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 a prior project: a Java implementation of the board game *Risk*. This project encapsulates all the necessary game logic, utilizing *standard input* for player commands such such as rolling dice, playing action cards, and moving armies.

1.1 Selection

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.

1.2 Scope

The existing Risk implementation is composed of 27 Java classes, including 6 abstract classes and one interface, totaling approximately $2,300$ lines of code (LOC). This size provides sufficient complexity to necessitate significant design improvements while remaining contained enough for comprehensive refactoring within the assignment's timeframe.

2 Goals of Refactoring: What Aspects We Aim to Improve and Why

The primary motivation for the refactoring process was to improve the overall structure, maintainability, and extensibility 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, error-prone to modify, and resistant to future evolution. 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* flaws. 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*.

3 Concrete examples of refactoring before and after

Originally, the `Menu` class acted as a *god class*, concentrating both the user interface logic and the complete game state. 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 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.

The `Menu` class also contained a 300+ line switch statement responsible for parsing all user input and dispatching the corresponding actions. Such a monolithic structure was hard to read, error-prone, and strongly coupled the input-handling logic to the rest of the class. To address this issue, we refactored the input system by introducing a set of Classes extending the *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.

Flaw	Count
LooseCoupling	71
SystemPrintln	59
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.

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 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 what was harder and what was simpler to refactor

6 Assesment of what else could be improved but wasn't