-Diagrams.pptx** (v5.2). That file should be available from the same source as this document.

## FGS, The Friendly Greeting System

FGS is the Friendly Greeting System, a fictional software product name for the classic “Hello, World!” software program. The //fgs [depot](http://www.perforce.com/perforce/doc.current/manuals/p4sag/03_superuser.html#1044923) is used in many examples in this document.

## Branching Strategies and the PDS

The core themes of branching theory have been well established since the 1990s. However, there are a great many variations on the theme, as the conventional wisdom is applied to different development organizations. This document provides background information and common terminology that will be helpful in understanding how the PDS can be applied for each team using Perforce. The physical directory structure and [branching](http://www.perforce.com/perforce/doc.current/manuals/p4guide/06_codemgmt.html#1065698) strategy are intimately related in Perforce.

Branching is achieved by making a tracked copy of a file in one location to another location. The “from” and “to” locations of the branch/copy operation are defined in the PDS. The PDS helps clarify the paths and destinations of software changes to be propagated by merging.

## Perforce Streams, Classic Perforce, and the PDS

Most of the concepts in the PDS apply to regardless of whether the Perforce feature known as Streams is used or not (commonly referred to as “Classic Perforce”). When using the Streams feature, some of the best practices promoted in the PDS are built into Perforce. Differences in Stream and non-Stream usage are identified throughout the document.

With Streams, the physical directory structure is flattened, and the recommended branching strategy is codified and indicated in the visual tools. In particular, the intended flow of change is communicated via the Stream Graph.

# Review of Branching Strategy Basics

What follows is a brief summary of branching strategy basics. The book [*Practical Perforce*](http://oreilly.com/catalog/9780596101855), by Laura Wingerd, is recommended for more detailed information.

## Branching Goals

Branching strategies are generally intended to meet some combination of the following objectives:

* Optimize productivity of the software development process by enabling parallel development.
* Allow for a variety of different types of changes to be made concurrently, such as urgent “hot fixes” made to Production code, separate from batches of new development changes.
* Support multiple versions of delivered/released software, delivery of 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, concurrent work on a 2.1 and a 3.0 version, after a 2.0 version has been released.
* Provide a clear promotion path for software changes as they evolve from development, through testing and Quality Assurance, and into Production.
* 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.

## The Mainline Model

The *Mainline Model* is a well-established standard branching strategy. The Mainline is a persistent branch follows the evolution of a software product through its entire history. The key concept of the Mainline Model is that branches are *justified* and *temporary*. For example, the desire to segregate code in early development from changes in Production justifies maintaining an extra branch. However, changes originating in Development and Production branches are encouraged to eventually return to the Mainline, reducing variations of the code base to only those needed to support the mission.

Visualizations of the mainline model generally start by illustrating a permanent main branch, with various streams branched from it. You might branch temporarily from main to maintain released software or start new development efforts. In these common software development workflows, the Mainline Model exerts a gentle convergence pressure, reducing unnecessary divergence of your code base.

There are scenarios where permanent divergence of some branches is part of your mission. In these cases the Mainline Model helps by illustrating the ancestry of any particular divergent branch.

There are many variations of the Mainline Model and how it is employed, and many factors that influence which variations make the most sense for a given product line.

## Back to the Mainline

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 illustrates one 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 streams of development 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 other branches. Changes from new development efforts are pushed to and through the Mainline on their way to being released.

In this sample, the **main** branch is the only long-lived branch, while all others have a defined lifespan. Development branches are closed when the feature set is complete and development activity transitions to the release branches. Release branches are closed when the business makes a decision to no longer support that release (though it can be re-opened later if necessary).

**Mainline**

Integration

3.2

Release

Development

R-3.0

R-3.1

R-3.2

3.1

Branch

Merge

Promotion

Figure 1: A Sample Mainline Model Branch Diagram

The equivalent stream representation is shown below.



Figure 2: Stream graph of mainline model

As Figure 2 indicates, the streams framework in Perforce models the relationship between streams (the hierarchy) and indicates the expected flow of change between streams.

With Streams, each stream is classified as a defined as being of a certain type (release, mainline, development, task, and virtual). Each type has an associated icon. With Classic Perforce, stream types are identified by naming convention and usage practices; the actual distinction is not known Perforce.

Note that an Integration branch as illustrated in the diagram is a stream of type ‘development’, positioned below the mainline and above other development streams.

## Merge Down, Copy Up

The phrase “Merge Down, Copy Up” has long been a mantra to help instill a key best practice of branching. Branch diagrams such as the one in Figure 1 are illustrated with horizontal lines depicting branches, with time moving forward from left to right. Lines drawn toward the top of the diagram are considered more “firm,” as they typically have a high cost associated with them being broken, and are generally viewed less tolerant of churn. Lines toward the bottom are considered “soft,” and have a greater ability to absorb change. Drawing diagrams consistently with the same up/down direction is helpful to stay in line with Perforce terminology, at least initially.

In the stream graph in Figure 2, the hierarchy follows the principle that more stable streams are shown towards the top of the graph.

It is helpful to classify integrations based on the intent of integration along a particular path. Below are some helpful classifications:

### Merge Down

A Merge Down:

* Is intended to integrate files changed in one branch with corresponding files in other branches. For example, a merge down might combine patches to a supported product with new development changes.
* Is an integration from a more firm to a less firm branch, in the direction away from release branches, e.g. from **main** to a development branch, or a release branch to **main**.
* Uses the 'p4 [merge](http://www.perforce.com/perforce/doc.current/manuals/cmdref/merge.html)' and 'p4 [merge](http://www.perforce.com/perforce/doc.current/manuals/cmdref/resolve.html)' commands.
* Requires potentially complex merge work, and may require manual resolution of conflicts. The resolve is usually started as a 'p4 [resolve](http://www.perforce.com/perforce/doc.current/manuals/cmdref/resolve.html) -am', causing Perforce to make its best guess at the merge result.
* Can introduce instability in the target branch. It is presumed that the target branch can accept the instability, as it is farther from release.
* Is best performed by someone familiar with the software and requirements, and who has some insight to the history of changes, and business and technical drivers for those changes.
* Is often done as a piecemeal operation, e.g. by subsystem or areas of subject matter expertise or areas of code ownership/responsibility.

In the stream graph shown in Figure 2, the arrows between streams indicate when a merge down is needed. The presence of an arrow showing change flowing down indicates that a merge down is possible, while a highlighted arrow indicates that a merge down is pending (Figure 3).



Figure 3: Pending merge down

### Copy Up (Promotion)

Copy Up (Promotion):

* Is intended to promote exact copies of tested, trusted files to the next step in the release process, one step closer to Production.
* Is an integration from a less stable to a more stable branch, in the direction toward Release branches, e.g. from a development branch to **main**.
* Does not require a resolve step, because the files are promoted as they are, verbatim.
* Uses the 'p4 [copy](http://www.perforce.com/perforce/doc.current/manuals/cmdref/copy.html)' command.
* Can be performed as a wholesale operation by a centralized Configuration Management or Release Engineering team. A promotion is done by people familiar with the release process, who don’t necessarily need to be familiar with the software.
* Promotes the entire branch 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.
	+ The initial promotion from **main** to a release branch might require that all tests available (regression, performance, stress, etc.) be run. It may be the case that more stabilization work is needed on the release branch before actual release of the software, so running all tests does not necessarily imply that the software must pass all tests.

In the stream graph shown in Figure 2, the arrows between streams indicate when a copy up is appropriate. The presence of an arrow showing change flowing up indicates that a copy up is possible, while a highlighted arrow indicates that a copy up is pending (Figure 4). Note that a copy up may not be allowed by the stream owner if it is not appropriate for a particular stream pair.



Figure 4: Pending copy up

### Selective Integration

Selective Integration:

* Is intended to “[cherry pick](http://www.perforce.com/perforce/doc.current/manuals/p4guide/06_codemgmt.html#1040533)” selected changes from a branch, such as extracting a generic bug fix from a branch normally used for custom development.
* Uses 'p4 [integrate](http://www.perforce.com/perforce/doc.current/manuals/cmdref/resolve.html#1040665)' with both *start* and *end* revision specifiers for the source of the integration, as in this example:

p4 [integrate](http://www.perforce.com/perforce/doc.current/manuals/cmdref/resolve.html#1040665) –b FGS-R1.0.B //fgs/main/src/foo.c#23,#24

Selective integrations introduce complexity into the integration history. Their use should be relegated to infrequent processes like cherry picking of selected individual changes. If you find selective integrations being used for routine merge down/copy up (promotion) activity, the branching strategy should be reviewed.

### Planned vs. Organic Development

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 promotes 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 more likely 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 tagged release v2.2 of the application. Instead the large applications’ components are modified independently as each change goes through a microcosm of a larger development life cycle.

### Organic Development

With organic development, work occurs directly in a /main branch. The primary advantage (and arguably disadvantage) of organic release processes is that they do not require branch planning or creation of branches to commence new development activity.

Organic development seems very simple at first. It incurs less overhead of managing branching activities, like merge downs and “copy up” promotions. It is most appropriate for small development teams, teams with no need for parallel development, and end-of-life products with limited development activity.

Merges are required when two developers modify the same file in /main at the same time. Perforce forces those merges to be resolved immediately, prior to accepting the submit. Merge downs are also required as well to pick up changes made in release maintenance branches, if they are used.

The simplicity of organic development is appealing. However, organic development has several limitations that make it inappropriate for many teams. Among the limitations are:

* Potentially Unstable Mainline. The stability and maturity of any given branch is generally determined by the least stable or mature chunk of work going on in that branch at any given time. Organic processes thus suffer from periods where the mainline is unstable. This can create a situation where efforts of one team impact the efforts of another. For example, say there is one change that is nearly ready for delivery, but another change is a work in progress. The work in progress can impact the ability to deliver the otherwise ready change.
* Not Agile. Organic processes do not support agile development well. Agile processes encourage keeping an “always ready” mainline, providing the ability to ship a product at any given time, with whatever features are ready to go at any given point in time. Here, “ready to go” may mean only “ready to go to our internal QA team.”
* Parallel Development is not well supported. If you have delivered 1.0 and want to simultaneously work on 1.1 and 2.0 as distinct efforts, the organic model doesn’t support this well, since both efforts share the same branch.

The idea of organic development isn’t strictly limited to the /main branch. There is always some degree of organic development in the lowest level branch of any given branching structure, no matter how sophisticated it is. It is common for software products to start with an organic process, and later evolve to adopt planned release processes. Common drivers of this evolution include:

* Increasing amount of change
* Increasing team size
* Need for parallel development
* Need for greater process agility
* Need for a stable mainline
* Desire for temporary isolation of distinct development activities.

### Planned Development

Planned development is characterized by the creation of one or more levels of branches from /main, prior to start of any work on a project. For instance, we may have development project branches associated with a particular set of features being developed, and integration branches that contain a series of development project branches.

It is typical in planned development to disallow direct edits on /main, allowing only integration operations. Thus, a policy might define that /main only receive promotions from software branches, or merge downs from firmer branches.

### Combining Planned and Organic Development

Combining planned and organic development processes is possible. It can be done using a permanent /dev/latest organic development branch in conjunction with planned development project branches (e.g. /dev/64Bit or /dev/3.2). In this manner, the actual mainline, /main, is kept stable, receiving promotions either the permanent /dev/latest organic branch or any of the project branches. Promotions occur from branches only when they are tested or at least thought to be ready for testing. Project branches are used for complex and potentially destabilizing changes which might need to be aborted, while smaller-scoped changes are made on the organic branch.

Alternately, the /main branch itself can be used for organic development, with project branches used for more complex changes. This approach can decrease stability of /main, but eliminates the need to maintain a separate /dev/latest organic development branch. It works best with smaller teams of well-disciplined developers, as it allows and requires individual developers to classify each change as being suitable for submitting directly to main, or being relegated to a project branch.

In some variations on the theme, different rules apply to different types of changes. For example, changes to art work and graphics files might be submitted directly on /main, perhaps by non-technical content producers. Meanwhile, changes to source code follow a different process, being initiated in a development branch and promoted to main only at discrete points, such as after testing at the end of a [SCRUM](http://en.wikipedia.org/wiki/Scrum_%28development%29) sprint.

# Branching Patterns and Release Processes

A *branching pattern* models the software development process of a product during some phase of its life cycle (e.g. R&D, development, maintenance). Each branching pattern has certain characteristics. For example, the Organic Development branching pattern can apply to any phase of the development process. Its characteristics are that it enables immediate development, with little up front planning (at least in terms of the branching strategy), but does not support parallel development well. The Patch Maintenance branching pattern applies to the maintenance phase of the software life cycle. It enables support of large, complex software products that are actively supported for years.

Table 1: Branching Patterns Supported by PDS

|  |  |  |
| --- | --- | --- |
| **Type** | **Pattern** | **Characteristics** |
| Development | Organic | Enables immediate implementation of changes without prior planning. Can be used by itself, or in conjunction with Basic/Advanced Planned Release models. Is used in Basic/Standard Hosted models. Serial development; parallel development is extremely limited. |
| Basic Planned | Requires planning prior to initiation of development activity. Provides feature sets to segregate various development activities. Enables parallel development during development phase of the life cycle. |
| Advanced Planned  | Requires planning prior to initiation of development activity. Provides feature sets to segregate various development activities, as well as Integration branches. Enables parallel development during development phase of the life cycle. Appropriate for large teams with long release cycles. |
| Maintenance | Basic | Used to support released software for scenarios where released software is not expected to change significantly. Enables parallel development during the maintenance phase of the life cycle. |
| Advanced | Used to support released software over extended periods of time, where large changes may occur in maintenance. Enables parallel development during the maintenance phase of the life cycle. |
| Patch | Used to support released software over extended periods of time, where large changes may occur in maintenance. Enables extensive parallel development during the maintenance phase of the life cycle. |
| Customization |  | Tracks changes of generic software product made by professional services organizations. Provides pathways to ‘cherry pick’ customizations back into the generic product. |
| Hosted | Basic | Provides **main** and **live** streams. Appropriate for smaller teams. |
| Standard | Provides **latest**, **main**, and **live** streams. Appropriate for hosted products for which QA processes are integral. |
| Standard w/EBF | Provides an Emergency Bug Fix workflow. |

A *release process* for any given product combines one or more branching patterns to achieve a comprehensive process with desired characteristics. For example, one team might combine the Basic Planned Development with Patch Maintenance and Customization branching patterns to form their release process. A larger team might combine Advanced Planned Development with Basic Maintenance.

Over the life cycle of a software product, its release process may change. For example, say a software product is in its first few years of life, with many active contributors. At that point, it may follow a release process consisting of Advanced Planned Development and Patch Maintenance branching patterns. Then a few years later that product transitions from active development to legacy support mode, with most of the people moved to other product lines. Changing the release process, perhaps moving to Organic Development and Basic Maintenance, would reflect this business change.

Table 1 shows the list of branching patterns supported in the PDS. Each branching pattern implies a certain number of levels in the directory structure. The accompanying PowerPoint presentation, **PDS-Diagrams.pptx** v4.8, illustrates the directory structures for each branching pattern and provides animated illustrations of sample release process scenarios.

# Terminology

This section provides terminology to help inform discussions about the PDS.

Product (*noun*): The result of software development activity, represented as a set of files in one or more directories that are built and delivered together. Large software development efforts typically consist of layers of products, with some products (e.g. core libraries) serving as components of other products. Each entity that is versioned separately can be thought of as a separate software product for the purposes of this document.

The use of the word “Product” implies that you are using Perforce to manage software development efforts that result in a software product. However, the term “Product” as used in this document should be interpreted very loosely. A Product could be a hosted web application, a third party or open source component, or even a set of files that represent the configuration settings of hardware devices, such as routers.

For purposes of this document, a Product is considered to be a long-lived entity.

Project (*noun*): A set of changes to a product with a clearly defined beginning and end. If FGS lived for decades, it would be a product, while the FGS-3.3 and FGS-3.4 projects came and went.

Baseline (*noun*): A reference to the set of files that constitute a given software product at a point in time. Baselines are typically tagged with a version identifier of some kind.

Branch (*noun*): A tracked copy of the set of files that represent a product. A baseline identifies the versions of files from which a branch is initially created (i.e. from which the file were initially copied).

Branch (*verb*): The activity of creating a branch from a given baseline.

Stream (*noun*): A named set of development activities, supported by branches of one or more software products. The terms *stream* and *branch* are often used interchangeably.

[Integrate](http://www.perforce.com/perforce/doc.current/manuals/cmdref/integrate.html#1040699) (*verb*): Propagating changes across development streams (and therefore across branches).

Merge (verb): Same as [Integrate](http://www.perforce.com/perforce/doc.current/manuals/cmdref/integrate.html#1040699).

Branch Diagram: A diagram illustrating planned or existing baselines and branches for a product. Branch diagrams help illustrate an organization’s plans for propagating changes across related streams of development. When using Perforce streams, the branch diagram is represented in the stream graph.

# Directory Structure Standard

Prior to Perforce streams, the directory structure was used to convey a great deal of information about the branch hierarchy. The streams framework embeds most of that information into the stream metadata directly, by storing the parent-child relationship between streams. Therefore, the stream directory structure if flattened.

When using streams, the recommended practice is to use one Perforce stream depot per product. All stream names form the second level of the directory structure. Some contextual information about the purpose of a stream (e.g. integration streams, QA streams) can be conveyed in the stream name rather than the directory path, and the stream graph helps to present an intuitive visual picture of the stream model. The 'p4 streams' command also supports filtering by stream name.

In some situations, using stream names to convey context in a single stream depot is not scalable. For instance, maintaining customized streams for several customers may require several de-facto mainlines. Consider using multiple mainlines in a single stream depot, or even a separate stream depot per customer.

The examples below include the PDS for non-streams usage and the equivalent streams usage.

## Overview for main

The PDS for **main** is:

//*TopLevelDir*/*Product*/main

or, optionally,

//*Product*/main

When using streams,

//*Product*/main

The *TopLevelDir* directory level is an optional directory level, though it should be included if it applies. This directory level organizes groups of software products that are somehow related, either mechanically or managed by the same business unit.

*ProductFamily* or *BusinessUnit* can be substituted for *TopLevelDir*. The ACME Enterprise Configuration Management team is solely responsible for determining the list of valid top level directory names, as well as determining cases where it is appropriate to use just a *Product* name at the top level.

The first directory in a Perforce directory structure (the text after the “//”) is called the *depot*[[1]](#footnote-1). Depot names are restricted to **lowercase** letters and numbers, but cannot be purely numeric. Special characters ‘.’, ‘-‘, ‘\_’, and ‘+’ are allowed, but the first character must be a number or lowercase letter. Spaces are *not* allowed in depot names. A *ProductFamily* directory level, if used, will always be a depot, and is thus subject to these limitations. A *Product* name is subject to these limitations if it is a top-level directory name (i.e., if it is a depot). If a *Product* directory appears under a *ProductFamily* directory, it may use mixed case (e.g. CamelCase).

If the *ProductFamily* directory level is used as a depot name, it may contain only *Product* directories.

Because top-level directories are a component of every pathname, they should be kept short (2-7 characters). Perforce can support arbitrarily large depot names, but long depot names result in longer pathnames for all files, which can eventually impair both usability (e.g. displays in GUIs and IDEs) and performance in large organizations.

## Under main

What appears under **main** will vary for each software product, and is beyond the scope of the PDS. Some examples are provided here to give a more clear sense of what the PDS looks like when implemented.

Example 1: Simple C Software Product:

 //fgs/main/src/server/…

 //fgs/main/src/api/…

 //fgs/main/src/ui/…

 //fgs/main/doc/…

In this and subsequent examples, underlined text indicates the parts of the path covered by this standard, while the remaining text is purely for illustration.

Example 2: Large application consisting of shared components that build in layers, with lower level components consumed API-style (e.g. header files and libraries in binary form, as opposed to being consumed in source code form):

 //fgs/main/build/…

 //fgs/main/common/…

 //fgs/main/doc/…

 //fgs/main/include/…

 //fgs/main/ComponentA/…

 //fgs/main/ComponentB/…

//fgs/main/ComponentC/…

When component-based or modular development is practiced, the inheritance of components defined under **main** can be controlled via the stream view when using streams.

Teams have complete ownership of what appears under **main.** Build systems may require certain files or directory structures relative to the root of the build tree, which typically appears just below **main**, or perhaps a directory level below that.

## Permanent Streams (Reserved Names)

### live

For hosted applications, **live** tracks the source files that generated the actual files deployed to the live production environment. In some cases where there is no build process, the source files are the actual files deployed to the live production runtime environment.

Example:

//fgs/live/www.acme.com

### main

The Mainline, or main stream of development.

Example:

//fgs/main/src

### latest

For Organic Development, the directory name that appears under the /dev container directory. The permanent **latest** tree may appear alongside temporary project branches.

Example:

//fgs/dev/latest/src

//fgs/dev/FGS34/src

//fgs/dev/64Bit/src

Equivalently when using streams:

//fgs/dev-latest/src

//fgs/dev-FGS34/src

//fgs/dev-64Bit/src

## Projects

### Project Branch Names

Project branches are identified with version identifiers and a suffix. Version identifiers typically consist of two or three digits, e.g. *Major*.*Minor* or *Major*.*Minor*.*Patch*. Components of a software product may also be used as part of the version identifier in the project branch name. For example, say a software product consists of a GUI, CLI, and Server components of the product FGS, each versioned separately. The project branches might be GUI-3.0, CLI-2.4, Server-2.0, as opposed to simply having a version identifier for the entire FGS product in the project branch name.

The suffixes are -P (Patch), -R (Release), -D (Development), and -I (Integration). The -R and -P suffixes are for projects under /rel container directories, in the maintenance phase of the life cycle. The -D suffix is used for development within a /rel container directory, as in the Advanced Maintenance branching pattern. The Advanced Maintenance pattern uses the -D branch to segregate work-in-progress for a planned maintenance from the -R branch, which contains only released software and possibly work-in-progress for an emergency, unplanned maintenance release. The -I suffix is for Integration branches under /int container directories. Development branches under /dev container directories do not use suffixes.

Branch names may vary among different software products. Project branch names are compliant with this PDS so long as they contain only alphanumeric characters and underscores, as well as ‘.’ and ‘-‘ characters used only as indicated here.

### Version Identifiers

Typically branch names include only 2 or 3 digits in version identifiers, *Major*.*Minor* or *Major*.*Minor*.*Patch*. Any given software product must define the number of digits that apply to each branch level. Branch names have one more digit in /rel \*-P streams than they in /rel \*-R streams, and one more digit in /rel \*-R streams than would apply to a point in time tagged on **main**. Version identifiers can tag **main** at certain points in time, but do not appear in the directory structure under /main since **main** has theoretically unlimited duration. For example, you might have rel/3.0.3-P at the patch branch level, branched from rel/3.0-R at the release branch level, tied to a 3.0 baseline tagged at a point in time on **main**.

 3 digits in /rel and 2 on **main**, or 4 in /rel and 3 on **main**.

Version numbering can be a complex subject for larger software products. So long as the number of digits is defined, the PDS is flexible on the number of digits used for each software product. Using some designation other than numeric values is also acceptable, e.g. naming releases after cities, rivers or mountains, with each starting with the next letter of the alphabet, Austin, Boston, Chicago, Denver, etc. Keep in mind that short tag names are best.

## Branch Container Directories

### svcs

The /svcs container directory contains customization branches, where customization work is performed by a professional services organization, distinct from the team that maintains the generic software product. The /main tree under the /svcs area is seeded by branching from a separate /main or /rel tree containing the generic (non-custom) software product.

Under /svcs, the next directory level has a tag identifying the specific customer or group of customers for whom the customization is done. Use short tags rather than company names in directory structures, to keep paths short, and reduce confusion when customers’ company names change.

Optionally, an extra organization directory level can be added if it is deemed of value to segregate source code changes on a per-work-order or per-SOW basis.

Examples:

//fgs/svcs/C44/main/src

//fgs/svcs/C44/main/test

//fgs/svcs/C44/CLIN25/main/src

//fgs/svcs/C44/CLIN25/main/test

Equivalently when using streams:

//fgs/svcs-C44-main/src

//fgs/svcs-C44-main/test

//fgs/svcs-C44-CLIN25-main/src

//fgs/svcs-C44-CLIN25-main/test

In the last two examples, BankOfBigtown (customer id C44) is the customer, and CLIN25 is a sample SOW identifier (CLIN = Contract Line Item Number, popular terminology in United States government contracts).

### custom

The /custom container directory contains customization branches, where customizations are developed by the same team that developed the generic software product.

Under /custom, the next directory level has a tag identifying the specific customer or group of customers for whom the customization is done. Use short tags rather than company names in directory structures, to keep paths short and reduce confusion when customers’ company names change.

Example:

//fgs/custom/C44/src

Equivalently when using streams:

//fgs/custom-C44/src

### rel

The /rel container directory contains projects that are in the maintenance phase of the life cycle. Projects are branched into the /rel tree shortly before they are ready to ship, at the point when the software is complete enough that it can transition from active development to relatively stable QA. Branches under the /rel container directory have ‘-R’, ‘-P’, or ‘-D’ suffixes, depending on the branching patterns used.

Examples:

//fgs/rel/3.0-R/src

//fgs/rel/3.0-D/src

//fgs/rel/GUI-4.0-R/src

//fgs/rel/GUI/4.0-R/src

Equivalently when using streams:

//fgs/rel-3.0-R/src

//fgs/rel-3.0-D/src

//fgs/rel-GUI-4.0-R/src

//fgs/rel-GUI-4.0-R/src

Which of these examples is most appropriate depends on the number of products being managed in the structure. The first two examples, with just a numeric identifier, make sense if fgs is a relatively small software product, always built and delivered as one cohesive releasable item. The third example would apply if, for example, fgs consisted of several components and the 4.0-R release contained only GUI components. The fourth example would apply if there were many independently versioned and released components within fgs. This variation can apply in ‘int’ and ‘dev’ areas as well, as may be needed to manage many products, whether they are released independently and/or combined to form larger products.

### int

The /int container directory contains integration branches, which appear in the Advanced Planned Development branching pattern. Integration branches always have a’-I’ suffix.

Examples:

//fgs/int/4.x-I/src

//fgs/int/CLI-2.x-I/src

Equivalently when using streams:

//fgs/int-4.x-I/src

//fgs/int-CLI-2.x-I/src

In these examples, the “.x” designation indicates that the integration branch is the source for a series of /dev branches. Use of “.x” is not dictated by this PDS; it is just an example.

To avoid confusion, the name of the /int branch must be selected carefully, and must make sense for all /dev branches to be created from it.

Consider this example with the FGS Product. We decide to build a new version of the project, FGS-3.3, which is expected to be complex, and to address a wide variety of features. In that case, a single /int branch might be created for the entire FGS-3.3 effort, named 3.3-I. From that, there might be just a single /dev/3.3 development branch. Or, there could be a series of development streams that start initially isolated, e.g. /dev/3.3-UI, (user interface enhancements), /dev/3.3-Perf (performance improvements), and /dev/3.3-Unicode (adding Unicode support).

In another scenario, you might have a series of less complex releases, 3.4, 3.5, and 3.6, all intended to release in quick succession. In that scenario, you might have an /int branch named /int/3.x-I, with /dev/3.4, /dev/3.5, and /dev/3.6 all created from it at the same time.

Having a naming convention for development branches that makes it clear which integration branch they were initially branched from is *optional*. It is acceptable if the development branch name bears no clear connection to its parent integration branch. The reasons for not requiring this connection are:

* Requiring a prefix for each development branch lengthens pathnames. In general, shorter pathnames are better.
* Sometimes a development branch may be retargeted for a different integration branch. This becomes confusing if the development branch names tie them to their original parent integration branch.

### dev

The /dev container directory contains development project branches. Each project branch is associated with a set of features, requirements, and/or bugs to fix. The lifespan of the project branch is related to the time it takes to implement the assigned features. The /dev container directory may also contain a permanent /latest organic development stream.

Examples:

//fgs/dev/64Bit/src

//fgs/dev/FGS34/src

//fgs/dev/FGS218/src

//fgs/dev/3.0/src

//fgs/dev/latest/src

Equivalently when using streams:

//fgs/dev-64Bit/src

//fgs/dev-FGS34/src

//fgs/dev-FGS218/src

//fgs/dev-3.0/src

//fgs/dev-latest/src

When naming development project branches, the name in the directory structure should be a short, unique tag name that refers to the set of features being developed in that branch. Feature set tags must be unique within a given /main tree, but may be reused across products. That would be the case if both the fgs and gizmo product lines, each with its own /main tree, each had a feature set named 64Bit. 64Bit is the short tag used in the directory structure. The fully qualified name of the feature sets (not referenced again in this document) are made unique by prefixing the product name, e.g. fgs-64Bit and gizmo-64Bit.

Various criteria are considered when determining what features are grouped together. Common factors for grouping features include:

* Features with common business drivers, e.g. requested by the same customer, may be grouped in the same feature set.
* Features with different priorities, e.g. “contractually required” vs. “optional”, may be grouped into separate feature sets.
* Features that will heavily overlap code areas, based on technical analysis of requirements by those familiar with system architecture. Perforce supports concurrent development of the same files, but it is still wise to avoid unnecessary or excessive concurrent development efforts from affecting many of the same files.

Ideally, project branch names should not imply release order. The “3.0” example above seems appropriate at first, since “3.0” can clearly refer to a certain group of features. However, using version IDs as development project branch names can be problematic, as version IDs imply a release order for feature sets. If the set of features initially developed under the moniker “3.0” is later decided to be released after something tagged “3.1”, this can cause confusion. The examples above illustrate using “FGS34” and “FGS218”, where the numeric portion (34 or 218) doesn’t have any implied connection with the order it will be released. The tag is just a reference to a set of features.

Renaming “3.0” to “3.2” after 3.0 has been used could be even more confusing, and is *not* recommended. Renaming an entire branch is not recommended because there are several unwanted side effects:

* The history trail is more difficult to follow for humans.
* The integration history leads to confusion when propagating changes into renamed branches. Using branch specs can alleviate this to some extent, but can not eliminate the problem.
* Development activity must stop and work-in-progress must be submitted prior to renaming.
* Developers will be impacted with workspace changes.
* Managing the flow of changes becomes more complicated, especially if there are other branches downstream from the one that changed.

The life cycle for a feature set starts when the development project branch is created for it, and ends when that feature set does its final promotion. Another key milestone in the life cycle is the first promotion from the project branch to its parent, e.g. /main or an /int/3.x-I stream. Project branches retain flexibility of release order until they make the first promotion to their parent stream. Before the first promotion, the feature set can easily be postponed to future releases, shelved, or even cancelled. After the first promotion, holding back a feature set from a release becomes more complex. At that point, holding it back requires the same sort of shuffling, such as rollbacks of selected files, that limits the organic model. **The decision to make that first promotion from a project branch, and thereby commit to a particular release order, should not occur too early in the life cycle of the feature set.**

Using version IDs (like “3.0”) under the /rel tree makes more sense. Version IDs don’t generally change once a product has been released, and can be considered absolute. The short tag names for development project branch names are restricted to /int and /dev areas. Feature sets or groups of feature sets combine at the point of promotion to /main, and they are assigned version IDs as they transition to /rel.

### exp

The /exp container directory contains experimental/prototype streams, branched from **main** or from /dev streams. Code from these streams may later be promoted to become regular /dev streams, or may simply be left alone.

In practical terms, there is little difference between /exp and /dev. The primary difference is that changes made in /dev are made in the hope and expectation that they will become part of the software product, whereas no such expectation exists in /exp branches. /exp branches might be used, for example, to determine the level of difficulty in attempting a port to a different platform, without formally committing to deliver the port as a product feature.

Examples:

//fgs/exp/GenX/src

//fgs/exp/Port-MacOSX/src

Equivalently when using streams:

//fgs/exp-GenX/src

//fgs/exp-Port-MacOSX/src

### demo

The /demo container directory contains any special throw-away modifications used to enable demonstrations. Demo projects may be used for particular demonstration events, or they may be maintained as long-lived branches from which many demonstrations are given. Where /exp branches which are treated similarly to /dev branches, /demo branches are treated more like /rel branches, because they have a greater need for stability.

Examples:

//fgs/demo/Comdex2012/src

//fgs/demo/latest/src

Equivalently when using streams:

//fgs/demo-Comdex2012/src

//fgs/demo-latest/src

The latter example illustrates a long-lived demo/latest branch. In order to demonstrate new software, this branch may be periodically refreshed from /main.

### sb

All /sb container directories contain individual developer sandboxes. The directory level below the container is the Perforce account of the user.

Use of sandboxes is optional. If sandbox branches are used, the developer is responsible for keeping the branch up to date, and for managing merges into, and promotions from, their branch. Developers may choose to make changes that are smaller, more straightforward, or easier in the regular dev branch. The sb branch is typically reserved for individual experimental or complex changes that could destabilize the dev branch.

Sandbox branches may optionally be deferred to the P4Sandbox tool, which maintains private local branches independently of the main Perforce server.

Note that the shelving feature in Perforce reduces the need for sandbox branches.

Example:

//fgs/sb/jdoe/FGS34/src

Equivalently when using streams:

//fgs/sb-jdoe-FGS34/src

## Sample Structures

Below are some sample structures illustrating valid combinations of branching patterns. The examples are for non-streams usage; the equivalent streams naming patterns are transparently derived by flattening the directory structure to two levels.

### Basic Planned Dev + Advanced Maintenance

This sample illustrates the FGS product:

//fgs/rel/1.0-R/src

//fgs/rel/1.0-D/src

//fgs/main/src

//fgs/dev/1.5/src

//fgs/dev/2.0/src

### Advanced Planned Dev + Advanced Maintenance

This example uses a single version identifier for the FGS product line.

//fgs/rel/1.0-R/src

//fgs/rel/1.0-D/src

//fgs/main/src

//fgs/int/2.x-I/src

//fgs/dev/2.5/src

//fgs/dev/2.6/src

This next example uses one version identifier for the FGS product line, and different values for components. This applies when components are developed separately, and the main product is released as a set of independently versioned components. In the examples that follow, FGS-1.0-R is used to support the entire FGS product, while CLI-4.0-R supports only the CLI component.

//fgs/rel/fgs-1.0-R/src

//fgs/rel/fgs-1.0-D/src

//fgs/rel/CLI-4.0-R/src

//fgs/rel/CLI-4.0-D/src

//fgs/rel/GUI-2.0-R/src

//fgs/rel/GUI-2.0-D/src

//fgs/main/src

//fgs/int/2.x-I/src

//fgs/dev/2.5/src

//fgs/dev/2.6/src

### Advanced Planned Dev + Basic Maintenance + Customization

In this sample, a professional services team owns the /svcs area, and uses it to make customizations for a particular customer. Here C2045 is a reference identifying the customer.

//fgs/svcs/C2045/main/src

//fgs/svcs/C2045/latest/src

//fgs/rel/fgs-1.0-R/src

//fgs/rel/CLI-4.0-R/src

//fgs/rel/GUI-2.0-R/src

//fgs/main/src

//fgs/int/2.x-I/src

//fgs/dev/2.5/src

//fgs/dev/2.6/src

### Advanced Planned Dev + Basic Maintenance + Sandboxes

In this sample, per-developer sandbox branches are used.

//fgs/rel/3.0-R/src

//fgs/main/src

//fgs/int/4.x-I/src

//fgs/dev/FGS34/src

//fgs/dev/FGS218/src

//fgs/sb/jdoe/FGS34/src

//fgs/sb/jdoe/FGS218/src

//fgs/sb/bruno/FGS34/src

 //fgs/sb/bruno/FGS218/src

## Import Structures

All Baseline & Branch Import (BBI) migrations are imported directly into the structures defined in this PDS.

For Detailed History Imports (DHI) from legacy source control systems, files will first be imported into a special import structure within the same depot, defined as follows:

 //*TargetDir*/**i\_***OriginalCMSystemTag*/

Sample *OriginalCMSytemTag* values are cvs, svn, cc (ClearCase), vss, syn, and mks. The *TargetDir* is the depot name for the import. This may be //*ProductFamily* or //*Product*, or //*ProductFamily*/*Product*, as appropriate. Examples:

//fgs/i\_cvs

//fgs/i\_svn

//fgs/i\_cc

For example, when migrating from ClearCase, the contents of the schip VOB might be migrated into:

//schip/i\_cc

The **i\_**\* import areas are read-only to humans, populated only by mechanical import processes. Once the imports are complete, these areas are made read-only for human users. Active branches are integrated into writable folders within the PDS structure.

Importing into stream depots may make sense if significant branch and component history can be imported.

# Version Control “Areas”

A *version control area* consists of one or more depots that are used for a similar purpose. The Perforce system is unaware of version control areas. They provide a way of classifying depots to simplify policy management and promote a clear organizational structure.

## Source Area

Depots that contain primarily source code (owned by the enterprise) are collectively referred to as the Source Area. The PDS applies in the Source Area only.

The Source Area contains files in a structure oriented toward a development view of the world. For example, the structure may separate files by system component, by technologies used, or may separate reusable vs. project-specific code. All artifacts used to create software products, including source code for compiled software, graphic files, scripts, and the like are versioned in the Source Area.

## Build Area

Use of Build Area depots is not required. This section is for informational purposes only; compliance is not required as of this version of this standard.

The Build Area consists of a series of optional //\*-bld depots that correspond to Source Area depots. For example, //fgs in the Source Area could have a corresponding //fgs-bld depot in the Build Area.

The Build Area is used to store the output of build processes, such as libraries, executables, and APIs, in Perforce. The Build Area is populated by fully automated build processes, and is read-only to human users.

Continuous integration build tools can be “pointed” to any stream. They are usually first used to automate builds in /dev streams. That’s where continuous integration adds the most immediate value for developers, giving fast feedback about bogus checkins. However, they are also valuable when pointed to streams that have more controlled submit processes, like /int, /main, or /rel. In those cases, you expect the continuous integration to always produce a good build, though they occur far less frequently than in /dev. The build processes, whether initiated manually, scheduled, or run by a continuous integration server, produce results in the Build Area.

The branch container directories in the Source Area directory structure are also used in the Build Area, to help make clear where the build artifacts (e.g. binaries) came from.

The general structure of the Build Area is:

//*SourceAreaDepot*-bld /<*Container*>/<*Component*>/<*Version*>

The use of the [‘+S’ file type modifier](http://www.perforce.com/perforce/doc.current/manuals/cmdref/o.ftypes.html#1040647) is strongly recommended in Build Areas to limit storage consumption in the Build Area.

Use of stream depots for build artifacts may have benefit if the stream view can be put to use, or if build artifacts are promoted upon completion of test procedures. The use of stream depots would require a flattened directory structure.

## Release Area

Use of Release Area depots is not required. This section is for informational purposes only; compliance is not required as of this version of this standard.

The Release Area consists of a series of optional //\*-rel depots that correspond to Source and Build Area depots. For example, //fgs in the Source Area could have a corresponding //fgs-bld depot in the Build Area and //fgs-rel depot in the Release Area.

The Release Area contains files in a structure oriented toward an operational view of the world. The Release Area contains compiled code in whatever form software is delivered in, such as WAR files, EXEs, DLLs, \*.so's, EAR/Zip files, etc. The Release Area read-only for human users, and is populated by automated build processes. The Release Area contains executables branched from the Build Area. It may also contain unique files that vary on a per-runtime-environment or per-deployed-configuration basis, such as database connection string files, files with configurable application settings, license files, and the like.

For web applications, the release area has the exact set of files deployed to runtime environments, or at least all the files that come from the version control system. For example, the release area for www.FriendlyGreetingSystem.com looks like:

//fgs-rel/live/www.FriendlyGreetingSystem.com

Use of stream depots for released artifacts may have benefit if the stream view can be put to use, if source code is included, or if released artifacts are promoted upon completion of test procedures. The use of stream depots would require a flattened directory structure.

## Third Party Area

The third party depot contains files from external vendors or partners. The general structure of the Third Party Area is:

//3rdparty/[*VendorProductFamily*/]*Product*/*VersionID*/src/...

//3rdparty/[*VendorProductFamily*/]*Product*/*VersionID*/[bin.*platform*]/...

The *VendorProductFamily* directory level is optional. The src directory is used only if source code is provided. The bin.*platform* directory contains binaries for the particular platform. The *.platform* suffix is omitted for platform-independent distributions such as Java.

Because third party code is rarely branched in these depots, and the third party vendor’s branching model is not of interest, the use of stream depots would not yield significant benefits.

### Binary and Source Distributions

The binaries are downloaded and submitted to a structure like:

//3rdparty/Jakarta/Tomcat/5.5.9/bin/...

If the products are obtained in source form and compiled locally, the source code also goes into this structure:

//3rdparty/Jakarta/Tomcat/5.5.9/src/...

Sometimes packages distributed in source form require simple tweaks to makefiles to get them to build, such as commenting or uncommenting lines appropriate to a build on a particular platform. Such changes are made in this area. Any version of a makefile used to compile a build should be submitted to Perforce along with the build results.

### Modifications to Third Party Software

For cases where the "as obtained from the vendor" configuration must be semantically changed, the rule is that //3rdparty is for "as obtained" software only, with no local modifications (other than commenting/uncommenting platform-specific lines in makefiles, or similar trivial tweaks required to build). This structure should contain only as-obtained files (\*.tar, \*.tgz, \*.zip, \*.msi, \*.exe, etc.), and unpacked tar/zip files, to simplify diffing in Perforce.

When modifications are necessary, the unpacked source files are branched to the appropriate Source Area depot to make clear the intent to semantically modify the code. The exact target structure is not defined here, since it will vary depending on factors like:

* How the branched 3rd party software is integrated into the build process, or
* How the branched 3rd party is to be shared among product line.

## Documentation Area

Due to efficient handling of binary files, Perforce is commonly used as a document management system, providing versioning, access control, and online publishing of documents.

Documents are generally classified into one of 2 types, regular and version-specific. Version-specific documents belong in the Source Area where they are versioned and branched alongside the software products they are related to. Examples of version-specific documents include:

* design documents
* functional specifications
* test plans
* “to do” checklist spreadsheets

Version-specific documents are typically stored in a ‘doc’ structure, parallel with the ‘src’ tree under /main. These are branched in the same structure as software and source code, though in some cases released on a different schedule.

Regular documents are not version-specific, and can include all manner of documents, not necessarily those related to software products. For such documents, a single “docs” depot can be established, something like:

//docs/*BusinessUnit*/

The directory structure that appears below Docs is outside the scope of this PDS. Such a structure should be established with care, planning for potential future growth so it can eventually be used by many business units within the enterprise.

In cases where there is a need to segregate “work in progress” documents from “Plan of Record” documents, a Perforce simplified branching structure can be used to promote documents, e.g.:

//docs/*BusinessUnit*/live

//docs/*BusinessUnit*/main

Or, for contract documents,

//docs/*BusinessUnit*/legal/draft

//docs/*BusinessUnit*/legal/final

Typically the /live tree would have relatively open access to published documents, while access to the /main tree might be restricted to document owners and publishers. This model works well for scenarios where there are clearly defined owners of documents.

In the second example, file histories in the draft folder would reflect the contract negotiation process, while PDFs of final, signed and counter-signed documents are stored in the final folder.

The use of stream depots may prove beneficial for documentation, as it facilitates an easy promotion model and also provides visual tools to the users. If stream depots are used, the directory structure would be flattened, e.g.:

//*BusinessUnitDocs*/live

//*BusinessUnitDocs*/main

Alternatively, documentation could be kept as a component in the same streams used for source code.

## User Area

A User Area may be established on a Perforce server to store work artifacts related to individual people, not related to software products. This can include things such as personal shell environment files, personal utility scripts, and a personal technology playground area to facilitate incubation of early-stage concepts not associated with formally planned releases.

Because of the ad-hoc nature of the files kept in these depots, the use of stream depots is not usually recommended.

The User Area typically consists of a single //user depot, with folders for individual users:

//user/john\_smith

# Perforce Depot Names

## Standard Depots

The initial set of depots will include the following:

* //spec – Standard name for a Perforce ‘[spec](http://www.perforce.com/perforce/doc.current/manuals/p4sag/03_superuser.html#1083233)’ depot. (See: <http://kb.perforce.com/AdminTasks/UsingTheSpecDepot>).
* //Perforce – Standard name for the depot containing the Perforce admin software and triggers, including those deployed by Perforce Consulting Services, such as the Perforce Server Deployment Package (SDP).

## Optional Depots

The initial set of depots might include the following:

* //3rdparty – Contains third party 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. See section 7.4, Third Party Area.
* //fgs – Source Code for the FGS product.
* //fgs-bld – 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 OSX/Linux/Solaris.
* //fgs-rel – Contains as-released software, suitable for distribution to runtime environments, burning to CDs or firmware, or otherwise delivered. This includes files branched from //fgs-bld, plus various configuration files, such as database connection strings or XML files defining app server settings.

Each product or product family might have a set of depots, one for source code, one for builds, and one for as-released files, e.g. //fgs, //fgs-bld, and //fgs-rel.

## How Many Branch Levels?

A key challenge in defining a branching strategy is to determine the right number of levels to have. Should the Branch Diagram that represents your branching strategy have **main** and release branches only, allowing work to occur in the Mainline? Should all work be relegated to planned Feature sets? Should you have separate Prototype, Development, Integration, **main**, and Release levels?

The goal is to ensure that your branching strategy is as sophisticated as your real-world business environment demands – and no more. You want “just enough process” to encourage optimal productivity. Too little process, and you find yourself wasting resources working around the limitations of an unsophisticated process. Schedules are unpredictable, and software quality degrades to unacceptable levels. Too much process, and release schedules become unnecessarily delayed, as the process institutionalizes inefficiency.

Each level on your branch diagram implies more process and a longer release cycle. Longer cycles are appropriate for products that need higher quality and fewer defects, and are necessary when integrating the efforts of a large number of contributors. Shorter cycles are important in fiercely competitive environments where timing releases according to market demand is balanced against the need to release defect-free products.

The number of levels on your Branch Diagram is an indication of the “sophistication vs. complexity” tradeoff of your release process.

Even for a team of one person on a small product, a minimum of two branch levels is a good idea. That allows you to separate latest & greatest (and quite possibly broken) code from stable code. Only in large and complex efforts, with hundreds or thousands of contributors and release cycles longer than a year, are more than about 4 branch levels needed, /rel, /main, /int, and /dev. This excludes personal development or experimental/prototype branches, as they are typically outside formal release processes.

1. Sometimes the term ‘depot’ is used to refer to a single instance of a Perforce server, and other times the term ‘depot’ refers to a top-level directory in Perforce. For purposes of this document, the latter applies. [↑](#footnote-ref-1)