Software Configuration Management Patterns

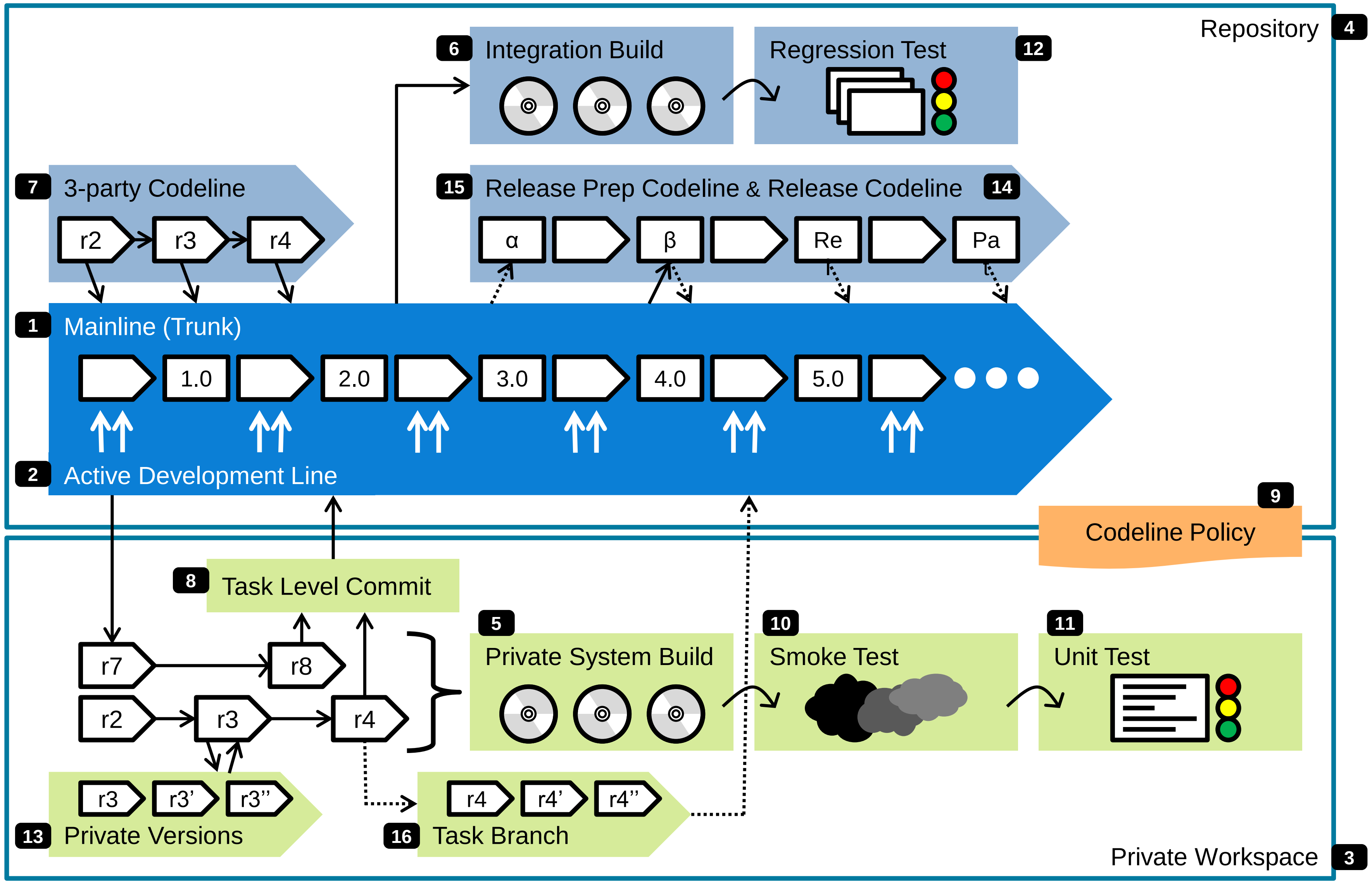
proces360
www.proces360.dk

Version 1.0

Software configuration management is a complex discipline because it is a bridge and an interface between crucial technical aspects of software development and management of the product and solution

These patterns* condensate important reusable strategies for organizing configuration management in the real world

*Based on Berczuk and Appleton: "Software Configuration Management Patterns"



Why implement software configuration management?

- Complete view of **scope** as deliverables (CI's)
- Scope management through **change control**
- Consistency between **plans** and the **status** of its work products
- System for **creating and changing** work products in parallel across team(s)
- **Information** about change, status of work products and relations between work products
- Easy **access** to project materials with a known status
- Ability to store **cross-organization** information
- Product level **change control** (e.g. by Product Managers)
- **Stakeholder management** regarding product change
- **Traceability**, supported by versions, status and change control

1 Mainline (or Trunk)

Why? The dynamics of the development could lead to a complex and cluttered version tree through many branches, but branching is a necessary mechanism to avoid serialization of work

How? Create a codeline for development to minimize integration effort from branching and merging

2 Active Development Line

Why? Development causing rapid and massive changes to mainline may also cause instability and thus making it useless

How? Establish frequent synchronization points, and mechanisms such as integration builds, to protect soundness through criteria for check-in

3 Private Workspace

Why? If all developers work directly on the mainline, they would be disturbed by many irrelevant activities and conflicting changes

How? Create a separate workspace for each individual and/or team to isolate developers from others work and do frequent synchronization to avoid outdated code

4 Repository

Why? It can be hard to identify the right version of code, components, and documents for a new workspace

How? Create repository as a single point of access to information. Also consider other useful mechanisms than CM system, e.g. file shares

5 Private System Build

Why? Changes added to mainline may break the build and thus create problems for other than the author

How? Isolated build

- Build locally similar to global integration build
- Include all dependencies
- Include dependent components

6 Integration Build

Why? Because the mainline is the home codeline, we need to protect it so it always builds reliably

How? Continuos integration

- Perform a complete build
- Do a clean build based from the CM system
- Do a central frequent build, e.g. nightly or continuously

7 Third Party Codeline

Why? Third party code needs to be coordinated into the mainline as releases are not synchronized and needs integration

How? Add third party code, components (e.g. reusable java beans), libraries, frameworks (e.g. NET), etc. to CM system and branch

8 Task Level Commit

Why? Committed changes at the task level align with work of teams and needs to be integrated, debugged and comprehended (by other that the team authors)

How? Commit new feature, solved issues, or refactored parts as whole. Commit at least once a day, if it makes sense

9 Codeline Policy

Why? A group of people needs to align with rules and expectations regarding the way to work

How? Communicate

- Which components are included in codelines
- How and when to check in/out and branch/merge
- Data management
- Promotion rules between codelines

10 Smoke Test

Why? Protect mainline integrity from changes but avoid significant overhead

How? Detect changes that cause obvious problems, using 80/20-effort testing

- Quick to run tests
- Automatic and self-evaluating tests
- Test broad rather than for deep coverage
- Base test on experience

11 Unit Test

Why? After introducing a change, other parts may have stopped working violating the full contract of the components

How? Run unit test on changed components

- Simple to run tests
- Automatic and self-evaluating tests
- Fine grained & isolated
- Testing the contract

12 Regression Test

Why? Protect mainline integrity from side-effects of changes and recurring problems avoiding full, and manual, test

How? Focus on

- Define a set of test cases reflecting risks
- Build test on cases that has failed before
- Verify implementation of requirements

13 Private Versions

Why? Developers needs to do rapid experimenting and discovery without breaking other work in progress, but a team requires its work isolated until quality is established

How? Provide local revision control area ("scratchpad") and support promotion mechanisms, e.g. from individual to team space

14 Release Codeline

Why? A release may require maintenance while development needs to continue

How? Support maintenance

- Keep each released version as a branch
- Allow branches to progress with bug-fixes
- Merge relevant bug-fixes back to mainline

15 Release Prep Codeline

Why? Stabilizing code for a release while development continues

How? Focus on

- Avoid freezing the branch when code approaches release quality
- Stabilize and update development codeline
- A branch may be promoted to become the release branch

16 Task Branch

Why? Multiple, long-term, and overlapping changes can occur unsynchronized with mainline, e.g. for an unknown future release, for product merges or major architectural refactoring

How? Create branch to hold the work and thereby encapsulate risk and planning if, how and when to merge into mainline again