Software Engineering: Implementation

Student No: 100571823

# Source Code Management

Using GitHub as a source code management tool for this project offers several compelling benefits. GitHub provides robust version control through Git, allowing for efficient tracking of changes and easy rollback to previous states. Its collaboration features, such as pull requests, code reviews, and issue tracking, facilitate teamwork and ensure high code quality. Additionally, GitHub’s cloud-based nature ensures that the project is accessible from anywhere, supporting remote work and offering secure hosting.

A screenshot of a computer

Description automatically generated

# Agreed coding conventions

**Naming Conventions:**

Classes: Class names should be in PascalCase. Example: PassengerRestService.

Variables: Variable names should be in camelCase. Example: $passengerId.

Constants: Constants should be in uppercase with underscores separating words. Example: DB\_SERVER.

Functions/Methods: Method names should be in camelCase. Example: performGet.

**Indentation and Spacing:**

Use 4 spaces for indentation.

Use spaces consistently around operators and after commas.

Maintain consistent spacing between methods, classes, and control structures for readability.

**Comments:**

Add comments to explain complex logic, algorithms, or non-obvious code blocks.

Comments should be clear, concise, and meaningful. Avoid unnecessary comments.

**Error Handling:**

Implement error handling for database operations, HTTP requests, and other critical functionalities.

Use appropriate try-catch blocks for handling exceptions.

Provide meaningful error messages or log entries to assist with debugging.

**Code Organization:**

Separate concerns by dividing code into logical modules, classes, and functions.

Group related methods together within classes.

Follow the Single Responsibility Principle to ensure each class or method has a single, well-defined purpose.

**Code Reusability:**

Encapsulate reusable functionalities into functions, methods, or classes.

Avoid duplicating code by extracting common functionalities into separate functions or utility classes.

Strive for modular design to facilitate code reuse and maintenance.

# Source Code Examples

## Design Pattern Evidence

The DatabaseConnection class implements the Singleton pattern, ensuring that only one instance of the database connection exists throughout the application. This pattern reduces overhead by reusing the same connection instance, promoting low coupling by centralizing database access logic.

/\*\*

 \* Singleton class to manage database connections.

 \*/

class DatabaseConnection {

    private static $instance = null;

    private $connection;

    /\*\*

     \* Private constructor to prevent multiple instances.

     \* Initializes the database connection.

     \*/

    private function \_\_construct() {

        global $DB\_SERVER, $DB\_USERNAME, $DB\_PASSWORD, $DB\_DATABASE;

        $this->connection = new mysqli($DB\_SERVER, $DB\_USERNAME, $DB\_PASSWORD, $DB\_DATABASE);

        if ($this->connection->connect\_error) {

            die("Connection failed: " . $this->connection->connect\_error);

        }

    }

    /\*\*

     \* Returns the single instance of the DatabaseConnection.

     \*

     \* @return DatabaseConnection

     \*/

    public static function getInstance() {

        if (self::$instance == null) {

            self::$instance = new DatabaseConnection();

        }

        return self::$instance;

    }

    /\*\*

     \* Returns the database connection.

     \*

     \* @return mysqli

     \*/

    public function getConnection() {

        return $this->connection;

    }

}

The Factory pattern is demonstrated through the PassengerFactory abstract class and the DefaultPassengerFactory concrete class. This pattern provides an interface for creating objects, allowing the instantiation logic to be centralized and decoupled from the client code that uses these objects. This enhances maintainability and scalability by allowing changes to object creation without modifying the client code.

// Define an interface for creating objects

class PassengerFactory {

    createPassenger(passengerData) {}

}

//  factory class for creating Passenger objects

class DefaultPassengerFactory extends PassengerFactory {

    createPassenger(passengerData) {

        return new Passenger(passengerData);

    }

}

// Define the Passenger class

class Passenger {

    constructor(passengerData) {

        this.passengerid = passengerData.passengerid;

        this.surname = passengerData.surname;

        this.name = passengerData.name;

    }

}

The Passenger class encapsulates the properties of a passenger, providing getter methods for each attribute. This encapsulation is a fundamental principle of object-oriented design, promoting data hiding and ensuring that object properties are accessed only through well-defined interfaces. As well as demonstrating high cohesion as it only relates to passenger data.

<?php

class Passenger

{

    public $passengerid;

    public $surname;

    public $name;

    public $Genre;

    public function \_\_construct($passengerid, $surname, $name)

    {

        $this->passengerid = $passengerid;

        $this->surname = $surname;

        $this->name = $name;

    }

    public function getId()

    {

        return $this->passengerid;

    }

    public function getsurname()

    {

        return $this->surname;

    }

    public function getname()

    {

        return $this->name;

    }

}

?>

Refactoring is demonstrated in my code as the PassengerRestService extends the RestService Class functionality without changing its external behaviour.

class RestService

{ …

class PassengerRestService extends RestService {