SOFTWARE CONFIGURATION MANAGEMENT

Dr. SELİN METİN

Orion Innovation Türkiye

Software Configuration Management Overview

What is Software Configuration Management?

- A Software Configuration Management (SCM) system is an essential part of almost any effective software development project. It can provide solutions to different challenges faced by the many roles in a development team - software engineers/developers, project leaders, release engineers, process engineers, engineering managers, and even engineering executives.

- Source control (or version control) is the practice of tracking and managing changes to code.

- A **Source Code Management (SCM)** is a software tool used by programmers to manage the source codes.

- **SCM** systems provide a running history of code development and help to resolve conflicts when merging contributions from multiple sources.

Source Control Basics

- Whether you are writing a simple application or a collaborating on a large software development project as part of a team, source control is a vital component of the **Software Development Life Cycle** (SDLC).
- SCM systems allow you to track your code change, see a revision history for your code, and revert to previous versions of a project when needed.
- With SCM systems, you can collaborate on code with your team, isolate your work until it is ready, and quickly trouble-shoot issues by identifying who made changes and what the changes were.
- SCM systems help streamline the development process and provide a centralized source for all your code.

Why Do We Need Source Code Management?

- You would have probably made up your own version control system without realizing it. Got any of these?
  - Source Code Management _ 20Jan2020
  - Source Code Management _ 22Jan2020
  - Source Code Management _ 10Feb2020
- It is why we use “Save As” when we want a new file without changing the old one.
- We can save the file with a different name if it’s our school project or one-time papers but for a well-equipped software development? Not a chance.
- Big projects need a version control system to track the changes and avoid misunderstanding.

Features of Source Code Management

- SCMs are used to give versions/revisions to the program.
  - Each version is given a timestamp and includes the person responsible for the change.
  - Even various versions can be compared and merged with other versions.
  - This is why SCM is also referred to Version Control, Revision Control or Source Control.

- Below are some of the basic features of a source code management system:
  - Authenticated access for commits
  - Revision history on files
  - Atomic commits of multiple files
  - Versioning/Tagging

Features of Source Code Management

- A good SCM does the following:
  - **Backup and Restore** – Files can be saved at any moment and can be restored from the last saved.
  - **Synchronization** – Programmers can get the latest code and fetch the up-to-date codes from the repository.
  - **Short-Term Undo** – Working with a file and messed it up. We can do a short-term undo to the last known version.
  - **Long-Term Undo** – It helps when we have to make a release version rollback. Something like going to the last version which was created a year ago.
  - **Track Changes** – We can track the changes as when anyone is making any change, he can leave a commit message as for why the change was done.
  - **Ownership** – With every commit made to the master branch, it will ask the owner permission to merge it.
  - **Branching and Merging** – You can create a branch of your source code and create the changes. Once the changes are approved, you can merge it with the master branch.

Time Machine

- When properly used, an SCM system will capture every key change in the evolution of a software system.
  - Not only checking in (committing) new versions of artifacts when a task is complete but also checking in intermediate states that capture notable progress.

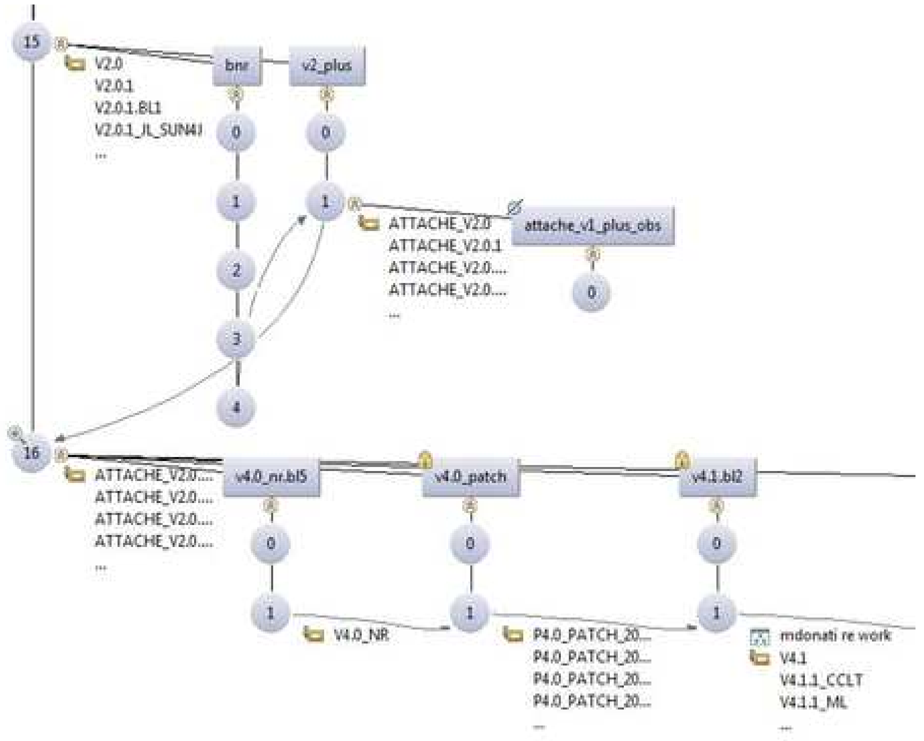
- An effective SCM system will make it easy to **reconstruct the software system at any point in the past**. In that sense, an SCM system functions as a time machine.

Parallel Development

- All development projects need a way to progress on one set of tasks in isolation from progress on other tasks.
  - Eg: One part of a team might have to implement a feature that will take a week to complete and introduces numerous incompatible changes. Being able to implement the feature in isolation from the rest of the project and then deliver the completed change all at once can save the rest of the team from constantly adapting to incremental changes or worse, being unable to make progress on their tasks until the project stabilizes again.
- Many software systems have multiple versions used by their customer base simultaneously. If a defect is discovered in a released version, the team must be able to reconstruct the software system as it was when that version was released, fix the defect, and then release an update to the customers. They must be able to do this in a way that does not pick up features that were implemented in later releases.

Parallel Development

- SCM Systems typically support parallel development through the use of branches or streams.
  - A starting point (often referred to as a foundation which is usually a baseline or equivalent, a set of versions of files that compose a consistent and compatible set)
  - Create a branch for a given file if and when a change is needed to that file.



Parallel Development

- In most circumstances, changes made on one branch are later needed on one or more other branches.
  - Eg: a defect fix to an early release of a software system is usually also needed in the later releases.
- SCM systems often provide a mechanism to merge changes from one branch to another.
  - The more powerful this mechanism is (including three-way merge support, effective common ancestor computation, powerful compare/merge GUIs and command line support), the lower the burden on the person performing the merge operation and the more effective the system will be handling large software systems with a huge number of files that may require merging.

Traceability

- **Project Lifecycle Traceability** allows related artifacts to be traced through much, if not all of the project lifecycle.

- **Change traceability** allows the history of changes in the SCM system to be retrieved. It can typically answer the “Who?”, “What?”, “Where”, and “When” questions of each change. It may also capture the “Why?” question (especially if there is a linkage to an associated task in the change management system). Examining the changes in the versions involved (greatly aided by effective compare/diff tools) can answer the “How?” question.

- **Build and Release Traceability** allows a release / build artifact to be traced back to the versions of files that were used to build that artifact. This can be a significant time saver when trying to track down numerous problems.

Other SCM Features

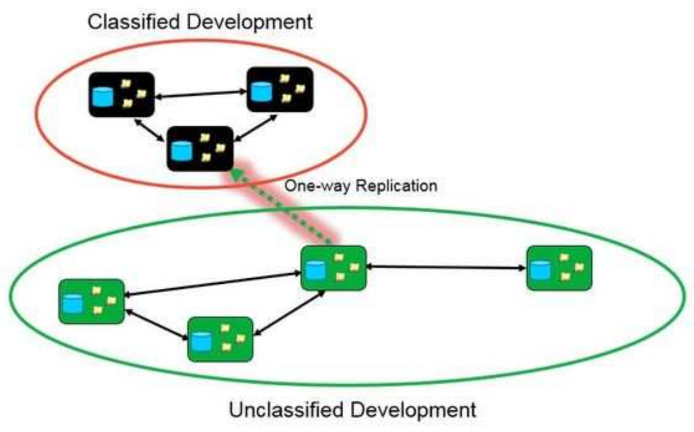
- **Process Automation and Enforcement:** Most software projects have a defined process. Teams can follow process checklists to manually ensure proper procedures are followed but this can be tedious and error prone. The degree to which the process can be automated and enforced by the tools involved can greatly increase the ease of adhering to the process and can avoid costly mistakes.

- **Security:** Different security requirements exist. E.g. Open source projects are open to all but still need to limit who can deliver a change to a stream that will be used to build a release. It may be important to restrict who has access to data that was copied out of the SCM repository to a client disk.

- **Subcontracting:** Many projects involve contracting at least portions of the project to an external organization. The contractors may work on-site or off-site. There are often additional security requirements that must be managed. Mechanisms to automate and enforce the development process enforced by the SCM system are important.

Other SCM Features (continued)

- **Classified Development:** Many military and government projects have stringent requirements that span security, process enforcement, and traceability. The degree to which the SCM tool provides solutions to classification challenges can greatly impact the cost of development and level of confidence that the project requirements are met.



- **Scalability:** The size of the software development team and the size of the software code base affect the load placed on the SCM system. It's important not only to understand the current size of a project but also to predict what it is likely to look like as it evolves over the years. Choosing an SCM system that is flexible and powerful enough to scale to the estimated needs of at least the next few years can avoid having to risk changing an important part of the development team's infrastructure at some inconvenient point in the future, which can be a very disruptive undertaking.

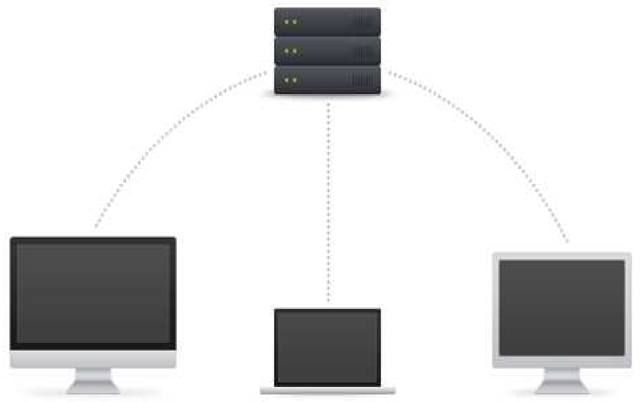
Types of Version Control Systems

- There are two main types of Version Control System:
  - Centralized
  - Decentralized (or Distributed)

Centralized Version Control

- Centralized Version Control works in a client and server relationship. The repository is located in one place and allows access to multiple clients.

- There is a single, (centralized) master copy of the code base, and pieces of the code that are being worked on are typically locked, (or “checked out”) so that only one developer is allowed to work on that part of the code at any one time. Access to the code base and the locking is controlled by the server. When the developer checks their code back in, the lock is released so it’s available for others to check out.
  – It’s very similar to FTP where you have FTP clients which connect to and commits have to pass through the central server.
  – E.g: IBM Rational Clearcase, CVS, Subversion (SVN).



- The benefits of centralized version control:
  – It’s easy to understand.
  – There are more GUI and IDE clients.
  – You have more control over the users and access.

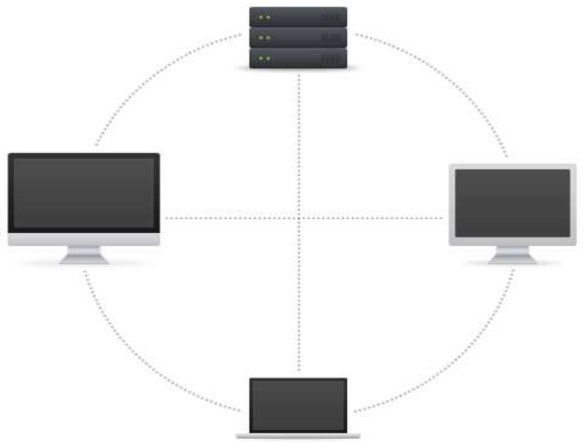
- We do have drawbacks also:
  – It is dependent on the access to the server.
  – It can be slower because every command from the client has to pass the server.
  – Branching and merging strategies are difficult to use.

Distributed Version Control

- These systems are newer to use.
- These systems work on a peer-to-peer model: the code base is distributed amongst the individual developers' computers. In fact, the entire history of the code is mirrored on each system - each user has their own copy of the entire repository as well as the files and history.
- There is still a master copy of the code base, but it's kept on a client machine rather than a server. There is no locking of parts of the code; developers make changes in their local copy and then, once they're ready to integrate their changes into the master copy, they issue a request to the owner of the master copy to merge their changes into the master copy.
- The emphasis switches from versions to changes, and so a new version of the code is simply a combination of a number of different sets of changes. That's quite a fundamental change in the way many developers work.
  - E.g: Git and Mercurial

- The benefits of distributed version control are:
  - More powerful and easy change tracking.
  - No need of a centralized server. Most of the functionalities work in offline mode also apart from sharing the repositories.
  - Branching and Merging strategies are more easy and reliable.
  - It's faster than the centralized one.

- Though we have drawbacks:
  - It is harder to understand.
  - It's new, so less GUI clients.
  - It is easier to make mistakes until you are familiar with the model.



Centralized vs. Distributed

- The key difference is that there is no locking of elements in a distributed system.
  - Every new set of changes that a developer makes is essentially like a new branch of the code, that needs to be merged back into the master repository.
- Distributed model
  - It's possible for two developers to be working on the same source file at the same time.
  - Performance of distributed systems is better, because there is no waiting for locks to happen across potentially slow network connections. Also, the complete code base is already on your local system.
  - Branching and merging is much easier to achieve in a distributed system, largely because it's built in to the way the system works.
  - You don't need to be connected to the network all the time.
- Centralized systems
  - Typically easier to understand.
  - Access control is easier, since everything is controlled from one place (the server).
  - Unless you want to, you don't have to merge different versions of the same code, which can be tricky.

SCM / Version Control Tools

- There are various tools associated with Source Code Management. Below are the few widely used tools:
  - GitHub
  - GitLab
  - BitBucket
  - IBM Rational Clearcase
  - SourceForge
  - Beanstalk
  - Apache Allura
  - AWS CodeCommit
  - Launchpad
  - Phabricator
  - GitBucket

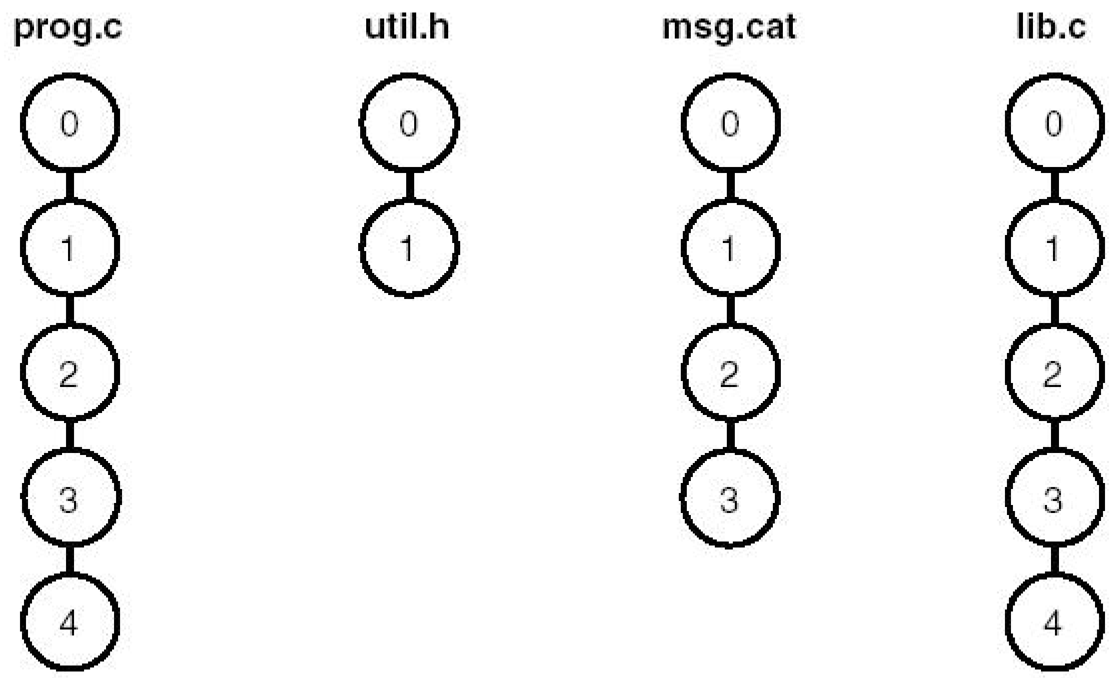
What is ClearCase ?

- It’s a comprehensive configuration management system that manages multiple variants of evolving software systems.
- ClearCase maintains a complete version history of all software development artifacts, including code, requirements, models, scripts, test assets, and directory structures.
- It performs audited system builds, enforces site-specific development policies, offers multiple developer workspaces, and provides advanced support for parallel development.
- ClearCase imbeds itself into the native file system of the platform’s operating system.

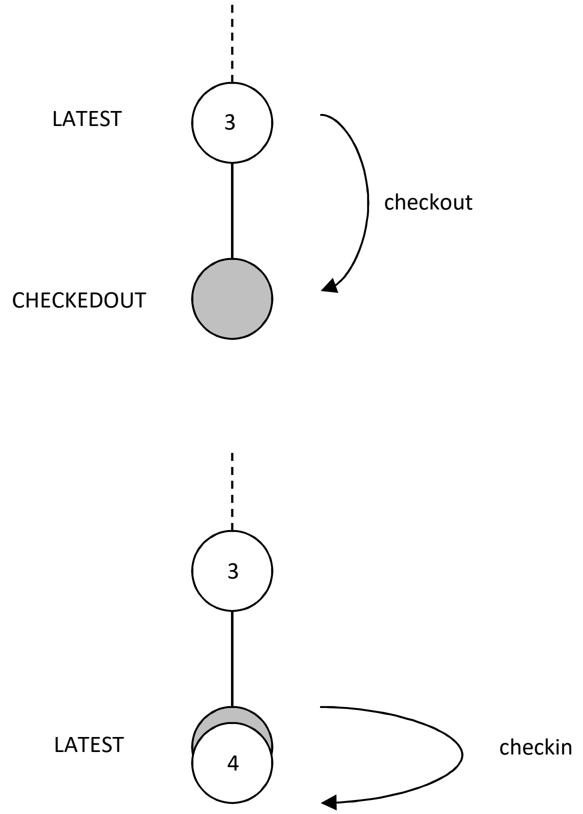
Basic Terminology of ClearCase

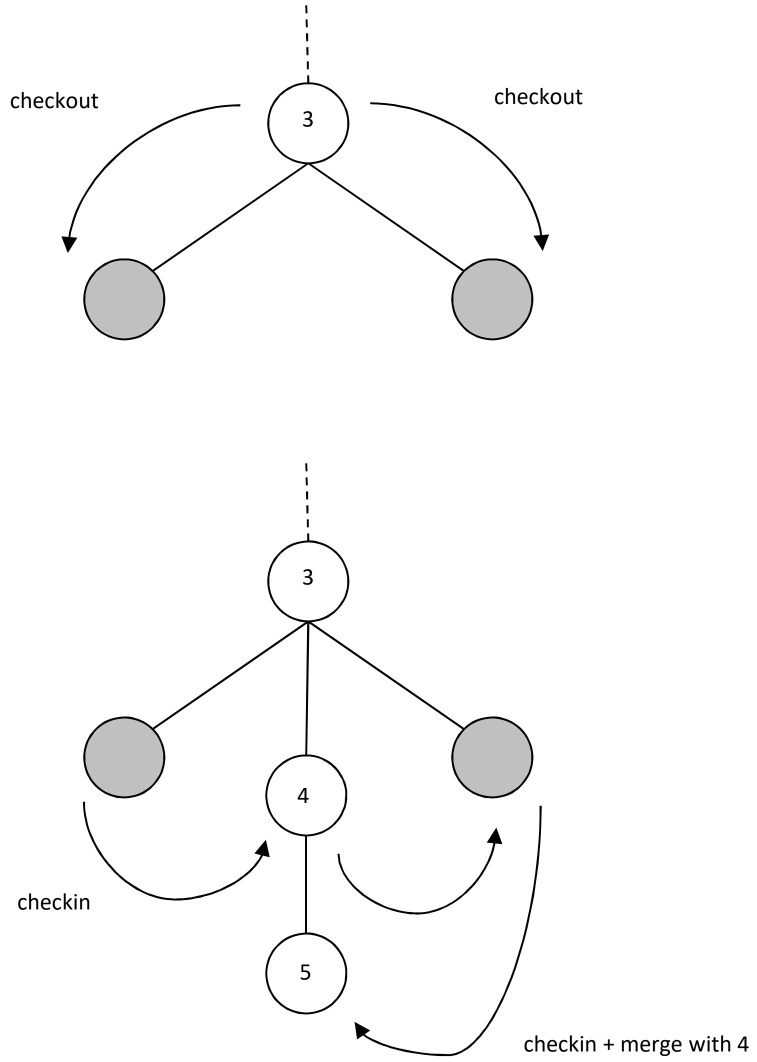
- File element
  - A file element is a file that contains software source code, a document, HTML code, XML code, or other data that can be stored in a file system. A directory element contains file elements and other directory elements.
- Version
  - A version is a specific revision of an element. For instance, instead of overwriting the same copy of your draft each time you work on it, you store a copy of the first version, the second version, and so on .
- Versioned object base (VOB)
- Check out-edit-check in model
- View

Elements and Their Versions

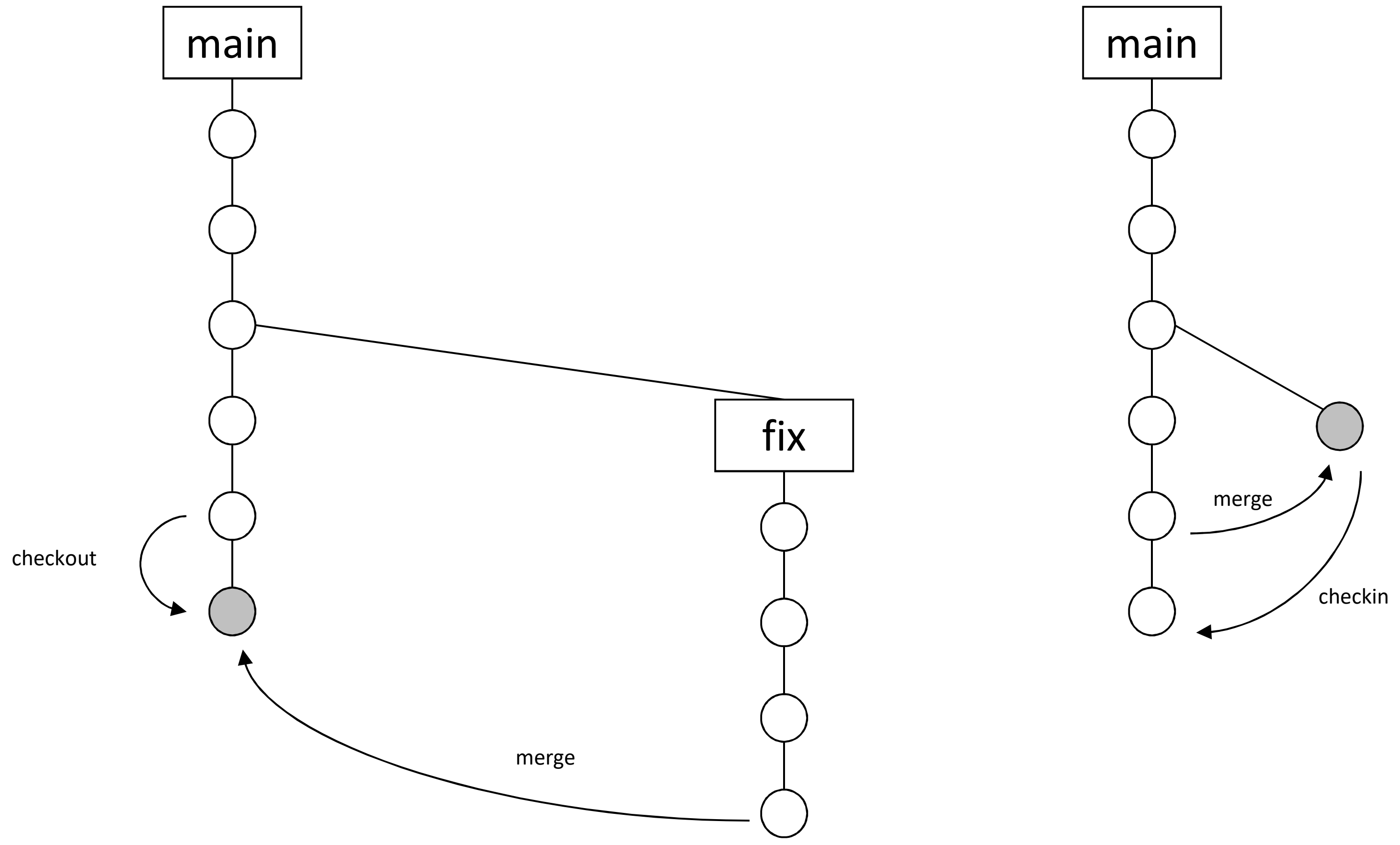


Check out-edit-check in model





Check out-edit-check in model



Reserved & Unreserved Checkout

- In some version-control systems, only one user at a time can reserve the right to create a new version on a branch (base ClearCase). In other systems, many users can compete to create the same new version.
- ClearCase supports both models by allowing two kinds of checkouts: reserved and unreserved.
- The view with a reserved checkout has the exclusive right to check in a new version for a given branch or stream.
- Many views can have unreserved checkouts.
- An unreserved checkout does not guarantee the right to create the successor version.
- If several views have unreserved checkouts, the first view to check in the element on a branch or stream creates the successor; developers working in other views must merge the checked-in changes into their own work before they can check in.

Merging Your Work

- **Merging** is the action of combining the contents of two or more files or directories into a single new file or directory.
  - You merge your work from your development branch to the integration branch when you want to make your changes visible to others on your team.
  - You merge from the integration branch to your development branch when you want to see other developers' changes.
- Merging Differences
  - The deliver operation compares the versions being delivered from the development stream with their counterparts in the target integration stream and invokes the ClearCase Merge Manager as needed which automatically resolves trivial differences.
    - Trivial differences are those that can be resolved without your manual intervention.
    - For nontrivial differences, ClearCase Merge Manager prompts you to resolve them manually when the merging occurs.

Merging Versions

- If another team member modified and delivered a version of the same file, you must perform a merge operation when you rebase to a baseline that contains the delivered version.
- As it does in a deliver operation, ClearCase merges all nonconflicting differences.
- For conflicting differences, ClearCase will prompt you to start the Merge Manager to resolve the conflicts.

How ClearCase Merges

- It identifies the base contributor.
- Next, it compares each contributor against the base contributor.
- For any line that is unchanged between the base contributor and any other contributor, ClearCase copies the line to the merge output file.
- For any line that has changed between the base contributor and another contributor, ClearCase performs a trivial merge by accepting the change in the contributor. Note, however, that depending on how you started the merge operation, ClearCase may copy the change to the merge output file. However, you can disable the automated merge capability for any given merge operation. If you disable this capability, you must approve each change to the merge output file.
- For any line that has changed between the base contributor and more than one other contributor, ClearCase requires that you resolve the conflicting difference.

Rebasing

- Project managers organize delivered activities into baselines. When baselines reach a satisfactory level of stability, after several cycles of testing and fixing defects, project managers will designate a recommended baseline.

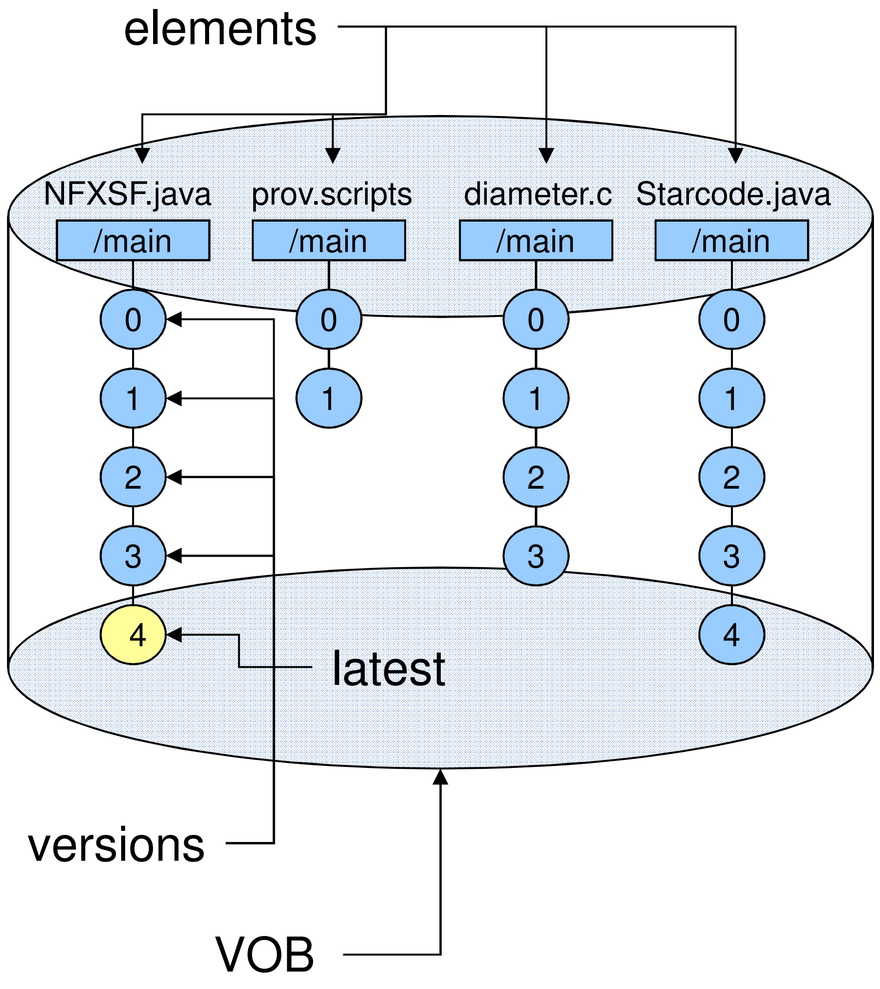
- As a developer, you need to update your development work area with the recommended baseline as soon as it becomes available.

Overview of the Rebase Process

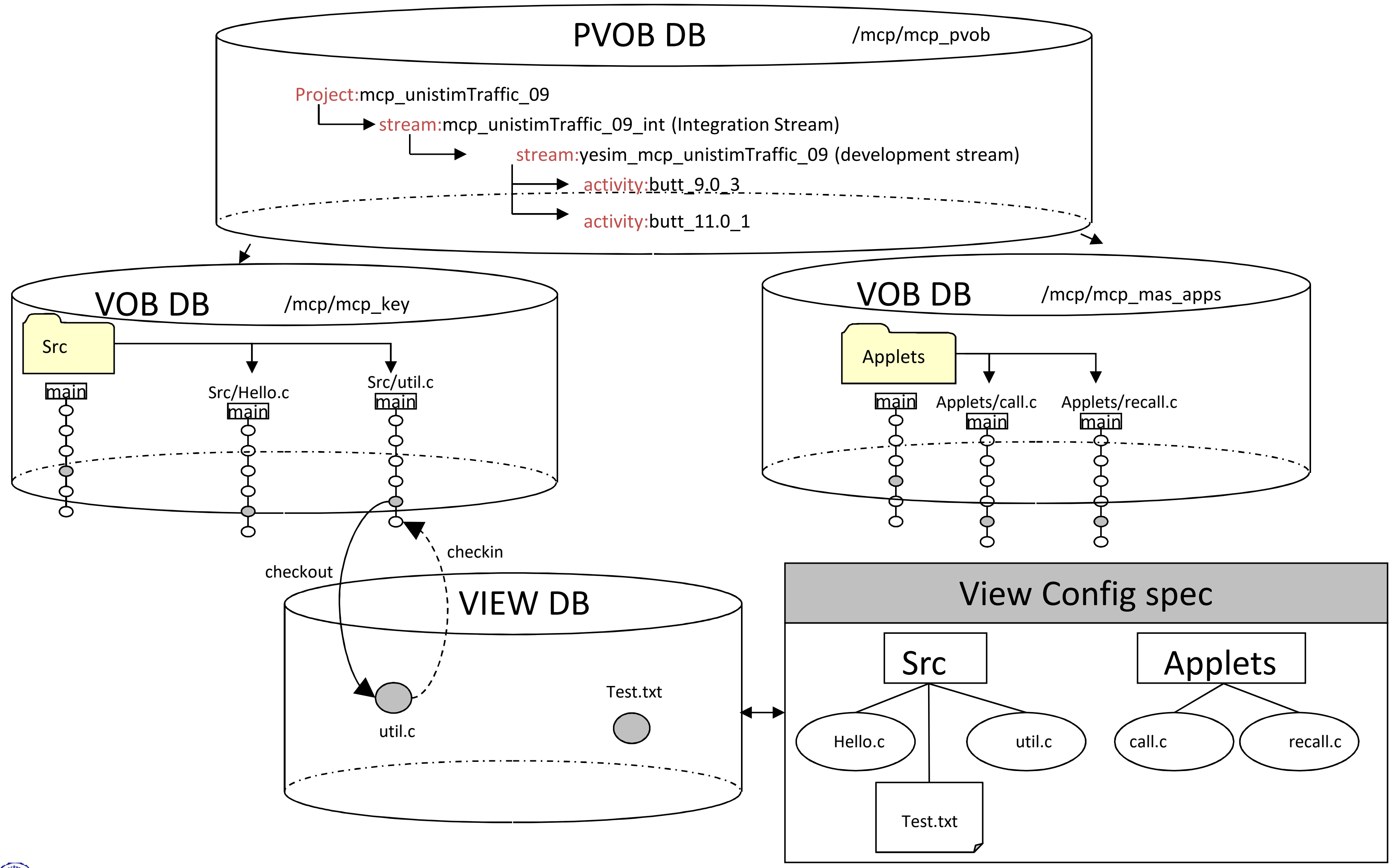
- **Prepare your work area** beforehand by finding all your checkouts and checking them in because the rebase operation cannot start from a view that contains checkouts. You can also check for differences between the earlier checked-in version, and see the check-in comments.
- **Start the rebase operation**. ClearCase will present you with the latest recommended baseline.
- **Be prepared to perform merge operations**. If another team member has modified and delivered a version of the same element you modified in your development stream, you will have to perform a merge operation when you rebase to a baseline that contains the delivered version. ClearCase merges all nonconflicting differences and will prompt you to decide how to resolve conflicting differences.
- **Test your undelivered work after the rebase**. After ClearCase reconfigures your work area and any conflicting merges have been resolved, you must verify that any undelivered work builds successfully in your work area using the most up-to-date elements of the new baseline. In the event of build errors, you will first need to resolve any conflicts before you can proceed to delivery your work.
- **Complete the rebase operation** when you are satisfied with your test builds.

ClearCase Versions, Elements and VOBs

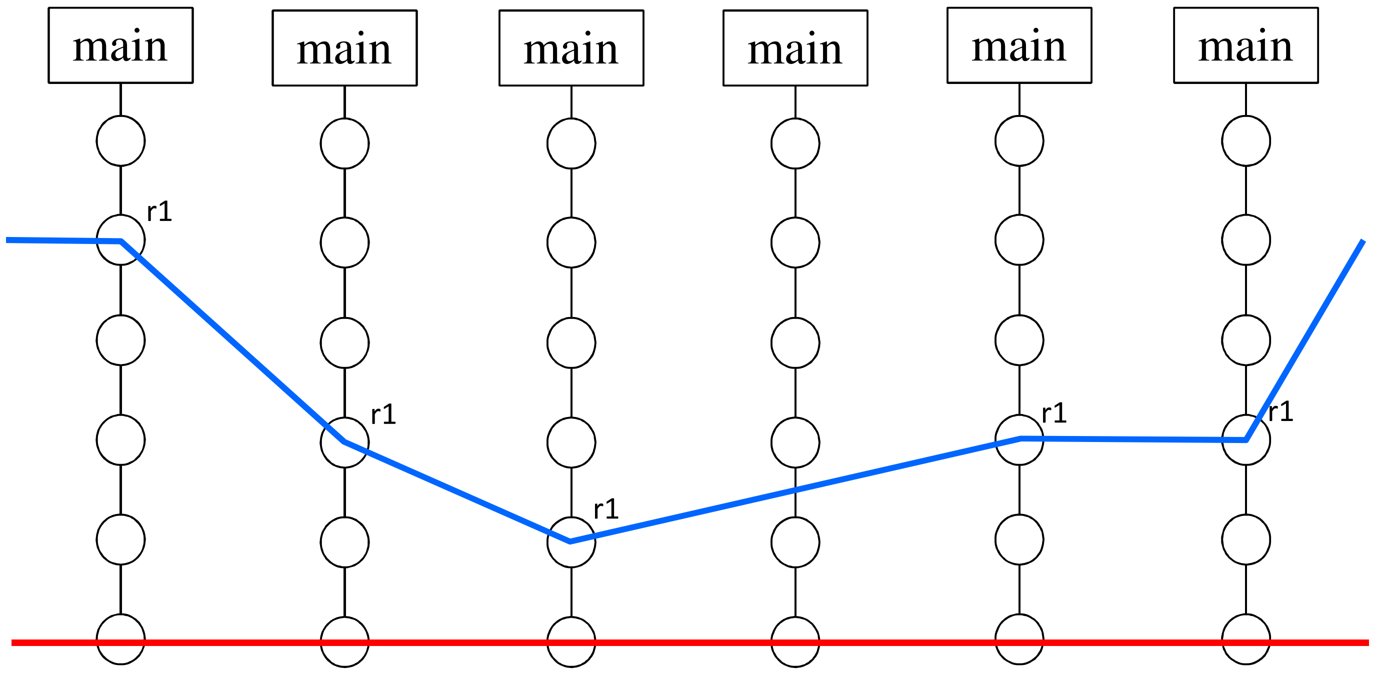
- Each time a file or directory is revised from a view, ClearCase creates a new version of it.
- Files and directories under ClearCase control are called elements and are stored in VOBs.
- Depending on the size and complexity of the development environment, ClearCase elements may be distributed across more than one VOB.



Elements and VOBs



Versions and Config Spec

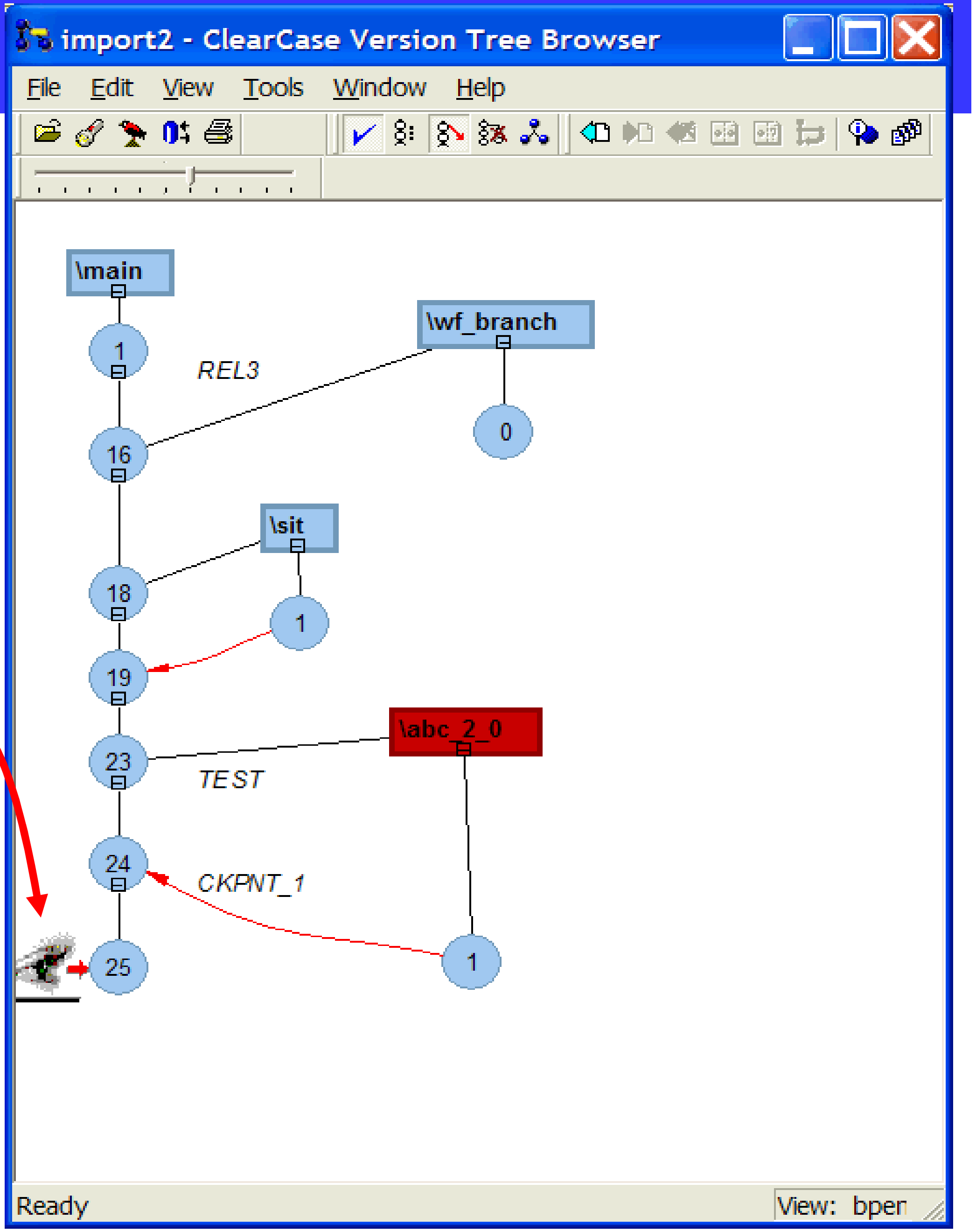


Config Spec

element * CHECKEDOUT
element * r1
element * /main/LATEST

Version Tree

- Displays a graphical view of the version history for an element.
- Each version of a resource is displayed as a node on the tree, and the version currently selected by your view is marked.
- You can use the ClearCase Version Tree view to perform ClearCase operations on resources by right-clicking on a version node, a branch, or a merge arrow and then selecting a ClearCase operation.



Controlling the Contents of Views

- A **view** is the mechanism ClearCase uses to provide access to a specific version of files and directories under source control.
- Rules determine which version of each element is visible and accessible through the view.
- The set of rules that determines which version of an element to display is referred to as the view's **config spec**.

Understanding the Role of the config spec

- Config specs are used to achieve a degree of control over project work.
- Project managers use config specs for any of the following reasons:
  - To control which versions developers see and what operations developers can perform in specific views.
  - To prohibit checkouts of all selected versions or restrict checkouts to specific branches.

Working with Branches

- Branches organize the different versions of files, directories and objects that are placed under version control.
- A **branch** is an object that specifies a linear sequence of versions of an element.
- The entire set of an element's versions is called a **version tree**.
- By default, ClearCase provides for every single element in a VOB one principal branch, called the main branch.

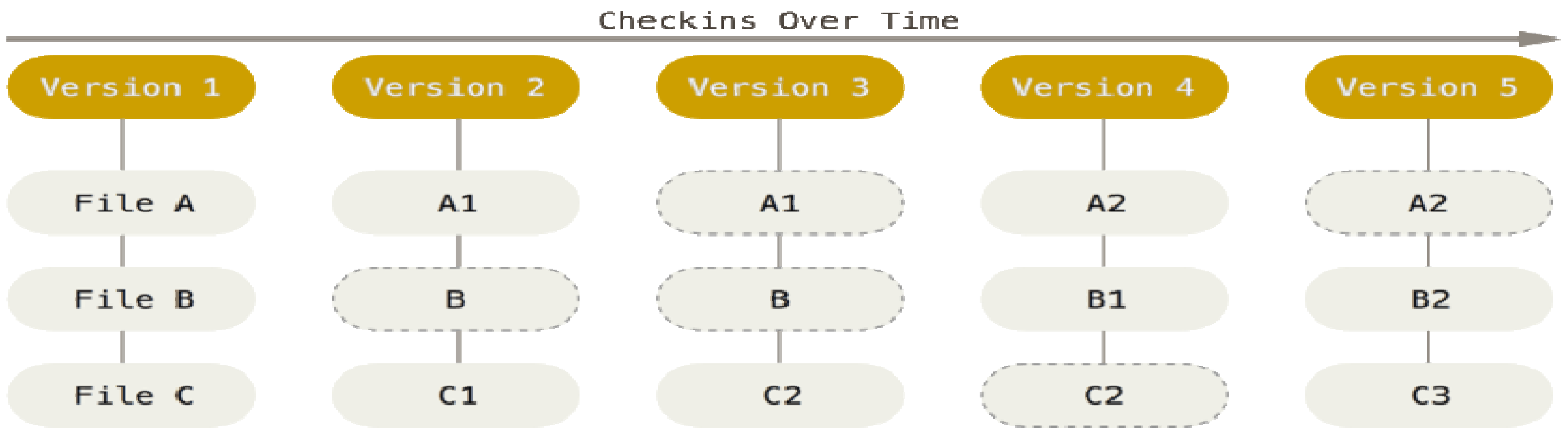
When to Use Branches

- A branch represents an independent line of development. Typically, different kinds of work is done on different branches.
  - E.g., to separate the work that is related to defect fixing from the regular development work, you can create a separate branch for the defect fixing of the main branch. The team working on the defect fixes can do their work without affecting or being affected by the work being done on the development branch.
- When the work on the subbranch is completed, you can integrate onto the main branch.
  - This is done by merging the most up-to-date version on the subbranch onto the elements’ main branch.
- The decision to use multiple branches for your project is part of a careful development planning strategy.

What is Git?

- The major difference between Git and any other SCM is the way Git thinks about its data. Conceptually, most other systems store information as a list of file-based changes. These other systems think of the information they store as a set of files and the changes made to each file over time.

- Instead, Git thinks of its data more like a series of snapshots of a miniature filesystem.
  - Every time you commit, or save the state of your project, Git basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot.
  - To be efficient, if files have not changed, Git doesn't store the file again, just a link to the previous identical file it has already stored.



Git – Most Important Aspects

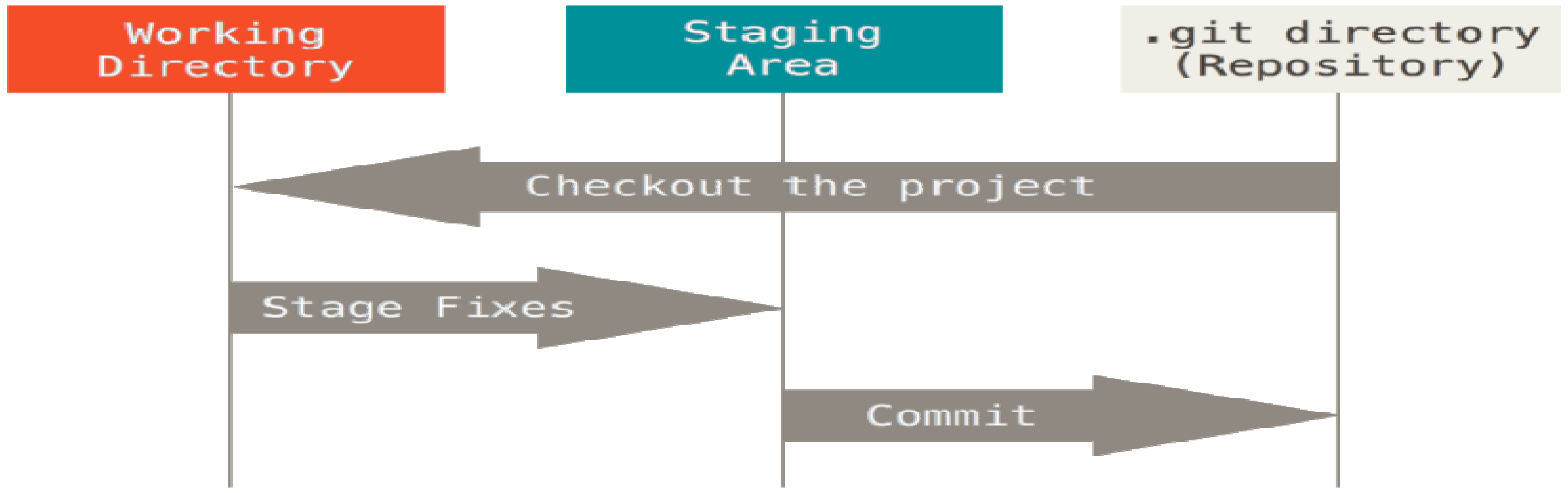
- Nearly Every Operation Is Local
  - Most operations in Git need only local files and resources to operate — generally no information is needed from another computer on your network.
  - There is very little you can't do if you're offline or off VPN.
- Git Has Integrity
  - Everything is checksummed before it is stored and is then referred to by that checksum.
- Git Generally Only Adds Data
  - Nearly all your actions in Git only add data to the Git database. After you commit a snapshot into Git, it is very difficult to lose, especially if you regularly push your database to another repository.

The Three States

- Git has three main states that your files can reside in:
  - **Modified** means that you have changed the file but have not committed it to your database yet.
  - **Staged** means that you have marked a modified file in its current version to go into your next commit snapshot.
  - **Committed** means that the data is safely stored in your local database.

Main Sections of a Git Project

- The **working tree** is a single checkout of one version of the project. These files are pulled out of the compressed database in the Git directory and placed on disk for you to use or modify.
- The **staging area** is a file, generally contained in your Git directory, that stores information about what will go into your next commit.
- The Git **directory** is where Git stores the metadata and object database for your project. This is the most important part of Git, and it is what is copied when you clone a repository from another computer.



The Basic Git Workflow

1. You modify files in your working tree.

2. You selectively stage just those changes you want to be part of your next commit, which adds only those changes to the staging area.

3. You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.

Basic Concepts - 1

- Repository
  - Think of a repository as a kind of database where your SCM stores all the versions and metadata that accumulate in the course of your project.
  - In Git, the repository is just a simple hidden folder named ".git" in the root directory of your project.
  - A "local" repository resides on your local computer, as a ".git" folder inside your project's root folder. You are the only person that can work with this repository, by committing changes to it.
  - A "remote" repository, in contrast, is typically located on a remote server on the internet or in your local network. Teams are using remote repositories to share & exchange data: they serve as a common base where everybody can publish their own changes and receive changes from their teammates.

- Commit
  - A commit is a wrapper for a specific set of changes. The author of a commit has to comment what he did in a short "commit message". This helps other people (and himself) to understand later what his intention was when making these changes.
  - Every set of changes implicitly creates a new, different version of your project. Therefore, every commit also marks a specific version. It's a snapshot of your complete project at that certain point in time. The commit knows exactly how all of your files and directories looked and can therefore be used, e.g., to restore the project to that certain state.

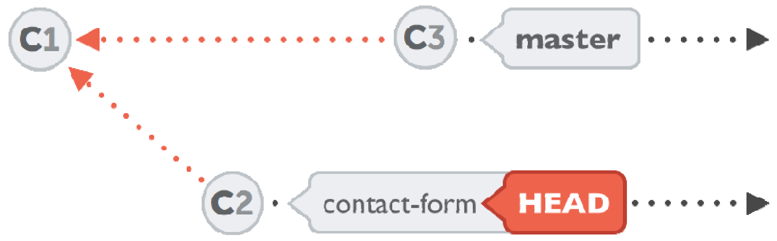
Basic Concepts - 2

- The Commit Hash
  - Every commit has a unique identifier: a 40-character checksum called the "commit hash". While in centralized version control systems like Subversion or CVS, an ascending revision number is used for this, this is simply not possible anymore in a distributed system like Git: The reason is that, in Git, multiple people can work in parallel, committing their work offline, without being connected to a shared repository. In this scenario, you can't say anymore whose commit is #5 and whose is #6.
  - Since in most projects, the first 7 characters of the hash are enough for it to be unique, referring to a commit using a shortened version is very common.

- The Stash
  - Think of the Stash as a clipboard on steroids: it takes all the changes in your working copy and saves them for you on a new clipboard. You're left with a clean working copy, i.e. you have no more local changes.
  - Later, at any time, you can restore the changes from that clipboard in your working copy - and continue working where you left off.
  - You can create as many Stashes as you want - you're not limited to storing only one set of changes. Also, a Stash is not bound to the branch where you created it: when you restore it, the changes will be applied to your current HEAD branch, whichever this may be.

Basic Concepts - 3

- Checkout, HEAD, and Your Working Copy
  - A branch automatically points to the latest commit in that context. And since a commit references a certain version of your project, Git always knows exactly which files belong to that branch.
  - At each point in time, only one branch can be HEAD / checked out / active. The files in your working copy are those that are associated with this exact branch. All other branches (and their associated files) are safely stored in Git's database.
  - To make another branch (say, "contact-form") active, the "git checkout" command is used. This does two things for you:
    - It makes "contact-form" the current HEAD branch.
    - It replaces the files in your working directory to match exactly the revision that "contact-form" is at.



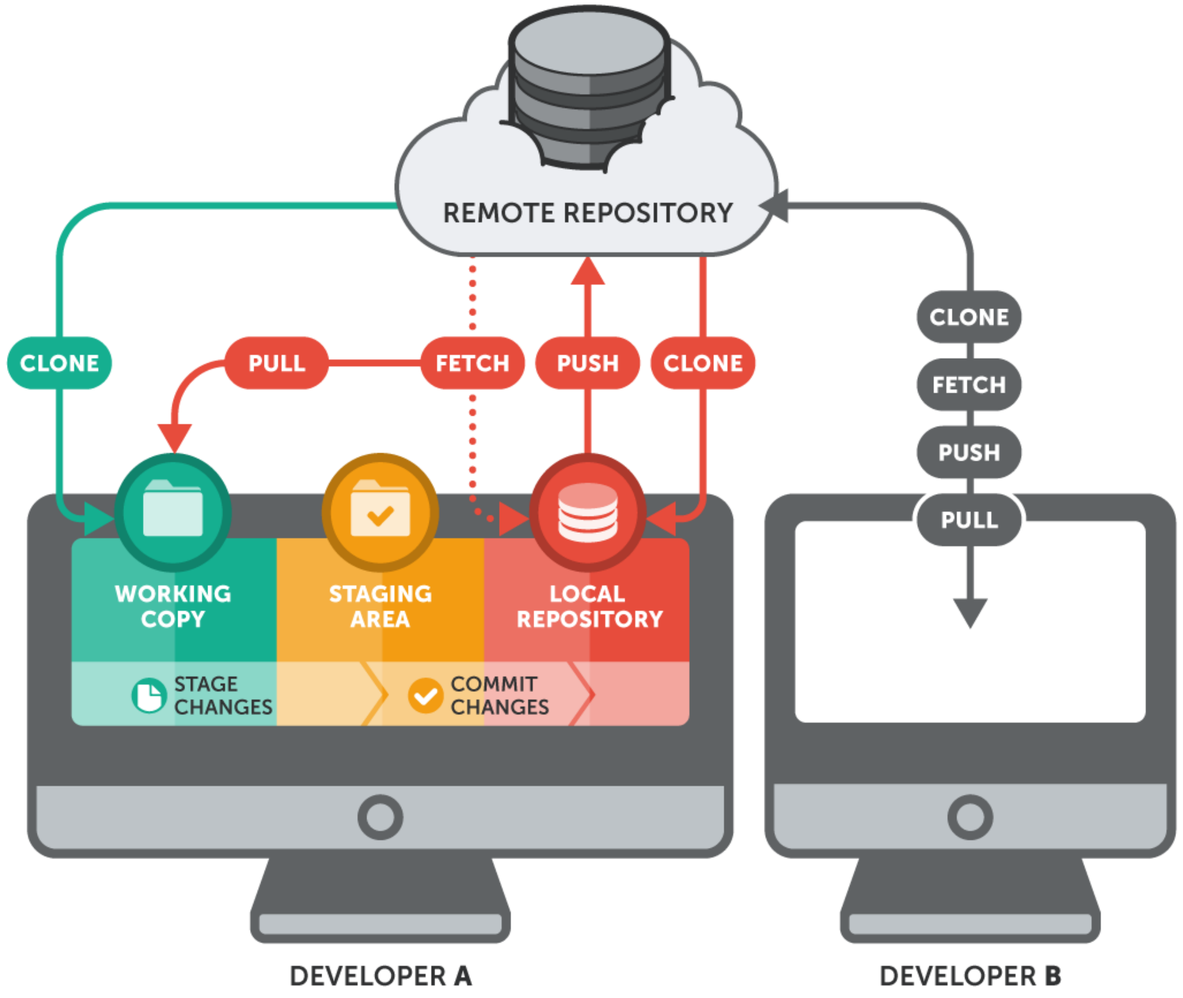
Working with Branches

- Branches aren't optional in Git: you are always working on a certain branch (the currently active, or "checked out", or "HEAD" branch).
- The "master" branch is created by Git automatically for us when we start the project. You can rename or delete it.

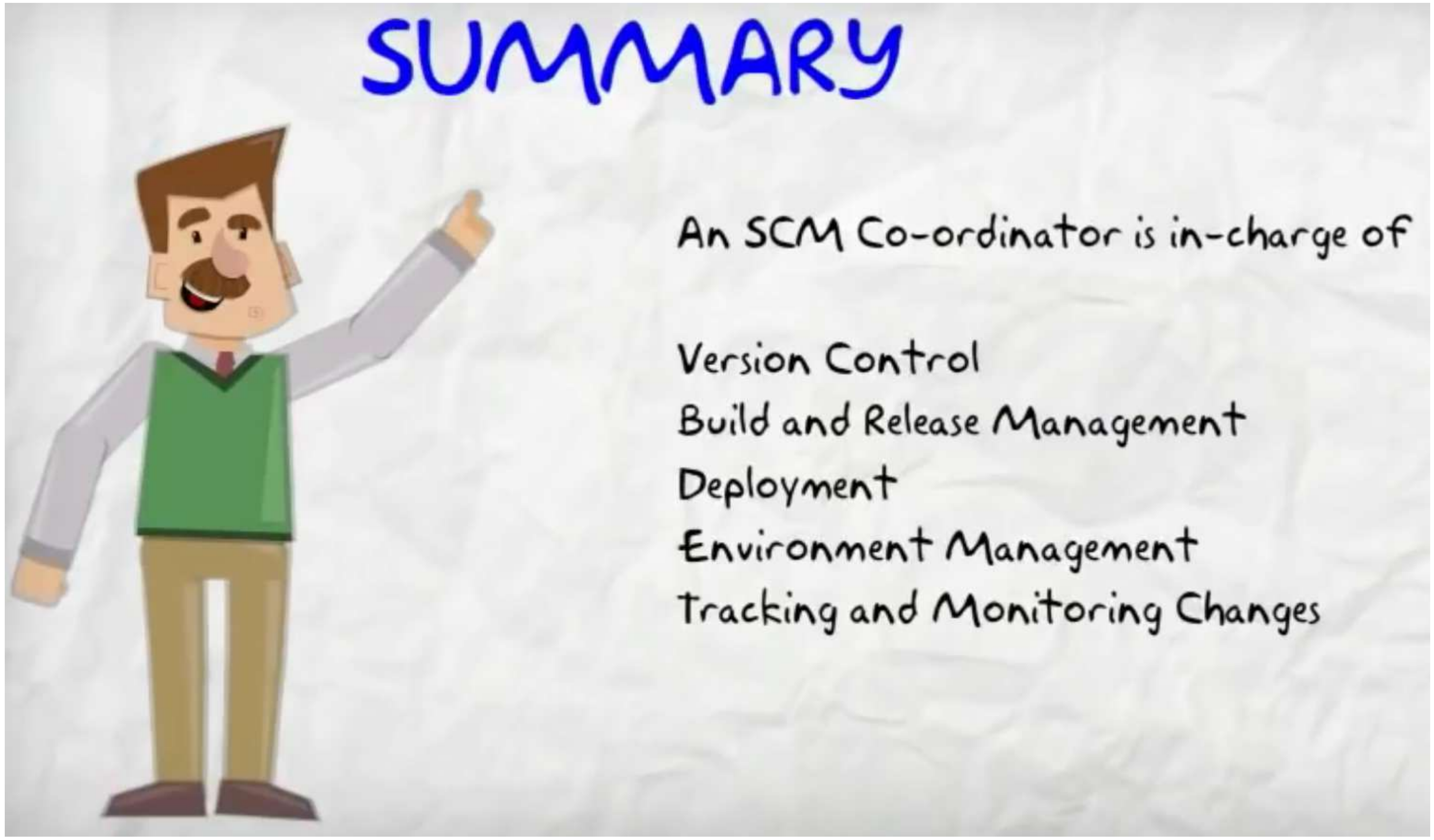
Integrating Branches

- When starting a merge, you don't have to (and cannot) pick individual commits that shall be integrated. Instead, you tell Git which branch you want to integrate - and Git will figure out which commits you don't have in your current working branch. Only these commits will then be integrated as a result.
- The target of such an integration is always your current HEAD branch and, thereby, your working copy.
- In some situations, merging will result in one or more "merge conflicts". In such a case, Git wasn't able to combine changes, e.g. because the exact same line was modified in two different ways. You'll then have to decide yourself which content you want.

Local / Remote Workflow



Software Configuration Management



SUMMARY

An SCM Co-ordinator is in-charge of

Version Control

Build and Release Management

Deployment

Environment Management

Tracking and Monitoring Changes

The Golden Rules of Version Control

- **Write Good Commit/Checkin Messages**
  - Time spent on crafting a good message is time spent well: it will make it easier to understand what happened for your teammates (and after some time also for yourself).
- **Use Branches Extensively**
  - Branches are the perfect tool to help you avoid mixing up different lines of development. You should use branches extensively in your development workflows: for new features, bug fixes, experiments, ideas…
- **Never Commit/Checkin Half-Done Work**
  - You should only commit code when it's completed. This doesn't mean you have to complete a whole, large feature before committing. Quite the contrary: split the feature's implementation into logical chunks and remember to commit early and often. But don't commit just to get half-done work out of your way when you need a "clean working copy".

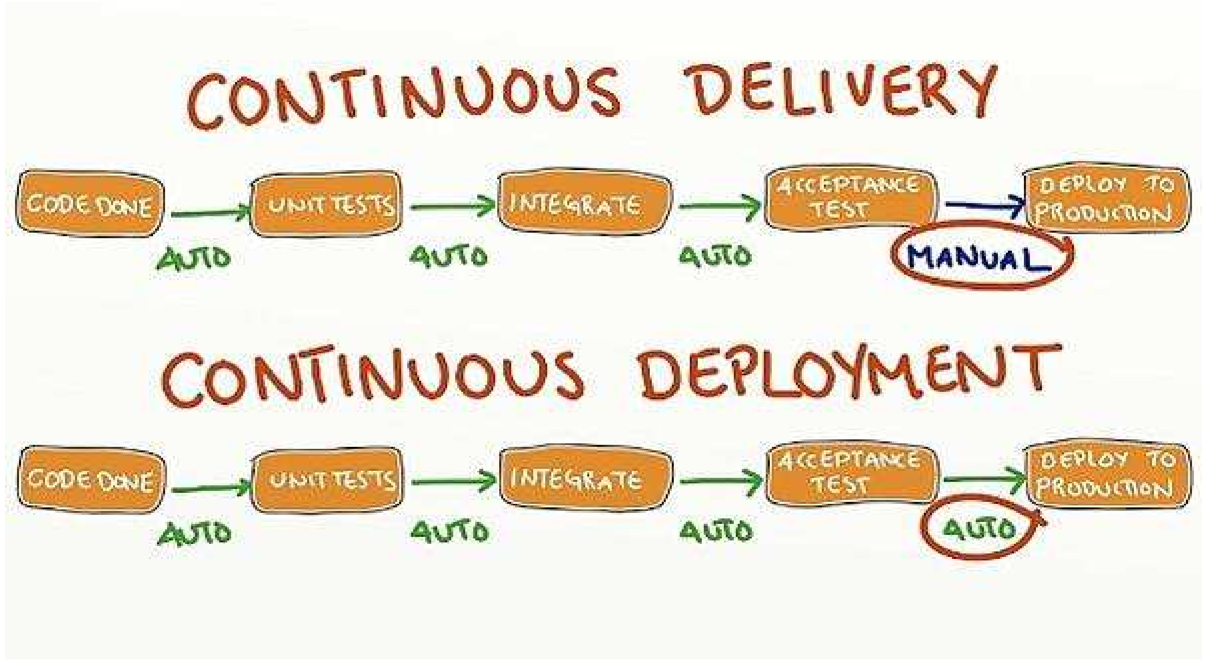
Continuous Integration

- Continuous Integration (CI) is a development practice where developers integrate code into a shared repository frequently, preferably several times a day.
- Each integration can then be verified by an automated build and automated tests. While automated testing is not strictly part of CI it is typically implied.
- One of the key benefits of integrating regularly is that you can detect errors quickly and locate them more easily. As each change introduced is typically small, pinpointing the specific change that introduced a defect can be done quickly.
- In recent years CI has become a best practice for software development and is guided by a set of key principles. Among them are revision control, build automation and automated testing.

Continuous Integration doesn't get rid of bugs, but it does make them dramatically easier to find and remove.

Continuous Deployment

- Continuous Deployment is closely related to Continuous Integration and refers to keeping your application deployable at any point or even automatically releasing to a test or production environment if the latest version passes all automated tests.



Continuous Delivery

- Continuous Delivery is the practice of keeping your codebase deployable at any point.

- Beyond making sure your application passes automated tests it has to have all the configuration necessary to push it into production. Many teams then do push changes that pass the automated tests into a test or production environment immediately to ensure a fast development loop.

Benefits of CI

- A good CI setup speeds up your workflow and encourages the team to push every change without being afraid of breaking anything.

- **Reduces Risk**
  - If you test and deploy code more frequently, it will eventually reduce the risk level of the project you are working on as you can detect bugs and code defects earlier. Defects are easier to fix and you can fix them sooner which makes it cheaper to fix them.
- **Better Communication**
  - With a CI process in place that is hooked into a Continuous Delivery workflow it's easy to share your code regularly. This code sharing helps to achieve more visibility and collaboration between team members. Eventually this increases communication speed and efficiency within your organization as everybody is on the same page, always.
- **Faster iterations**
  - As you release code often, the gap between the application in production and the one the developer is working on will be much smaller. As every small change will be tested automatically and the whole team can know about these changes you will want to work on small, incremental changes when developing new features. This results in less assumptions as you can build features quicker and test and deploy them automatically for your users to see as soon as possible, thus gaining valuable feedback from them faster. → Agile thinking
- **Faster feedback on business decisions**
  - As a software development manager, you have more data available which you can analyze to check if the product is heading into the right direction. This continuous data flow and the timeline of metrics (like dependency, unit tests, complexity, and code smell) can also help to reflect on the progress of the project more frequently which enables faster technological and business decisions.

Some other benefits of using CI and CD

- Reduces overhead across the development and deployment process
- Reduces the time and effort for integrations of different code changes
- Enables a quick feedback mechanism on every change
- Allows earlier detection and prevention of defects
- Helps collaboration between team members so recent code is always shared
- Reduces manual testing effort
- Building features more incrementally saves time on the debugging side so you can focus on adding features
- First step into fully automating the whole release process
- Prevents divergence in different branches as they are integrated regularly
- If you have a long running feature you're working on, you can continuously integrate but hold back the release with feature flags.

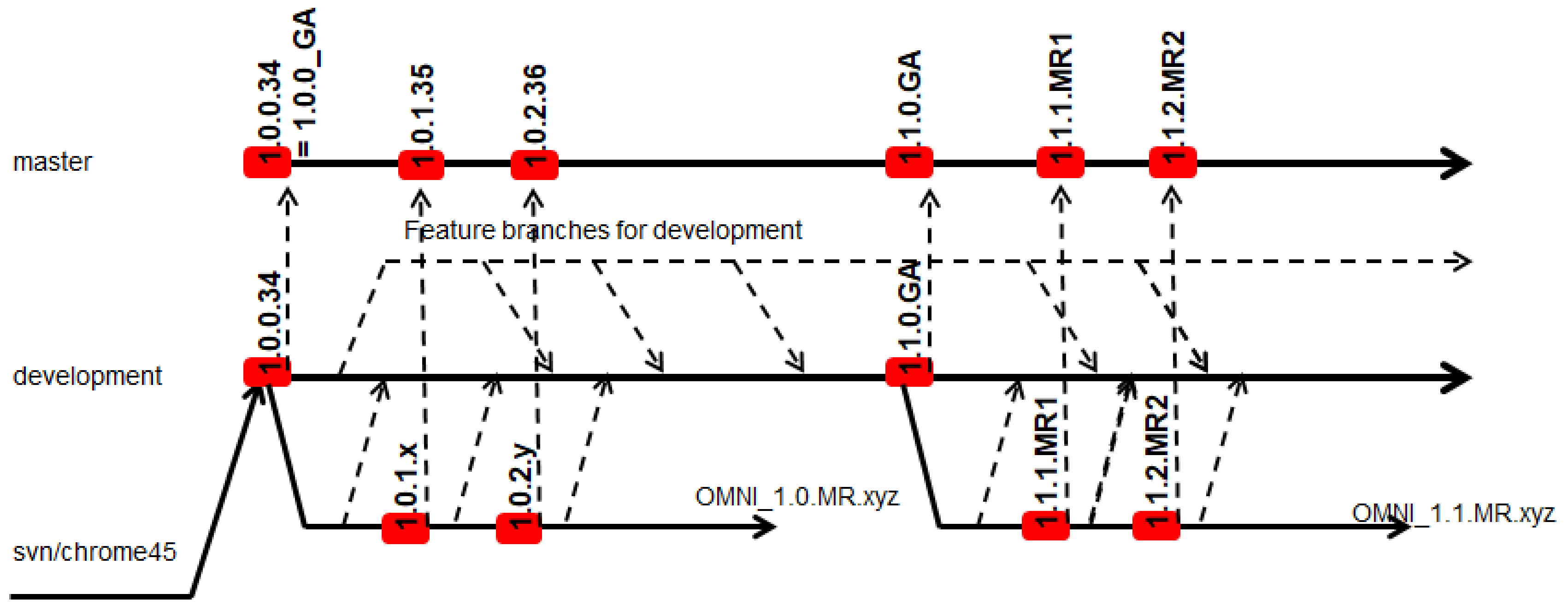
Continuous Delivery Checklist

1. Before submitting changes, check to see if a build is currently in the "Successful" status. If not, you should assist in fixing a build before submitting new code.
2. If the status is currently "Successful", you should rebase your personal workspace to this configuration.
3. Build and test locally to ensure the update doesn't break functionality.
4. If Successful, check in new code.
5. Allow CI to complete with new changes.
6. If build fails, stop and fix on your machine. Return to step 3.
7. If build passes, continue to work on the next item.

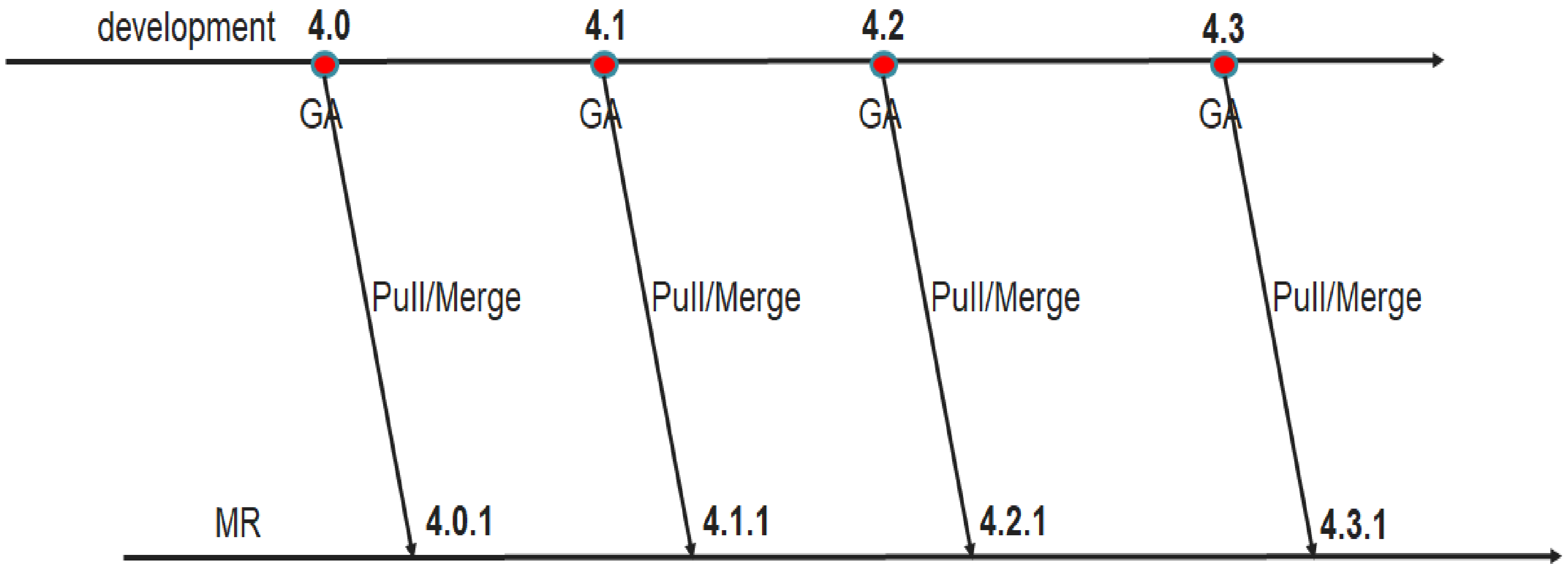
Key Tools & Best Practices

- Jenkins for automated builds
- Automated testsuites for the technologies you are using, for example Rspec and Jasmine.
- Version control systems like Git, Subversion or Perforce.
- Desktop Subversion and Git Clients like Cornerstone
- Code reviews like Github pull requests or Assembla merge requests, etc.
- Deployment & staging setup scripts like Capistrano, Shipit, Fabric, etc.
- Integrations with collaboration tools you're already using like Slack so you can deploy via commands directly in your channel.

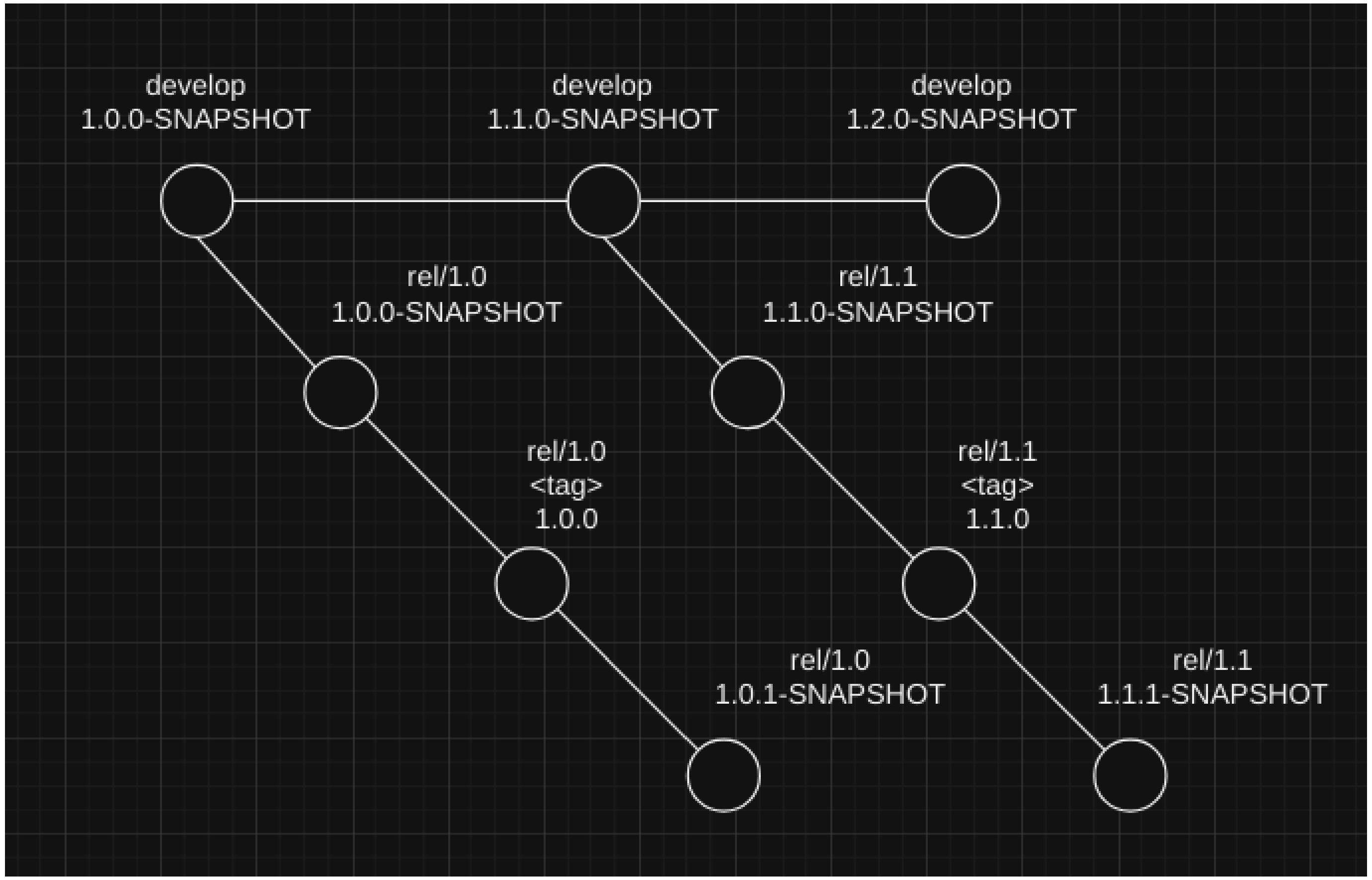
E.g. Branch Strategy for Releases - 1



E.g. Branch Strategy for Releases - 2



E.g. Branch Strategy for Releases - 3



Summary

- What is Software Configuration Management?
- Source control (or version control) is the practice of tracking and managing changes to code.
- **SCM** systems provide a running history of code development and help to resolve conflicts when merging contributions from multiple sources. SCM systems allow you to track your code change, see a revision history for your code, and revert to previous versions of a project when needed.
- An effective SCM system will make it easy to **reconstruct the software system at any point in the past**.
- Centralized vs. Distributed version control
- **Merging** is the action of combining the contents of two or more files or directories into a single new file or directory.
- A **branch** is an object that specifies a linear sequence of versions of an element.
- You need to update your development work area with the recommended baseline as soon as it becomes available.
- ClearCase
  - Check out-edit-check in model
  - ClearCase Versions, Elements and VOBs
  - Versions and Config Spec
- Git - data is more like a series of snapshots of a miniature filesystem
  - Three states: **Modified, Staged, Committed**
  - Commits, commit hash, repository, stash
  - Local / Remote Workflow: push, pull
- The Golden Rules of Version Control
- Continuous Integration (CI) is a development practice where developers integrate code into a shared repository frequently, preferably several times a day.
- Continuous Deployment refers to keeping your application deployable at any point automatically.
- Continuous Delivery is the practice of keeping your codebase deployable at any point.

References

- https://www.slideserve.com/carys/configuration-management
- https://www.youtube.com/watch?v=AaHaLjuzUm8