Icon

Description automatically generated

**Software Maintenance and Evolution**

**Assignment 3**

**SWE473**

**Dr. Mohmmad Khan**

|  |  |
| --- | --- |
| **Names** | **Student ID** |
| **Fares Mohammed Al Areefi** | **1082090** |

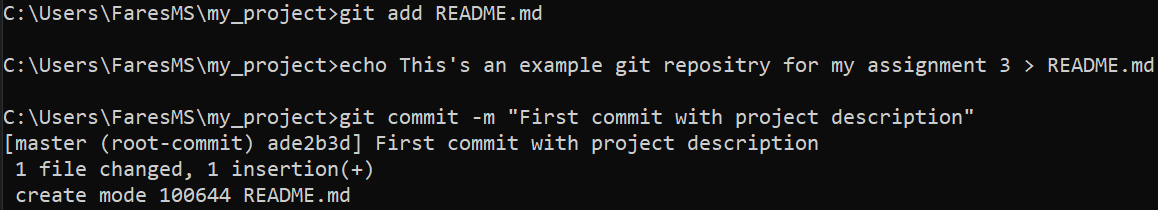
1. Setup Git Repository:

* Initialize a new Git repository in a folder named my\_project.

A black screen with white text

Description automatically generated

* Create a file named README.md with a brief description of the project.



* Stage and commit the README.md file to the repository.

A black screen with white text

Description automatically generated

1. Making Changes and Version Control:

* Create a new branch named feature-1.



* Switch to the feature-1 branch.

A black background with white text

Description automatically generated

* Add a new file named app.py with a simple Python script (e.g., a "Hello, World!" program).



* Stage and commit the app.py file to the feature-1 branch.

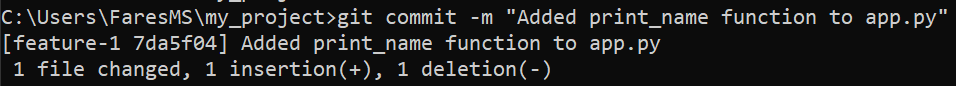


* Modify the app.py file to include a new function that prints your name.



* Stage and commit the changes.





1. Merging Branches:

* Switch back to the main branch.

A black background with white text

Description automatically generated

* Merge the feature-1 branch into the main branch.

A screen shot of a computer code

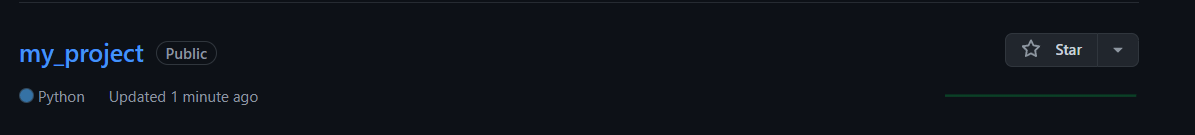
Description automatically generated

* Resolve any merge conflicts if they arise.

No merge conflicts occurred.

1. Using GitHub (or any other remote repository service):

* Create a new repository on GitHub named my\_project.



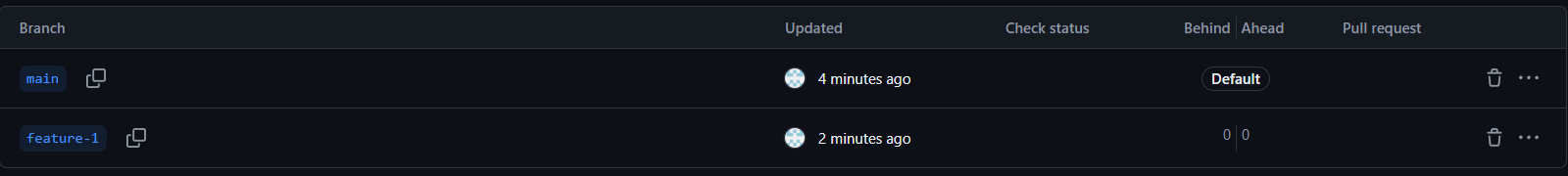
* Push the local my\_project repository to GitHub.

A screen shot of a computer

Description automatically generated

* Ensure that both main and feature-1 branches are available on the remote repository.

A screen shot of a computer program

Description automatically generated 

1. Collaboration Simulation:

* Create another branch named bugfix-1.

A black background with white text

Description automatically generated

* Simulate a bug fix by making a small change to the app.py file (e.g., correcting a typo).

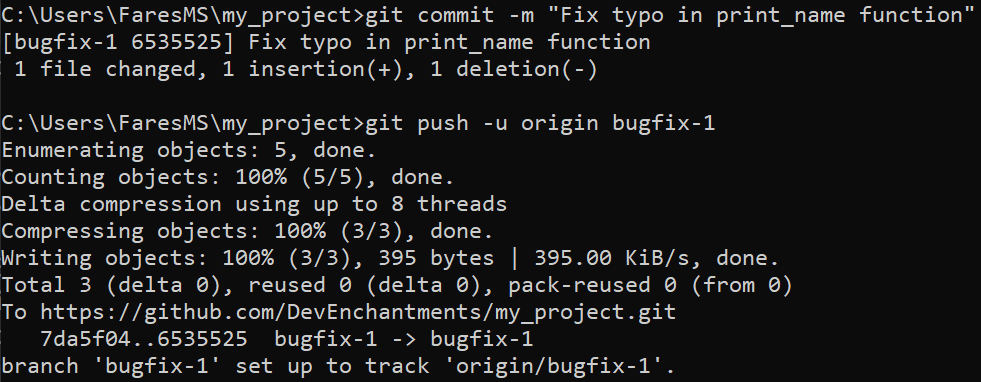


* Stage and commit the changes on the bugfix-1 branch.

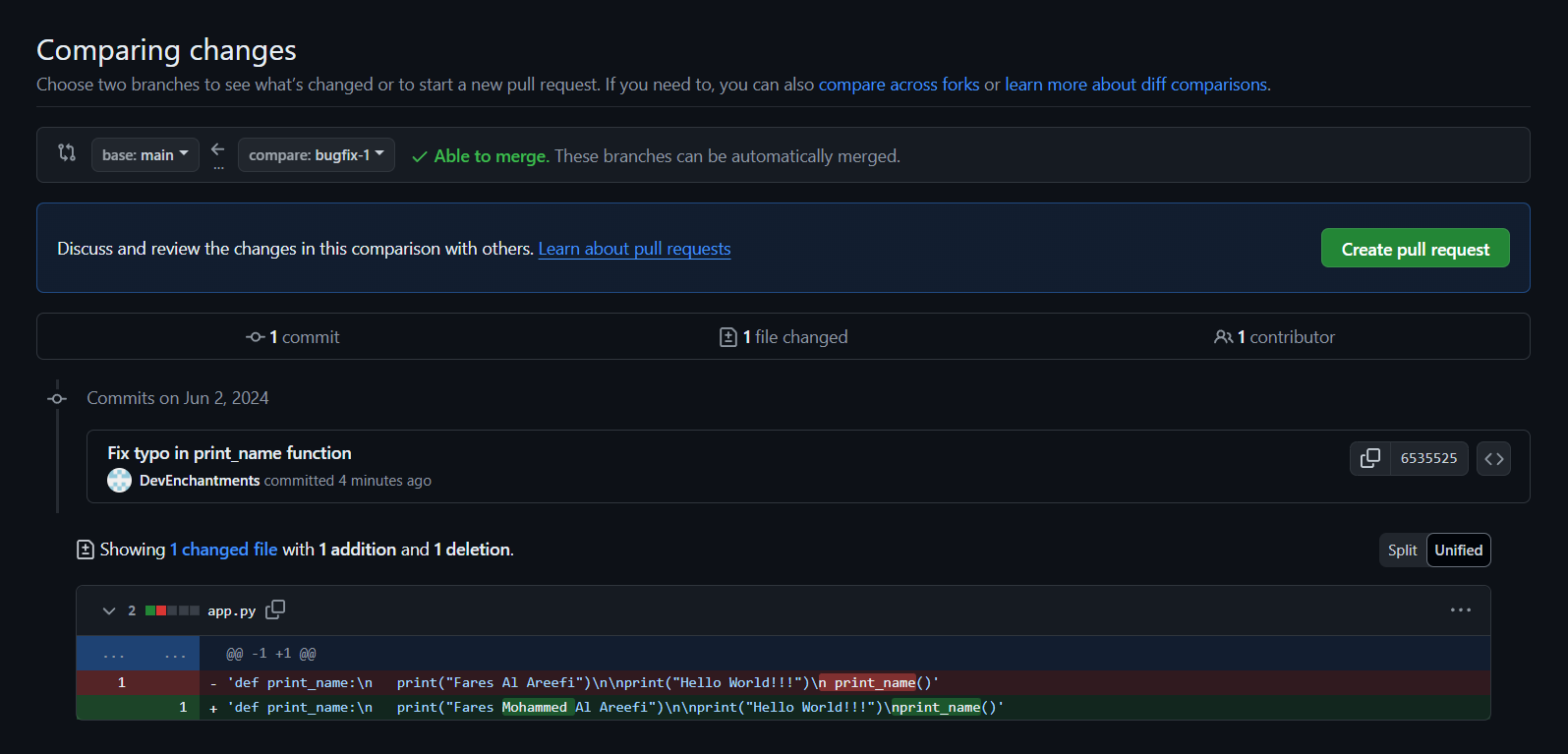
A screen shot of a computer

Description automatically generated

* Push the bugfix-1 branch to the remote repository.



* Open a pull request on GitHub to merge bugfix-1 into main.



* Complete the pull request.

A screenshot of a computer

Description automatically generated

1. History and Reversion:

* Use Git to view the commit history.

A screen shot of a computer program

Description automatically generated

* Revert the last commit on the main branch.

A black background with white text

Description automatically generated

* Explain how to undo the revert if necessary.

To undo a revert, we can use Git to revert the revert commit. which creates a new commit that undoes the changes made by the previous revert.

Q2)

There are many tools available to help with writing clean, maintainable code and assist with maintenance tasks such as refactoring and performance enhancements.

**Static Code Analysis Tools**

1. **ESLint**:
   * **Languages Supported**: Primarily JavaScript, TypeScript
   * **Features**: Identifies problematic patterns in code, enforces coding standards, and integrates with various editors and CI/CD pipelines.
   * **Use Case**: Ensures code quality and consistency by catching errors early and enforcing coding standards.
2. **SonarQube**:
   * **Languages Supported**: Multiple languages including Java, C#, JavaScript, TypeScript, Python, and more.
   * **Features**: Continuous inspection of code quality, identifying bugs, vulnerabilities, and code smells. Provides detailed dashboards and reports.
   * **Use Case**: Comprehensive code quality management and technical debt analysis.
3. **Pylint**:
   * **Languages Supported**: Python
   * **Features**: Checks for errors in Python code, enforces a coding standard, looks for code smells, and offers simple refactoring suggestions.
   * **Use Case**: Python-specific code quality tool that integrates well with other development tools and CI/CD systems.

Q3)

**Compilation vs. Decompilation: Forward and Reverse Engineering**

**Compilation**:

* **Forward Engineering**: Compilation is considered a forward engineering process because it involves translating high-level source code (written by developers) into lower-level machine code or bytecode that can be executed by a computer. This transformation moves in a "forward" direction, from human-readable code to a form that the machine can execute.
* **Purpose**: The primary goal is to produce an executable program from the source code, ensuring that it runs efficiently on the target platform.

**Decompilation**:

* **Reverse Engineering**: Decompilation is the process of translating machine code or bytecode back into a higher-level, human-readable source code. This is considered reverse engineering because it moves in the opposite direction of compilation, trying to reconstruct the original source code from the compiled code.
* **Purpose**: The main goal of decompilation is to understand, analyze, or modify the behavior of compiled programs when the original source code is not available.

**Uses for Decompilers**

1. **Debugging and Analysis**: Developers can use decompilers to debug and analyze compiled code, especially when the source code is lost or unavailable.
2. **Security Auditing**: Security researchers use decompilers to inspect software for vulnerabilities, ensuring that there are no hidden malicious elements.
3. **Understanding Third-Party Libraries**: When working with third-party libraries without source code, developers can decompile them to understand how they work and how to integrate them into their projects.
4. **Recovering Lost Source Code**: In cases where source code is lost due to accidents or mismanagement, decompilers can help recover the functionality of the software.
5. **Educational Purposes**: Decompilers are used in educational contexts to teach students how machine code corresponds to high-level programming constructs.

**Methods to Prevent Decompilation**

1. **Obfuscation**: Code obfuscation involves transforming the code into a form that is difficult to understand while preserving its functionality. This makes it harder for decompilers to produce useful, readable source code.
   * **Example**: Renaming variables, methods, and classes to meaningless names, encrypting strings, and using complex control flow constructs.
2. **Encryption**: Encrypting the bytecode and decrypting it at runtime can make it more challenging for decompilers to generate meaningful code.
3. **Code Packing**: Packing the code into a compressed and encrypted format that is unpacked at runtime can prevent straightforward decompilation.
4. **Native Code Conversion**: Converting critical parts of the code into native code can make decompilation more difficult, as native code is harder to reverse-engineer than bytecode.

**Accessible Decompiler Tool: IntelliJ IDEA's Built-in Decompiler**

**Functionality and Features**:

* **Integrated Decompiler**: IntelliJ IDEA comes with an integrated decompiler that allows you to view the source code of compiled Java classes directly within the IDE.
* **User-Friendly Interface**: The decompiler provides a user-friendly interface, seamlessly integrated into the IDE, making it easy to navigate and understand decompiled code.
* **On-the-Fly Decompilation**: As soon as you open a compiled .class file, IntelliJ IDEA automatically decompiles it and shows the source code in the editor.
* **Syntax Highlighting**: The decompiled code is displayed with syntax highlighting, making it easier to read and understand.
* **Navigation and Search**: You can navigate through decompiled code and search for specific classes, methods, or variables just like you would with regular source code.
* **Integration with Debugger**: You can set breakpoints and debug decompiled code, which is particularly useful for analyzing third-party libraries or debugging issues in compiled code.