Multi Threaded Card Game Report

Amirali Famili

November 14, 2023

Introduction

This report provides a brief explanation and justification regarding the design choices made for source code, it's implementation and tests associated with the project.
This report has three main sections ; design choices and performance issues, test suites and problems associated with the code and design of the tests.

Production Code Design & Problems

After the publication of the specification, me and my partner analysed the requirements and design choices that we had, we soon realized that the coding part shouldn't be a problem with this project but the design of the code is the most important aspect of the code, we had to plan carefully how to approach the game.
We decided to use the V-module for the development of the project since we could write the code and test it concurrently which meant we can develop the project more thoroughly and with caution, however we also took advantage of the waterfall design approach, we started by writing the code essential for the generation of the pack and distributing it's content to players and decks along with some helper methods implemented in a class called Card (Card.java), then we began testing the functionality of the Card class and made sure that it's working properly.
With Card class done we had some data that we can manipulate, we started working more on the structure of the code, we decided to make the Card class as automated as possible, then we began working on the player class and tried to build the structure of the game, Although we had some confusion about how to assign each player with a thread of their own in a round robin fashion, we first decided to build a game which had real players, so we can better understand the game structure and what the threads would have to do if it was automated, soon it became clear that this game can easily get caught in a race condition due to the circular structure of the game play (round robin) and the fact that players are using a shared resource (their decks) to keep the game going.
We decided to make a thread-pool of size players to demonstrate the thread associated to each player, this was a challenging problem since we were implementing the game inside the player class and we needed to have a run method so that threads could be executed, here we chose to use a nested class represented as player Thread, using a simple for loop we

executed player threads.
Here we were faced with big problems players were playing in an unpredictable order and the code would crash after a few seconds by getting caught in a race condition, after searching for solutions in lecture notes and the internet we noticed that we can take advantage of many methods and tools to control the behaviour of the threads, in our case we used synchronized blocks and lock objects to control and restrict the access of players to the shared resources. After many attempts and tests we finally got result and the game no longer got caught in a race condition, however we still had one problem, when a player won the game they weren't able to notify all players successfully and as a result in some runs of the game we would have more than one winner, the problem seemed to be that when a player matches all cards in their hand and they notified other players, some players were still in the middle of their turn and they had to finish their current turn before being able to respond properly.
We attempted to address this problem by notifying players from another location within the code but we would often get an error saying that the current thread is not the owner and therefore they can't notify other players, after many failed attempts we decided to address this problem by checking the win condition in all the helper methods that players were interacting with then we got result and got the game running. Not long after we began unit testing the game to make sure that it's working properly and we don't get an unsuspected error in time, we ran the test and initiated the game countless times to make sure that it would perform properly and wouldn't cause any errors or crash due to unstable nature of the threads.
After finishing the main game we started attempting to print the data to the files in the correct format and since access to shared resources was already restricted with synchronized blocks and locks we placed the code for writing to the files in the same place and made sure that it wouldn't be any conflict between threads for writing to the files, since we could get an fatal error regarding that matter and crash the program.
However, after implementing output files as well we had no concern nor problems and started reviewing the overall design of the code and how these methods and classes should communicate with each other.

Unit Testing & Design Choice

Testing the code was a challenging process, neither of us was familiar with unit testing and aspects related to it so we started researching and finding sample test online and from workshops, we also went through the lecture notes and with all that information in our minds we decided to use JUnit 4 for implementing the test suites of our project.
The reason for our choice is that it was recomended in the lecture notes to do so and there were many examples on how we can write tests with JUnit 4. Moreover we noticed that JUnit 4 has only one huge package which we can place in our tests folder and take advantage from by running our tests more easily, in other words we wouldn't have to import specific libraries which we don't know and it's difficult to implement by using JUnit 5, we could avoid complexity and since we didn't developed our project with java 7 which JUnit 5 supports, we thought it's unnecessary to go through the trouble of using JUnit 5 and we were satisfied with the properties and services that JUnit 4 offered.
After finding the right testing framework we started writing the test code for our classes, for simplicity we decided mimic the structure of the code that we already wrote into our test suites, therefore we created 3 test Classes (since we only have 3 main classes for running the game), and as mentioned in the lecture notes we followed the correct naming conviction by adding the word "test" in the beginning of all our test cases , classes and methods (even the file names).
Writing the test code was a challenging part of the project we had to think of every possible scenario were something might go wrong, since most of the code is automated and self sufficient we had to manipulate many instances of the main classes to make sure that all methods are independent and they can handle any problem that could possibly occur.
For testing the generator code in card class (Card.java), we didn't had much problem since all the input that we were getting were integers and there aren't many scenarios where something might go wrong, we rapped around many parts of the code that we suspected something might go wrong in try and catch claws and we tested the methods thoroughly with unusual input that we manually entered as mock objects and used built in methods of JUnit 4 such as assertEquals and assertTrue to make sure that we are reciving the correct output from our methods.

One of the most challenging aspects of Unit testing was testing the main game method, despite the fact that main game method is not receiving any values and is working with predefined instances of the Player class and we had to manipulate the values within the Player class, we were also face with the problem of multi threading and the fact that the game is running in it's own nested class.

Some of the test we could think of were changing the values of the hand and decks whilst the game was playing and checking to see if it causes the game to crash or overloading the game with entering a large number as the number of players.
One error that we raised during our tests for which we didn't find an answer for was an error related to memory, that the system doesn't have enough space for storing the values, another interesting error that we faced was an IndexOutOfBounds error which we could fix with many attempts of restricting threads access to it, eventually whilst we were doing the

unit test we pin pointed the cause of the problem, it turned out that we had many print statements in our main game method (for debugging purposes) and since printing out many data to the terminal is computationally expensive, the compiler usually crashed at some point, but after removing them the game improved it's robustness.
We discovered many problems and bugs in the project whilst doing the Unit test from which we were able to better structure the code, find and fix many known and unknown issues and make sure that our code works fine if we run it multiple times event with the unnatural behaviour of the threads.
Unit testing also made us better developers by making us think of all the possible problems that could cause and all possible paths that the compiler might take, or traps that it can get stock on and cause a live project to fail, these learning made us realize the importance of Unite testing as a part of software development project.