Matthew Smith

Derek Schissler

**Clue Refactor Comments**

ISSUE #1: Didn’t declare a type for Set or Map in instance variable declaration and in function return types. While the code works, you want to include the type that will be stored in these variables so that you, other programmers, and the complier can know more about what each Set or Map can and should include. In the case of your code, not declaring the type did result in an error. In the non-default constructor, you instantiated targets as a HashSet containing Integer types, when in fact it should contain BoardCell type objects.

EXAMPLE:

**private** Set targets;

**public** Board(String mapName, String legendName) {

/\*

\* OTHER CODE ...

\*/

targets = **new** HashSet<Integer>();

}

SOLUTION: Add appropriate type for each declaration of Set or Map.

**private** Set<BoardCell> targets;

**public** Board(String mapName, String legendName) {

/\*

\* OTHER CODE ...

\*/

targets = **new** HashSet<BoardCell>();

}

ISSUE #2: Code in Board constructor was used to determine whether or not it was Rader’s test files or your own test files. This is not a big issue but solutions like this often lead to unintentional consequences that sometimes become difficult to trace when the programmer forgets to change the variable (raderTests in this case).

EXAMPLE:

**if** (raderTests) {

mapName = "ClueLayout.csv"; //used in CR tests

legendName = "ClueLegend.txt";

} **else** {

mapName = "ClueMap.csv"; //default names- change to your own

legendName = "legend.txt";

}

SOLUTION: It is a much more appropriate solution to use the non-default constructor to determine which files to load. Here, we deleted the code above (left the default names part) from the default constructor and changed the code in CR\* test files to load Rader’s config files using the non-default constructor.

*board* = **new** Board("ClueLayout.csv", "ClueLegend.txt");

ISSUE #3: Code in the “error” function in BadConfigFormatException was repeated. You should always look for ways to simplify functions like this to avoid repeated code. As well, the function name “error” is not very self-explanatory.

EXAMPLE:

**public** **void** error() {

**try** {

FileWriter fstream = **new** FileWriter("ErrorLog.txt");

BufferedWriter out = **new** BufferedWriter(fstream);

out.write("An input file is in an invalid format. Check the documents.");

out.close();

} **catch** (Exception e) {

System.*out*.println("Error: " + e.getMessage());

}

}

**public** **void** error(String name) {

**try** {

FileWriter fstream = **new** FileWriter("ErrorLog.txt");

BufferedWriter out = **new** BufferedWriter(fstream);

out.write(name + " is in an invalid format. Check the document.");

out.close();

} **catch** (Exception e) {

System.*out*.println("Error: " + e.getMessage());

}

}

SOLUTION: We combined the error functions into one, and used the constructor call super to create the different error messages. We then used these messages as the lines that get written to the log file. We also re-named the function to “logError” to give ourselves and other programmers a bit more info on what the function does.

**public** BadConfigFormatException() {

**super**("An input file is in an invalid format. Check the documents.");

logError();

}

**public** BadConfigFormatException(String fileName) {

**super**(fileName + " is in an invalid format. Check the document.");

logError();

}

**public** **void** logError() {

**try** {

FileWriter fstream = **new** FileWriter("ErrorLog.txt");

BufferedWriter out = **new** BufferedWriter(fstream);

out.write(getMessage());

out.close();

}

**catch** (Exception e) {

System.*out*.println("Error: " + e.getMessage());

}

}

ISSUE #4: The function getAdjList should just be a getter. In your code, you performed the calculations for the adjacency list within getAdjList. This means that every call to getAdjList would recalculate adjacencies. Processing can be saved by calculating the list a single time, then using getAdjList to return the pre-calculated list.

EXAMPLE:

**public** LinkedList<Integer> getAdjList(**int** index) {

// CALCULATE CODE

}

SOLUTION: We created a new function, calcCellAdjacency, to perform the calculations. Then we changed getAdjList to be a getter that returns the list that was calculated in calcCellAdjacency.

**private** LinkedList<Integer> calcCellAdjacency(**int** index) {

// CALCULATE CODE

}

**public** LinkedList<Integer> getAdjList(**int** index) {

**return** adjMtx.get(index);

}

ISSUE #5: Somewhat of a code repetition in the getRoomCellAt and getCellAt function. You have two versions of each function, one for a single (int cell) parameter and another for two (int row, int col) parameters. Instead of repeating similar code, it is a good idea to use one function within the other. Then if/when the function needs to be modified, it only needs to be changed in one location.

EXAMPLE:

**public** RoomCell getRoomCellAt(**int** row, **int** col) {

**int** index = calcIndex(row, col);

**if** (cells.get(index) **instanceof** RoomCell)

**return** (RoomCell) cells.get(index);

**else**

**return** **null**;

}

**public** RoomCell getRoomCellAt(**int** index) {

**if** (cells.get(index) **instanceof** RoomCell)

**return** (RoomCell) cells.get(index);

**else**

**return** **null**;

}

**public** BoardCell getCellAt(**int** row, **int** col) {

**int** index = calcIndex(row, col);

**return** cells.get(index);

}

**public** BoardCell getCellAt(**int** index) {

**return** cells.get(index);

}

SOLUTION: We took advantage of the calcIndex function to use the single parameter function as the base function. What I mean is that in the two parameter function, we call calcIndex to get a cell index that can be passed to the single parameter version of the function.

**public** RoomCell getRoomCellAt(**int** row, **int** col) {

**int** index = calcIndex(row, col);

**return** getRoomCellAt(index);

}

**public** RoomCell getRoomCellAt(**int** index) {

**if** (cells.get(index) **instanceof** RoomCell)

**return** (RoomCell) cells.get(index);

**else**

**return** **null**;

}

**public** BoardCell getCellAt(**int** row, **int** col) {

**int** index = calcIndex(row, col);

**return** getCellAt(index);

}

**public** BoardCell getCellAt(**int** index) {

**return** cells.get(index);

}

ISSUE #6: Logic in calcTargets is needlessly complex. In your code you iterated through adjacent cells, and if some conditions were met, then you added it to another “unvisited” array that would be iterated through again. From a performance standpoint, this is not ideal because we are iterating more than necessary.

EXAMPLE:

**for** (Integer i : adjMtx.get(index))

**if** (visited[i] == **false**) unvisited.add(i);

**for** (**int** i : unvisited) {

// SOME CODE

}

SOLUTION: We used the same iteration and conditions from the first of your iterations, but then used the code within the second of your iterations. This saves a length of the unvisited array number of iterations.

**for** (Integer i : adjMtx.get(index)) {

**if** (visited[i] == **false**) {

// SOME CODE

}

}