**POM 10 – Software Configuration Management I**

*Why configuration management?*

* Multiple people work on artifacts that are changing
* More than one version of the artifact has to be supported
  + Released software systems
  + Custom configured systems (different functionality)
  + Systems under development
  + Software running on different machines & operating systems
* Need for coordination
* Software configuration management (SCM)
  + Manages evolving software systems
  + Controls the costs involved in making changes to a system

*What is software configuration management?*

Definition:

* **set of management disciplines** within a software engineering process to develop a **baseline**
* It encompasses (umfasst) the disciplines and techniques **of initiating, evaluating and controlling change** to work products during and after a software project

IEEE 828-2012: IEEE standard for configuration management in systems and software engineering

**Baseline**: a work product that can be changed only through a change control procedure

*Administering software configuration management:*

* Software configuration management is a **project function** with the goal to make technical and managerial activities more effective
* Software configuration management can be administered in several ways:
* Organization-wide
* Project-specific
* Distributed among the project members
* Mixture of all of the above

*Configuration management roles:*

* **Configuration manager**
  + Responsible for identifying configuration items
  + Often responsible for defining the workflows for creating promotions and releases
* **Change control board member**
  + Responsible for approving or rejecting change requests
* **Developer**
  + Creates promotions triggered by change requests or the normal activities of development
  + Checks in changes and resolves conflicts
* **Auditor**
  + Responsible for the selection and evaluation of promotions for release and for ensuring the consistency and completeness of this release

*Software configuration management activities:*

* **Configuration item identification**
  + Modeling the system as a set of evolving components
* **Promotion management**
  + The creation of versions for other developers
* **Build and Release management**
  + The creation of versions for clients and users
* **Change management**
  + The handling, approval & tracking of change requests
* **Branch management**
  + The management of concurrent development
* **Variant management**
  + The management of coexisting versions

*Terminology: configuration item:*

An aggregation of software, hardware, or both, designated for configuration management and treated as a single entity in the configuration management process

* **Software configuration items**: source files, models, tests, documents, configurations
* **Hardware configuration items**: CPUs, sensors, actuators

*Configuration item identification:*

Not every entity needs to be under configuration control all the time

Two Issues:

1. What: Selection of configuration items

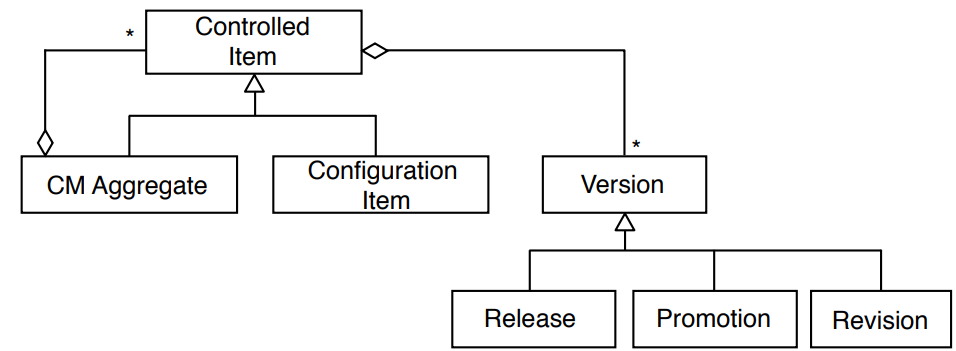
* What should be under configuration control?

1. When: When do you start to place entities under configuration control?

* In early days, it was an activity
* Nowadays it should be a project function

*Terminology: version:*

* **Version**: an initial release or re-release of a configuration item associated with a complete compilation or recompilation of the item
* **Release**: the formal (external) distribution of an approved version, e.g. a potentially shippable product increment in Scrum
* **Promotion**: a version that is made available (internally) to other developers, e.g. a new commit in the version control system
* **Revision**: change to a version that corrects only errors in the design/code, but does not affect the documented functionality

*Object model for configuration management (UML class diagram):*

*Terminology: baseline:*

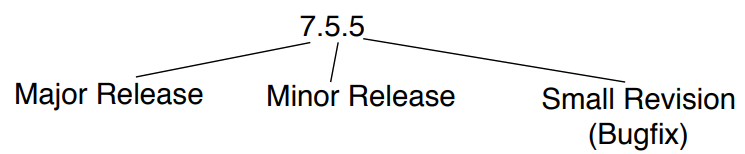
* A specification or product that has been formally reviewed and agreed to by responsible management
* Thereafter it serves as the basis for further development, and can be changed **only** through change control procedures
* Examples
  + Baseline A: the API has been completely been defined; bodies of methods are empty
  + Baseline B: all data access methods are implemented and tested
  + Baseline C: the GUI is implemented

*Types of baselines:*

* As systems are developed, a series of baselines is developed, usually after a review (analysis review, design review, code review, system testing, ...)
  + **Developmental baseline**
  + **Functional baseline**
  + **Product baseline**
* Branch Management allows to transition between these baselines

*Naming Schemes for Baselines (Tagging):*

* Many naming scheme for baselines exist (1.0, 6.01a, ...)
* A 3 digit scheme is quite common:



*History of software configuration management tools:*

* RCS: The first on the block [Tichy 1975]
* CVS (concurrent version control)
  + Based on RCS, allowed concurrent working without locking
* Perforce
  + Repository server; allows to keep track of developer’s activities
* ClearCase
  + Multiple servers, process modeling, policy check mechanisms
* Subversion
* Git

*Version control systems (VCS):*

* VCS allow many software developers to collaborative work on the configuration items
* VCS store different versions of configuration items (e.g. source code and configuration data) in a commit history and allow to restore previous versions
* The commit history allows developers to see how the configuration items changed over time and to see who changed a certain item
* Revisions are stored in a repository and developers can check out a revision into a working copy
* Distributed version control systems (DCVS), also known as distributed revision control or decentralized version control, provide more flexibility and features

*Monolithic architecture for version control:*

* Developers have simple local database, keeps all changes to files under revision control
* Example: RCS (Revision Control System)
* Still distributed with many computers today

*Repository architecture for version control:*

* A single (central) server contains all the versioned files
* Developers check out files from the server to their computer, change them and check them back into the central server
* Administrators have fine-grained control over who can do what
* **Problem**: single point of failure in the central VCS server: possibility of loosing all the versions and their history if the server crashes

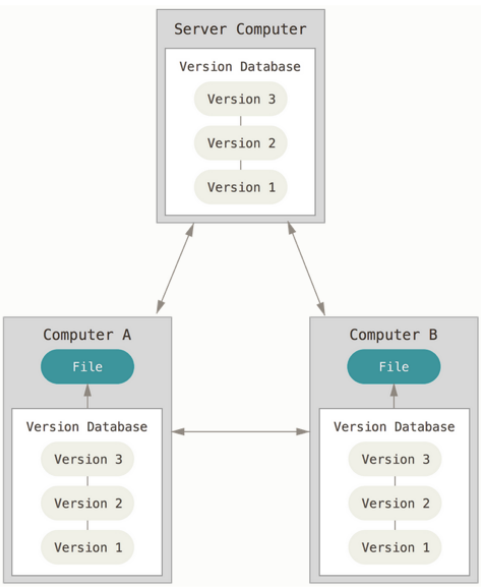
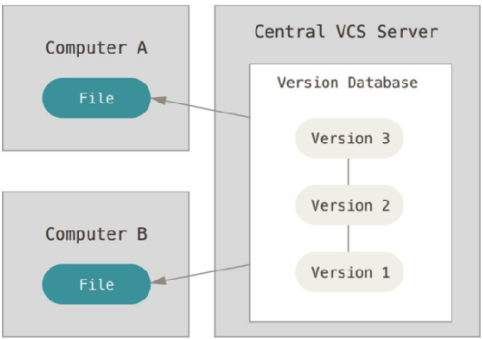
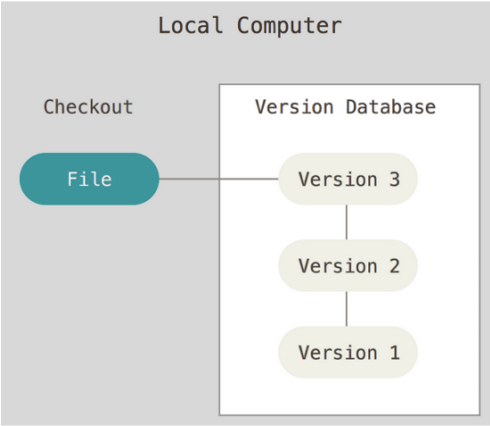
*Example of a repository architecture: Subversion:*

* Open source project
* Based on CVS (Concurrent Versions System)
* Distinction between programmer’s directory (working copy) and master directory (central repository)
* Commands:
  + Checkout: Check out a programmer’s working copy from the server repository
  + Add: Add a file to the programmer’s working copy
  + Delete: Delete a file in the programmer’s working copy
  + Commit: Commit changes from the programmer’s working copy to the server repository (create a new version and promote it)
  + Diff: Comparison between 2 versions
* The time for branch management is independent of the size of the system (unlike CVS, which creates physical copies of the files, Subversion uses only tags)

*Peer-to-peer architecture for version control:*

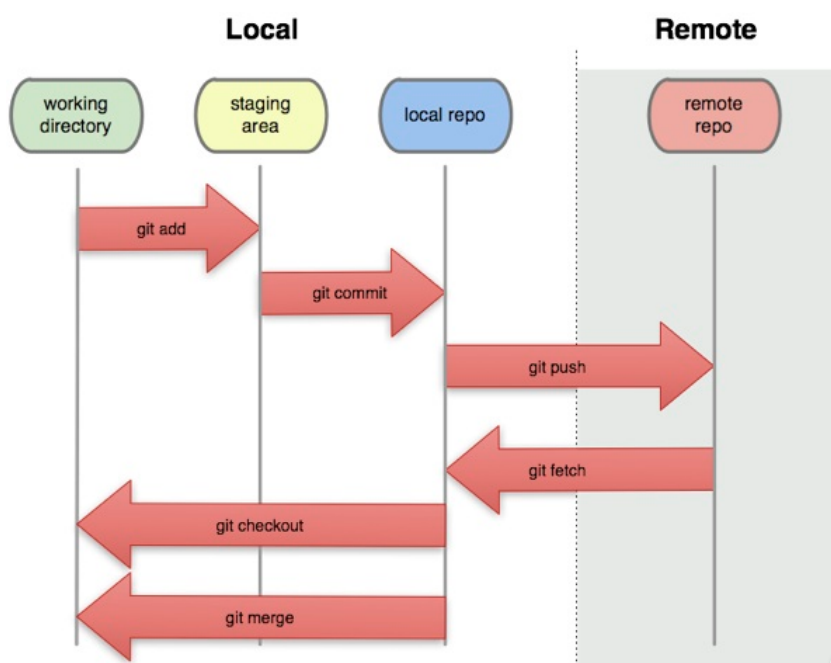
* Addresses the single point of failure problem
* Each programmer’s computer (Computer A, Computer B, …) fully mirrors the repository (Server Computer)
* Programmers can work offline and create versions (commits and branches)
* Not all versions are promoted to the master directory (they need to be pushed)
* If the server dies and a programmer has a full copy of the repository, it can be copied back to the server computer
* Example: git

**monotholic** **repository peer-to-peer**



*Example of a peer-to-peer architecture: git*

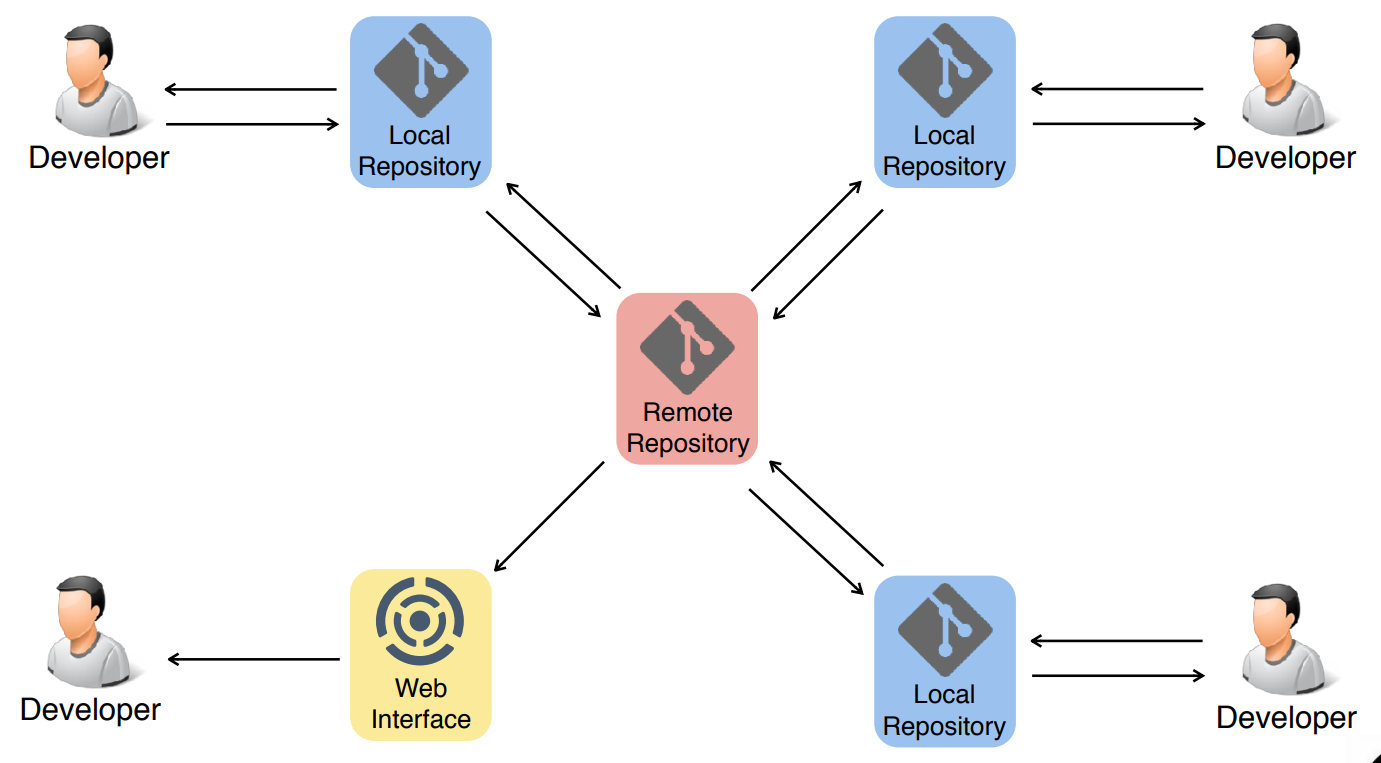
* Open source project
* Supports light-weight local branching
* Commands: clone, commit, push (=promote), fetch, merge, pull
* Differences to Subversion:
  + Support for multiple repositories, subversion supports only single repositories
  + Branches are light-weight



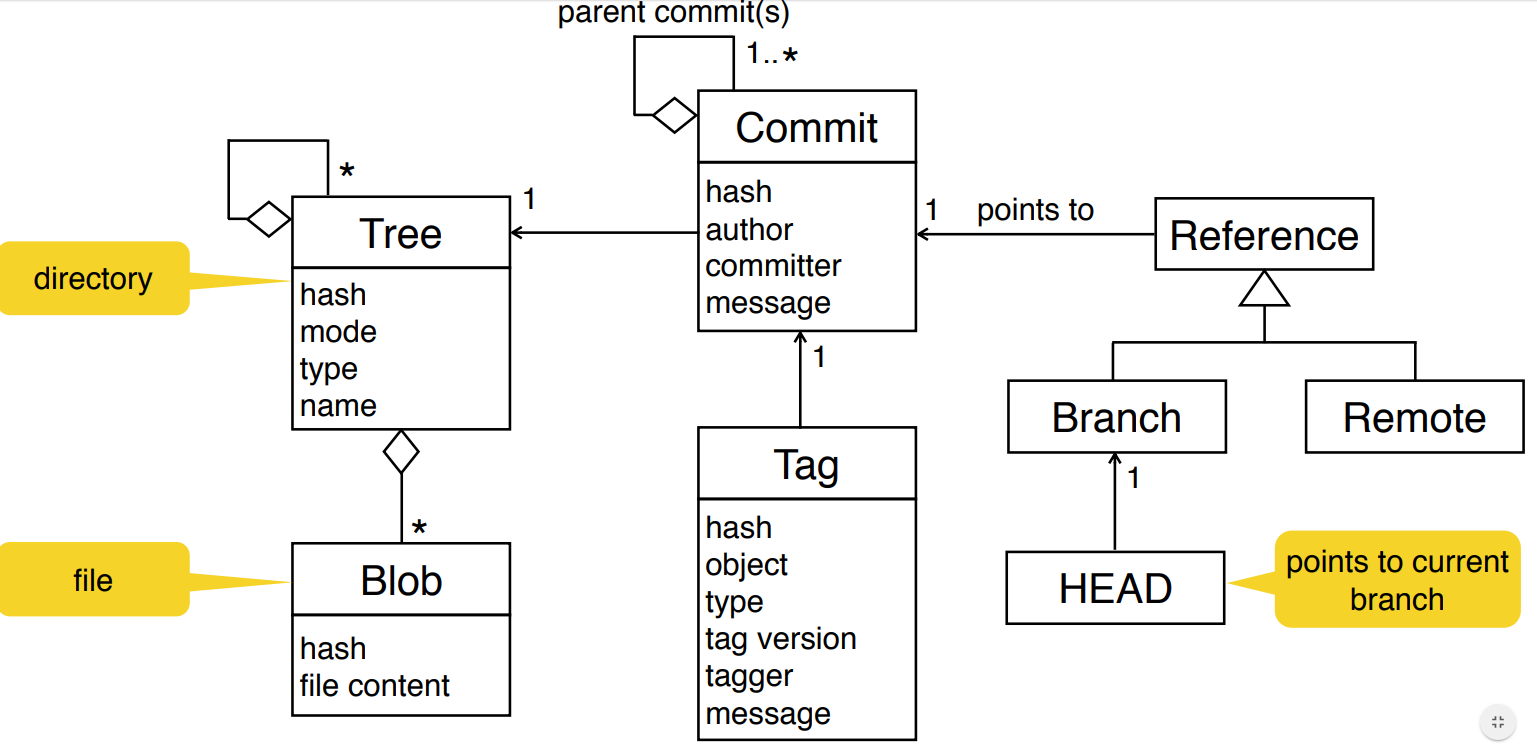
*Comparison of distributed vs centralized version control systems:*

* Advantages of distributed VCSs
  + Ability to work offline (local commits)
  + Ability to work incrementally (small commits)
  + Ability to context switch efficiently (lightweight branching)
  + Ability to do exploratory coding efficiently (lightweight branching)
* Disadvantages of distributed VCSs
  + High learning curve
  + Scaling issues
  + Less administrative control

**Distributed version control**



*Git object model (UML class diagram):*



*Resolve a merge conflict:*

1. Take mine (i.e. my changes and ignore the other changes)
2. Take theirs (i.e. the other changes and ignore my changes)
3. Merge the overlapping changes manually and decide per case possibly taking both changes

*Distributed version control:* ***best practices****:*

* Commit related changes (small commits, example: fixing 2 bugs should produce 2 commits)
* Commit and push often (share core more frequently with others)
* Do not commit half done work (commit code when it is completed)
* Test before you commit (… at least build and make sure it compiles)
* Write meaningful and understandable commit messages (summary, body)
* Do not use version control as a backup system
* Keep your working copy of the repository up to date (regularly pull and push)
* Use branches
* Agree on a workflow
* Do not change published (promoted) history