Overview

We have implemented a set of classes to solve the LogicGridPuzzle, in which we provide several categories, each associated with specific values. After creating these categories, various clues are provided. Based on these clues, we fill the grid with cross and tick marks, eventually revealing relationships between categories and solving the puzzle. Furthermore, in this puzzle, there exists a one-to-one relation for each value of a category to every other value of a different category. Please refer to the assignment specification for details on the required methods.

The problem decomposes into four main components: filling the grid based on clues, handling the mapping of each value of every category, and after completing all the clues filling the remaining grid boxes using various logics, and creating a solution for the grid.

A LogicGridPuzzle handles two types of clues: One utilizes a method called valuePossibilities(), in which the input contains a value on which we have to perform an operation. For that value, what are the possible options? This set determines what the things, value like or might be like. The other type of clue uses a method called listOrder(), in which we have two values. The method also provides information about the order gap between those two values, similar to mathematical concepts. Finally, it contains the category name to which the order gap belongs.

Files and External Data

The implementation is divided into 6 classes (Figure 1):

- LogicGridPuzzle- It the driver class that contains all the necessary methods, such as adding a category, adding a clue, checking a solution, and solving the puzzle.

- LogicHelper - Checks whether the input clue, options, and exclusions are valid. If they are valid, it then calls the functions represented below, which implement the necessary processes.

- Clue - Contains the implementation for adding clues to the global structure that holds all the past inputted clues.

- Mapping - Contains the code that generates the mapping of existing data, applies ticks or crosses to that mapping, and also includes the implementation for filling the remaining boxes by deducing logic from all the provided clues.

- Solution - Implementation for creating a solution, which helps in the check and solve methods, and includes the method to check whether the given solution in the check function is valid or not as per the given categories.

- Utils - Contains methods for input validation, which are used by all the above methods.

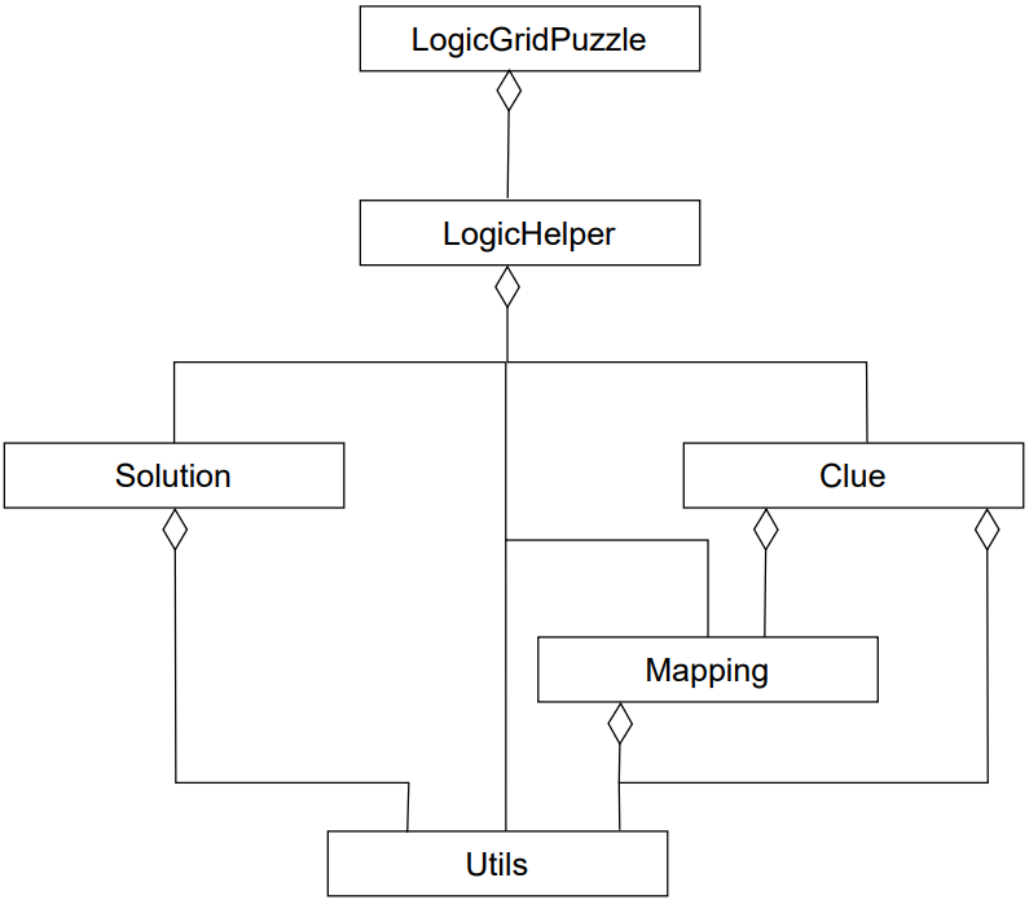

Figure 1 Class interactions

Data structures and their relations to each other

Data

- Contain the category name and all the values of that category.
- It will be created in the LogicHelper class and set as a global variable so that every method of that class access and use it.
- I implement this data in Map<String, List<String>> in which the key is the category name, and the list contains the values of that category.

DataMapping

- Contain on-to-one mapping for each value of one category to all the values of remaining categories.
- I implement this mapping in Map<String, Map<String, String>> in which the key of the outer map is the value from the category and all the values other than that category are set in the inner map with the initial value set as null.
- This will create in the same LogicHelper class as the above data created but it will only fill if the solve and check function is called.

OptionMap and ExclusionMap

- It will store the clues of this puzzle, the optionMap store options from the value possibilities, and the exclusionMap store the exclusions from value possibilities and list order.
- When the solve or check method is called this map is used to fill the data mapping.
- I implement this in Map<String, Set<String>> to store the values, where the key is the base value from value possibilities and the first or second value from list order while the set contains the option, exclusion, and values from orderGap logic.

Assumptions

- In check() method the given solution is a full solution, so no partial solution is allowed, and no null value is present in the solution.
- Clues are not misleading the puzzle means two puzzles do not contradict each other.
- All the strings given in inputs are case-sensitive.
- The puzzle contains only a single correct solution.

Choices

- I don’t use the state space approach to reach the solution instead I use the elimination method.
- If in options two values of different categories are present I consider both to be correct.
- I choose the Map to implement this entire puzzle.
- I choose to return null if any single null is present in the solution generated in check() function.
- I didn’t apply the clues as they come, I store them and when check or solve is called I apply all the clues to the mapping and then generate a solution. I choose this to counter the state space approach.
- I choose null, “yes”, and “no” to show the status in the mapping.

Key algorithms and design elements

AutoFillRowCol

1. Traverse through data mapping.
2. Now for each mapping iterate through every category except the category of this mapping key.
3. Now make the count of the number of null, yes, and no for each category
4. If the number of “no” for that particular category is equal to the number of values present in each category minus 1 then mark yes in that remaining null value.

(If A=B and B=C, than A=C)
IntutionMappingForTick

1. Traverse through data mapping.
2. Now for each value of yes we go to that mapping
3. Now for each yes from this mapping we mark yes from the key from step 1 and the key of yes in this iteration.

(If A=B and A!=C, than B!=C)
IntutionMappingForCross

1. Traverse through data mapping.
2. Now for each mapping make a map values of yes and no
3. Now for each yes make no for every value present in the no map for this particular mapping.

Limitations

- This puzzle only handles string inputs.
- If one clue contradicts with another clue then this puzzle is not able to handle that situation.
- After a certain level, human intervention is required to solve this puzzle.
- Only handle two types of clues, while in real life we can give many ways to give clues that confuse the user and make the puzzle harder.

Test Cases

setCategory(String categoryName, List<String> categoryValues)

Input validation

- Both category name and category values are null
- Empty category name
- Empty categories value

Boundary cases

- Add one category that contains only one value

Control flow

- Call for categories that already exist.
- Call for categories having repeated values.
- Call for categories having different sizes.

Data flow

- Call before using any other function of this puzzle.

valuePossibilities(String baseValue, Set<String> options, Set<String> exclusions)

Input validation

- All values are null or empty
- Options are null or empty
- Exclusions are empty

- Both option and exclusion are empty
- Base value is null or empty while option and exclusion are present.

Boundary cases
- Read data only one value is present in either option or exclusion
- Read data when one value for every category except the category of base value.

Control flow
- Call for the same base value present either option and exclusion.
- Call for the same category as the base value belongs, present in either option or exclusion.
- Call for the base value which is not present in existing data.
- Call for the values present in option or exclusion not present in existing data.
- Call for the values in which at least one value common in option and exclusion

Data flow
- Call after setCategory()
- Call before check() and solve()

listOrder(String orderCategory, String firstValue, String secondValue, int orderGap)

Input validation
- All values are null or empty
- First value is null or empty
- Second value is null or empty
- Both First and Second value are null or empty
- Order Category is null or empty

Boundary cases
- Read data with orderGap as a negative value.
- Read data with orderGap as a positive value.
- Read data with orderGap as a zero.

Control flow
- Call when orderGap is less than the size of values present in each category.
- Call when orderGap is more than the size of values present in each category.
- Call when the first value and the second value are from the same category.
- Call when orderCategory, first value, and second value all are from the same category.
- Call when anyone from ordercategory, the first value and second value is not present in existing data.

Data flow

- Call after setCategory()
- Call before valuePossibilities()
- Call after valuePossibilities()
- Call before check() and solve()

solve(String rowCategory)

Input validation

- Null parameter
- Empty Parameter

Boundary cases

- Read data when the category contains only one value
- Read data when the category contains multiple values

Control flow

- Call before single call of valuePossibilities() and listOrder()
- Call with the rowCategory not present in existing data.
- Call when the partial solution is present
- Call when the entire solution is obtained
- Call when two boxes are already marked and the third automatically gets a tick.
- Call when more deduction is required from the above-provided clues.

Data flow

- Call after valuePossibilties() and listOrder().
- Call before solve()
- Call after solve()

check(String rowCategory, Map<String, Map<String, String>> solution)

Input validation

- Null category
- Empty category
- Null solution provided.
- Empty solution provided.

Boundary cases

- Read data when the category contains only one value
- Read data when the category contains multiple values

Control flow

- Call before single call of valuePossibilities() and listOrder()
- Call with the rowCategory not present in existing data.
- Call when the partial solution is present
- Call when the entire solution is obtained
- Call when two boxes are already marked and the third automatically gets a tick.
- Call when more deduction is required from the above-provided clues.
- Call with the solution containing a null value.
- Call with the solution is different from the values present in existing data.
- Call with the solution for a different category from what it passed in the rowCategory parameter.

Data flow

- Call after valuePossibilties() and listOrder().
- Call before check()
- Call after check()