SCM End Term Study Material

**UNIT IV: DVCS Extended**

**1. Introduction to Distributed Version Control Systems (DVCS)**

1. **Definition**
   * A **Distributed Version Control System** (DVCS) is a type of version control where every contributor has a full copy of the repository (including its entire history).
   * **Examples**: Git, Mercurial, Bazaar.
2. **Key Advantages**
   * **Easier Merging**: Distributed systems handle branching and merging more efficiently due to local operations.
   * **Scalability**: Each user’s local repository reduces load on a central server.
   * **Offline Work**: Developers can commit, branch, and explore history without network connectivity.
   * **Speed**: Most operations are local (e.g., viewing history, comparing commits).
3. **Core Concepts**
   * **Local Repository**: Each user has a full clone of the repository, including all commits, branches, and tags.
   * **Remote Repository**: A shared repository (e.g., on a server or GitHub) that developers push and pull from.
   * **Synchronization**: DVCS workflows rely on “push” (sending changes) and “pull/fetch” (retrieving changes) operations between local and remote repositories.

**2. Easier Merging in DVCS**

1. **Why Merging is Easier**
   * **Local Branches**: In DVCS, creating and switching branches is fast and cheap. Developers can isolate changes and merge them frequently.
   * **Three-Way Merge**: Tools like Git automatically detect common ancestors, reducing merge conflicts.
   * **Frequent Commits**: Encourages smaller, more frequent merges, which lowers the complexity of each merge.
2. **Merge Conflict Handling**
   * **Conflict Markers**: When a file is modified in parallel by multiple contributors, Git will add markers (e.g., <<<<<<< HEAD) to highlight conflicting lines.
   * **Resolution Strategies**: Manually edit conflicts, use a merge tool (e.g., git mergetool), or decide to accept one version over the other.
3. **Best Practices for Merging**
   * **Commit Early and Often**: Frequent commits and merges reduce large conflict sets.
   * **Rebase vs. Merge**: Rebasing can create a linear history, while merging preserves the full branching structure.
   * **Pull Requests**: Use code reviews and pull requests (in platforms like GitHub) to ensure merges are vetted.

**3. Scaling Horizontally**

1. **Definition of Horizontal Scaling**
   * In a DVCS, horizontal scaling refers to the ability for multiple repositories to exist and interact without a single central bottleneck.
2. **Advantages**
   * **Distributed Collaboration**: Teams can maintain separate repositories for different components or sub-projects.
   * **Redundancy**: Every clone is a full backup of the repository history, reducing the risk of data loss.
   * **Flexible Workflows**: Large projects can be broken into smaller modules, each with its own repository.
3. **Implementation Approaches**
   * **Monorepo**: A single repository containing multiple projects or components. Often used with specialized tooling to manage subprojects.
   * **Polyrepo**: Each service or module resides in its own repository, which can scale independently.

**4. Multiple Repositories Model**

1. **Single vs. Multiple Repositories**
   * **Single Repository**: Central place for all code. Simple to manage but can become large and complex.
   * **Multiple Repositories**: Each project or service in its own repository. Allows independent versioning and modular development.
2. **When to Use Multiple Repositories**
   * **Microservices**: Each service can have its own lifecycle and dependencies.
   * **Different Technologies**: Separate repositories for front-end, back-end, mobile apps, etc.
   * **Team Autonomy**: Allows different teams to manage their release cycles independently.
3. **Challenges**
   * **Dependency Management**: Coordinating changes across repositories can be tricky if services are tightly coupled.
   * **Consistent Standards**: Ensuring consistent coding guidelines and workflows across multiple repos.
   * **Version Synchronization**: Need to ensure that interdependent repositories are updated in lockstep when necessary.

**5. Multiple Repositories for Different Services**

1. **Microservices Architecture**
   * **Definition**: Breaking an application into small, independently deployable services.
   * **Repository Strategy**: Typically each microservice resides in its own repository, enabling separate versioning and deployment.
2. **Pros and Cons**
   * **Pros**:
     + Independent deployment.
     + Smaller, more focused codebases.
     + Clear ownership and boundaries.
   * **Cons**:
     + More overhead in managing multiple repos.
     + Potential for version drift if services have interdependencies.
3. **Example Workflow**
   * **Service Repos**: service-auth, service-payment, service-inventory.
   * **Coordination**: Use a continuous integration pipeline that triggers tests across all services if an interface changes.
   * **Version Tags**: Each repo can have its own semantic version (e.g., v1.2.3).

**6. Resetting the Local Environment**

1. **Why Reset?**
   * Developers may want to discard local changes, revert to a known good state, or fix a broken workspace.
   * Common scenario: A developer pulls the latest changes but encounters conflicts or corrupted files.
2. **Methods to Reset**
   * **Discarding Changes**: In Git, git checkout -- <file> discards uncommitted changes in a file.
   * **Hard Reset**: git reset --hard <commit> moves the HEAD pointer to a specific commit, discarding all local changes.
   * **Clean Command**: git clean -df removes untracked files or directories, restoring a pristine workspace.
3. **Caution**
   * **Data Loss**: A hard reset or cleaning can permanently delete changes if they are not committed.
   * **Safe Approach**: Stash or commit changes before resetting if you might need them later.

**7. Revert and Canceling Out Changes**

1. **Revert**
   * **Definition**: Creates a new commit that undoes the changes of a previous commit without altering the commit history.
   * **Usage**: git revert <commit>
   * **Benefits**: Safe way to undo a commit in a shared repository because history remains intact.
2. **Difference from Reset**
   * **Reset**: Moves the branch pointer to an older commit, potentially discarding commits.
   * **Revert**: Appends a new commit that reverses the effect of a previous commit, preserving the existing history.
3. **Canceling Out Changes**
   * **Partial Revert**: You can revert a specific file or hunk from a commit by manually editing or using interactive tools.
   * **Best Practice**: Revert changes in a new branch and merge that branch into the main line if you want to preserve code review or CI checks.

**UNIT V: Git**

**1. Introduction to Git**

1. **Definition**
   * **Git** is a **distributed version control system** originally created by Linus Torvalds for Linux kernel development. It tracks changes to files and coordinates work among multiple developers.
2. **Data Structure Used in Git**
   * **Snapshots, Not Diffs**: Unlike older VCS tools (e.g., SVN), Git stores snapshots of the entire project at each commit.
   * **Directed Acyclic Graph (DAG)**: Commits form a DAG, where each commit references one or more parent commits.
   * **Objects**: The four main object types in Git are **blob**, **tree**, **commit**, and **tag**.
3. **Core Features**
   * **Branching and Merging**: Extremely cheap and fast due to Git’s lightweight branching model.
   * **Staging Area**: A unique concept in Git where changes are first “staged” before being committed.
   * **Local Operations**: Most operations are performed locally, making Git very fast.

**2. Git Commands with Proper Explanation**

**2.1 git add**

1. **Purpose**
   * Moves changes from the working directory into the staging area (also called the “index”).
   * **Syntax**: git add <file> or git add . (adds all changes in the current directory).
2. **How It Works**
   * **Staging**: Allows you to selectively include changes in the next commit.
   * **Best Practice**: git add -p for an interactive mode to review chunks of changes before staging.

**2.2 git commit**

1. **Purpose**
   * Creates a new commit object from the staged changes.
   * **Syntax**: git commit -m "Commit message".
2. **Commit Message Best Practices**
   * **Short Summary**: 50 characters or fewer.
   * **Detailed Description**: If needed, add a blank line, then provide more context.
   * **Reference Issues or Tickets**: E.g., #123 for cross-referencing bug trackers.
3. **Amending Commits**
   * git commit --amend modifies the most recent commit.
   * **Caution**: Avoid amending commits that have already been pushed to a shared repository.

**2.3 git reset**

1. **Definition**
   * Moves the current branch pointer to a specific commit, optionally modifying the staging area and working directory.
2. **Types of Reset**
   * **--soft**: Keeps changes in the staging area.
   * **--mixed** (default): Keeps changes in the working directory but clears the staging area.
   * **--hard**: Discards changes in both the staging area and the working directory.
3. **Use Cases**
   * **Soft Reset**: If you committed too early and want to re-stage changes for a better commit message.
   * **Hard Reset**: Discard all local changes, returning to a known commit.

**2.4 git revert**

1. **Definition**
   * Creates a new commit that undoes the changes introduced by a specific commit.
2. **Usage**
   * git revert <commit-hash>
   * **Scenario**: You discover a bug introduced in commit abc123; revert that commit to remove the bug while preserving history.
3. **Advantages**
   * **Preserves History**: Everyone can see the revert commit, which clarifies the timeline of changes.
   * **Safe for Shared Repos**: Avoids rewriting commits that others may have based work on.

**2.5 git stash**

1. **Definition**
   * Temporarily saves uncommitted changes in a “stash” so you can switch branches or pull changes without losing work.
2. **Usage**
   * **Create Stash**: git stash or git stash push -m "message".
   * **Apply Stash**: git stash apply (leaves stash in place) or git stash pop (removes stash after applying).
3. **Common Use Case**
   * You’re in the middle of a feature, but need to switch to a different branch to fix a quick bug. Stash your work, switch branches, then return and apply the stash.

**2.6 git config**

1. **Definition**
   * Configures user information, editor preferences, merge tools, and more.
2. **Scope**
   * **System**: Affects all users on the system (usually /etc/gitconfig).
   * **Global**: Affects the current user (usually ~/.gitconfig).
   * **Local**: Affects a specific repository (.git/config in the repo).
3. **Examples**
   * git config --global user.name "Your Name"
   * git config --global user.email "you@example.com"
   * git config --global core.editor "vim"

**2.7 git diff**

1. **Definition**
   * Shows changes between commits, commit and working tree, etc.
2. **Usage Examples**
   * **Compare Staging and Working Directory**: git diff
   * **Compare Staging and Last Commit**: git diff --staged
   * **Compare Two Commits**: git diff <commit1> <commit2>
3. **Helpful Options**
   * --stat for a summary of changed files and lines.
   * --color for colorized output (often enabled by default).

**2.8 git rebase**

1. **Definition**
   * Re-applies commits on top of another base tip, creating a linear history.
2. **Usage**
   * **Simple Rebase**: git rebase <branch> moves your current branch commits on top of <branch>.
   * **Interactive Rebase**: git rebase -i <commit> allows you to edit, squash, reorder, or drop commits.
3. **Advantages & Warnings**
   * **Advantages**: Cleaner, linear commit history, easier to read logs.
   * **Warnings**: Do not rebase commits that have been pushed to a shared repository; it rewrites history.

**2.9 git log**

1. **Definition**
   * Shows the commit history for the repository or a given branch.
2. **Common Options**
   * --oneline: Shows each commit in a single line (abbreviated commit hash + message).
   * --graph: Visual ASCII graph of the branch structure.
   * --pretty=format:"...": Customize output format, e.g., showing only commit hash and author.
3. **Filtering**
   * git log --author="Alice" to see commits by a specific author.
   * git log --since="2 weeks ago" to see recent changes.

**2.10 git push, git pull, git fetch**

1. **git push**
   * **Definition**: Sends local commits to a remote repository.
   * **Syntax**: git push <remote> <branch>
   * **Example**: git push origin main pushes the local main branch to the remote named origin.
2. **git pull**
   * **Definition**: Fetches changes from the remote and merges them into the current branch.
   * **Equivalent**: git fetch + git merge in a single step.
   * **Conflict Potential**: If local changes conflict with remote changes, a merge conflict may occur.
3. **git fetch**
   * **Definition**: Retrieves commits, files, and refs from a remote repository without merging them.
   * **Use Case**: Inspect remote changes before merging: git fetch origin then git log origin/main.

**2.11 git merge**

1. **Definition**
   * Combines changes from one branch into another, creating a merge commit if necessary.
2. **Fast-Forward vs. Merge Commit**
   * **Fast-Forward**: If the current branch is behind another branch in a linear fashion, Git simply moves the pointer forward.
   * **Merge Commit**: If branches have diverged, Git creates a commit that merges both histories.
3. **Example**
   * git checkout main
   * git merge feature-branch merges the feature branch into main.

**2.12 git branch, git checkout**

1. **git branch**
   * **Definition**: Lists, creates, or deletes branches.
   * **Examples**:
     + git branch: lists local branches.
     + git branch feature-xyz: creates a new branch called feature-xyz.
     + git branch -d feature-xyz: deletes a merged branch.
2. **git checkout**
   * **Definition**: Switches the working directory to a different branch or commit.
   * **Examples**:
     + git checkout feature-xyz: moves HEAD to feature-xyz.
     + git checkout <commit-hash>: checks out a specific commit in a “detached HEAD” state.
3. **git switch**
   * **Modern Alternative**: In newer Git versions, git switch is recommended for switching branches.

**2.13 Commit ID, Hash Value Generation**

1. **SHA-1 Hash**
   * Git uses **SHA-1** (though transitioning to SHA-256 in some newer versions) to uniquely identify commits.
   * The hash is calculated based on the commit’s content (author, date, parent commits, tree, etc.).
2. **Uniqueness**
   * Each commit has a unique hash, ensuring that any change in content or metadata produces a different commit ID.
3. **Finding Commit Hash**
   * **Short Hash**: Usually the first 7-10 characters are shown in logs or references.
   * **Full Hash**: 40-character string representing the SHA-1 digest.

**UNIT VI: GitHub**

**1. Introduction to GitHub**

1. **Definition**
   * **GitHub** is a web-based hosting service for Git repositories, offering collaboration features like pull requests, issue tracking, and project management tools.
2. **Data Structure in GitHub**
   * **Underlying Technology**: GitHub uses Git as its backbone for storing and tracking repository data.
   * **Enhanced Features**: GitHub adds layers of collaboration, such as pull requests, code reviews, and integrated CI/CD with GitHub Actions.
3. **Key Differences from Git**
   * **Git**: A version control tool installed on your local machine.
   * **GitHub**: A platform (SaaS) that hosts Git repositories, provides a web UI, and offers additional services (e.g., issues, wikis, project boards).

**2. Difference Between Git and GitHub**

1. **Git**
   * **Type**: DVCS software running locally.
   * **Core Function**: Track changes, manage branches, commits, merges.
   * **Installation**: On developer machines or servers.
   * **Offline Capability**: Full local repository, can commit and branch without internet.
2. **GitHub**
   * **Type**: Cloud-based hosting service for Git repositories.
   * **Core Function**: Repository hosting, collaboration features (pull requests, issues, wikis).
   * **Accessibility**: Requires an internet connection to interact with hosted repos.
   * **Extended Ecosystem**: GitHub Actions (CI/CD), GitHub Pages, code scanning, and more.
3. **How They Work Together**
   * Developers use **Git** locally to track changes.
   * They **push** their repositories to **GitHub** for backup, collaboration, and review.
   * **GitHub** is effectively a remote repository host with extra features.

**3. Types of Repositories on GitHub**

1. **Public Repositories**
   * **Visibility**: Open to everyone on the internet.
   * **Use Cases**: Open-source projects, community collaboration.
   * **Forking**: Anyone can fork and contribute via pull requests.
2. **Private Repositories**
   * **Visibility**: Only accessible to invited collaborators.
   * **Use Cases**: Proprietary or confidential code, personal projects.
   * **Access Control**: Granular permissions (e.g., read, write, admin).
3. **Internal Repositories** (GitHub Enterprise)
   * **Visibility**: Accessible only to users within the same organization on GitHub Enterprise.
   * **Use Cases**: Company-wide code sharing without exposing to the public.

**4. Pull Requests on GitHub**

1. **Definition**
   * A **pull request** (PR) is a GitHub feature that allows developers to notify others about changes they’ve pushed to a branch in a repository.
   * Encourages discussion, code review, and continuous integration checks before merging.
2. **Workflow**
   * **Fork or Branch**: Developer creates a branch or fork for their changes.
   * **Commits**: Developer commits code to the new branch.
   * **Open PR**: They open a pull request to merge their branch into a target branch (e.g., main).
   * **Review & Discussion**: Reviewers can comment on specific lines, suggest changes, or approve the PR.
   * **CI Checks**: Automated tests, linting, or security scans run on the proposed changes.
   * **Merge**: Once approved and passing checks, the branch is merged into the base branch.
3. **Benefits**
   * **Code Quality**: Mandatory reviews improve quality and catch bugs early.
   * **Transparency**: Everyone can see changes, rationale, and discussion history.
   * **Documentation**: The PR itself acts as a record of why and how changes were made.

**5. Forks on GitHub**

1. **Definition**
   * A **fork** is a personal copy of another user’s repository on your GitHub account. You can freely experiment without affecting the original repo.
2. **Usage**
   * **Open-Source Contributions**: Fork an open-source project, make changes, then submit a pull request to the original repository.
   * **Private Customizations**: Keep your modifications separate from the main repository if you don’t intend to merge them back.
3. **Syncing a Fork**
   * **Upstream**: The original repository.
   * **Downstream**: Your fork.
   * **Workflow**: git remote add upstream <URL> allows you to fetch and merge changes from the original repo into your fork.

**6. Cloning a Repository from GitHub**

1. **Definition**
   * **Cloning** creates a local copy of a remote Git repository.
   * **Syntax**: git clone <repo-URL>
2. **HTTPS vs. SSH**
   * **HTTPS**: Simpler for public repos, but requires username/password for private repos.
   * **SSH**: Requires setting up SSH keys, often more convenient for frequent pushes/pulls.
3. **Typical Workflow**
   * **Clone**: git clone git@github.com:username/repo.git
   * **Develop Locally**: Make commits on your local machine.
   * **Push Changes**: git push origin <branch> to update the remote on GitHub.

**7. Best Practices and Workflows**

1. **Branching Strategy**
   * **Git Flow**: Branches for features, releases, hotfixes, etc.
   * **GitHub Flow**: Simple workflow with short-lived feature branches merging into main.
   * **Trunk-Based Development**: Single main branch with frequent merges.
2. **Commit Message Conventions**
   * **Short, Descriptive Titles**: Summarize changes in ~50 characters.
   * **Reference Issues**: E.g., “Fix #123: Correct the user login bug.”
   * **Detailed Body**: If needed, provide extra context in subsequent lines.
3. **Continuous Integration**
   * **GitHub Actions** or external CI systems (Jenkins, Travis, CircleCI) can run automated tests on each pull request.
   * **Automated Checks**: Linting, unit tests, security scans, coverage reports.
4. **Release Management**
   * **Tags**: Mark specific commits as versions (e.g., v1.0.0).
   * **Releases**: GitHub’s release feature can bundle release notes, binaries, or source archives.
   * **Changelogs**: Maintain a record of what’s new, changed, or fixed in each release.