Sudoku Architectural Documentation

Overview of rules:

1. Fill a 9 x 9 matrix with numbers 1 to 9.
2. Each 3 x 3 sub matrix contains all digits no duplicates.
3. Each row contains all digits no duplicates.
4. Each column contains all digits no duplicates.

Algorithm Design Choices

Steps 2 through 4 are each boolean checks we can implement as helper functions to run as we iterate through potential solutions. The function names are:

1. isRowSafe
2. isColSafe
3. isSubMatSafe

The search algorithm itself is a backtracking algorithm that works similarly to Depth first search, using the stack frame to backtrack since it is recursive.

The basis step of the recursive function is to check if there are no more opencells left in the Sudoku matrix. If completely filled we end execution. Otherwise we search for a number between 1 and 9 (inclusive) that satisfies the isSafe rules. We then try to place the number we think is safe in the matrix. If we can make progress with our solution given this number we re-run solve to confirm and reset the cell placement if the confirmation is negated.

The puzzle generator used to test SudokuSolver works similarly, with a few caveats. First when we solve we solve with shuffled numbers between 1 and 9 instead of iterating between 1 and 9. Second, I use a derivative of the backtracking algorithm to instead check if I can randomly remove an index in the matrix and still have a unique solution.

Implementation Design Choices

Much of the code is written using the Java 8 stream apis. The stream apis were a conscious choice in the advent that this project may need to support larger Sudoku matrices and parallelism could be readily available to me by just calling the parallel() function. Further, for testing I extended the SudokuSolver class to also generate puzzles and lambdas are a great way to force side-effect free code by preventing local and member variable modification inside a lambda function scope. Other than that it was largely an aesthetic choice to both more cleanly and succinctly describe the behavior of each function.

Build System

The build system is setup with gradle. The intent was to make the build system an active part of producing cleaner code. As such every build runs a linter, a static code analyzer for security best practices, junit tests, and code coverage monitoring. This data can be found in the build/reports/ portion of every project. The plugins used to pull this off are:

1. Checkstyle (linter using google's style sheet)
2. FindBugs (static code analyzer)
3. Findsecbugs (plugin to findbugs, checks 80 additional vulnerability types)
4. ErrorProne (google static analysis)
5. Jacoco (code coverage)

The build system also generates documentation and can fix some style check violations. See `gradle task` for more info on options.

Continuous Integration:

This project uses Travis.CI to test builds remain building and passing tests. It uses lgtm to run static code analysis and score the code quality. Further code coverage is tracked on every commit using codecov.io.

Testing

Testing is done using Juint. Both the easy and difficult tests were included to start. They were good baselines to confirm I had a working solution, however they were not very exhaustive. To have more exhaustive testing I started by using an existing data set of sudoku csvs. To do this I wrote a bare bones csv reader, with minimal error handling. This was not sufficient testing for me, as I wanted something that would be more random while testing. In past compiler and schema validation I have done we develop expression generators and fuzzers to expand test coverage. I wanted something similar, but assuming valid Sudoku puzzles. So I wrote a Puzzle generator that would generate puzzles with a single solution and then return those. I would then try and solve the puzzle with the Sudoku solver and compare the results to the puzzle solution key. 10 of these tests are done per build.

Difficulties

I decided to go with a simple solution, so the hardest part was to prove to myself the solution was correct. There was some trial in error on the PuzzleGenerator class seeing which functions needed to be overridden and what pieces needed to be randomized inorder to not only generate new puzzles, but to generate puzzles with different solutions while still maintaining the requirement that there was only one unique solution. I needed this so that when I compared the output of the SolutionSolver to the PuzzleGenerators key it would always match.

Future work

I know this isn't the most optimal solution in a world where you can solve the puzzle with dancing links, Constraint Satisfaction (CSPs), or and optimization problem using Integer Linear Programing (ILPs). However, while these maybe more optimal they were not intuitive so I chose not to re-write the solver. If the prompt was expanded further I would investigate these methods for a future re-write.