LECTURE 7

Software requirements Engineering

Evolutionary requirements

Instructor:
Dr. Natalia Chaudhry,
Assistant Professor, PUCIT, University of the Punjab, Lahore.

Requirement evolution refers to the process of changes and adaptations in the requirements of a system or project over time.

Here are some key points to note about requirement evolution:

Dynamic Nature: Requirements for a system or project are rarely static. They can evolve due to various factors such as changing stakeholder needs, technological advancements, market trends, regulatory requirements, and feedback from users.

Iterative Process: Requirement evolution often occurs iteratively throughout the lifecycle of a project or system. As stakeholders gain a better understanding of the problem domain or as new information becomes available, requirements may need to be refined, expanded, or even modified significantly.

Feedback Loops: Feedback loops play a crucial role in requirement evolution. Stakeholder feedback, user testing, market research, and other forms of feedback provide valuable insights that drive changes in requirements.

Impact Analysis: Changes to requirements can have ripple effects throughout the project or system. Performing impact analysis helps assess the consequences of proposed changes on other requirements, project scope, schedule, budget, and resources.

Prioritization: Not all changes to requirements are equally important or feasible within a given timeframe. Prioritizing requirements based on their importance, value to stakeholders, and feasibility is essential for managing requirement evolution effectively.

Communication and Collaboration: Effective communication and collaboration among stakeholders, including customers, users, developers, testers, and project managers, are critical for managing requirement evolution successfully. Regular meetings, workshops, and other collaborative activities help ensure alignment and shared understanding.

Change Management: A structured change management process is essential for handling requirement evolution efficiently. This process typically involves capturing change requests, evaluating their impacts, obtaining approvals, implementing changes, and communicating updates to relevant stakeholders.

Today’s era

- Today's technology landscape demands agility and flexibility in requirements management.
- With the rise of agile and DevOps methodologies, requirements are constantly evolving based on user feedback, market trends, and technological advancements.
- Teams need to adapt quickly to changing requirements to stay competitive.
- Rapid advancements in technologies such as artificial intelligence, machine learning, Internet of Things (IoT), and blockchain are continuously reshaping requirements.
- Businesses are leveraging these technologies to innovate and deliver new capabilities, leading to the evolution of requirements to incorporate these cutting-edge features.

some DevOps tools that can help in requirements evolution

Issue Tracking and Project Management Tools: Tools like Jira enable teams to capture, track, and manage requirements effectively. They provide features for creating user stories, defining acceptance criteria, assigning tasks, and tracking progress. These tools facilitate collaboration among team members and stakeholders, allowing for the continuous evolution of requirements based on feedback and changing priorities.

Version Control Systems (VCS): Version control systems such as Git are essential for managing changes to requirements documents, code, and other project artifacts. By using branches, pull requests, and merge requests, teams can review and incorporate changes to requirements in a controlled manner. VCS also provide a history of changes, enabling traceability and accountability throughout the requirements evolution process.

Continuous Integration (CI) Tools: CI tools like Jenkins automate the process of building, testing, and integrating code changes. While CI primarily focuses on the development and testing aspects of the software delivery pipeline, it can indirectly contribute to requirements evolution by providing rapid feedback on the implementation of new features or changes. This feedback loop helps teams iterate on requirements and address any issues early in the development cycle.

Continuous Deployment (CD) Tools: CD tools such as Ansible, Puppet automate the deployment and configuration of applications and infrastructure. While CD primarily focuses on streamlining the deployment process, it can also facilitate requirements evolution by enabling teams to deploy changes to production quickly and reliably. This allows for faster feedback from users and stakeholders, which can inform further iterations of requirements.

Collaboration Platforms: Collaboration platforms like Slack, Microsoft Teams promote communication and knowledge sharing among team members and stakeholders. These platforms provide real-time messaging, file sharing, and integration with other tools, making it easier for teams to discuss requirements, share updates, and solicit feedback. By fostering open communication, collaboration platforms support the continuous evolution of requirements in a transparent and inclusive manner

GiT

- Git is a version control system.
- It helps you keep track of code changes.
- GIT is an acronym for Global Information Tracker
- With Git, 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 thinks about its data more like a stream of snapshots.

Git Has Integrity

- Everything in Git is checksummed before it is stored and is then referred to by that checksum.
- You can’t lose information in transit or get file corruption without Git being able to detect it.
- The mechanism that Git uses for this checksumming is called a SHA-1 hash. This is a 40-character string composed of hexadecimal characters (0–9 and a–f) and calculated based on the contents of a file or directory structure in Git.
- A SHA-1 hash looks something like this:
  24b9da6552252987aa493b52f8696cd6d3b00373

What is a git repository?

A repository is a file structure where git stores all the project-based files. Git can either stores the files on the local or the remote repository

GITHUB has a wide range of features. These features expand the functions of GIT. The user interface of GITHUB allows users to do everything that GIT does without having to write the git commands. It makes communicating with collaborators or developers easier.

Although GIT and GITHUB are capable of running independently, they both go hand in hand. Using GITHUB looks easy for a non-programmer but as a programmer, learning GIT commands is necessary.

Learning both GIT and GITHUB is always encouraged for a developer.

What does Git do?

- Manage projects with **Repositories**
- **Clone** a project to work on a local copy
- Control and track changes with **Staging** and **Committing**
- **Branch** and **Merge** to allow for work on different parts and versions of a project
- **Pull** the latest version of the project to a local copy
- **Push** local updates to the main project

What does git clone do?

The command creates a copy (or clone) of an existing git repository. Generally, it is used to get a copy of the remote repository to the local repository.

Head

Head is the reference to the last commit object of the branch

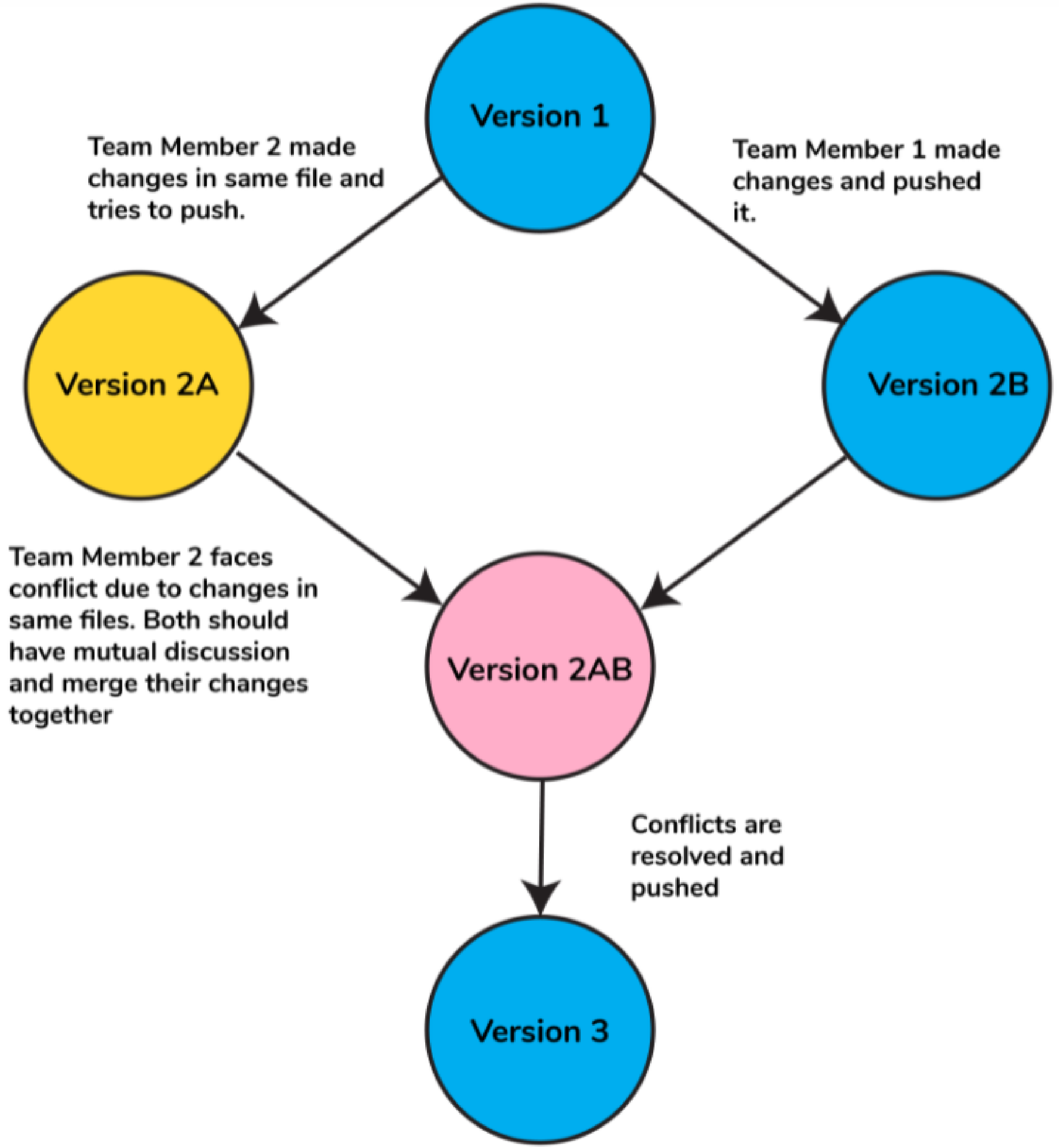
There is always a default head referred to as master or main

What is a conflict?

Sometimes while working in a team environment, there might be cases of conflicts such as:

1. When two separate branches have made changes to the same line in a file
2. A file is deleted in one branch but has been modified in the other.

These conflicts have to be solved manually after discussion with the team as git will not be able to predict what and whose changes have to be given precedence.

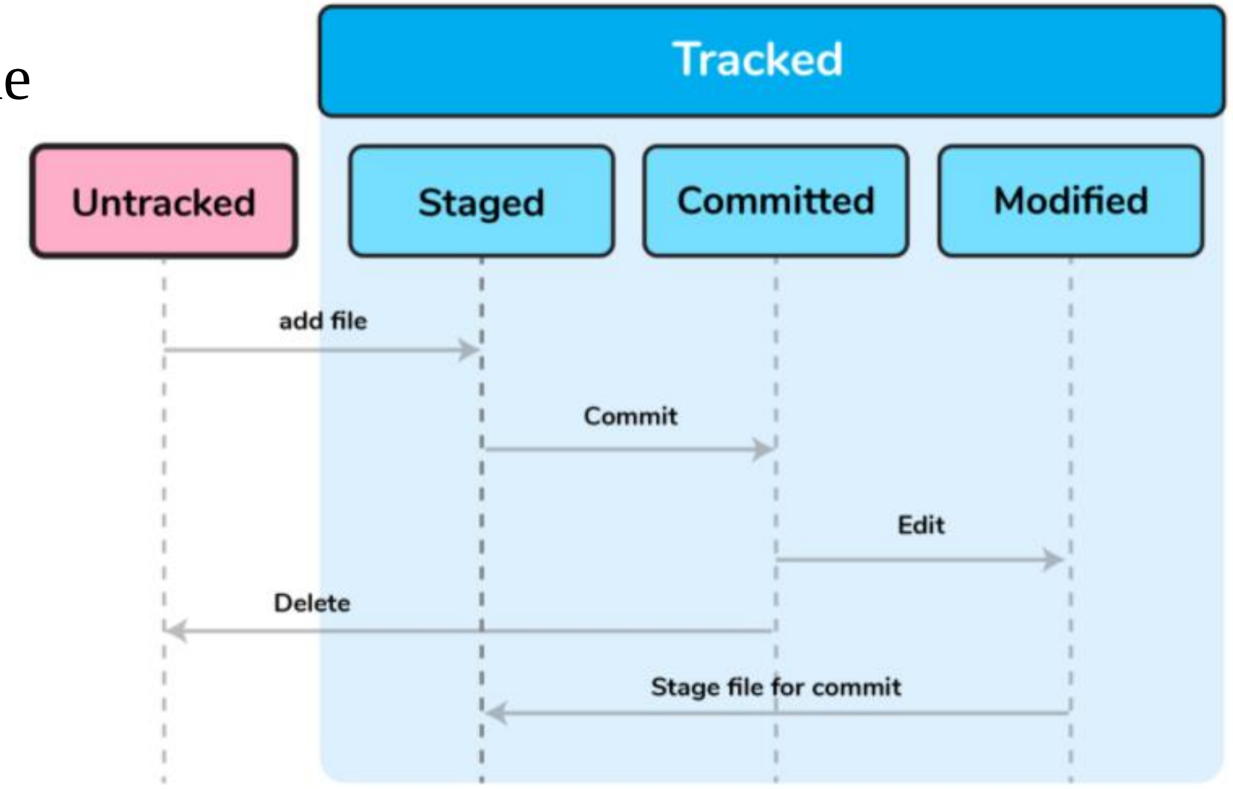

Version 1
Team Member 2 made changes in same file and tries to push.
Team Member 1 made changes and pushed it.
Version 2A
Version 2B
Team Member 2 faces conflict due to changes in same files. Both should have mutual discussion and merge their changes together
Version 2AB
Conflicts are resolved and pushed
Version 3

Before making commits to the changes done, the developer is given provision to format and review the files and make innovations to them. All these are done in the common area which is known as **'Index'** or **'Staging Area'.**

- **Staging area** gives the control to make commit smaller. Just make one logical change in the code, add the changed files to the staging area and finally if the changes are bad then checkout to the previous commit or otherwise commit the changes.

- It gives the flexibility to split the task into smaller tasks and commit smaller changes. With staging area it is easier to focus on small tasks.

- Git has three main states that your files can reside in: modified, staged, and committed:
- **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.



The basic Git workflow goes something like this:

- You modify files in your working tree.
- You selectively stage just those changes you want to be part of your next commit, which adds only those changes to the staging area.
- You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.

If a particular version of a file is in the Git directory, it’s considered **committed.** If it has been **modified** and was added to the staging area, it is **staged**. And if it was changed since it was **checked out** but has not been staged, it is modified.

Git Commit

- Since we have finished our work, we are ready move from stage to commit for our repo.
- Git considers each commit change point or "save point".
- It is a point in the project you can go back to if you find a bug, or want to make a change.
- When we commit, we should always include a message.

Contd..

- You can even switch between branches and work on different projects without them interfering with each other.
- **Branching in Git is very lightweight and fast!**
- You create branches to isolate your code changes, which you test before merging to the main branch
- Main branch is the first branch made when you initialize a Git repository using the **git init** command.

- Branching means you diverge from the main line of development and continue to do work without messing with that main line. In many VCS tools, this is a somewhat expensive process, often requiring you to create a new copy of your source code directory, which can take a long time for large projects.

- When you make a commit, Git stores a commit object that contains a pointer to the snapshot of the content you staged. This object also contains the author’s name and email address, the message that you typed, and pointers to the commit or commits that directly came before this commit (its parent or parents): zero parents for the initial commit, one parent for a normal commit, and multiple parents for a commit that results from a merge of two or more branches.

A branch in Git is simply a lightweight movable pointer to one of these commits. The default branch name in Git is master. As you start making commits, you’re given a master branch that points to the last commit you made. Every time you commit, the master branch pointer moves forward automatically.

Branching

- In Git, a branch is a new/separate version of the main repository.
- Let's say you have a large project, and you need to update the design on it.
- Branches allow you to work on different parts of a project without impacting the main branch.
- When the work is complete, a branch can be merged with the main project.

How will you create a git repository?

- Have git installed in your system.
- Then in order to create a git repository, create a folder for the project and then run git init.
- Doing this will create a .git file in the project folder which indicates that the repository has been created

A **branch** is nothing but a separate version of the code.

What consists of a commit object?

A commit object consists of set of files that represents the state of a project at a given point in time

Initialize Git

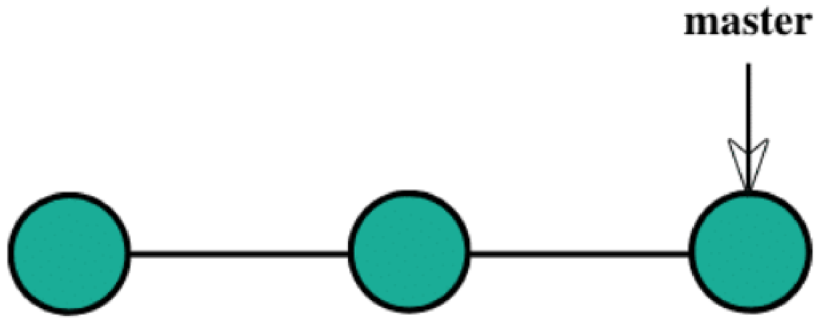
Once you have navigated to the correct folder, you can initialize Git on that folder:

```
git init
```

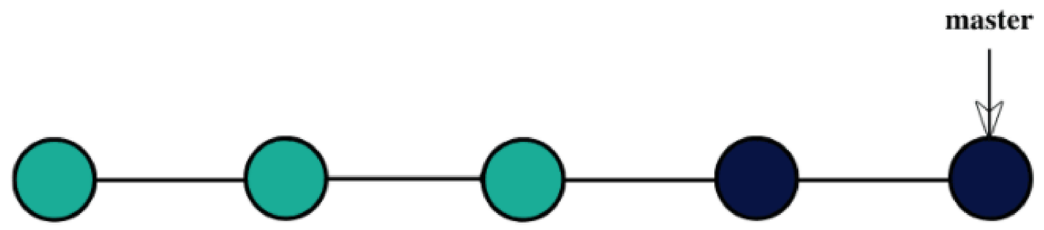
Consider that you are developing a project with your team, and you finish a feature. You contact the client to request them to see the feature, but they are too busy, so you send them the link to have a look at the project.

Project Development through linear development.

You have been working on a project with the client being happy until this point.

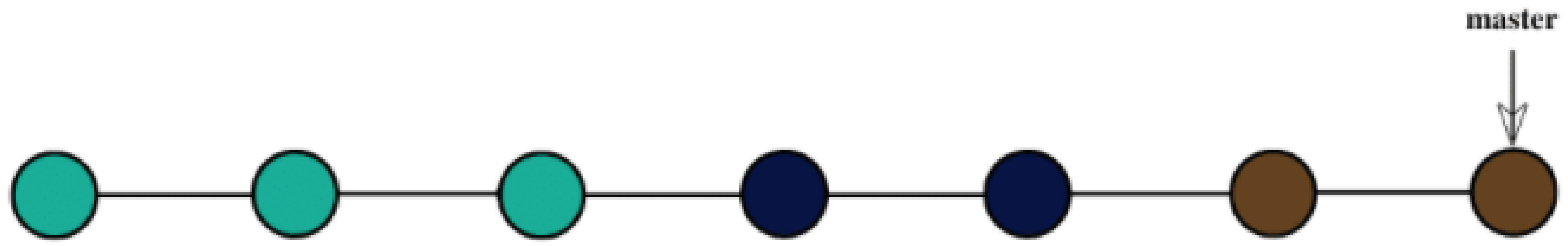


Now, you decide to develop a feature and start developing it on the same code (*denoted by black commits*).

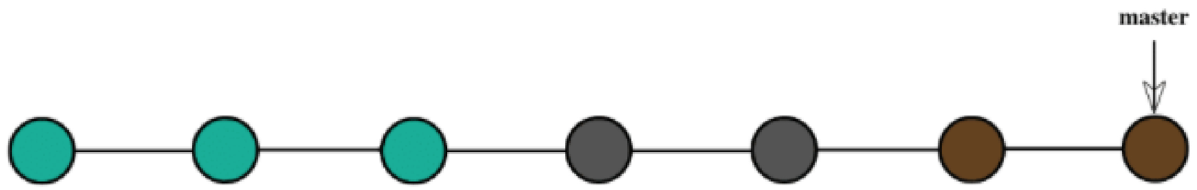


You show this to your client.

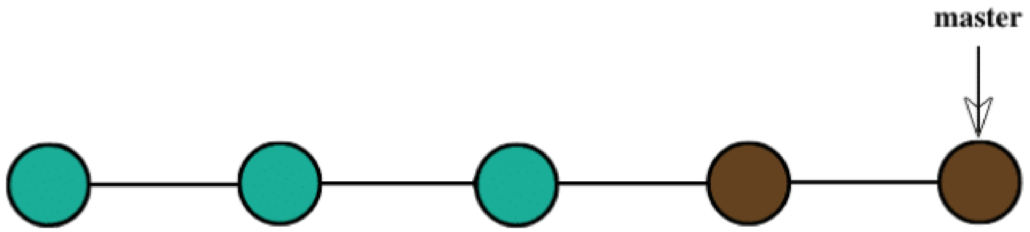
In the meantime, you decide to develop another feature (let say xyz) and wait for the client's approval (xyz denoted by brown commits).



The client disapproves of the feature (*black commits*) and requests to delete it (*denoted by grey color depicting deletion*).



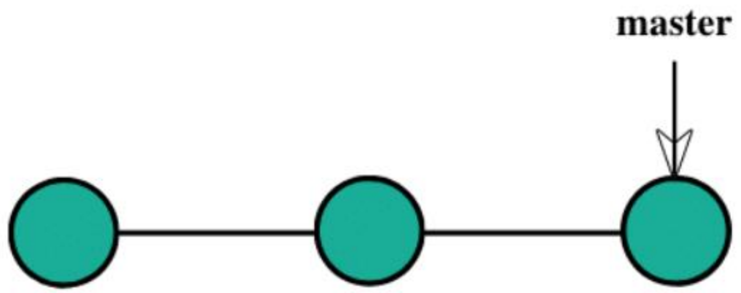
Now, since you were following the linear development method, you need to delete the complete code and go through the hectic process of adjustments and removing glitches repeatedly to achieve the following:



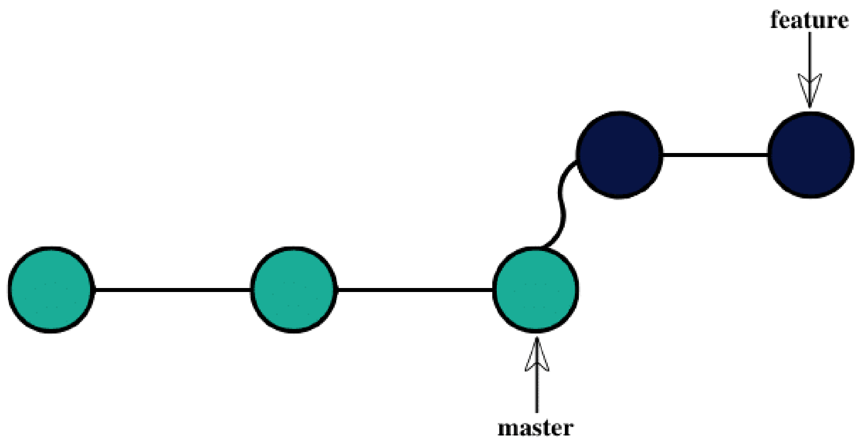
Developing the project through branching.

Let see the same scenario by using the Git branching technique.

You have been working on a project with the client being happy until this point.

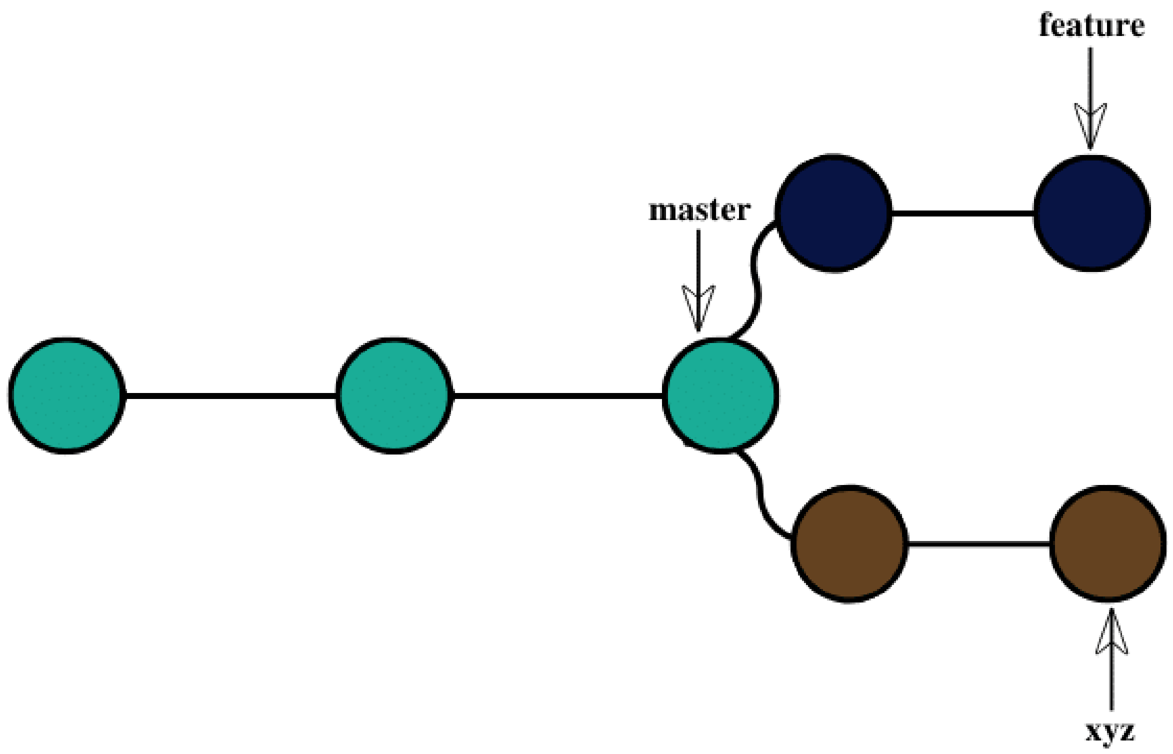


After that, you decide to develop a feature and create a new branch called feature for the same purpose and start working on it.



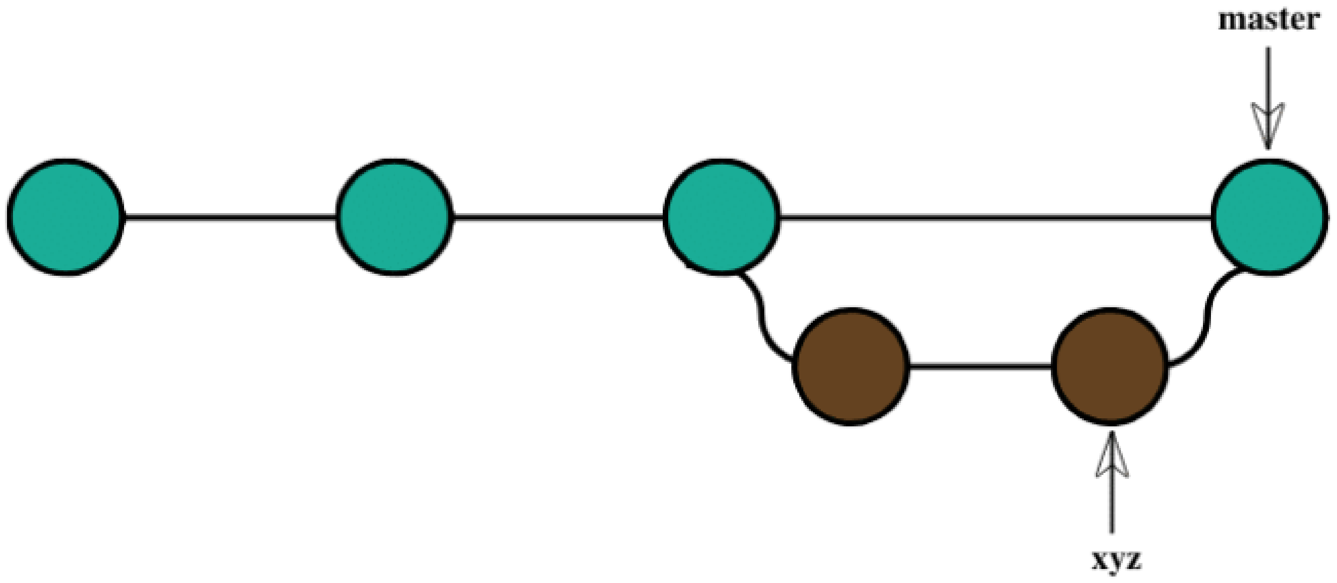
You show this to your client.

In the meantime, you decide to develop another feature and wait for the client's approval.



The client disapproves of the feature and requests to delete it.

Now, since you were following the branched strategy, you need to remove the branch, and all the remaining code remains as it is. The new feature can be easily added to the master branch to achieve the following.

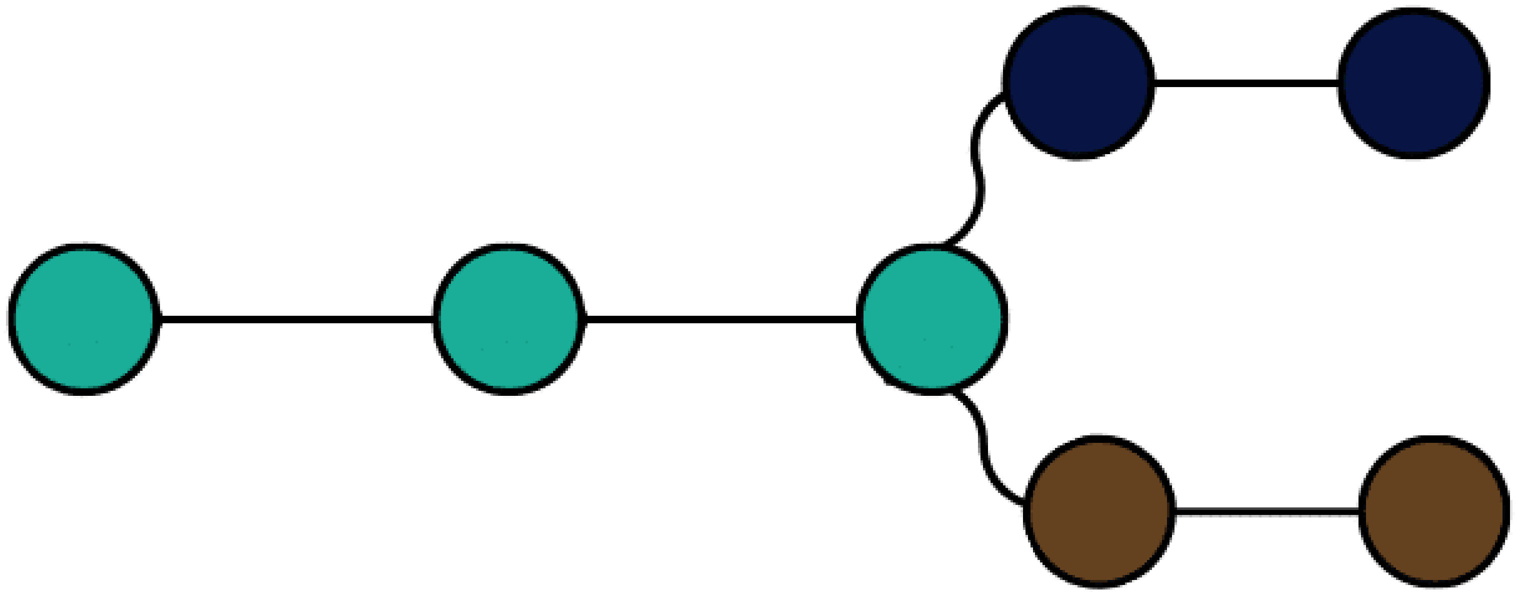


- Branches give you the freedom to independently work on different modules and merge the modules when you finish developing them.
- It might sound a cumbersome process, but git branches are swift to be created and destroyed.
- Just a simple command can perform these processes, and they are very cheap, considering the size they take.
- Branches in Git help the team, which are in different parts of the world, work independently on independent features that would ultimately combine to produce a great project.

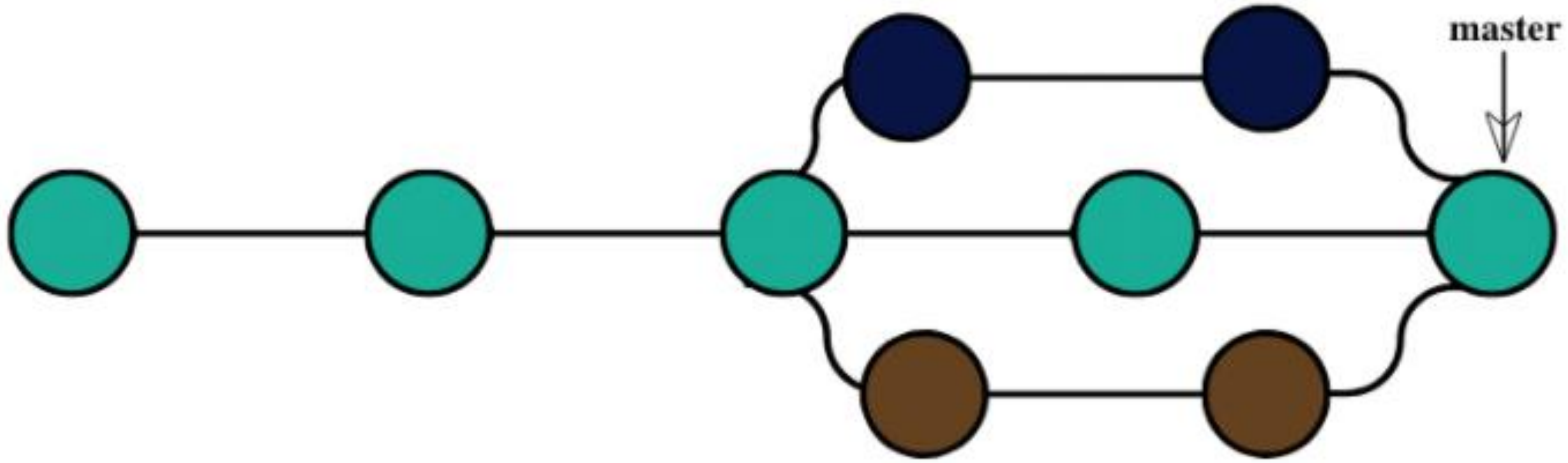
- We can use the branch in git for any reason we want.
- We create different branches for different teams working on the project (*or the same module*).
- Additionally, one can create them for any other feature you are creating in the project.

Real-Life Depiction Of a Git Branch

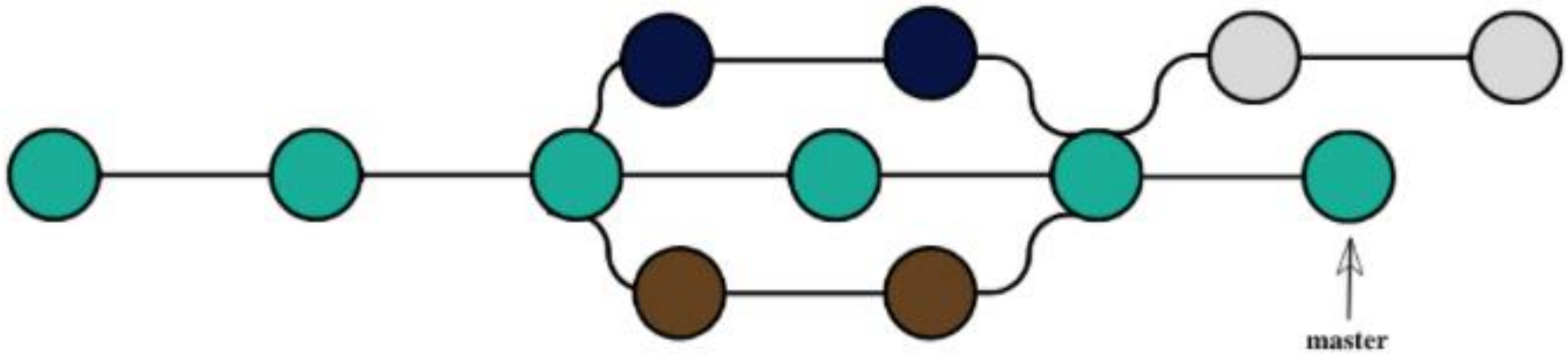
Let's assume you are working on a project along with your friend. You both are working on two different features and hence are working on two different branches. We can see it in the below image.



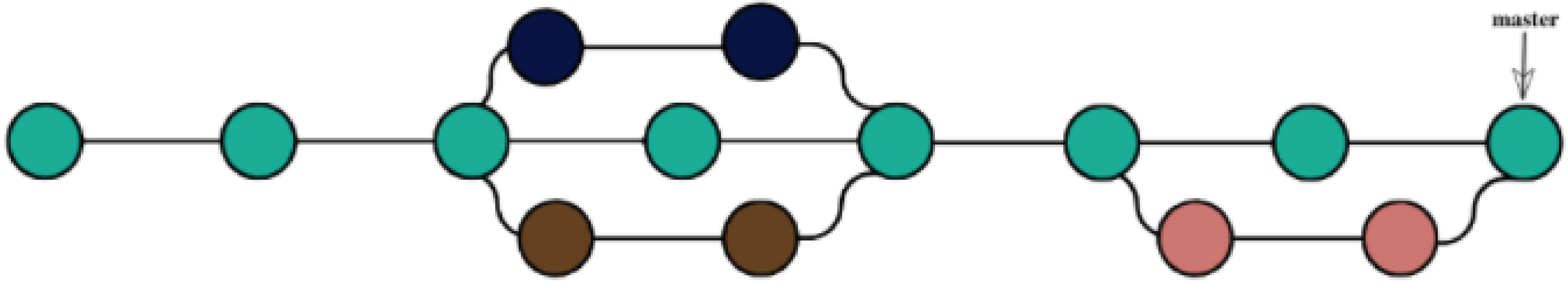
Once you both finish work on your particular features, these feature branches were merged into the master branch and accepted into the main stable code.



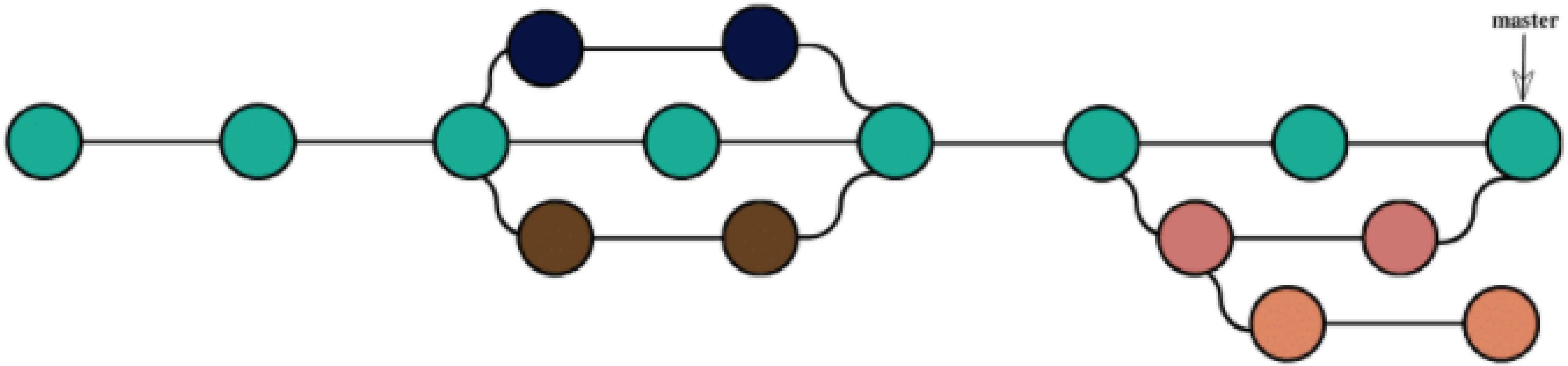
Later on, you start working on a different branch and a distinct feature. But, due to some reason, the feature is not required this time. Therefore, the main master branch does not include it. The branch below in grey shows that it has to delete without inclusion.



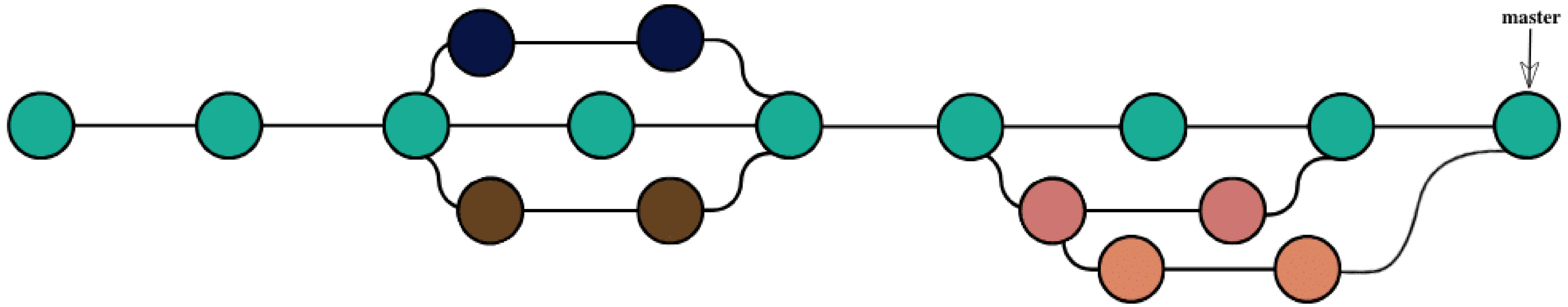
Although the team is working on the master branch continuously, sometime later, an urgent fix pops up that should urgently address. The team fixes it and merges it into the master.



Meanwhile, when this fix came, to add some additional functionalities, it was pulled to a feature branch.



After working on this pulled branch, it finally merges to the master branch.



Different Operations On Branches

- ***Create a Branch***: *This is the first step in the process, you can start on a default branch or create a new branch for the development.*
- ***Merge A Branch***: *An already running branch can merge with any other branch in your Git repository. Merging a branch can help when you are done with the branch and want the code to integrate into another branch code.*
- ***Delete A Branch***: *An already running branch can delete from your Git repository. Deleting a branch can help when the branch has done its job, i.e., it's already merged, or you no longer need it in your repository for any reason.*
- ***Checkout A Branch***: *An already running branch can pull or checkout to make a clone of the branch so that the user can work on any of them. Pulling a branch can help when you don't want to disturb the older branch and experiment on the new one.*

ToDo

- Run basic commands to get hands on

Here are some basic Git commands along with brief explanations:
git init: Initializes a new Git repository in the current directory. This command creates a hidden directory called ".git" where Git stores its internal data for version control.
git clone <repository_url>: Clones an existing Git repository from a remote server to your local machine. This command creates a copy of the remote repository on your local machine, allowing you to work on the codebase locally.
git add <file(s)>: Adds file(s) to the staging area, preparing them to be included in the next commit. This command stages changes for commit.
git commit -m "commit message": Commits the changes staged in the current branch to the repository. A commit represents a snapshot of the project at a specific point in time.
git status: Displays the current status of the repository, including the state of tracked and untracked files, changes in the working directory, and the branch you are currently on.

git pull: Fetches changes from the remote repository and merges them into the current branch. This command is used to update your local repository with changes made by others.
git push: Pushes committed changes from your local repository to the remote repository. This command is used to share your work with others and update the remote repository with your changes.
git branch: Lists all local branches in the repository. By default, this command shows the current branch with an asterisk (*) next to it.
git checkout <branch_name>: Switches to the specified branch. This command is used to navigate between branches or restore files in the working directory to their state in a specific branch.
git merge <branch_name>: Merges changes from the specified branch into the current branch. This command combines the changes made in the specified branch with the changes in the current branch.
git remote -v: Lists the remote repositories associated with your local repository along with their URLs. This command is useful for viewing the remote repositories you are working with.

Sample

Initialize a Git Repository:

```
mkdir my-website
cd my-website
git init
```

Create Some Files:

Create some files for your website. For this example, let's create an index.html file:

```
<!-- index.html -->
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width,
initial-scale=1.0">
   <title>My Website</title>
</head>
<body>
   <h1>Hello, World!</h1>
</body>
</html>
```

Add and Commit Changes:

Add the index.html file to the staging area and commit it to the repository:

```
git add index.html
git commit -m "Initial commit: Added index.html"
```

Collaboration: Imagine your colleague wants to contribute to the project. They clone the repository to their local machine:

```
git clone <repository_url>
```

Make Changes:

Your colleague makes some changes to the index.html file. They add an additional paragraph to the webpage:

```
<!-- index.html -->
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width,
initial-scale=1.0">
   <title>My Website</title>
</head>
<body>
   <h1>Hello, World!</h1>
   <p>Welcome to our website!</p> <!-- New line added -->
</body>
</html>
```

Add and Commit Changes (Colleague): Your colleague adds and commits their changes:

```
git add index.html
git commit -m "Added welcome message"
```

Pull Changes (You): You pull the changes made by your colleague to your local repository:

```
git pull origin master
```

Review Changes:
You review the changes made by your colleague in the index.html file

Make Further Changes:

You decide to make further changes to the index.html file. You update the heading to make it more prominent:

```
<!-- index.html -->
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>My Website</title>
</head>
<body>
    <h1>Welcome to Our Website!</h1> <!-- Updated heading --
>
    <p>Hello, World!</p>
    <p>Welcome to our website!</p>
</body>
</html>
```

Add and Commit Changes (You): You add and commit your changes:

```
git add index.html
git commit -m "Updated heading"
```

Push Changes: Finally, you push your committed changes to the remote repository:

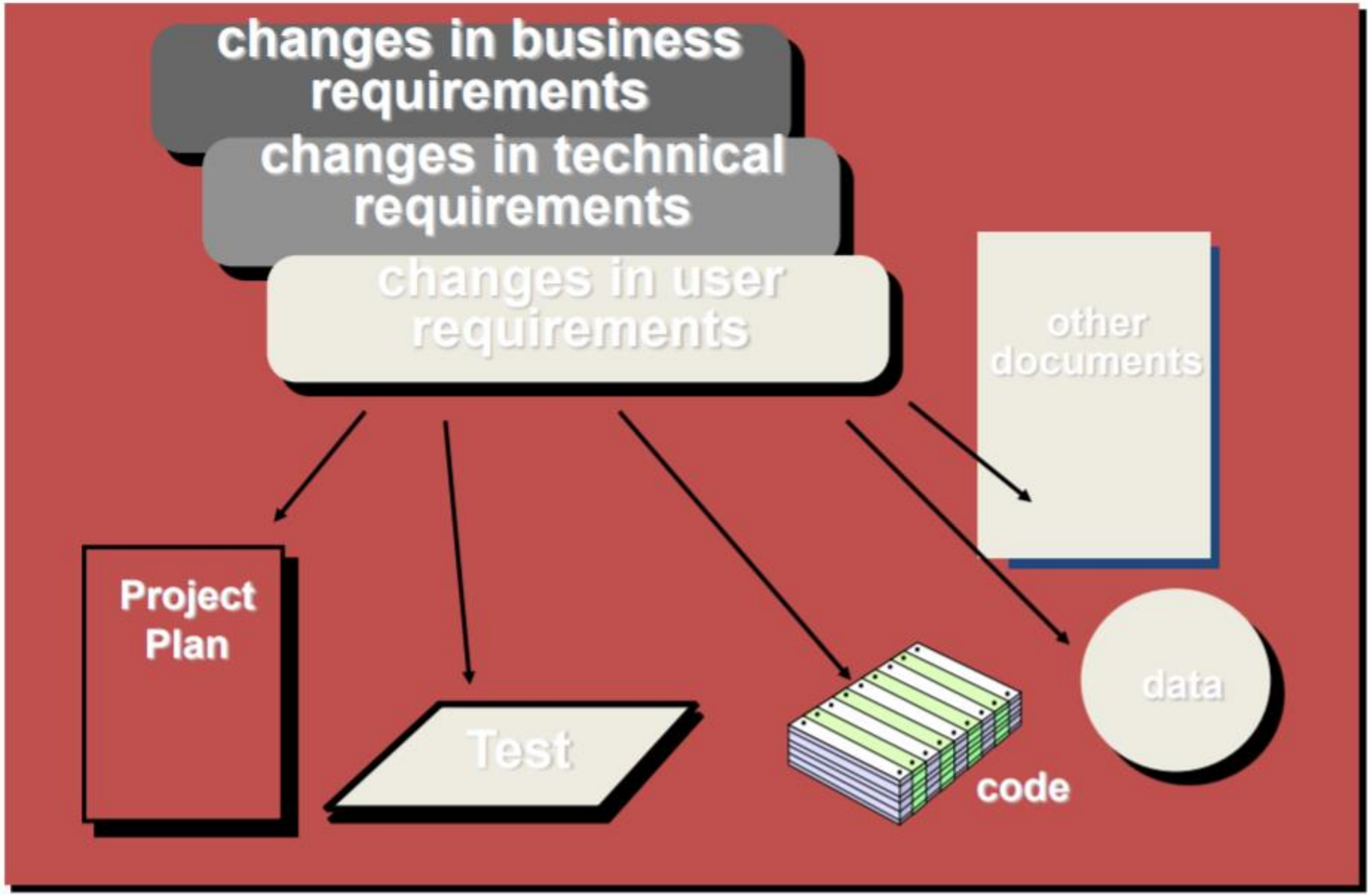
```
git push origin master
```

Software Configuration Management

- Definition of CM
- What is CM
- Why do we have CM

No matter where you are in the system life cycle, the system will change, and the desire to change it will persist throughout the life cycle.

Bersoff, et al, 1980


changes in business requirements
changes in technical requirements
changes in user requirements
other documents
Project Plan
Test
code
data

Configuration management is the art of identifying, organizing, and controlling modifications to the software (& documentation) being built by a programming team.

SCM activities:

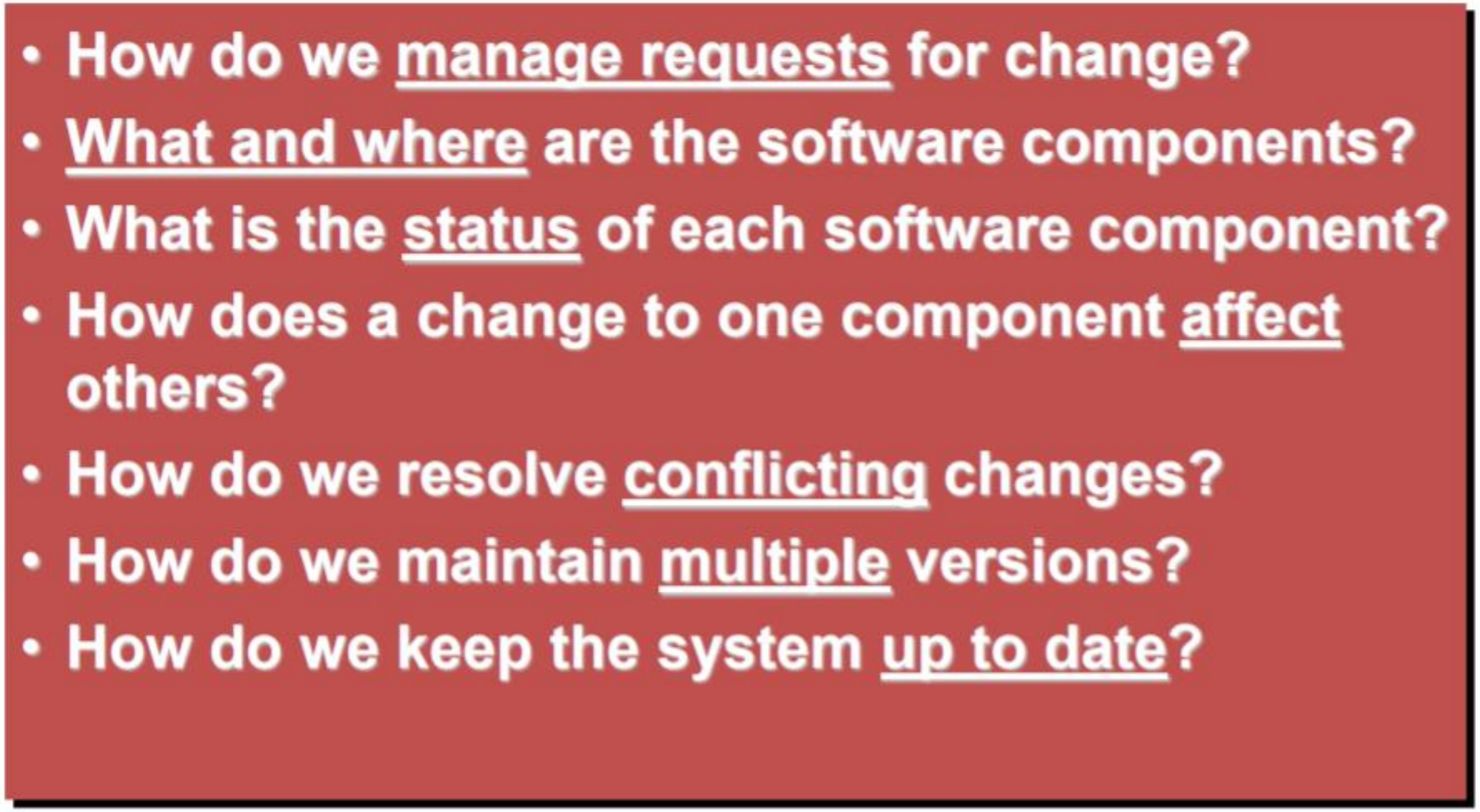
- identify change
- control change
- ensure that change is being properly implemented
- report change to others who may have an interest

Baselines

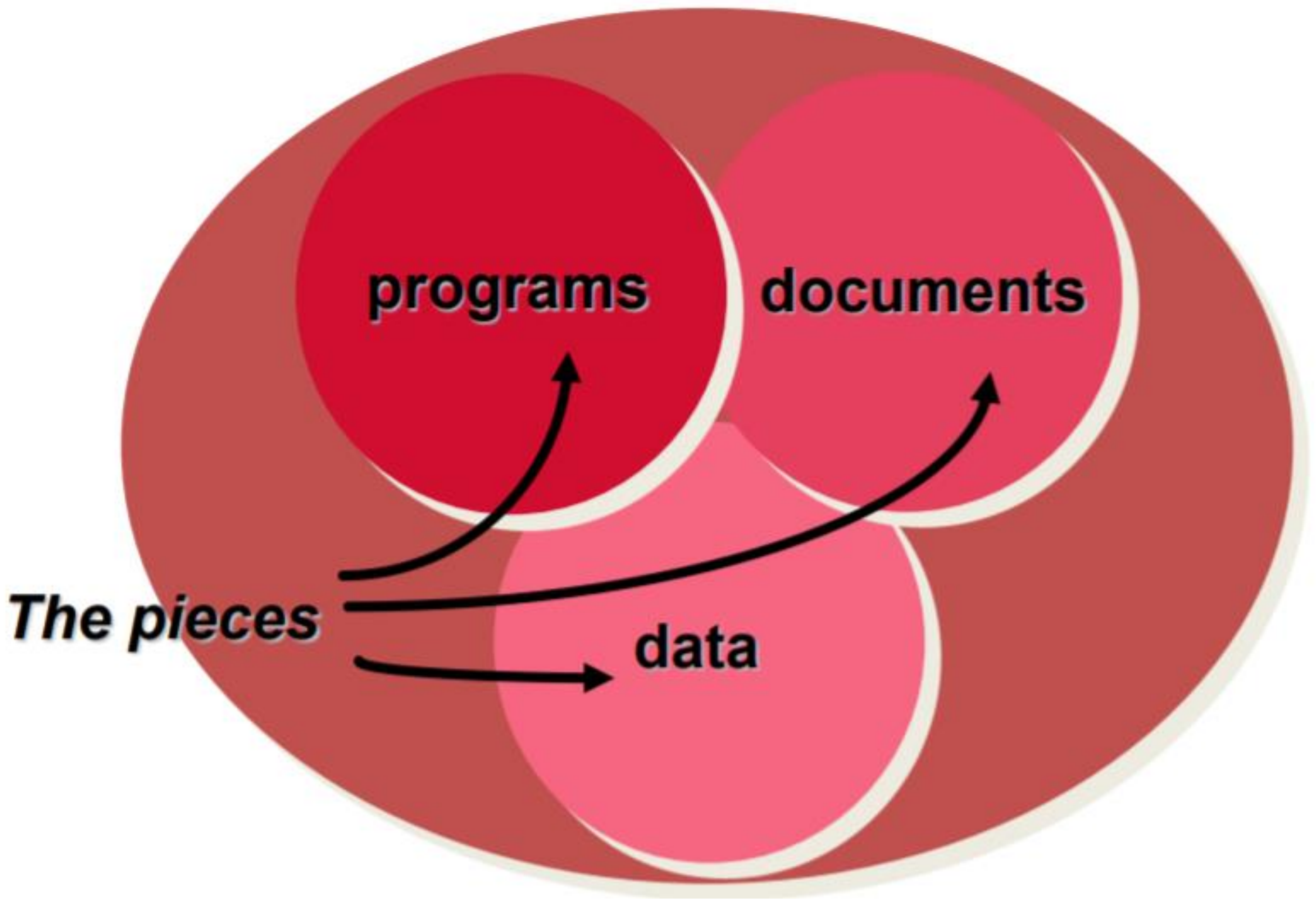
“A specification or product that has been formally reviewed and agreed upon, that thereafter serves as the basis for further development, and that can be changed only through formal change control procedures.”

- A baseline is a specification or product that has been reviewed and agreed upon
- Once a baseline is established, changes can be made only via a formal procedure, which evaluates and verifies each change.

The Questions Which SCM Must Answer



- How do we manage requests for change?
- What and where are the software components?
- What is the status of each software component?
- How does a change to one component affect others?
- How do we resolve conflicting changes?
- How do we maintain multiple versions?
- How do we keep the system up to date?


programs
documents
The pieces
data

SCM Tasks

- According to IEEE we can define these primary activities in CM
  1. Identification of objects
  2. Management
  3. Configuration auditing
  4. Status reporting

1. Identification of objects

- Identification is the task of identifying our divisional artifacts i.e. the items that make up our system e.g.
  - SDLC Documents
  - Software
  - Hardware

2. Management

- Management is the introduction of controls (procedures and quality gates) to ensure that product evolves appropriately
- Keep your focus on
  - Version control
  - Change control

Change Control Procedure

- Need for change is recognized
- User creates Change Request (CR)
- Developer evaluates CR, and produces report
- Change Control body decides, either
  - Yes: ECO (engineering change order) is queued
  - Maybe: Report rejected – need more info.
  - No: CR is denied; user is informed

3. Configuration Audit

- Audit is the review of the organizational process against the defined/required standards
- Areas of focus includes
  - – Adherence to process
  - – Conformance to security
  - – Configuration Verification

4. Status reporting

- Configuration Status Reporting
- A CSR report is generated on regular basis and is intended to keep management and practitioners appraised of important changes
- What happened?
- Who did it?
- When did it happen?
- What else will be affected?

Link of SCM and Evolution of requirements

SCM (Software Configuration Management) plays a crucial role in requirement evolution by providing a structured approach to managing changes to requirements throughout the software development lifecycle.

Version Control
Traceability: SCM systems enable traceability between requirement changes and other artifacts in the development process, such as code changes, test cases, and documentation. This traceability ensures that all stakeholders can understand the rationale behind requirement changes and their impact on the overall project.
Branching and Merging: SCM systems support branching and merging, allowing teams to manage parallel streams of development and experiment with different versions of requirements. Teams can create branches to work on specific features or changes independently, and then merge them back into the main codebase once they are complete.

Collaboration: SCM systems facilitate collaboration among team members by providing mechanisms for code review, discussion, and feedback. Teams can use features like pull requests (in Git-based systems) to review requirement changes before they are merged into the main codebase, ensuring that they meet the necessary quality standards.
Integration with Other Tools: SCM systems can integrate with other tools commonly used in the software development process, such as issue tracking systems (e.g., Jira), CI/CD pipelines, and project management tools. This integration streamlines the requirement evolution process by providing seamless workflows between different tools and environments.