Università della Svizzera Italiana

Evaluating the Design of a Project

Pablo Landrove Pérez-Gorgoroso

November 26, 2025

1 Project Selection and Scope

The preceding assignments involved analyzing the design flaws and patterns within the **Google Guava** library. While Guava offers a valuable case study in robust software design, its extensive size and highly optimized structure make deep refactoring infeasible for the scope of this project.

For this refactoring assignment, we have therefore selected one of our own prior projects: a Java implementation of the board game *Risk*. This project encapsulates some of the game logic, utilizing *standard input* for reading player commands such as viewing the board, assigning armies and countries to players.

The primary reasons for choosing this codebase are:

- **Language Alignment**: It is written in Java, which allows us to effectively demonstrate the use of object-oriented principles, specifically *inheritance* and *polymorphism*, in the refactoring process.
- **Codebase Familiarity**: As a known project from prior academic work, it significantly reduces the time required for initial codebase analysis, allowing us to focus on deep structural modifications.
- **Manageable Scope**: The project's moderate scale facilitates substantial, impactful modifications that would be impossible in a larger commercial library.
- **Refactoring Potential**: Preliminary analysis indicates numerous opportunities for applying refactoring techniques to improve code quality, maintainability, and design adherence.

Some details of the project are:

- 37 Java files.
- 6 abstract classes.
- 1 interface.
- Around 3,200 lines of code.
- More than 600 comments.

Although the project is relatively small in size, its structure reveals several design weaknesses. In particular, the very limited use of interfaces and the reliance on only a few abstract classes indicate a suboptimal application of Java's object-oriented features and inheritance mechanisms. Many responsibilities are concentrated in concrete classes, and polymorphism is used only minimally. As a result, the codebase exhibits rigid dependencies and duplicated logic, leaving significant room for improvement. This makes the project an excellent candidate for refactoring: it is compact enough to be fully understood and restructured, yet sufficiently flawed in its design to benefit greatly from a more principled use of abstraction, encapsulation, and modular decomposition.

2 Goals of our refactoring

The primary motivation for the refactoring process was to improve the overall structure and maintainability of the application. During the initial analysis of the codebase, we identified that the `Menu` class acted as a *god class*, concentrating responsibilities such as game logic, game state management, and user input processing. This high degree of coupling made the system difficult to understand and modify,. A major goal of the refactoring was therefore to decompose this class into smaller, more cohesive components with clearly defined responsibilities.

We also made use of the PMD static analysis tool. PMD helped us identify potential design flaws, such as excessive class size, duplicated code, long switch statements, and violations of object-oriented design principles. These insights served as valuable indicators of where the system suffered from poor separation of concerns or insufficient modularity. In Table 2.1 we can see a summary of the PMD output, counting each of the flaws that where found.

We can see that there is a very high number of *LooseCoupling* violations. These refer to uses of implemenations of an interface instead of the interface itself, for example using *ArrayList* instead of *List* or *HashMap* instead of *Map*. This makes the code highly coupled with a specific implementation, making it harder to add new features that would require another type of *List* or *Map*.

There is also a high number of violations of the *LiteralsFirstInComparisons* rule. These occur when a *String* variable is compared to a literal using `variable.equals("literal")`. This pattern is not null-safe, since calling `equals` on a potentially null variable may lead to a `NullPointerException`. The recommended and safer practice is to place the literal first, i.e., `"literal".equals(variable)`, which avoids such failures even when the variable is `null`. We modified all the occurrences of these violations throughout the whole codebase.

3 Refactoring examples

3.1 Menu God Class

Originally, the `Menu` class acted as a *god class*, concentrating both the user interface logic and the complete game state in Class attributes. The class consists of a loop to read user input and a 300 line switch statement to select the code for running each command. This design hindered maintainability and made further extensions increasingly difficult. As an initial step toward decomposing this overly-centralized structure, we separated the game state from the menu-related functionality. Although this refactor does not yet represent

Flaw	Count
LooseCoupling	71
LiteralsFirstInComparisons	57
UnusedAssignment	37
ExceptionAsFlowControl	27
OneDeclarationPerLine	20
CyclomaticComplexity	19
CognitiveComplexity	13
RelianceOnDefaultCharset	9
PrimitiveWrapperInstantiation	8
NcssCount	7
AvoidPrintStackTrace	7
NPathComplexity	6
NonExhaustiveSwitch	4
AvoidCatchingGenericException	4
ForLoopCanBeForeach	3
UseUtilityClass	3
GodClass	2
SwitchDensity	2
UseCollectionIsEmpty	2
UnusedLocalVariable	2
CouplingBetweenObjects	1
TooManyMethods	1
TooManyFields	1
SimplifyBooleanExpressions	1
UnusedPrivateField	1
MutableStaticState	1
	1

Table 2.1: Design Patterns reported by Pattern4J.

the final architectural solution, it provides a cleaner separation of concerns and creates a more modular foundation. This intermediate step significantly facilitates the subsequent restructuring phases, where additional responsibilities will be delegated to dedicated components.

To address the switch statement responsible for parsing all user input and dispatching the corresponding actions we made use of the Command and Factory patterns. We refactored the input system by introducing a *CommandFactory* 1 Class, which processes the input with a list of Classes extending a *CommandParser* abstract class, each responsible for recognizing a specific type of command. These parsers instantiate objects implementing the *Command* interface, which encapsulate the actual behavior associated with each action. This new architecture improves modularity, enhances testability, and aligns the design more closely with established object-oriented principles. This represents a substantial improvement over the previous implementation.

3.2 Exceptions Duplicated Code

In the original implementation, the exceptions defined within the `errors` package contained a significant amount of duplicated code and included methods with high cognitive

Code 1: CommandFactory Implementation

```java
public class CommandFactory {
    private final List<CommandParser> parsers;
    public CommandFactory() {
        // Initialize the parser list by adding instances of
            CommandParser
    }
    public Command createCommand(String rawInput) throws
        CommandException {
         Command out;
         for (CommandParser parser: this.parsers) {
             out = parser.parse(rawInput);
             if (out != null)
                 return out;
         }
    }
}
```

Code 2: CommandParser Implementation

```java
public abstract class CommandParser {
    String commandName;
    String[] arguments;
    void splitCommand(String input) {
         String[] parts = input.trim().split("\\s+");
         this.commandName = parts[0];
         this.arguments = (parts.length > 1) ? java.util.Arrays.
            copyOfRange(parts, 1, parts.legth) : new String[0];
    }
    public static void throwIncorrectCommand() throws
        CommandException {
         CommandException error = new CommandException("101");
         throw (error);
    }
    public abstract Command parse(String input) throws
        CommandException;
}
```

Code 3: GeoException Before

```
public class GeoException extends Exception{
    GeoException(String code){
        super(code);
    }
    public void printError(){
        String out = new String();
        switch(super.getMessage()){
            case "100":
                out = "{\nerror code: 100,\ndescription: \"Color no
                    permitido\"\n}";
                break;
            ...
            case "112":
                out = "{\nerror code: 112,\ndescription: \"Los
                   paises no son border\"\n}";
                break;
            default:
                out = super.getMessage();
                break;
        }
        Risk.console.Print(out);
    }
}
```

complexity. In particular, the `printError()` method used a large `switch` statement to determine the appropriate error message for each exception, making the code difficult to maintain and extend.

As part of the refactoring, we introduced an abstract class `MyException`, which centralizes the storage of the error code and corresponding error message, and provides a single method to print the error. Each concrete exception now passes its error code to the `MyException` constructor, which retrieves the appropriate message from a *map*. This approach eliminates repetitive switch statements, simplifies the exception classes, and makes it easier to modify or add new error messages in the future. Overall, the new structure reduces complexity, improves maintainability, and aligns better with object-oriented design principles. We can see a comparison between the code before and after the refactor in Snippets 3 and 4

4 Ensuring Behavioral Consistency During Refactoring

Throughout the refactoring process, it was essential to guarantee that no functional changes were introduced and that the program's behavior remained identical to the original implementation. To achieve this, we adopted a systematic testing strategy based on input–output comparison.

First, we produced a representative sequence of commands that exercises the main functionalities of the program. We executed the original, pre-refactoring version of the application with this input and stored the resulting output as the *expected* reference.

After each refactoring step, we used a custom Python script to automatically compile the program, run it with the same predefined command sequence, and capture its output. The

Code 4: GeoException After

```
public class GeoException extends MyException {
    private static final Map<String, String> ERROR_MESSAGES = Map.
        ofEntries(
         Map.entry("100", "Color no permitido"),
         ...
         Map.entry("112", "Los paises no son border")
    );
    public GeoException(String code){
         this.errorCode = code;
         this.errorMessage = ERROR_MESSAGES.get(code);
    }

}
```

script then compared the newly generated output with the expected one using a line-by-line diff. If both outputs matched, we could be confident that the refactoring had not altered the program's external behavior. Any discrepancy immediately signaled a regression or unintended modification.

This automated procedure allowed us to validate every intermediate refactoring step, ensuring that structural improvements did not compromise correctness. As a result, we maintained a high level of assurance that the final refactored version behaves exactly as the original program.

5 Discussion of Refactoring Challenges

The most challenging aspect of the refactoring process was decoupling the `Menu` class into multiple components. Originally, the `Menu` class concentrated a wide range of responsibilities, including game state management, user input parsing, and command execution. Separating these concerns required careful analysis to ensure that the behavior of the system remained unchanged, as well as finding opportunities for abstraction and polymorphism.

In contrast, some refactoring tasks were relatively straightforward:

- Replacing string literals in comparisons to follow null-safe practices.
- Switching from concrete types (like `ArrayList` or `HashMap`) to their corresponding interfaces (`List`, `Map`).

These changes improved code quality and maintainability with minimal risk of introducing errors.

6 Assessment of Further Improvements

Although the current refactoring significantly improved the structure of the code, there remain areas that could benefit from additional decomposition. In particular the classes:

- `Game`: Holds most of the state and is highly coupled with the rest of the program.
- `Map`: Contains a high amount of complex and very long methods, so it could benefit from extracting these methods.

A deep refactoring of these components would require a substantial amount of time and effort and is particularly challenging because it is difficult to make structural changes without inadvertently altering the existing functionality. As a result, this more extensive restructuring was considered beyond the scope of the current project, but it could be pursued in future work to achieve a fully modular design.