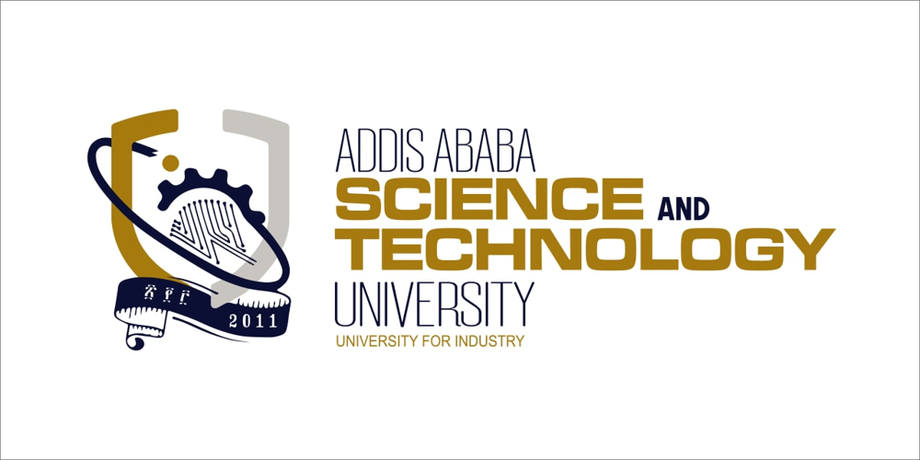
****

**COLLEGE OF ELECTRICAL AND MECHANICAL ENGINEERING**

**DEPARTMENT OF SOFTWARE ENGINEERING**

**Software evolution and maintenance**

**Group members ID No**

1. Gelila Adugna ETS0312/12
2. Hara Birhanu ETS0336/12
3. Kokebe Negalgn ETS0419/12
4. Mastewal Tesfaye ETS0435/12

**SECTION B**

**Submitted to Mr. Ashenafi**

**Submission date May 13,05,2024**

3.**Duplicate code**

duplicate code is also called duplicate code smell or duplicated code, is a software development concept where identical or similar code is written in multiple places within a program or across different programs. It is sections of code that appear in more than one place within a software system or application. It occurs when identical or very similar code blocks are present in different parts of the codebase.

It can occur due to

* Copy-paste code: When developers copy and paste code from one location to another, often without modifying it, to reuse it in a similar context.
* Similar but not identical code: When developers write similar code in different places, but with minor variations, to achieve a similar functionality.

Duplicate code can lead to several issues, including:

* Maintenance and update challenges: When duplicate code is updated, it needs to be updated in multiple places, which can be time-consuming and prone to errors.
* Code duplication: Multiple instances of the same code can lead to redundancy, increasing the overall size of the codebase.
* Debugging difficulties: When issues arise in duplicate code, it can be challenging to identify the root cause, as the problem may be present in multiple locations.
* Code quality decline: Duplicate code can lead to decreased code quality, as developers may not prioritize refactoring or maintaining the code.
* Code smell: Duplicate code is considered a code smell—a sign that there might be design flaws or missed opportunities for code reuse.

To avoid duplicated code developers can do.

* Code reuse: Implementing modular and reusable code modules that can be easily integrated into different parts of the program.
* Code abstraction: Creating abstract interfaces or classes that define a common behavior, making it easier to reuse and modify the code.
* Code Refactoring: Regularly review the codebase and refactor it to eliminate duplicate code. Refactoring involves restructuring the code without changing its external behavior, making it cleaner, more maintainable, and less prone to duplication Code analysis tools: Utilizing static analysis tools that can identify duplicate code and suggest improvements.

To avoid code duplication, use refactoring.

Refactoring is the process of restructuring existing code to improve its maintainability, scalability, and efficiency, while preserving its original functionality the most common methods are.

* Extract Method/function: Break down a large method into smaller, more manageable methods. This can help eliminate duplicate code by identifying and extracting reusable code blocks.
* Extract Class/Module: When there is a group of related methods or functions that are duplicated across different classes or modules, you can extract them into a separate class or module. This consolidates the common functionality in one place, allowing for easier maintenance and reducing duplication.
* Replace Duplicate Code with Inheritance: Use inheritance to create a base class that contains common functionality, and then extend it to create subclasses that inherit the common behavior.
* Replace Duplicate Code with Composition: Use composition to create objects that contain other objects, allowing you to reuse code without duplicating it
* Utilize Libraries or Frameworks: If you find yourself duplicating functionality that is already available in a library or framework, it's best to leverage those existing components instead of reinventing the wheel. Use the provided APIs or modules to avoid duplicating code.
* Remove Dead Code: Remove code that is no longer used or is unreachable, as it can lead to confusion and maintenance issues.

Example for code duplication

For checking the user is authorized to or not

In one function use

function isAuthorized(username, resource) {

if (username === "admin") {

return true;

} else if (username === "user") {

return resource === "resource1" || resource === "resource2";

} else {

return false;

}

}

duplicate code in another function

function isAuthorized2(username, resource) {

if (username === "admin") {

return true;

} else if (username === "user") {

return resource === "resource1" || resource === "resource2";

} else {

return false;

}

}

But instead of use the same code multiple times use code refactoring method of

function authorize (username, resource) {

if (username === "admin") {

return true;

} else if (username === "user") {

return resource === "resource1" || resource === "resource2";

} else {

return false;

}

}

function isAuthorized(username, resource) {

return authorize (username, resource);

}

Dead code

Dead code is code that is never executed or has no effect. It is also known as "dead code" or "unreachable code." It includes variables, functions, or entire blocks of code that have become redundant due to changes in requirements, logic, or refactoring.

* Unused variables

Eg let un=”hello” if th un is declared and not used in the program then it is dead code.

The variable is not contributing to the program's functionality.

The memory allocated for the variable is wasted.

* Unreachable code

Example: if (false) {console.log('This will never be executed'); }

* Unused functions
* Unused modules
* Commented codes.

**Impacts of dead code on the program**

* . **Memory Waste:** Dead code can occupy memory space, even if it's not being executed. This can lead to increased memory usage, which can cause performance issues and slow down the program.
* **Code Complexity**: Dead code can make the codebase more complex and harder to understand. It can create unnecessary dependencies, variables, and functions, making it harder to maintain and debug the program.
* **Increased Compile Time:** Dead code can increase compile time, especially if it's not properly optimized. It can take longer for the compiler to parse and analyze the code.
* **Reduced Code Readability:** Dead code can reduce code readability, making it harder for developers to understand the codebase.