

WD401 - L1

Submitted by - Swanand Bhuskute

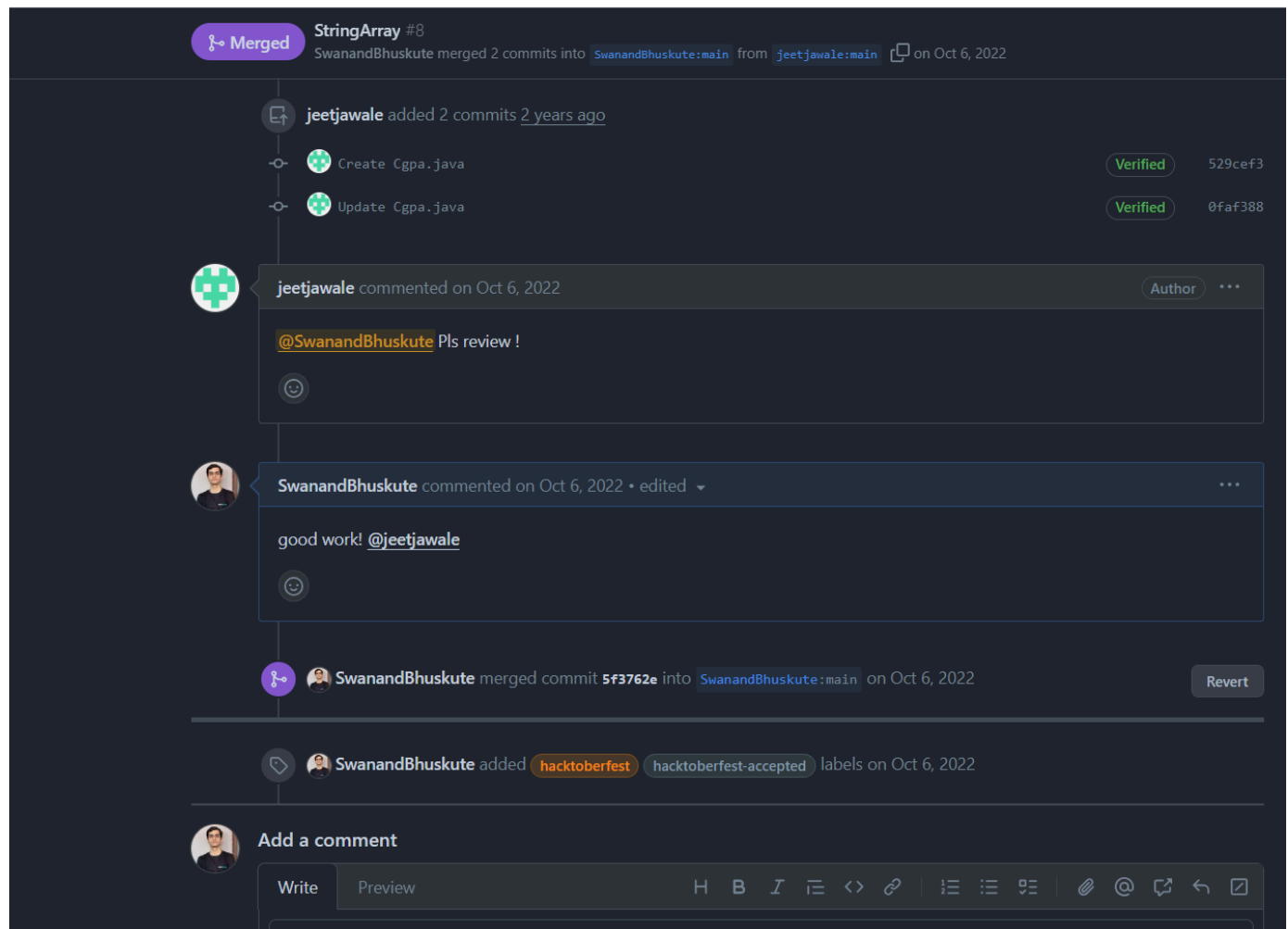
Problem statement:

You've been assigned to a project that involves enhancing a critical feature for a web application. The team places a strong emphasis on the pull-request workflow, with a focus on code reviews, merge conflict resolution, and the recent integration of CI/CD. As you navigate through the development task, you encounter challenges such as feedback during code reviews and discussions on effective merge conflict resolution. The team looks to you to demonstrate your understanding of these challenges and your ability to adapt to the added complexity of CI/CD integration.

Handling Code Review Feedback:

The screenshot shows a GitHub pull request interface. At the top, the repository is 'SwanandBhuskute / Leetcode_and_GFG'. The pull request is titled 'StringArray #8' and is in a 'Merged' state. It shows that 'SwanandBhuskute' merged 2 commits into 'SwanandBhuskute:main' from 'jeetjawale:main' on Oct 6, 2022. Below the title, there are tabs for 'Conversation' (2), 'Commits' (2), 'Checks' (0), and 'Files changed' (1). A comment from 'jeetjawale' dated Oct 6, 2022, is visible. The comment text is: 'It is CLI based program in java', 'Fixes #6', and 'Screenshots of relevant screens'. Below the text is a screenshot of a terminal output showing a CLI program in Java. The terminal output is as follows:

```
Number of students: 3
CGPA of students: 8
9
7
Name of students with 8+ CGPA:
Sam
Rick
User Input:
Morty
Morty is not a 8.0 pointer
```



So, let me explain this how it is working

1. The other developer provided the code solution for the issue I had raised.
2. He created a Pull request with the necessary code and outputs.
3. He pinged me for code review and merge.
4. I checked the PR and the whole code.
5. I found it worthy according to the issue.
6. I wrote “good work” and pinged him.
7. Finally I merged the PR.

This is a very simple and basic example and explanation of how Pull requests work and developers work collaboratively.

This process can only be sped up if code reviews happen fast. And there won't arise any merge conflicts

This was all about **handling code review feedback**

Iterative Development Process:

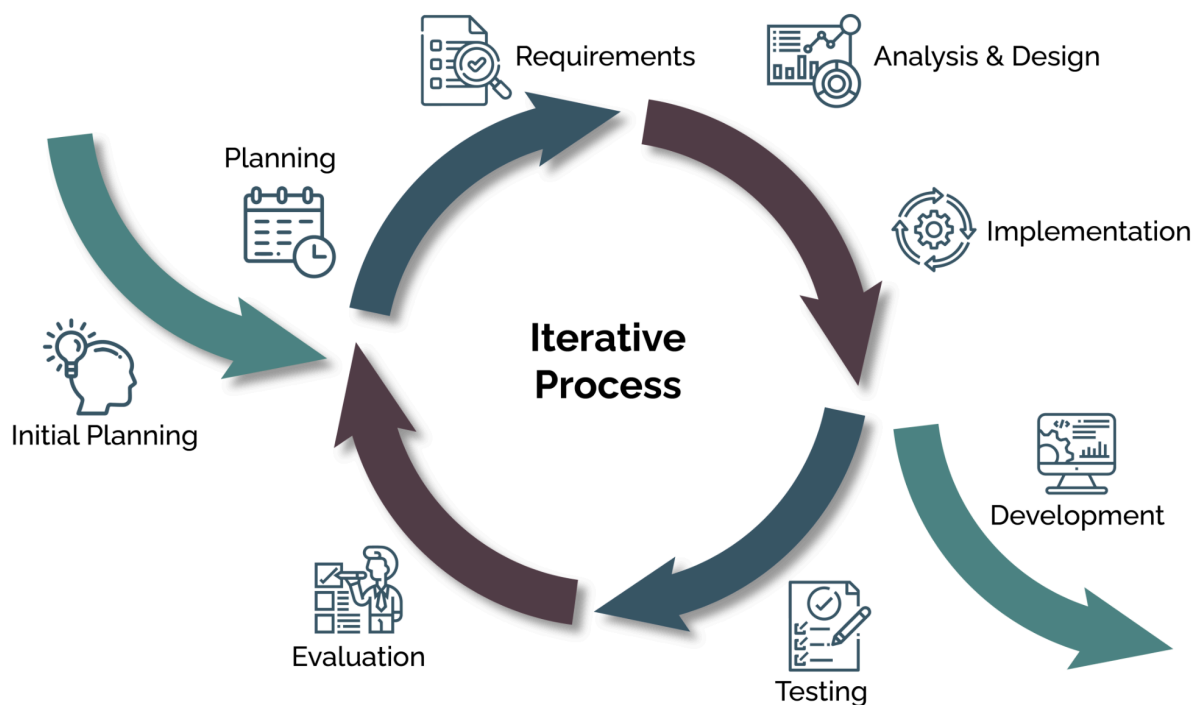
Iterative development is a way of breaking down the software development lifecycle (SDLC) of a large application into smaller chunks. It is typically used in conjunction with incremental development in which a longer SDLC is split into smaller segments that build upon each other.

In iterative development, feature code is designed, developed and tested in repeated cycles. These cycles or iterations (also known as sprints) give the concept its name.

With each iteration, additional features can be designed, developed and tested to add to the program's functionality. These iterations continue until a fully functional software application is created and ready for deployment to customers or end users.

This can be understood using simple standard diagram:

Iterative Process Model



So my approach will be:

1. I will raise requirements in the "Issues"
2. When other developers will provide solutions after developing, I will test them according to the problem statement and verify its authenticity and worthiness.
3. If I find any further issues, I will mention them in the comment of code review.

4. Then the respective developer will again develop/modify the solution and again ping me for check.
5. I will again test them according to the problem statement and verify its authenticity and worthiness.
6. We will keep repeating this process until I am satisfied with any given solution.
7. Finally, I will merge/deploy.

This is how the whole iterative development process works.

I explained this in a very naive way for this level, just to give a high level idea.

Resolving Merge Conflicts:

A git merge conflict is an event that takes place when Git is unable to automatically resolve differences in code between two commits. Git can merge the changes automatically only if the commits are on different lines or branches.

Merge conflicts occur when competing changes are made to the same line of a file, or when one person edits a file and another person deletes the same file.

To resolve a merge conflict caused by competing line changes, you must choose which changes to incorporate from the different branches in a new commit.



Common steps to resolve:

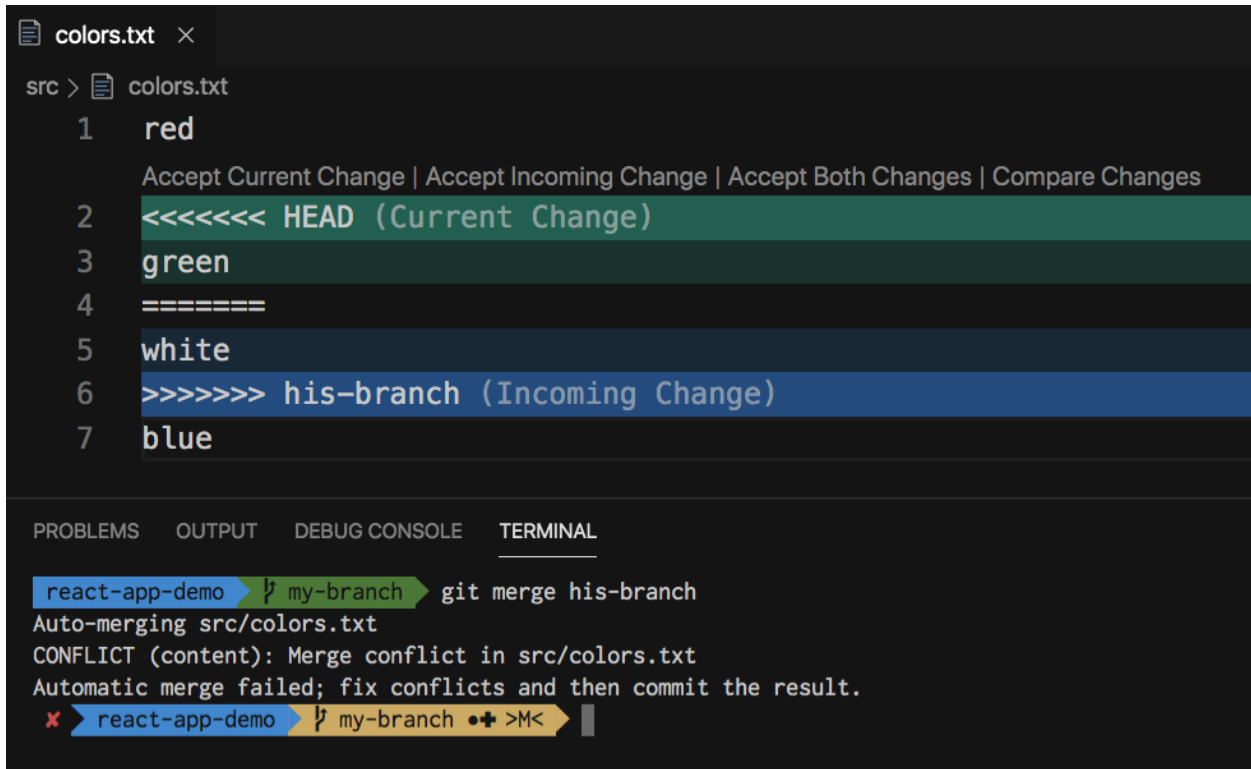
1. Identify the Conflict - Git will indicate there is a conflict and show which files are conflicting. We can use command like

`git merge feature-branch`

to merge branches.

2. Locate the Conflict - Open the conflicting files. Conflicts are marked by <<<<<<, =====, and >>>>>> lines.

This can simply be viewed in any code editor like VScode.



```
src > colors.txt
1  red
   Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes
2  <<<<<< HEAD (Current Change)
3  green
4  =====
5  white
6  >>>>>> his-branch (Incoming Change)
7  blue

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
react-app-demo my-branch git merge his-branch
Auto-merging src/colors.txt
CONFLICT (content): Merge conflict in src/colors.txt
Automatic merge failed; fix conflicts and then commit the result.
x react-app-demo my-branch >M<
```

3. Resolve the Conflict - Edit the conflicting files to resolve the conflicts. Choose between the changes or merge them manually.

This step usually involves removing the conflict markers (<<<<<<, =====, and >>>>>>).

4. Add the Resolved Files - This usually means adding new files / resolved files after resolving conflicts.

```
git add .
git add -all
git add filename
```

5. Commit the work - Commit all the changes

```
git commit -m "any message"
```

6. Continue working as normal as before, if again encountered with conflicts, repeat the same steps and go back to normal.

These are the simple steps to resolve conflicts. It totally depends on the kind of conflict the developer encounters and they choose respective commands to resolve them. I mentioned some.

CI/CD Integration:

CI/CD, which stands for continuous integration and continuous delivery/deployment, aims to streamline and accelerate the software development lifecycle.

In very simple terms, CI is a modern software development practice in which incremental code changes are made frequently and reliably. Automated build-and-test steps triggered by CI ensure that code changes being merged into the repository are reliable. The code is then delivered quickly and seamlessly as a part of the CD process. In the software world, the CI/CD pipeline refers to the automation that enables incremental code changes from developers' desktops to be delivered quickly and reliably to production.

CI/CD allows organizations to ship software quickly and efficiently. CI/CD facilitates an effective process for getting products to market faster than ever before, continuously delivering code into production, and ensuring an ongoing flow of new features and bug fixes via the most efficient delivery method.

Common CI/CD Issues

1. Failing builds - This usually occurs due to compilation errors, failed tests, missing dependencies, or misconfigured build scripts.
2. Test failures - This can be caused due to wrong tests or flaky tests.
3. Environment Configuration Issues - This occurs due to differences between development, staging, and production environments. This can be resolved by setting environment variables correctly.
4. Deployment failures - This can occur due to incorrect deployment scripts, insufficient permissions, network issues. This can be resolved by giving proper permissions, staying with proper network connectivity and writing proper deployment scripts.

Code example (not real):

```
# Example GitLab CI configuration (.gitlab-ci.yml)
stages:
  - build
  - test
  - deploy

build:
  stage: build
  script:
    - npm install
    - npm run build
  only:
    - main

test:
  stage: test
  script:
    - npm install
    - npm test
  only:
    - merge_requests

deploy:
  stage: deploy
  script:
    - ./deploy.sh
  only:
    - main
```

Thank you.