

**Course No: CSE-3106**

**Course Title: Software Development Project**

Submitted to,

Associate Professor

Dr. Amit Kumar Mondal

Computer Science and Engineering Discipline, Khulna University,

Khulna

Submitted By,

Parthoraj Nandi (210202)

Susmita Biswas (210224)

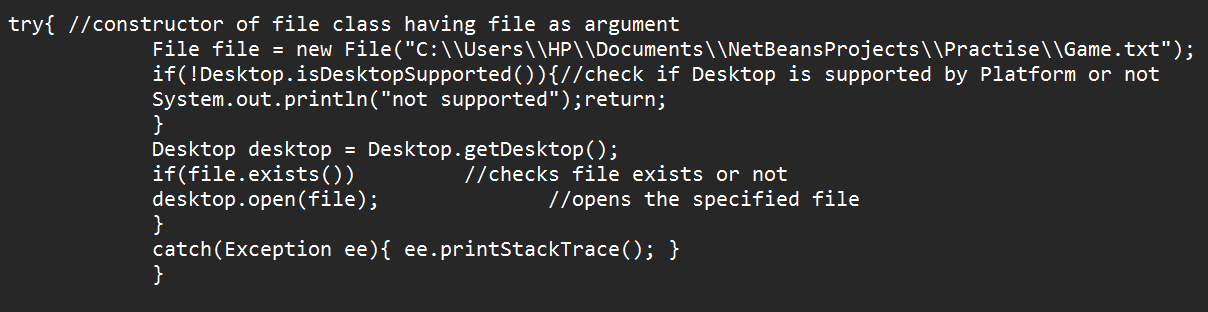
CSE Discipline, Khulna University, Khulna

Code review &Checklist for Code Reviews of “Snake game**” (**210215, 210231**)**

**Code smells:**

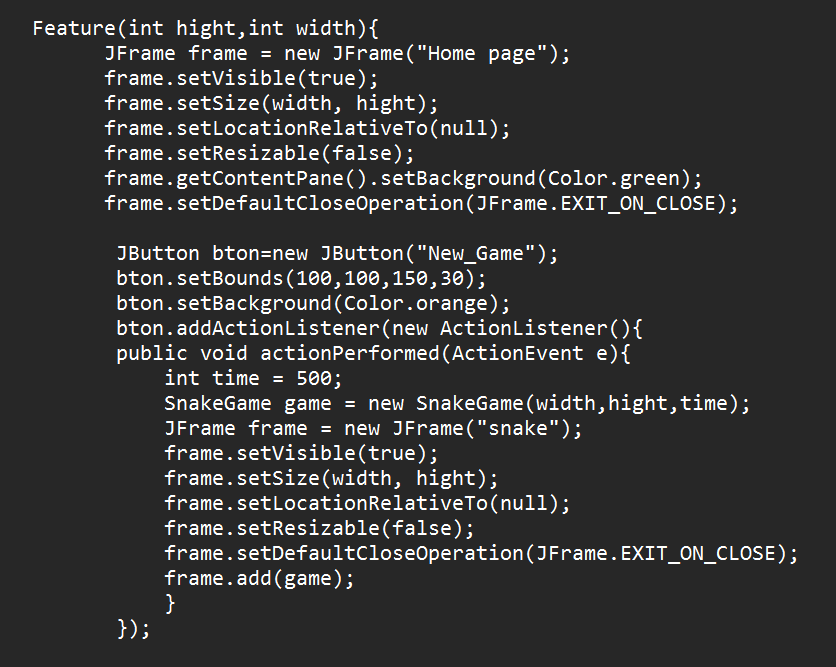
1. **Incomplete error handling:** Failing to handle errors properly can result in unpredictable behavior and errors. It can also make the code more difficult to test and maintain. Properly handling errors can make the code more reliable and easier to maintain.

Let’s talk an example of feature.java class:



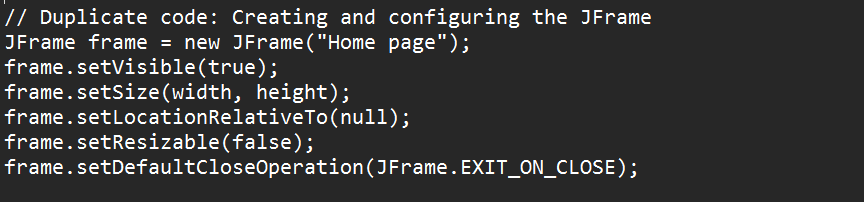
The problem with the code is that it doesn’t handle the error properly. If there is an error while making the API call, the function simply logs the error to the console and returns nothing. This can be a problem for the calling code, which might expect some meaningful response.

**2. Large or Complex Methods:** In the Feature class, the constructor method seems to be large and contains multiple sections of code for setting up different components.

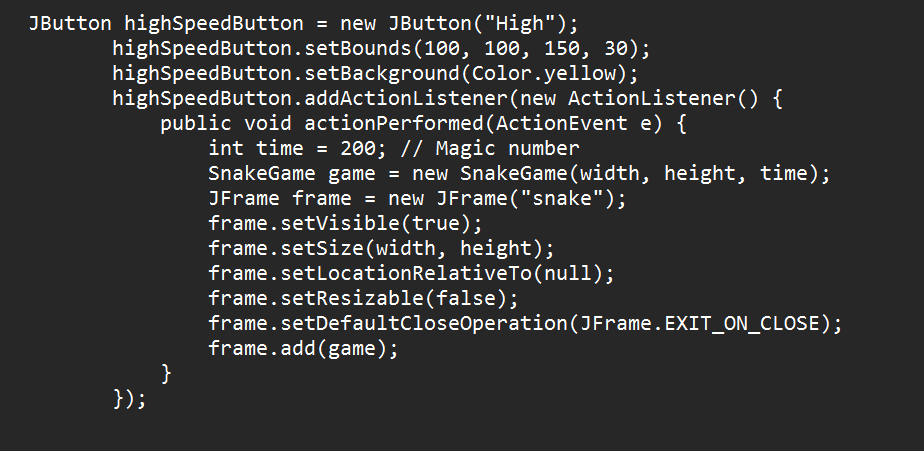
****

This code violates the Single Responsibility Principle (SRP) and is difficult to read and maintain.

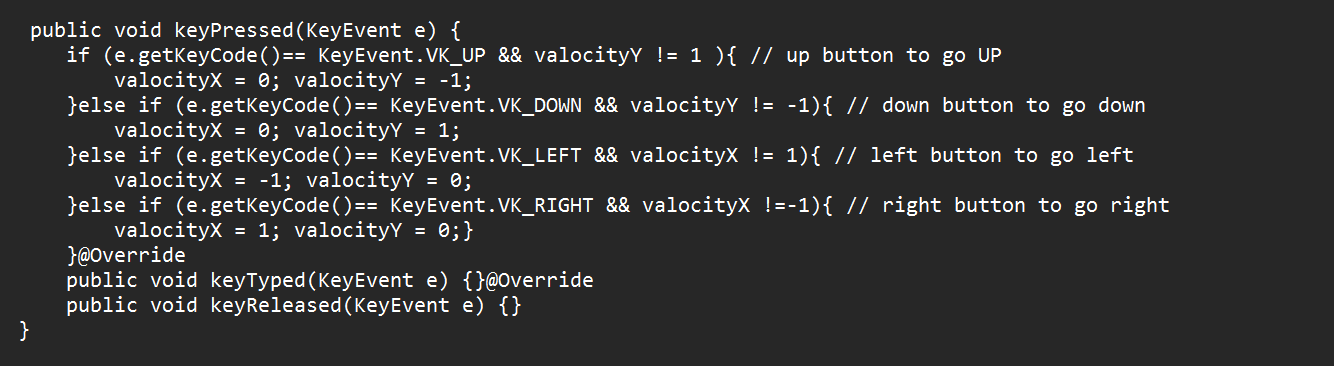
**3.** **Duplicate code:** Upon reviewing the code, one instance of duplicate code is the configuration of the JFrame in both the Feature class and the Speed class. Here's the duplicated code:



**4. Magic numbers or hard-coded values:** Using magic numbers or hard-coded values in code can make it more difficult to understand and maintain. The Speed class where magic numbers are used for time values:



**5. Too many if/else:** If/else statements can make code difficult to read and maintain, especially when they are nested or numerous. Code that contains too many if/else statements can be a sign of poor design and could benefit from refactoring using techniques like polymorphism or strategy patterns. The SnakeGame.java class contains too many if else.



**Architecture:** The code does not follow the defined architecture.

**Absence of Concern Separation**: In a layered design, various layers handle different concerns, like data access, display, and business logic. Nevertheless, these issues are not distinguished clearly in the provided code. For example, the Feature class mixes UI-related code (creating buttons and frames) with game logic (such as starting a game). Similarly, the SnakeGame class combines game logic with UI rendering.

**Violation of Single Responsibility Principle (SRP):** The classes in the provided code take on multiple responsibilities, making them harder to understand, maintain, and extend. For example, the SnakeGame class handles both game logic (e.g., collision detection, movement) and UI rendering (e.g., painting components), violating the SRP.

**Modularity check:**

Feature.java responsible for displaying all kinds features like: new game, speed, score history save, exit game etc. All features are implemented in different different methods.

Speed.java class maintain different speed of the game in different method.

SnakeGame.java responsible for visible the snake & food. It controls where the food is given and increases the size of snake.

At last, the game.java contains the main class.

Finally, I can say that the code is modular.

**Coding best practices:** The code does not fully adhere to some coding best practices:

**No hard coding, use constants/configuration values:** Hard-coded values are present in the code, such as time values (time = 200, time = 500, time = 1000). These should be replaced with named constants for better maintainability and flexibility.

**Avoid multiple if/else blocks:** The SnakeGame class contains multiple if/else blocks, particularly in the keyPressed method where key events are handled. Refactoring these if/else blocks into a cleaner structure, such as using a switch-case statement, could improve readability and maintainability.

Overall, while the code performs its intended functionality, there are opportunities for improvement in terms of adhering to coding best practices for readability, maintainability, and flexibility.

**Object-Oriented Analysis and Design (OOAD) Principles:** Let's examine how well the provided classes adhere to the OOAD (Object-Oriented Analysis and Design) principles:

**1. Single Responsibility Principle (SRP): ‘**SnakeGame’ class handles multiple responsibilities such as game logic, rendering, event handling, and user input. It could be refactored to separate these responsibilities into distinct classes to adhere to SRP more strictly.

**2. Open Closed Principle (OCP):** Similar to the ‘SnakeGame’ class, the ‘Speed’ class may require modifications to add new functionality, violating the OCP. Encapsulating GUI creation and event handling into separate modules could make the class more open for extension without modification.

**3. Liskov Substitution Principle (LSP):** No explicit inheritance relationships are evident in the provided code snippet. Therefore, the LSP cannot be evaluated.

**4. Interface Segregation Principle (ISP):** ‘SnakeGame’ class implements multiple interfaces (ActionListener, KeyListener) which may result in lengthy interfaces. It could benefit from segregating the interfaces into smaller, more focused ones**.**

**5. Dependency Injection:** The code does not demonstrate explicit dependency injection. Dependencies are tightly coupled within the classes, limiting flexibility and testability.

Overall, while the code partially adheres to some OOAD principles. Refactoring and redesigning the codebase could enhance its adherence to these principles and improve overall maintainability and extensibility.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Generic Checklist for Code Reviews:** | | | | | | |
| **Structure** | | | | | | |
|  |  | **Description of Item** | **Pass** | **Fail** | **Comments** |  |
|  | | | | |
| 1 | | Does the code completely and correctly implement the design? |  | ✓ | Not all circumstances may be handled by the code in accordance with the design specifications. | |
| 2 | | Does the code conform to any pertinent coding standards? | ✓ |  | There is no excessive comments. | |
| 3 | | Is the code well-structured, consistent in style, and consistently formatted? | ✓ |  | Consistent indentation. | |
| 4 | | Are there any uncalled or unneeded procedures or any unreachable code? |  | ✓ | There is some unreachable code in the Speed class | |
| 5 | | Can any code be replaced by calls to external reusable components or library functions? |  | ✓ | several parts of your code can be improved by using reusable components or library functions | |
| 6 | | Are there any blocks of repeated code that could be condensed into a single procedure? |  | ✓ | There are some repetitive patterns in the code, indicating opportunities for refactoring to condense repeated code into reusable procedures. | |
| 7 | | Is storage use efficient? |  | ✓ | To improve storage usage efficiency, you can reuse the existing JFrame instead of creating new ones. | |
| 8 | | Are symbolics used rather than “magic number” constants or string constants? | ✓ |  | Variables such as hight, width, and tilesize are used to represent dimensions rather than using direct numeric values. | |
| 9 | | Are any modules excessively complex and should be restructured or split into multiple routines? |  | ✓ | SnakeGame class contains overly complex functions, need to split. | |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Documentation** | | |  |  |  | |
|  |  | **Description of Item** | **Pass** | **Fail** | **Comments** |  |
|  | |  |  |  |
| 1 | | Is the code clearly and adequately documented with an easy-to-maintain commenting style? |  | ✓ | The code provided has some comments, but they are sparse and not consistently applied throughout the codebase. | |
| 2 | | Are all comments consistent with the code? |  | ✓ | Not all comments are consistent with the code. | |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Variables** | | |  |  |  | |
|  |  | **Description of Item** | **Pass** | **Fail** |  |  |
|  | |  |  |  |
| 1 | | Are all variables properly defined with meaningful, consistent, and clear names? |  | ✓ | Instead of time for the delay time in the ‘Feature’ class, consider defaultDelay or defaultGameDelay. | |
| 2 | | Do all assigned variables have proper type consistency or casting? | ✓ |  | All assigned variables have proper type consistency | |
| 3 | | Are there any redundant or unused variables? |  | ✓ | In the SnakeGame class, the variable gameOver is unused. The variable value is redundant. There are also some variable like this. | |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Style** | | |  |  |  | |
|  |  | **Description of Item** | **Pass** | **Fail** | **Comments** |  |
|  | |  |  |  |
| 1 | | Does the code follow the style guide for this project? | ✓ |  | The code seems to be a Java Swing application for a Snake game. However, there are several aspects of the code that don't adhere to standard Java style guidelines and best practices | |
| 2 | | Is the header information for each file and each function descriptive enough? |  | ✓ | There are no header comments in the file to evaluate. | |
| 3 | | Is there an appropriate number of comments?  (frequency, location, and level of detail) |  | ✓ | There are not enough comments in each file, so there's no commentary on their frequency, location, or level of detail. | |
| 4 | | Is the code well structured?  (typographically and functionally) | ✓ |  | The code structure appears to be fine. | |
| 5 | | Are the variable and function names descriptive and consistent in style? |  | ✓ | Variable and function names are inconsistent and not always descriptive. | |
| 6 | | Are "magic numbers" avoided? (use named constants rather than numbers) |  | ✓ | There are magic numbers present in the code. | |
| 7 | | Is there any “dead code” (commented out code or unreachable code) that should be removed? | ✓ |  | Here is no dead code in the provided section of the file. | |
| 8 | | Is it possible to remove any of the assembly language code, if present? | ✓ |  | There is no assembly language code in the provided section of the file. | |
| 9 | | Is the code too tricky? (Did you have to think hard to understand what it does?) | ✓ |  | SnakeGame class is very tricky. | |
| 10 | | Did you have to ask the author what the code does? (code should be selfexplanatory) |  | ✓ | Without proper documentation and clear naming conventions, it's difficult to discern the code's purpose without additional context. | |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Architecture** | | | | | | | | | |
|  |  | **Description of Item** | | **Pass** | | **Fail** | | **Comments** |  |
|  | | | | | | | |
| 1 | | Is the function too long? (e.g., longer than fits on one printed page) |  | | ✓ | | Feature method is too long. | | |
| 2 | | Can this code be reused? Should it be reusing something else? | ✓ | |  | | The code can be reused, but it would benefit from refactoring to improve its reusability, maintainability, and overall structure. | | |
| 3 | | Is there minimal use of global variables? Do all variables have minimum scope? | ✓ | |  | | The code minimizes the reliance on global variables and ensures that variables are appropriately scoped, promoting better code organization, readability, and maintainability. | | |
| 4 | | Are classes and functions that are doing related things grouped appropriately?  (cohesion) |  | | ✓ | | Classes and functions that are doing related things are grouped appropriately | | |
| 5 | | Is the code portable? (especially variable sizes, e.g., “int32” instead of “long”) | ✓ | |  | | The code appears to be portable. | | |
| 6 | | Are specific types used when possible?  (e.g., “unsigned” and typedef, not just "int") | ✓ | |  | | Specific types are used when possible. | | |
| 7 | | Are there any if/else structures nested more than two deep? (consecutive “else if” is OK) | ✓ | |  | | There are no if/else structures nested more than two deep in the. | | |
| 8 | | Are there nested switch or case statements? (they should never be nested) | ✓ | |  | | Here are no nested switch or case statements in the provided code snippet. | | |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Variables** | | |  |  |  | |
|  |  | **Description of Item** | **Pass** | **Fail** | **Comments** |  |
|  | |  |  |  |
| 1 | | Does the code avoid comparing floatingpoint numbers for equality? | ☓ | ☓ | Since the code doesn't involve numerical computations or comparisons of floating-point numbers, there's no explicit need to compare floating-point numbers for equality. | |
| 2 | | Does the code systematically prevent rounding errors? | ☓ | ☓ | Given the absence of explicit numerical computations or operations prone to rounding errors, there's no direct need to systematically prevent rounding errors in the code. | |
| 3 | | Does the code avoid additions and subtractions on numbers with greatly different magnitudes? | ☓ | ☓ | Since the code doesn't involve numerical computations or operations on numbers, there's no scenario where additions or subtractions with greatly different magnitudes would occur. | |
| 4 | | Are divisors tested for zero or noise? | ☓ | ☓ | The code doesn't perform division operations involving divisors that could potentially be zero or noisy. | |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Maintainability** | | | | | | |
|  |  | **Description of Item** | **Pass** | **Fail** | **Comments** |  |
|  | | | | |
| 1 | | Does the code make sense? | ✓ |  | Overall, the code makes sense and appears to function as intended for a text editor GUI application. | |
| 2 | | Does the code comply with the accepted Coding Conventions? | ✓ |  | The code generally follows java coding conventions. | |
| 3 | | Does the code comply with the accepted Best Practices? |  | ✓ | The code you provided does not fully comply with best practices. | |
| 4 | | Does the code comply with the accepted Comment Conventions? |  | ✓ | The code contains less comments. That’s why the code can’t explain complex logic or functionality. | |
| 5 | | Is the commenting clear and adequate? |  | ✓ | Comments are not clear and adequate. | |
| 6 | | Are ideas presented clearly in the code? | ✓ |  | Yes, the ideas are presented clearly. | |
| 7 | | Is encapsulation done properly? | ✓ |  | Encapsulation appears to be appropriately implemented, with functions and classes encapsulating related functionality. | |
| 8 | | Is the code not too complex? |  | ✓ | The code can be considered complex due to lack of modularity, repetitive Code, Hardcoded File Paths, Limited Error Handling. | |
| 9 | | Are there no unnecessary global variables? |  | ✓ | There are a few unnecessary global variables like: In the Speed class, frame is declared multiple times unnecessarily. | |
| 10 | | Is the reading order in source code from top to bottom? | ✓ |  | The code is organized in a top-to-bottom manner, with functions, classes, and imports appearing in a logical order. | |
| 11 | | Are there unused variables or functions? |  | ✓ | There are some unused variables but no any functions in the provided code. | |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Reusability** | | |  | |  | |
|  |  | **Description of Item** | **Pass** | **Fail** | **Comments** |  |
|  | |  | |  |
| 1 | | Are all available libraries being used effectively? | ✓ |  | The code effectively uses the library for creating the GUI components and handling user interactions. | |
| 2 | | Are available **OpenMRS** util methods known and used? | ☓ | ☓ | The code does not appear to utilize any **OpenMRS-**  specific utility methods, so this criterion is not applicable. | |
| 3 | | Is the code as generalized/abstracted as it could be? |  | ✓ | The code appears to be designed for a simple Snake game with Java Swing GUI. | |
| 4 | | Is the code a candidate for reusability? |  | ✓ | It serves its purpose but its reusability in other contexts may be limited. | |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Robustness** | | |  |  | | |
|  |  | **Description of Item** | **Pass** | **Fail** | **Comments** |  |
|  | |  |  | |
| 1 | | Are all parameters checked? | ✓ |  | The code seems to adequately check parameters where necessary to ensure validity and prevent errors. | |
| 2 | | Are error conditions caught? |  | ✓ | Error conditions areappropriately handled not throughout the code. | |
| 3 | | Is there a default case in all switch statements? | ✓ |  | All switch statements appear to include a default case, ensuring that all possible cases are covered. | |
| 4 | | Is there non-reentrant code in dangerous places? | ✓ |  | It serves its purpose but its reusability in other contexts may be limited. | |
| 5 | | Is the usage of macros proper? (Readability, complexity, portability...) | ✓ |  | There are no macros visible in the provided code section. If macros were used, their usage should be evaluated based on readability, complexity, and portability. | |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Error Handling** | | | | | | |
|  |  | **Description of Item** | **Pass** | **Fail** |  |  |
|  | | | | |
| 1 | | Does the code comply with the accepted Exception Handling Conventions? |  | ✓ | The code does not include exception handling, which is essential for robust error management and fault tolerance. | |
| 2 | | Does the code make use of exception handling? |  | ✓ | Exception handling is not implemented in the codebase, which could lead to runtime errors and unexpected behavior. | |
| 3 | | Does the code simply catch exceptions and log them? | ☓ | ☓ |  | |
|  | |  |  |  |  | |
| 4 | | Does the code catch general exception? | ☓ | ☓ |  | |
|  | |  |  |  |  | |
| 5 | | Does the code correctly impose conditions for "expected" values? | ☓ | ☓ |  | |
|  | |  |  |  |  | |
| 6 | | Are input parameters checked for proper values (sanity checking)? | ☓ | ☓ |  | |
|  | |  |  |  |  | |
| 7 | | Are error return codes/exception generated and passed back to the calling function? | ☓ | ☓ |  | |
|  | |  |  |  |  | |
| 8 | | Are error return codes/exceptions handled by the calling function? | ☓ | ☓ |  | |
|  | |  |  |  |  | |
| 9 | | Are null pointers and negative numbers handled properly? | ☓ | ☓ |  | |
|  | |  |  |  |  | |
| 10 | | Do switch statements have a default clause used for error detection? | ☓ | ☓ |  | |
|  | |  |  |  |  | |
| 11 | | Are arrays checked for out-of-range indexing? Are pointers similarly checked? | ☓ | ☓ |  | |
|  | |  |  |  |  | |
| 12 | | Is garbage collection being done properly, especially for errors/exceptions? | ☓ | ☓ |  | |
|  | |  |  |  |  | |
| 13 | | Is there a chance of mathematical overflow/underflow? | ☓ | ☓ |  | |
|  | |  |  |  |  | |
| 14 | | Are error conditions checked and logged? Are the error messages/codes meaningful? | ☓ | ☓ |  | |
|  | |  |  |  |  | |
| 15 | | Would an error handling structure such as try/catch be useful? (depends upon language) | ☓ | ☓ |  | |
|  | |  |  |  |  | |