Module: CS3318 – Advanced Programming with Java

Assignment 1: Technical Debt Analysis (Report)

Codebase: Risk Code

Introduction:

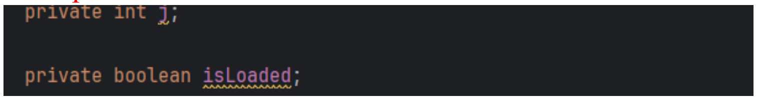
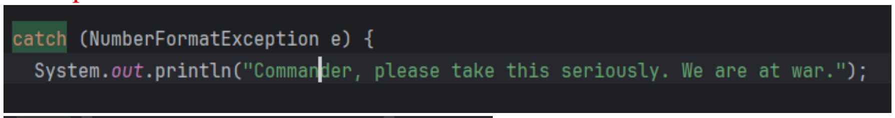
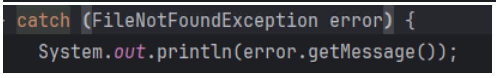
Technical debt refers to the compromises that are made in software development that allow for code to be developed and delivered to stakeholders quicker, done at the cost of future maintainability. For this report, we will look at the technical debt found within the Risk project code. For each type of technical debt, observations of each if found in the code will be discussed, the impact the presence of these debts have on the code, and improvements that can be made.

Examinations of the Code in the Different Types of Technical Debts:

1. Style Clash Debt:

Style clash debt refers to the technical debt caused by inconsistencies in code styling and design, which leads to reduced code readability and maintainability.

Observations:

- Inconsistent Programming Styles: Involves discrepancies that makes the code less uniform and harder to follow, such as inconsistent naming conventions (camelCase, PascalCase, etc), parameters being named with descriptive names vs short/less intuitive names, indentation discrepancies, etc.
  Example:
  
  ```java
  private int j;

  private boolean isLoaded;
  ```
  As we can see here, j is used to name a integer, while isLoaded names a Boolean. IsLoaded is descriptive, whereas j isn't. This is found using IntelliJ's find-in-files function (ctrl + shift + f) and searching for variable declarations.b
  Impact: Creates visual dissonance and can also slow down the process of others understanding the code during reviews or collaborations. It also suggests that many programmers may have work on the code together without all following a certain style guide.
  Improvement: Establish project-wide style guide. Also make use of IntelliJ IDEA's inspection setting to automatically enforce style consistency.
- Disdain for approaches that don't match one's own philosophy: Individual developers have their own preferences when it comes to things like error handling, data manipulation and validation. The code will reflect these varying philosophies. All leads to code where one section handles error with detailed logging and custom exceptions, while another uses minimal error handling.
  Example:
  
  ```java
  catch (NumberFormatException e) {
    System.out.println("Commander, please take this seriously. We are at war.");
  ```
  
  ```java
  catch (FileNotFoundException error) {
    System.out.println(error.getMessage());
  ```
  Here we have two different exception handling methods: one with a custom message, the other simply printing out the message from the exception itself. Identified by searching for and comparing error handling in the code using find-in-files feature. While a inconsistency exists

here, it is hard to prove whether this comes from a distain of approaches that don't match ones philosophy or a shift in the same developer's style (as all commits are made by the same developer). We would need to interview the developer to understand their reasoning behind this.
Impact: Leads to confusion about what the preferred method to use when handling similar scenarios. This will lead to further inconsistencies when further contributions are made.
Improvement: Develop clear guides for common coding tasks. Use code reviews to ensure adherence to the established practices.

Overview:

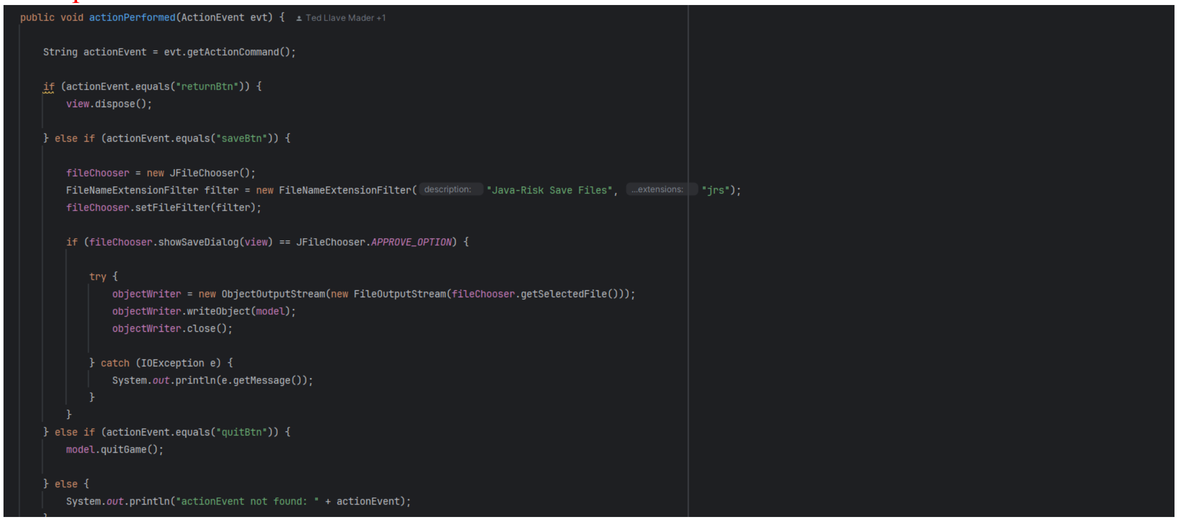
Occurs when there is a lack of consistency in the programming styles and architectural patterns throughout the code. Addressing style clash debt will involve establishing standardised coding guidelines, ensuring collaborative development environments, and promoting code consistency through regular review and automated tools.

2. Skill Deficit Debt:

Arises from lack of experience among developers, leading to code that is poorly structured, difficult to maintain, and inefficient.

Observations:

- Poorly structured, hard to navigate code: Code that lacks logical structure or organisation can be hard for developers to read and understand. Occurs with lack of clear formatting, organisation, logical flow, making it difficult for developers to follow code logic.
  Example:



```java
public void actionPerformed(ActionEvent evt) {  Ted Llave Mader +1

    String actionEvent = evt.getActionCommand();

    if (actionEvent.equals("returnBtn")) {
        view.dispose();

    } else if (actionEvent.equals("saveBtn")) {

        fileChooser = new JFileChooser();
        FileNameExtensionFilter filter = new FileNameExtensionFilter( description: "Java-Risk Save Files", ...extensions: "jrs");
        fileChooser.setFileFilter(filter);

        if (fileChooser.showSaveDialog(view) == JFileChooser.APPROVE_OPTION) {

            try {
                objectWriter = new ObjectOutputStream(new FileOutputStream(fileChooser.getSelectedFile()));
                objectWriter.writeObject(model);
                objectWriter.close();

            } catch (IOException e) {
                System.out.println(e.getMessage());
            }
        }
    } else if (actionEvent.equals("quitBtn")) {
        model.quitGame();

    } else {
        System.out.println("actionEvent not found: " + actionEvent);
```

  Here we see deep nesting with if-else statements and a mix of responsibilities – returning, saving, and quitting the game all in a single block, violating the Single Responsibility principle. This was found by conducting manual reviews of the code.
  Impact: Leads to slower development, increasing risk of errors, making onboarding new developers more difficult.
  Improvement: Consistent documentation, well defined method implementations, and maintaining code structure standards help enhance readability.
- Fragile systems prone to bugs and failures: When code lacks validation, error handling, or testing, it is prone to failure under unexpected circumstances. This is often due to lack of knowledge of how to implement robust error handling or safeguarding mechanisms.
  Impact: Fragile code can cause frequent errors, which can be hard to resolve, which means that greater maintenance efforts need to be taken.

Example: In the above code (actionPerformed) the catch blocks are merely print exceptions without providing any meaningful feeadback of attempting recovery.
Improvement: Error handling needs to be used. Regular code reviews can also help detect areas prone to bugs and failures.

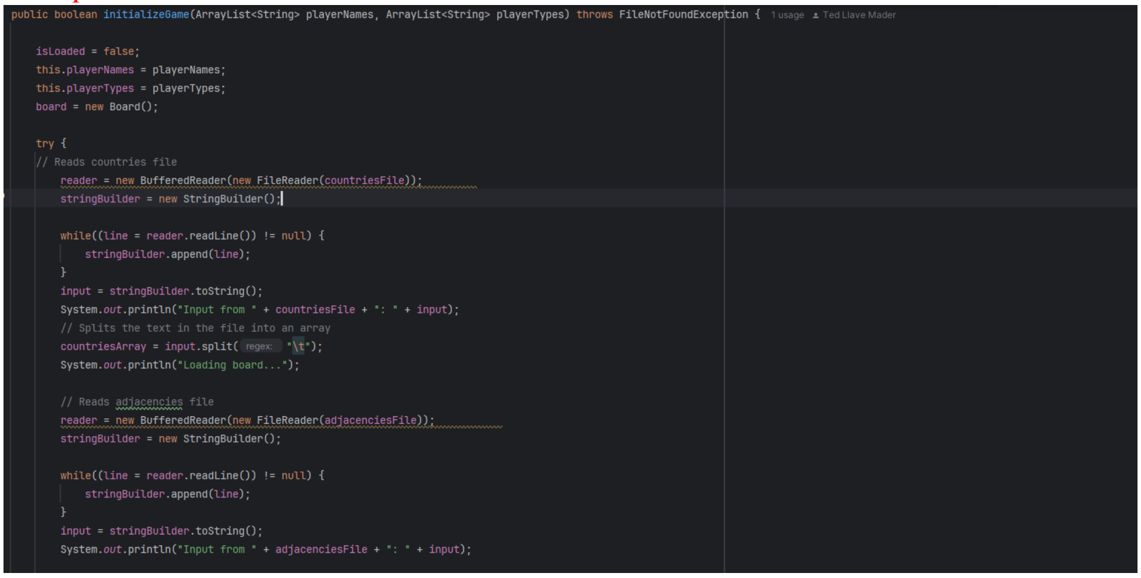
Overview:

Occurs when there is a lack of experience among developers, leading to hard-to-navigate code, which is prone to bugs. Addressing this requires continuous learning to ensure team members gain the necessary skills to produce high-quality code.

3. Strategic Debt:

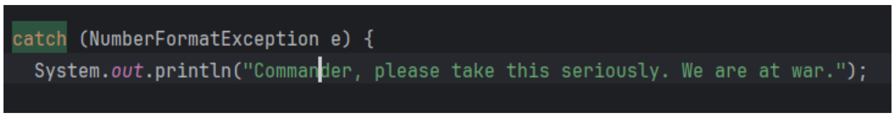
This occurs when deliberate shortcuts are taken to achieve quick results, such as meeting deadlines or releasing a product faster.

Observations:

- Architectural and testing shortcuts: To speed-up development, teams may skip testing, allowing for rapid feature delivery, at the code of that is harder to test and run.
  Example:
  

```java
public boolean initializeGame(ArrayList<String> playerNames, ArrayList<String> playerTypes) throws FileNotFoundException {   1 usage  ≛ Ted Llave Mader

    isLoaded = false;
    this.playerNames = playerNames;
    this.playerTypes = playerTypes;
    board = new Board();

    try {
    // Reads countries file
        reader = new BufferedReader(new FileReader(countriesFile));
        stringBuilder = new StringBuilder();

        while((line = reader.readLine()) != null) {
            stringBuilder.append(line);
        }
        input = stringBuilder.toString();
        System.out.println("Input from " + countriesFile + ": " + input);
        // Splits the text in the file into an array
        countriesArray = input.split( regex: "\t");
        System.out.println("Loading board...");

        // Reads adjacencies file
        reader = new BufferedReader(new FileReader(adjacenciesFile));
        stringBuilder = new StringBuilder();

        while((line = reader.readLine()) != null) {
            stringBuilder.append(line);
        }
        input = stringBuilder.toString();
        System.out.println("Input from " + adjacenciesFile + ": " + input);
```

  The length and complexity here show that the code was developed quickly without proper architectural planning. This shortcut can lead to difficulties in testing individual components. Found by analysing method lengths – used alt + 7 to access the structure to see the different method names.
  Impact: When code lacks test or well-designed structure, changes can increase unexpected bugs. This can increase the risk and cost of maintenance and makes it difficult to ensure reliability.
  Improvement: Through tests for crucial code and refactor code into smaller parts. Test-driven development may help this type of debt in the future.
- Lack of quality checks or failure handling: Involves bypassing quality checks or implementing minimal error handling to accelerate code development.
  Example:



```
catch (NumberFormatException e) {
   System.out.println("Commander, please take this seriously. We are at war.");
```

This simply prints messages but does not provide recovery options or informative logs, which can make it difficult for developers to identify and fix issues when they arise. Found using the find-in-files method again.
Impact: Can lead to vague error messages and makes fixing code difficult.
Improvement: Use logging of detailed messages and provide recovery options where applicable. Using exception handling and following best practices in error management should be used.

Overview:

This is where deliberate shortcuts are taken during the development process to achieve quick results. To address this, the code will need to be refactored, and any missing practices must be implemented to ensure long-term stability and maintainability.

4. Product Debt:

Refers to code that no longer aligns with user needs or have become outdated due to evolving requirements. This type of debt can lead to bugs and incidents related to stale functionality.

Through examination of the Risk codebase, product debt has not been observed.

Here are some causes of product debt:

- Features no longer aligned to current user goals: As user needs evolve, certain features or logic in the code may become obsolete or misaligned with what the application currently requires – keeping this code is unnecessary and leads to confusion. Can make developers maintain and modify code that isn't needed. Leading to waste of time and contributes to code clutter. To improve this we would conduct code reviews to remove or update obsolete features.
- Extraneous complexity from outdated requirements: Code grows in complexity as new features are added to meet evolving requirements. When these change, old logic that was once necessary may remain, leading to complexity that no longer serves its original purpose. Makes code harder to read, understand, maintain. Developers will need to spend extra time deciphering code. To improve this we need to simplify or remove outdated logic. Maintain modular designs that can be updated more easily as requirements change.

To identify product debt:

- User feedback could be used to understand their current needs and pain points.
- By examining how the different features are being utilised by users, we can use this see where users' expectations are not being met.
- Conducting market research and addressing trends can highlight where the Risk codebase is lacking when it comes to its user's needs.

Experiences of Code Analysis Tools for Identifying Technical Debt:

Utilising IntelliJ IDEA's built in code analysis tools has been instrumental in identifying and managing technical debt. Using these has allowed for real time insights into code quality and allows developers to be pro-active in maintenance and improvement of their code.

Here is IntelliJ IDEA's tools used to complete this report:

- Code inspection: Found under Code > Inspect code, and then deciding desired scope (typically chose whole project). Allows for detection issues such as potential bugs. They will continuously run as you write code and gives immediate feedback.

  How this was used for the report is after running the inspection, the results will appear in the 'Inspection Results' tool window. Issues are categorised by severity and type, which has allowed for efficient navigation and review.

  Some common examples of inspection results were code styling issues, pointless Boolean expressions and unused declarations.
- Version Control Analysis: Found using Alt + 9, this allows the review of commit history, allowing for the tracing of evolution of the codebase.

  This was checked to see if there were any areas with frequent changes or bug fixes, indicating potential instability. It also allowed for confirmation that only one person created and developed the codebase.
- Search Functionality: Using 'Find in Files' feature (Ctrl + Shift + F), searched occurred for specific patterns and keywords. This enabled the locating of exception handling, variable declarations to identify potential problem areas.
- Manual Code Review: Systematically examined the code to access its structure and adherence to best practices. This allowed to identify issues such as mixed responsibility, deep nesting, etc.

Conclusion:

In this analysis of the Risk codebase, several forms of technical debt have been observed. These issues manifest as inconsistent programming styles, poorly structured code, and shortcuts when it comes to testing. To address these, standardised coding guidelines, implementing robust error handling mechanisms, adopting best practices in software development is crucial. By doing this, the Risk codebase can achieve enhanced quality, maintainability, and adaptability to future requirements.