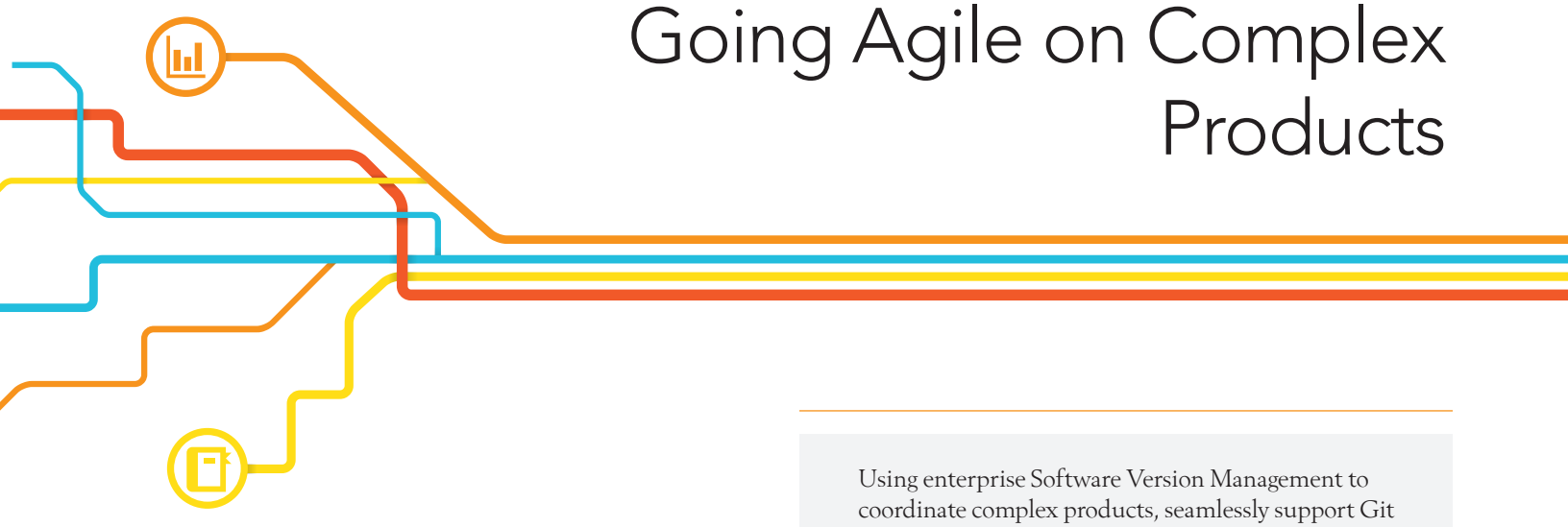
PERFORCE
Version everything.

Code Reuse and Refactoring: Going Agile on Complex Products



Using enterprise Software Version Management to coordinate complex products, seamlessly support Git developers, and solve refactoring problems.

Table of Contents

Modern Agile workflows adopted by the most dynamic software teams often require rapid task branching. Rapid delivery of small incremental changes overcomes the scalability and planning bottlenecks associated with traditional Agile workflows, giving you the flexibility to deliver incremental improvements as often as several times a week.

Supporting this workflow requires rapid and accurate creation of new private working areas. When a private Git repository is used as the local workspace, Perforce Git Fusion seamlessly supports release managers, product architects, and developers. Release managers and architects can design the product and its release cycle in Perforce, taking account of all component relationships, and refactor when necessary. The product architect's knowledge of how the next generation of the product is assembled is embedded in the enterprise repository. This information is made available on demand to developers working on backlog tasks in Git repositories.

COMPONENT-BASED DEVELOPMENT

Component-based development (CBD) adopts production line techniques for software development, emphasizing sharing and reuse of work whenever possible. A complex product may be composed of tens or hundreds of component building blocks. Each component may be developed by a specialized team, working on an independent iteration cycle. A product architect will define how the complete product is assembled from specific versions of the necessary components. The CBD process provides some insulation from the vagaries of scheduling such a complex effort, just as a car may be shipped with the latest stable version of a GPS navigation system in order to maintain the overall delivery schedule. CBD also encourages sharing of work whenever possible, so that effort is not duplicated.

However, just as changing manufacturing specifications require close coordination of the production line, the components of a complex software product must be compatible at some interface level. Additionally, in many industries a finished product can be configured in specific ways for particular customers. In the cellular communication industry, a chip set may be tailored for different networks and carriers, resulting in a complex feature matrix. Collaboration (hopefully automated) between the producers and consumers of components is essential if the model is to work at a large scale.

CBD PRODUCT MODEL

Consider a scenario with a product with three components developed internally and one imported library (Figure 1).

For the next generation of the product, the architects wish to use a new version of log4net and combine the CORBA and COM modules into a new REST module (Figure 2). As in many such cases, the public interfaces will change significantly but the business logic stays relatively unchanged.

This refactoring operation has many downstream impacts on development, build and release, and other teams.

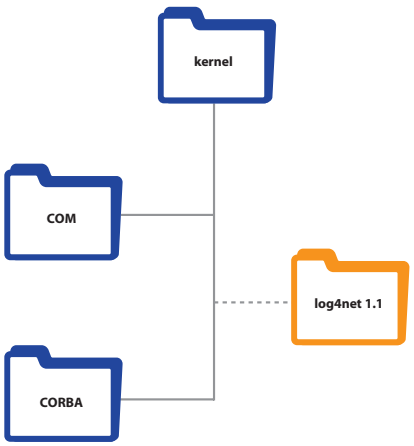


Figure 1: Simple Component Model

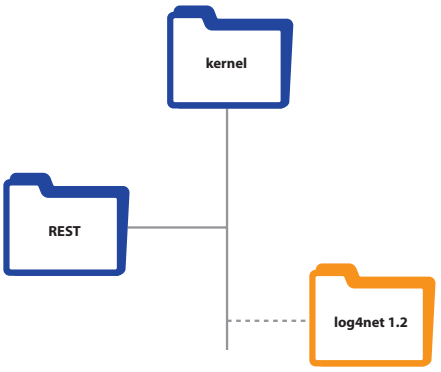


Figure 2: Refactored Components

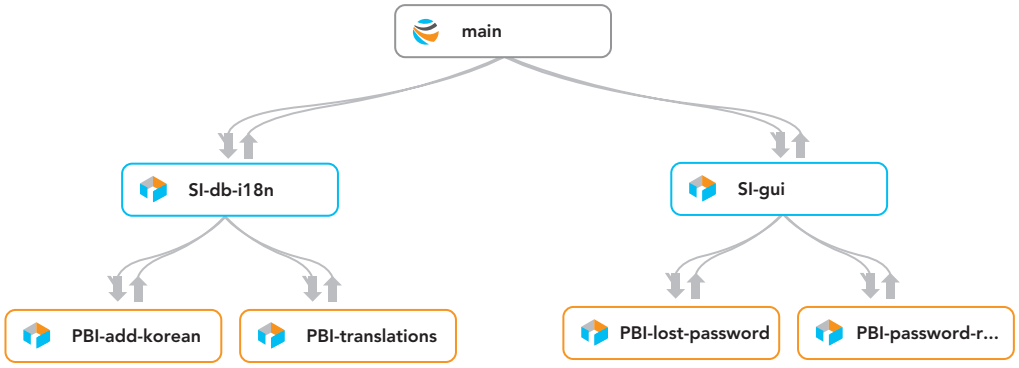


Figure 3: Task-based agile model in Perforce Streams

THE REPOSITORY MODEL

The task-based scalable Agile model is shown in Figure 3. In this figure each task is developed in a private stream, branch, or repository, and promoted upstream when ready for continuous integration and deployment. In the most advanced cases, the task branches are realized as distributed version management repositories. Distributed technology gives all developers the freedom of always-on versioning and private branching.

A necessary prerequisite of employing distributed repositories is seamlessly providing all developers the right view of data to work on: the right version of the right files, with the right access, to tackle the task at hand. This problem becomes increasingly difficult to manage as the scope of the project becomes larger. In component-based development, the relationships between components and customer-driven customizations spawns a large matrix of possible variations of a product.

One manifestation of this problem is the difficulty of successfully refactoring a complex project without disrupting active development. Refactoring is often necessary to improve legacy software or adopt a new generation of technologies.[1] But refactoring can become a development and release management headache, and should only be undertaken to improve the product, not respond to limitations of development tools.

MANAGING CBD IN GIT

Without an enterprise repository to guide the project, substantial overhead is involved in the CBD model. A team working on the kernel module will require the latest version of the kernel, plus stable versions of the CORBA, COM, and log4net modules as determined by the product architect.

1 http://www.martinfowler.com/bliki/TechnicalDebt.html

Setting Up the CBD Model

In Git, each component is managed in its own repository and subtrees are used to bring these components into a kernel developer's working repository (Figure 4). Alternatively, the entire project could be kept in a single repository. That would lead to a potentially large and difficult to clone repository. More importantly, because Git branches operate at the repository level, it would be almost impossible to mix and match versions of components in a single Git repository.

Setting up the 3 components takes 21 commands, or 7 per component (Table 1).

The architect can control when new stable versions are pushed, and whether bug fixes can be submitted back to the component repositories, assuming access control is set up on the component repositories. Note, however, that since Git does not provide directory-level access control a developer could easily submit changes to the imported subtrees, which may cause a discrepancy between the version of a component used by the team and the version in the official component repository.

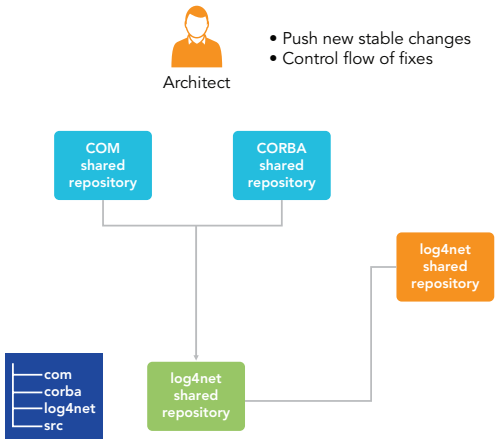


Figure 4: Components Managed as Subtrees

Table 1: Component Setup

Command	Purpose
Git remote add	Add remote reference to component repository
Git fetch	Pull remote component into local repository
Git checkout –b	Checkout component into new branch
Git checkout master	Switch back to local
Git read-tree	Import component into local branch as subdirectory
Git add	Add new subdirectory
Git commit	Commit change

Incorporating Non-Software Components

Many digital products include non-software components, such as art and animation assets, documentation, and CAD or EDA design files. Git is not well suited to versioning these types of assets, so a third-party asset manager must be used and incorporated into the development and release process.

Updating Component Baselines

Assuming that a new stable version of a component has been released, a lead kernel developer would need to run 5 commands to get the new version into her working repository (see Table 2).

Once she shares that work, the rest of the team receives the changes when they *pull* from the shared repository. As you can see, there are several steps required to make sure that component changes flow to the rest of the team in the proper fashion.

Table 2: Component Update

Command	Purpose
Git checkout	Checkout component branch
Git pull	Pull latest component baseline
Git checkout master	Switch back to local branch
Git merge –s subtree	Run subtree merge to get component changes into working branch
Git commit	Commit updated component

Submitting Patches to Components

If a developer submitted a patch to the imported com component, the architect could use the *cherry-pick* command to pull that patch into the component repository. This solution is simple and works well when patches are infrequent. Frequent patches would require advanced use of the subtree merge strategy.

Developer View

Note that using subtrees in this way greatly restricts the developer's view of the product. A developer who was not aware of the origins of the COM module would see it as simply a subdirectory of the product. In order to examine the full history of the COM module, she would need to clone and browse the COM repository.

Refactoring: In Practice

The architect would need to perform several steps to combine the COM and CORBA modules. One technique is to import the CORBA module as a subtree of the COM module.

After combining the two components and performing any other necessary work on the code, the combined repository can be treated as the new REST module, or it can be cloned into a new repository altogether.

Refactoring to split a repository is sometimes done to accommodate Git's repository size limitations. In this case, the architect would need to clone a copy of the repository and use the filter-branch command to remove unwanted portions of the original and new repositories. Other cleanup commands may be necessary to remove unwanted tags and branches.

Refactoring: Developer Impact

After the refactoring, several steps are necessary to produce new working repositories for developers (Table 3). A lead developer would need to remove the CORBA module to start with. Then, if the old COM module was serving as the new REST module, she would repeat the steps in Table 2 to pull in the refactored components. If a new REST repository was defined, she would delete the old COM module and then repeat the steps in Table 1 to pull in the new REST component.

Then, each developer would pull the updated changes.

Table 3: Refactoring

Command	Purpose
Git remote add –f	Add CORBA remote in COM repository and fetch commits
Git merge –s ours	Prepare a subtree merge
Git read-tree	Add CORBA into working branch
Git commit	Commit changes

MANAGING CBD IN PERFORCE SOFTWARE VERSION MANAGEMENT AND PERFORCE GIT FUSION

Assume now that the entire product is versioned in Perforce Software Version Management. Each component is managed in a Perforce branching structure, while the import is kept in its own area (Figure 5).

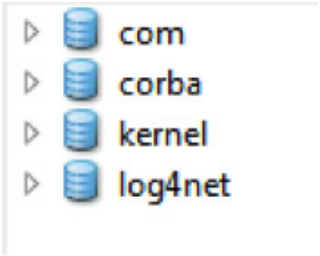


Figure 5: Perforce Depot Structure

Setting Up the CBD Model

The kernel team uses Git for daily work, and a Perforce view defined in one operation by the architect determines their working data. It is simple to include the latest kernel code (main branch), plus stable (rel or released) versions of the other components.

```
View:
//kernel/main/...          //git_dev/kernel/...
//com/rel/...              //git_dev/com/...
//corba/rel/...            //git_dev/corba/...
//log4net/rel1.1/...       //git_dev/log4net/...
```

In this simple example the view *git_dev* is exposed as a git repository for developers to clone from. A developer can issue a single *git clone* command and obtain the proper set of working files. Access control in Perforce can easily determine whether patches are accepted into the component areas.

Incorporating Non-Software Components

Perforce is able to version any type of digital asset, meaning that a single development and release process can be used for the entire product (Figure 6). A non-software component can easily be included in the working view presented to Git repositories, while the content creators can use interfaces more relevant for their line of work, such as Perforce's Graphical Tools plugin or Perforce Commons.

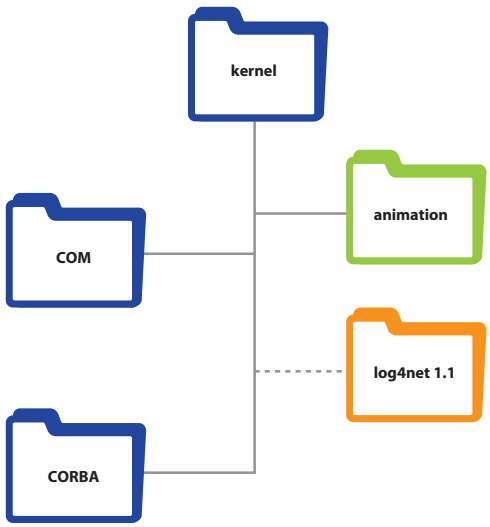


Figure 6: Animation Component in Product

If the digital assets are too large to fit into a Git repository, some subset can be exposed.

Updating Component Baselines

As new versions of components are released, the developer receives them automatically when they pull from the shared repository. No additional steps are required.

Submitting Patches to Components

A developer wishing to submit a patch to the COM component would simply commit as usual. It then falls to the component maintainers to merge that patch to other versions of the component, and of course a code review system could be used to accept or reject the patch in the first place.

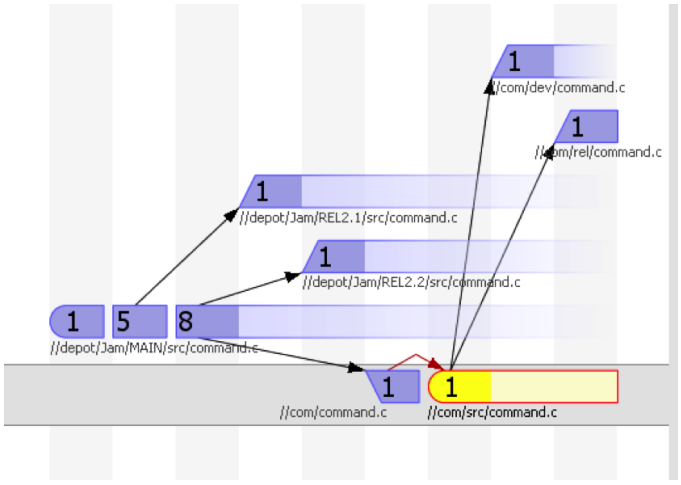


Figure 7: Revision Graph

Developer View

Developers who look at the project view in Perforce can quickly see that each component is developed in a separate depot (Figure 5), and could use tools like the Revision Graph (Figure 7) to examine file history without browsing to a separate repository.

Refactoring: In Practice

The refactoring is done using standard Perforce rename or branching commands. Since all components are versioned in the enterprise repository, the version management aspects of refactoring are straightforward. After the refactoring, the updated Perforce repository reflects the new component structure (Figure 8).

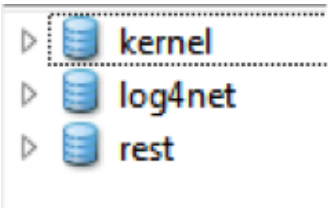


Figure 8: Post-Refactoring Depot Structure

The product architect simply reconfigures Git Fusion.

```
View:
//kernel/main/...          //git_dev/kernel/...
//rest/rel/...             //git_dev/rest/...
//log4net/rel1.1/...       //git_dev/log4net/...
```

The view is simply updated and a new git repository is produced, available for immediate cloning.

Likewise, a refactoring involving splitting part of a component into a new area is handled easily by changing the views used by Perforce Git Fusion. Since Perforce has no effective limitation on the size of a component, this form of refactoring is only undertaken when it improves the product. If a component grows too large for a single Git repository, a smaller portion of it can be exposed through Git Fusion.

Refactoring: Developer Impact

Each developer would run a single *git clone* command to obtain a new working copy with the updated product structure.

CONCLUSION

Component-based development is a common and necessary software development technique. Rapid development of a CBD product is often accomplished with a distributed Agile workflow and distributed repositories. Supporting CBD effectively for architects, release managers, and developers is greatly simplified with Perforce Git Fusion, which provides:

- Simple sharing of reusable software components, both when a project is created and during ongoing maintenance.
- Strong access control to prevent mistaken updates of shared components.
- Quick response to dynamic product architecture.
- Total view of the project.

Simple and Dynamic Sharing

Setting up or restructuring a CBD product is accomplished with simple Perforce view changes, rather than running a linearly scaling sequence of commands. Future updates to imported components are available automatically without any complicated subtree merge steps, while controlling patches to the imported components is substantially easier.

Ensure that Updates are Coming from the Right Team Members

Granular access control in Perforce prevents divergence between the working copy of a component and the officially maintained copy.

Refactor at Will

Refactoring a complex product is often necessary but always bears a cost. The product architects must handle the planning and implementation of refactoring as it affects the source code and component relationships. Again, Git Fusion reduces the architect's burden from a per-component cost to a fixed cost, while giving developers Git repositories that are simpler and faster to use.

The combination of Perforce Software Version Management and Perforce Git Fusion also reduces the frequency of refactoring, as no architectural changes are necessary to accommodate repository restrictions. By modeling the product architecture in Perforce and making the correct view of the data accessible on demand, Perforce and Git Fusion allow development teams to resume rapid development cycles after the refactoring is complete.

Total View

By providing a total view of the project, Perforce and Git Fusion allow release managers and architects to design the product to best fit the business and technology objectives of the team. The right view of the data is presented to developers seamlessly.

perforce.com

North America
Perforce Software Inc.
2320 Blanding Ave
Alameda, CA 94501
USA
Phone: +1 510.864.7400
info@perforce.com

Europe
Perforce Software UK Ltd.
West Forest Gate
Wellington Road
Wokingham
Berkshire RG40 2AT
UK
Phone: +44 (0) 845 345 0116
uk@perforce.com

Australia
Perforce Software Pty. Ltd.
Suite 3, Level 10
221 Miller Street
North Sydney
NSW 2060
AUSTRALIA
Phone: +61 (0)2 8912-4600
au@perforce.com