Component Based Development in Perforce

Randy DeFauw
Senior Product Manager

Sven Erik Knop
Lead Consultant
What is Component Based Development?

CBD Defined
What are Components?

- Well defined interface and hidden internals
- High cohesion and low coupling
- Independent test and release schedules
- Often developed by separate teams or third-party

Examples:
- Drivers, Services, Car parts, Phone modules, …
What is Component Based Development?

- Products build out of components
  - Assembly and configuration
- Separation of concerns
- Specialization of skills
- Agile workflow and high flexibility
Why use CBD?

- Break complex problems into smaller units
- Reassemble and reconfigure for new products
- Reduced cost through
  - Reuse
  - Simplified maintenance
- Minimal impact of change
Example of a CBD project

- **Components**
  - Can be swapped for testing
  - Might need to be configured
  - Can be assembled locally during development
  - Can be upgraded independently
Challenges

- Projects need to be prepared for components
- Dependencies and interfaces
- Requires special tooling
  - Creation of configurations for development and test
  - Reproducibility of delivered configurations
LARGE projects can contain 1000s of components
- Often grouped into super-components

Examples
- Car Infotainment Systems
- Mobile Phones
- Financial services application
Pieces of the Solution

Building blocks
A framework, not a complete building

Scalable repository

Flexible data model

Robust extension points
Scalable repository

- Hold all the components
- Simplified management
- Easy updates as the projects and components change
Flexible data model

- The way that users work with data shouldn’t have to align with how you store the data
- Respond quickly to changing code base or requirements
Extension points

- APIs for tooling
  - Building projects
  - Interacting with other processes
- Process control
  - Triggers, broker
  - Enforce policies
- User tools
  - Simplify workflow
  - Available in command line or GUI
Patterns

Putting the pieces together
Workspace generation

- Assemble the right view for a project
  - The right dependencies
  - The right versions
- Update the view if the project changes
Access control

- Beyond the protections table
- Per-role, per-project permissions
- Read in one project, write in another
Merge management

- Define pathways for merges
  - Based on role
  - Based on how a component is used in the project
- Define merge rules
  - Only bug fixes for some components
  - Regular merge down, copy up for active components
Workspace storage

- Leverage storage solutions for workspace cloning
  - ZFS clones or commercial equivalent
  - FUSE
- Provides efficient data management for large workspaces (> 1 GB)
- Fast workspace creation and duplication
A Reference Framework

A starting point with visual tools
Goals

- A starting point, not a complete solution
- Illustrate how the tools can solve different parts of the problem
  - Triggers for policy enforcement
  - Broker as a centralized substitute for end user tools
  - P4V applets for visual guidance
Component Data Model

Pool of components in depots

Configuration:
- Per-project
- Change at any time
- Each component’s location, use, and optional view

Workspace
- drivers
- fileio
- libs-db
- libs-gui
<configuration>
  <component>
    <depotPath>//depot/comp/audio/...</depotPath>
    <access>active</access>
    <clientLocation>audio/...</clientLocation>
  </component>
  <component>
    <depotPath>//depot/comp/encoder/...</depotPath>
    <access>write</access>
    <clientLocation>encoder/...</clientLocation>
  </component>
  <component>
    <depotPath>//depot/comp/test/...</depotPath>
    <view>comp.test.l</view>
    <access>read</access>
    <clientLocation>test/...</clientLocation>
  </component>
</configuration>
Use Cases

- Configuration CRUD (CLI/P4V)
- Generate workspace
- Update workspace when configuration changes
- Switch workspace to new configuration
- Smart sync
- Intercept edit/submit to enforce configuration rules
- List component consumers (dependency relationships)
## Configuration CRUD

<table>
<thead>
<tr>
<th>Name</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>acme</td>
<td>View</td>
</tr>
<tr>
<td>bigproj</td>
<td>View</td>
</tr>
<tr>
<td>myproj</td>
<td>View</td>
</tr>
<tr>
<td>rd</td>
<td>View</td>
</tr>
</tbody>
</table>

### Configuration: acme

<table>
<thead>
<tr>
<th>Component Path</th>
<th>Access</th>
<th>Location</th>
<th>View</th>
</tr>
</thead>
<tbody>
<tr>
<td>//depot/comp/abcd/...</td>
<td>active</td>
<td>abcd/...</td>
<td></td>
</tr>
<tr>
<td>//depot/comp/def/...</td>
<td>write</td>
<td>def/...</td>
<td></td>
</tr>
<tr>
<td>//depot/comp/ghvl/...</td>
<td>read</td>
<td>ghvl/...</td>
<td>comp.ghvl</td>
</tr>
<tr>
<td>//depot/lib/xyz/...</td>
<td>binary</td>
<td>lib/xyz/...</td>
<td></td>
</tr>
</tbody>
</table>
# Trace Dependencies

<table>
<thead>
<tr>
<th>Component</th>
<th>Consumed By</th>
</tr>
</thead>
<tbody>
<tr>
<td>abc</td>
<td>• acme</td>
</tr>
<tr>
<td></td>
<td>• bigproj</td>
</tr>
<tr>
<td></td>
<td>• myproj</td>
</tr>
<tr>
<td></td>
<td>• rd</td>
</tr>
<tr>
<td>def</td>
<td>• acme</td>
</tr>
<tr>
<td></td>
<td>• bigproj</td>
</tr>
<tr>
<td></td>
<td>• myproj</td>
</tr>
<tr>
<td>ghi</td>
<td>• acme</td>
</tr>
<tr>
<td></td>
<td>• bigproj</td>
</tr>
<tr>
<td></td>
<td>• myproj</td>
</tr>
<tr>
<td>xyz</td>
<td>• acme</td>
</tr>
<tr>
<td></td>
<td>• bigproj</td>
</tr>
</tbody>
</table>
Nuts & Bolts

- Uses triggers, broker, and P4V applets
  - Custom commands defined
  - Guard rails in place
- Defines storage location for scripts and configuration in depot
- Uses XML data formats
- Uses spec depot and attributes
- Written with P4Python
Enhancements

- Component hierarchies / nesting
- More flexible command usage – specify file arguments, override defaults
- Support streams
- Immutable / Automatic labels
- Multiple depot paths per component
Questions?

Contact us!