**Lebanese American University**

Text

Description automatically generated

COE599F – Test. For Digital Integrated Circuits

Dr. Maria Abi Saad

**Final Project**

- Ali El Dor ID: 202002180

- Mohammad Al Moukhallalati ID: 202000311

Thursday 21/12/2023

Table of Contents

[Introduction: 4](#_Toc154049368)

[Phase 1: Benchmark Parsing: 5](#_Toc154049369)

[Wire and WireType Classes: 5](#_Toc154049370)

[Gate class: 8](#_Toc154049371)

[Circuit class: 14](#_Toc154049372)

[Phase 2: Logic Simulation: 20](#_Toc154049373)

[Phase 3: Fault Simulation: 23](#_Toc154049374)

[Results: 31](#_Toc154049375)

[Conclusion: 35](#_Toc154049376)

Table of Figures

[Figure 1: Wire class part 1. 6](#_Toc154049383)

[Figure 2: Wire class part 2. 7](#_Toc154049384)

[Figure 3: Wire class part 3. 7](#_Toc154049385)

[Figure 4: WireType enum class. 8](#_Toc154049386)

[Figure 5: Gate class part 1. 9](#_Toc154049387)

[Figure 6: Gate class part 2. 10](#_Toc154049388)

[Figure 7: NAND Gate class. 11](#_Toc154049389)

[Figure 8: AND Gate class 11](#_Toc154049390)

[Figure 9: NOR Gate class. 12](#_Toc154049391)

[Figure 10: OR Gate class. 12](#_Toc154049392)

[Figure 11: XNOR Gate class. 13](#_Toc154049393)

[Figure 12: XOR Gate class. 13](#_Toc154049394)

[Figure 13: NOT Gate class. 14](#_Toc154049395)

[Figure 14: Circuit class parameters. 15](#_Toc154049396)

[Figure 15: c17 benchmark. 15](#_Toc154049397)

[Figure 16: c17 benchmark in the code. 16](#_Toc154049398)

[Figure 17: Parsing method part 1. 17](#_Toc154049399)

[Figure 18: Parsing method part 2. 17](#_Toc154049400)

[Figure 19: Parsing method part 3. 18](#_Toc154049401)

[Figure 20: 3 crucial lines of code. 18](#_Toc154049402)

[Figure 21:replaceWireInGates method. 19](#_Toc154049403)

[Figure 22: Printing wires methods. 19](#_Toc154049404)

[Figure 23: Printing fanout wires. 20](#_Toc154049405)

[Figure 24: Methods for logic simulation part 1. 21](#_Toc154049406)

[Figure 25: Methods for Logic simulation part 2. 21](#_Toc154049407)

[Figure 26: createGate method. 22](#_Toc154049408)

[Figure 27: User Interface methods part 1. 22](#_Toc154049409)

[Figure 28: User Interface methods part 2. 23](#_Toc154049410)

[Figure 29: User Interface methods part 3. 23](#_Toc154049411)

[Figure 30: Serial Fault Simulator algorithm. 24](#_Toc154049412)

[Figure 31: Options 1 and 2 code. 25](#_Toc154049413)

[Figure 32: Options 1 and 2 code part 2. 25](#_Toc154049414)

[Figure 33: Options 1 and 2 code part 3. 26](#_Toc154049415)

[Figure 34: Options 1 and 2 code part 4. 26](#_Toc154049416)

[Figure 35: Options 1 and 2 code part 5. 27](#_Toc154049417)

[Figure 36: Options 1 and 2 code part 6. 27](#_Toc154049418)

[Figure 37: Serial fault simulator code part 1. 29](#_Toc154049419)

[Figure 38: Serial fault simulator code part 2. 29](#_Toc154049420)

[Figure 39: Serial fault simulator code part 3. 30](#_Toc154049421)

[Figure 40: compareOutputs method. 30](#_Toc154049422)

[Figure 41: processFault and reportAllFaults methods. 31](#_Toc154049423)

[Figure 42: getInputWiresCount method. 31](#_Toc154049424)

[Figure 43: c17 circuit format. 32](#_Toc154049425)

[Figure 44: circuit c17 results part 1. 32](#_Toc154049426)

[Figure 45: circuit c17 results part 2. 33](#_Toc154049427)

[Figure 46: Another circuit format. 33](#_Toc154049428)

[Figure 47: Circuit results part 1. 34](#_Toc154049429)

[Figure 48: Circuit results part 2. 34](#_Toc154049430)

[Figure 49: Last circuit format. 35](#_Toc154049431)

[Figure 50: Last circuit results. 35](#_Toc154049432)

# Introduction:

In this project, we are required to build a fault simulator circuit where this will be done using 3 phases. In the first phase, we will be able to parse the benchmark given in order to interpret the circuit and gates used for this benchmark. In the second phase, we will simulate the true-value simulation. In the third and last phase, we will develop a serial fault simulator in order to determine whether the different stuck at faults can be detected or not. Our project is written in Java and is made up of 13 classes which we will elaborate on in the report.

# Phase 1: Benchmark Parsing:

In this phase, we were able to parse benchmarks as explained later in this section. It should be noted that we took a few assumptions into considerations which are: all gates will be 2-input gates (except NOR gate which will take one input), and the benchmark will be inputted as a String and will be handled in a Circuit class. Moreover, we assumed we are simulating single stuck-at-faults.

For this phase, we used 12 classes (all except the FaultResult class) to parse the benchmark.

## Wire and WireType Classes:

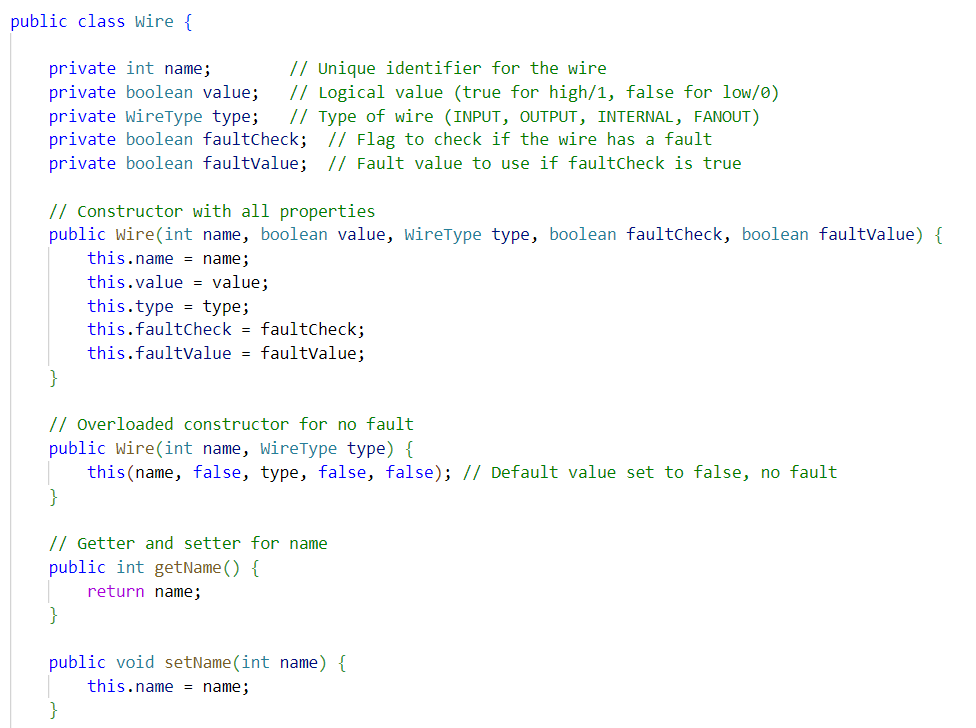


Figure : Wire class part 1.

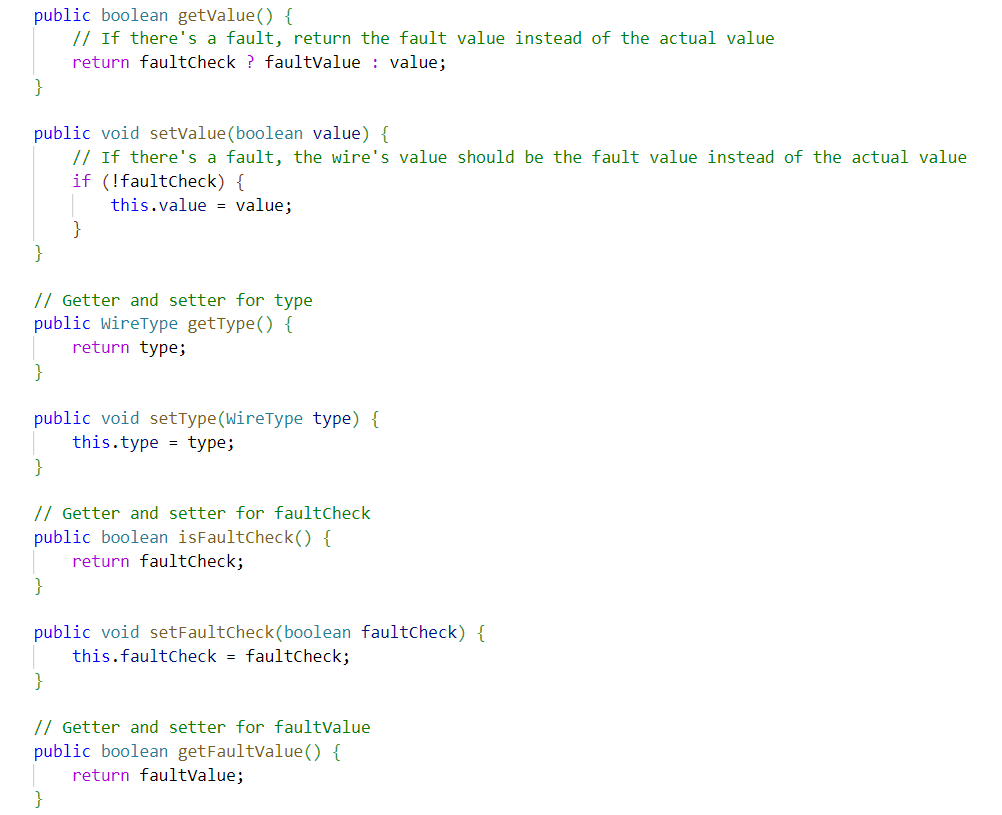


Figure : Wire class part 2.

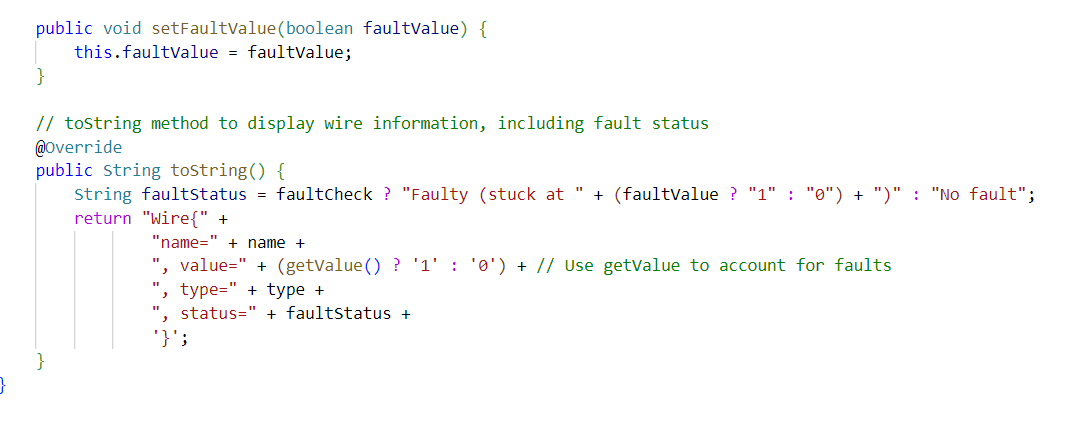


Figure : Wire class part 3.

This Wire Object class takes as parameters name of the wire (1, 11, 23), its Boolean value (logic 0 or 1), type of wire which is of type Object WireType class (whether the wire is an input wire, output wire, branch of fanout, or internal wire), and Boolean faultCheck and faultValue. It should be noted that faultCheck and faultValue are flags which will be used in the third phase of the project which is fault simulation. They are essential flags to determine if the specific wire has a fault (faultCheck), and to determine the value of the fault, logical 0 (false) or 1 (true) (faultValue). Also, there will be getters and setters for each parameter.

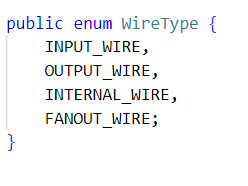


Figure : WireType enum class.

## Gate class:

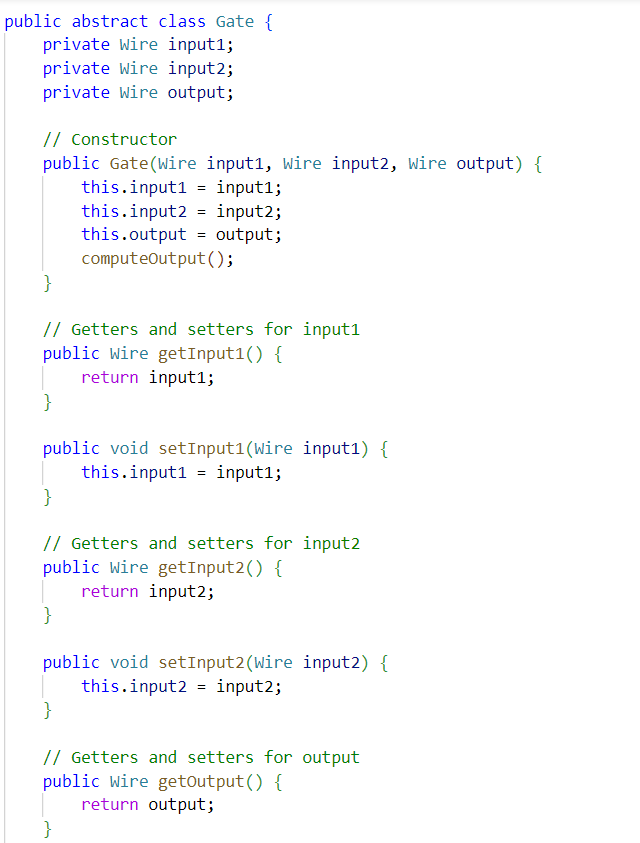


Figure : Gate class part 1.



Figure : Gate class part 2.

The Gate Object class represents the parent of the different types of gates used in the given benchmarks and their inputs and outputs. The Gate Object class takes as parameters the inputs and output of each gate of type Object Wire, and there will be getters and setters for each parameter. As shown in figure 6, the enum class GateType represents the different gates which will be used in the benchmarks and in the project.

Now we will take a look at these GateType classes with the following screenshots where each of the gates will be assigned their proper functionality and these classes will extend or inherit the Gate class.

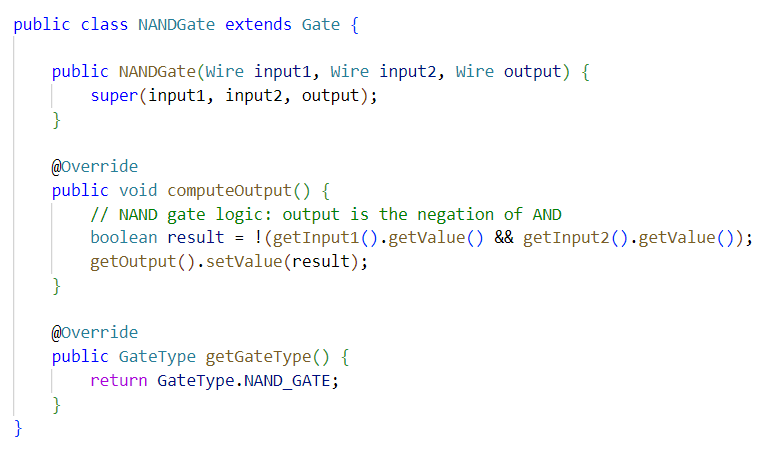


Figure : NAND Gate class.

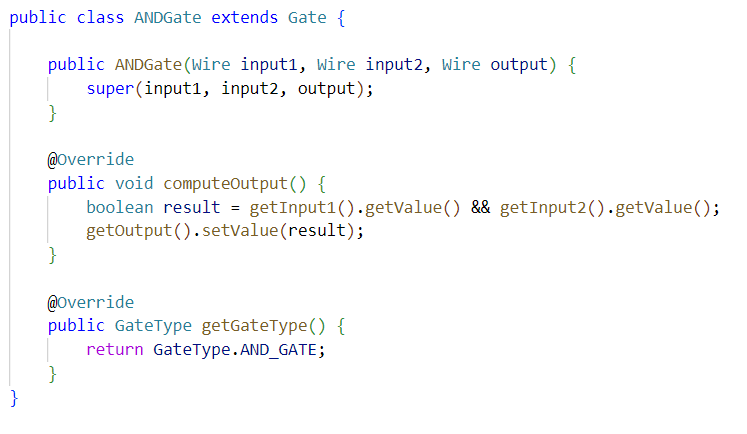


Figure : AND Gate class

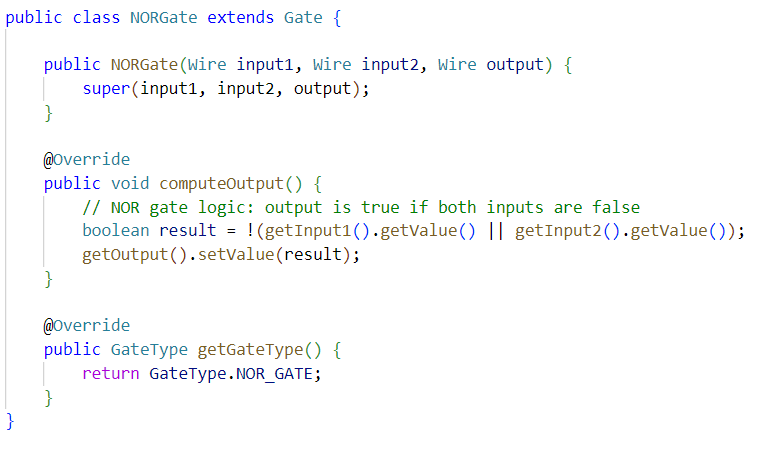


Figure : NOR Gate class.

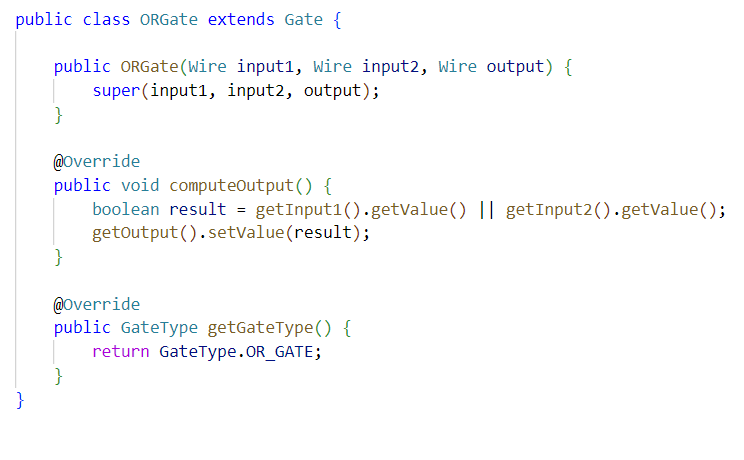


Figure : OR Gate class.

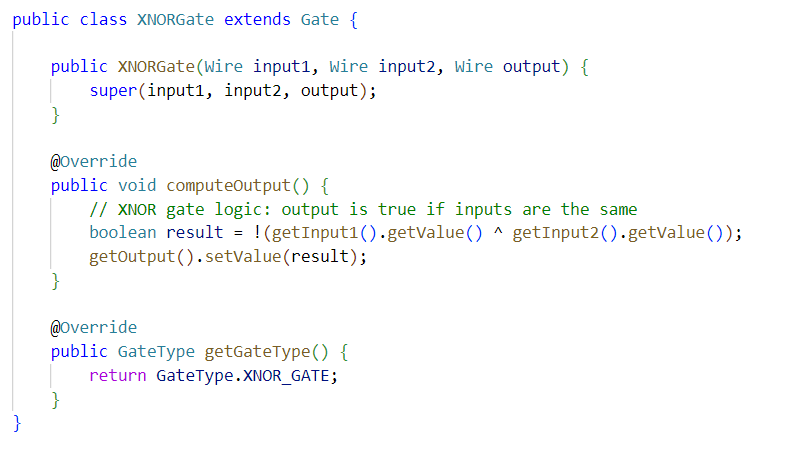


Figure : XNOR Gate class.

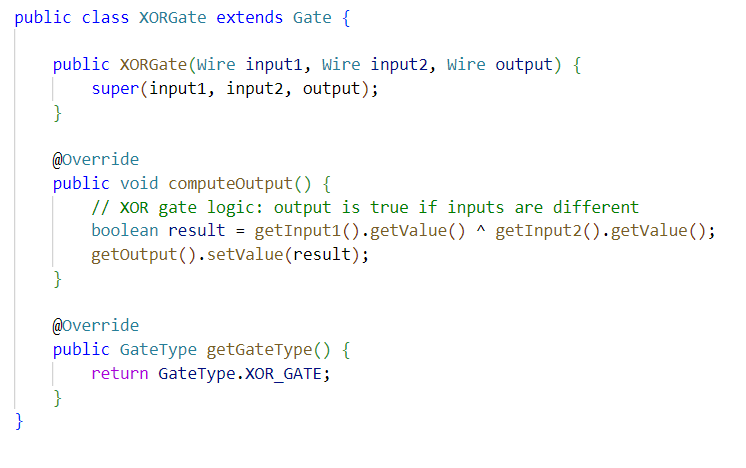


Figure : XOR Gate class.

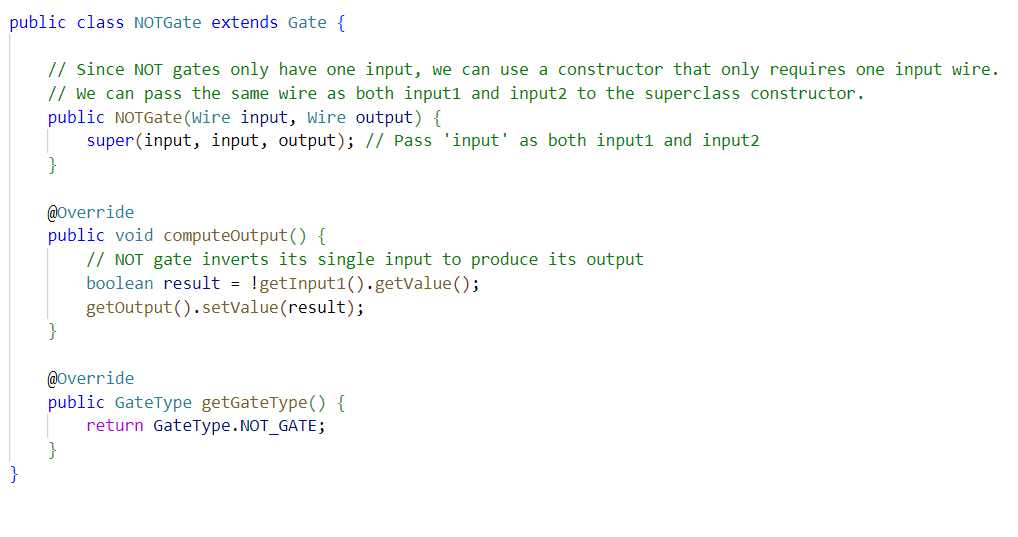


Figure : NOT Gate class.

It should be noted that in figure 13 showing the NOT Gate class, the NOTGate constructor takes one input parameter which is called input instead of 2 inputs for the other gates called input1 and input2. This is because the NOT gate takes only one input.

## Circuit class:

This class is the most important class in our project as it contains all the methods used for all of the phases. We will first introduce the class then provide explanation for the methods related to each phase alone.



Figure : Circuit class parameters.

The Circuit class takes as parameter definition which is the input file or string of the benchmark as seen in the figure below as example.

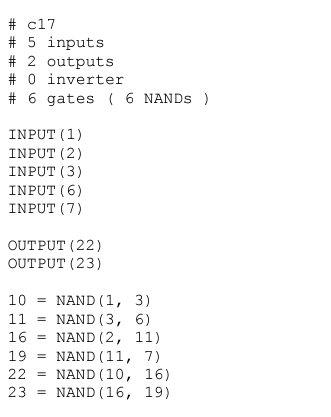


Figure : c17 benchmark.

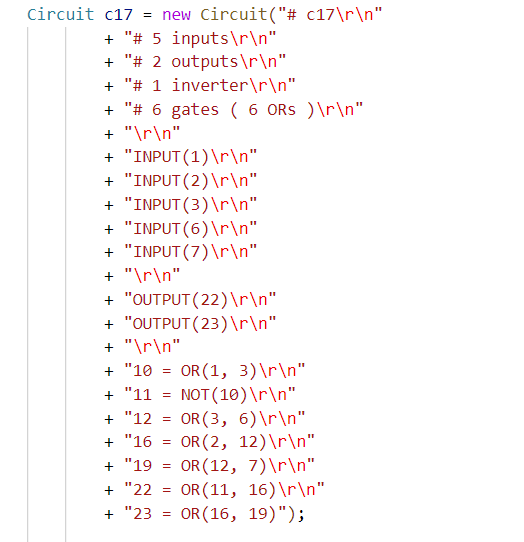


Figure : c17 benchmark in the code.

The private inputs for this class are: String name which will be the name of the benchmark, Map<Integer, Wire> wireMap, which is a map having a key of type integer (wire number) and value of type Wire (the wire object for this wire), List<Gate> which is the list of gates used, and private Map<Integer, List<Wire>> fanoutWireMap, which is the map used for fanout wires having key of type integer (wire number) and value which is a list of type Wire (Wire Object class for fanouts).



Figure : Parsing method part 1.

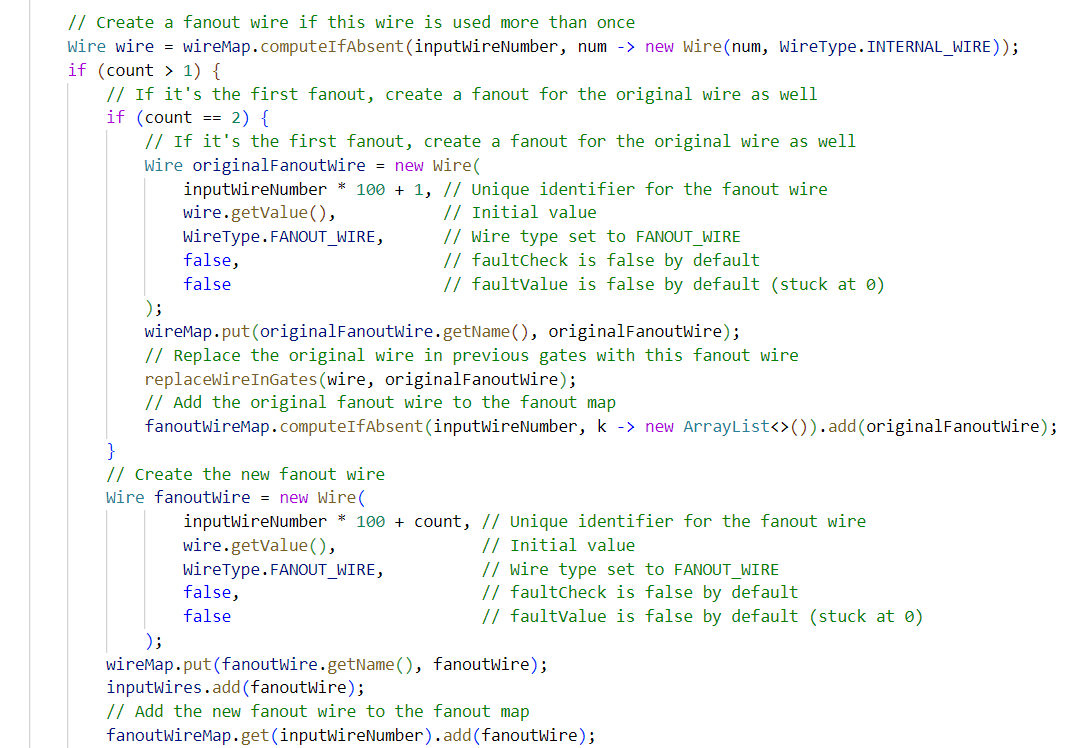


Figure : Parsing method part 2.

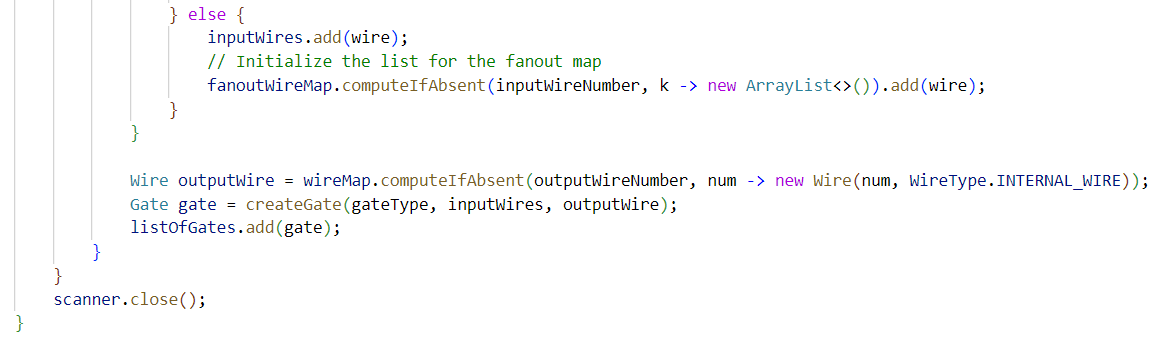


Figure : Parsing method part 3.

This method is the method used for parsing the benchmark file. At first, the already existing Java Pattern class was used in order to determine which wires are inputs, outputs, and gates for the benchmark when scanning the file. Moreover, the fanoutCounts map was used in order to determine the number of fanouts a wire has (if count=1 it has no fanouts but if more, it does have fanouts). Next, we iterate over each line of the benchmark string and use the already existing Java Object class Matcher to get the inputs, outputs, and gates by comparing the Pattern instances. For the inputs and output wires that matched, we classify the wire number or wire as an output wire or input wire and insert them in the wireMap. For the gate matcher, we iterate over each line and group the inputs and outputs for the gates.

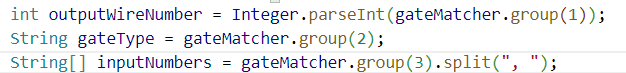


Figure : 3 crucial lines of code.

For an example 10 = OR (1,3), the first line of code of figure 20 would give 10 as the outputWireNumber, the 2nd line of code will give OR as gateType, and the third line of code will give an array of type String having 1 and 3 as elements in it. Now, we iterate over this array in order to get the count of each input wire to the gate from the fanoutCounts map and update the fanout count in order to put it again in the fanoutCounts map. Next, we define the wire in the wireMap as an internal wire if it is absent already in this wireMap. If it is not absent, it is already defined as an output or input wire. Now, if the fanout count is greater than 1, meaning the wire is used more than once in the benchmark, we enter a conditional statement where a new name for this wire will be assigned having the format 1601 for example for a wire 16 having 1 fanout. If it has 2 fanouts, then it will have a format 1602, etc. Through this name, we create a new Wire instance called fanoutWire which we will insert in the wireMap and we will add it to the fanoutWireMap too.

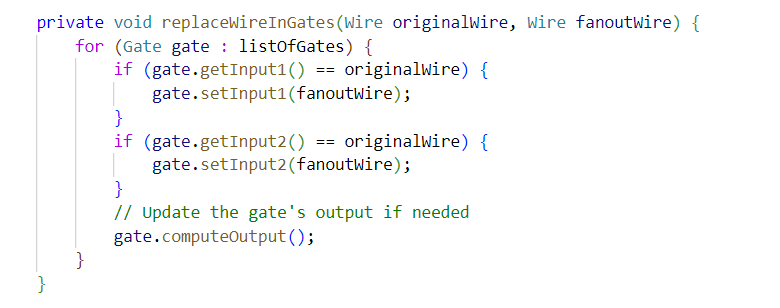


Figure :replaceWireInGates method.

The replaceWireInGates method will help us know which wires are fanout for the fault simulation later on.



Figure : Printing wires methods.



Figure : Printing fanout wires.

The methods in figures 22 and 23 represent the methods to print the different type of wires and gates.

# Phase 2: Logic Simulation:

In this phase, we ask the user to input a specific test vector in order to simulate the true-value of the circuit for this test vector. Also, true-value simulation for all possible test vector inputs will be done in phase 3 which is the fault simulation in order to check which input vectors detect the specific fault. This phase will be done purely in the Circuit class.

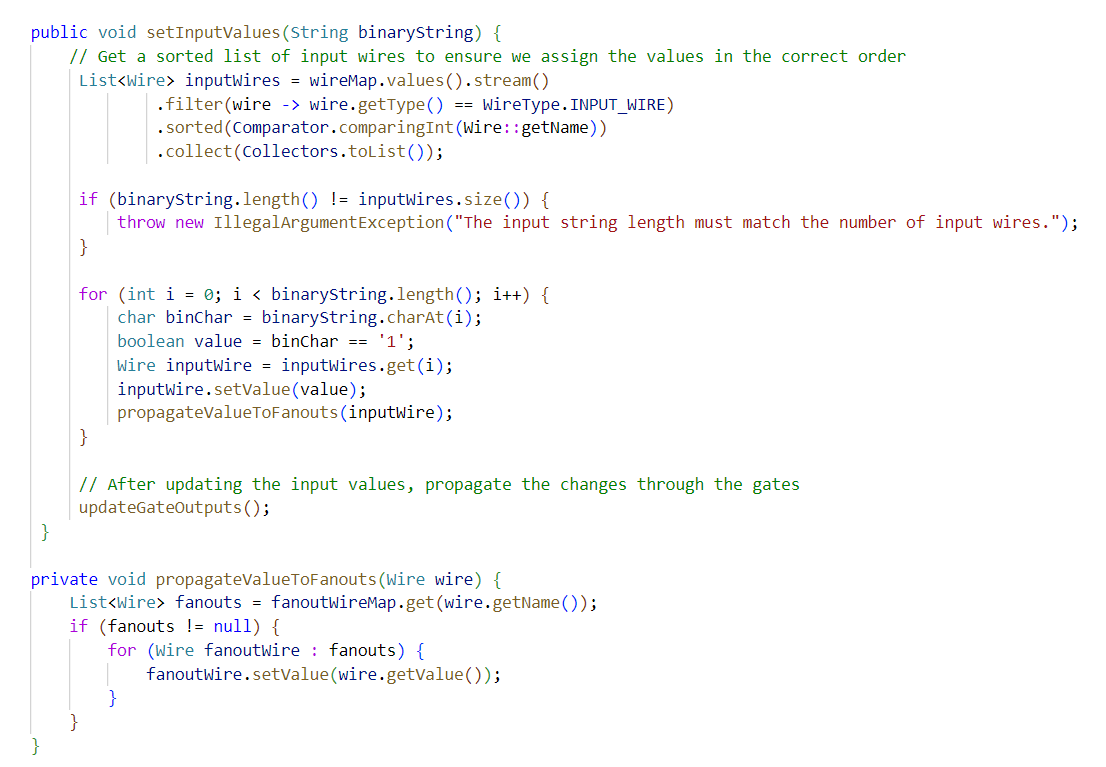


Figure : Methods for logic simulation part 1.



Figure : Methods for Logic simulation part 2.

The setInputValues method sets the inputs to the circuit based on what the user gave as a test vector input. The propagateValuesToFanouts method propagates the values from the fanout stems to the fanout branches. This is done by using the list of fanouts which was filled earlier with fanout branches and their corresponding fanout stem. We get the value of fanout stem and propagate the value of it to the fanout branches by iterating over the list of fanouts. The updateGateOutputs() method computes the output of the circuit based on the value of inputs at each gate till we get to the output.

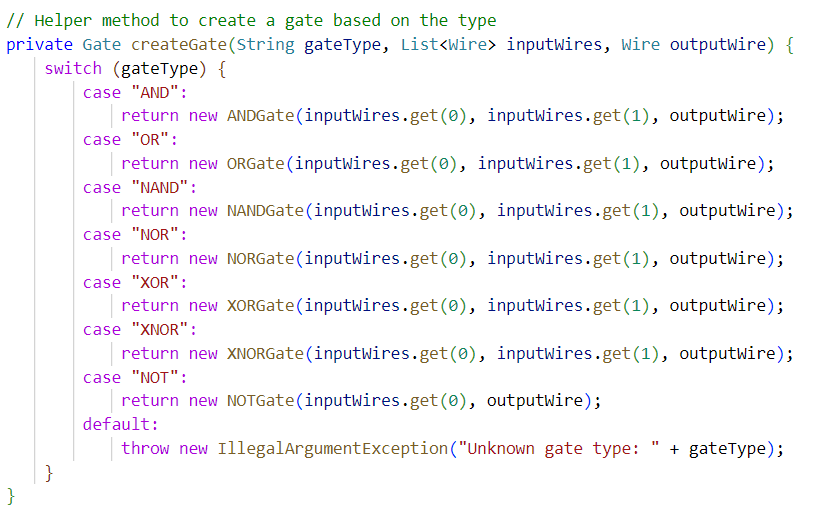


Figure : createGate method.

The createGate method helps us create the gates based on the benchmark String in order to perform the true-value simulation.

We also did as extra a User Interface where the user can visually see the circuit through a dot file. The codes for that are shown below:

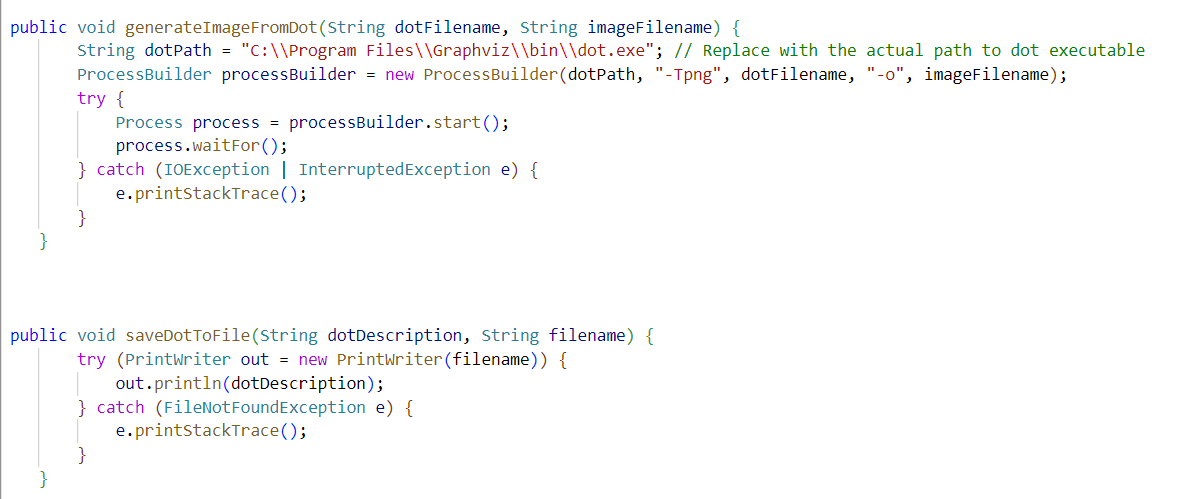


Figure : User Interface methods part 1.

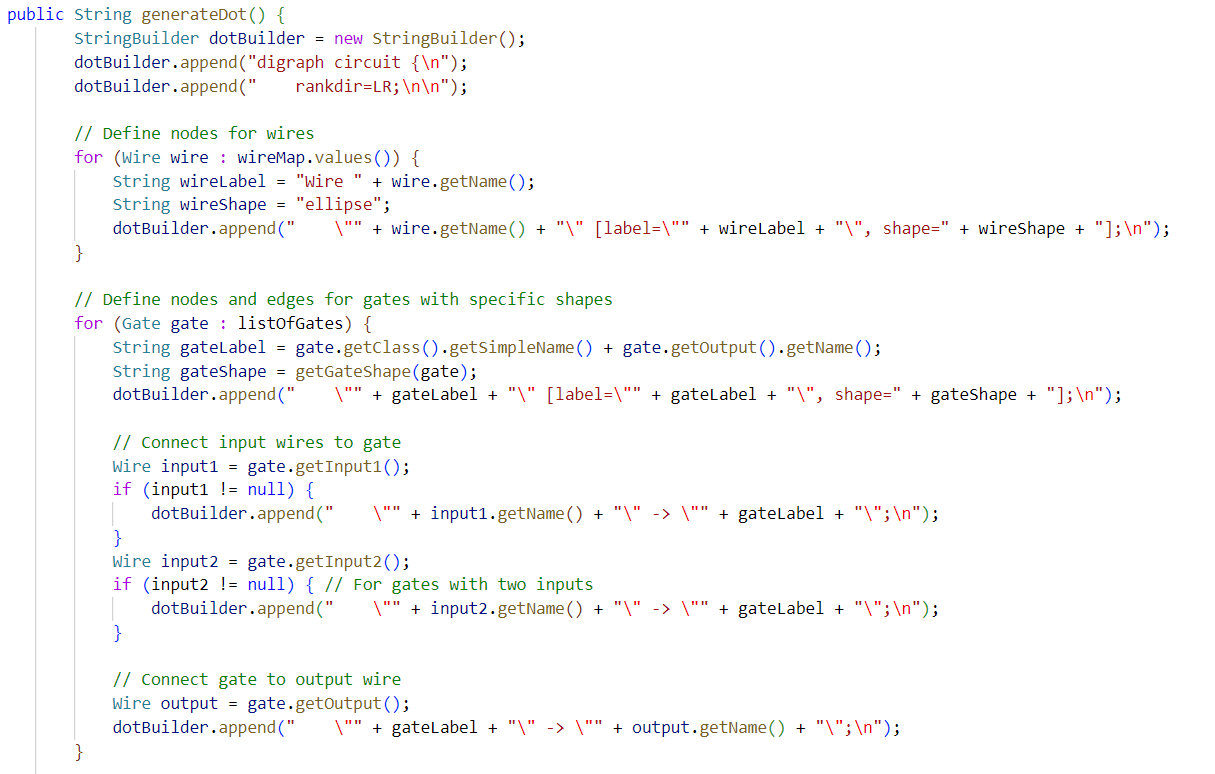


Figure : User Interface methods part 2.

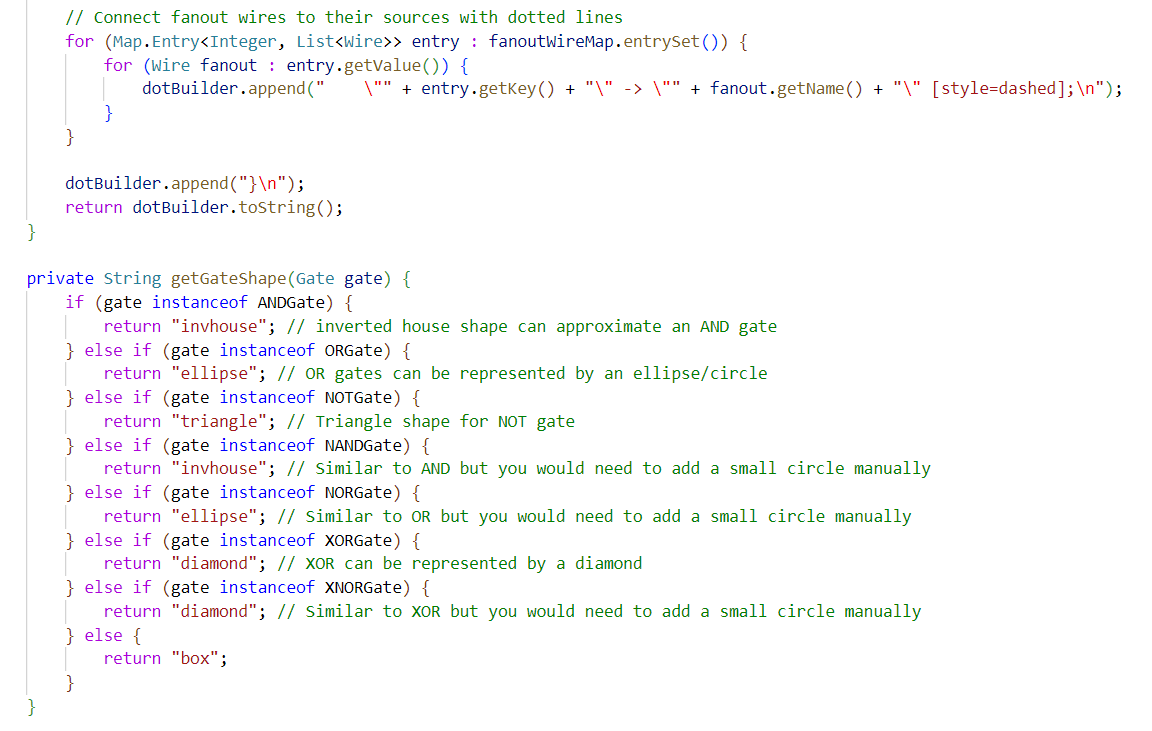


Figure : User Interface methods part 3.

# Phase 3: Fault Simulation:

In this phase, we are required to develop a serial fault simulator based on the flowchart picture show below:

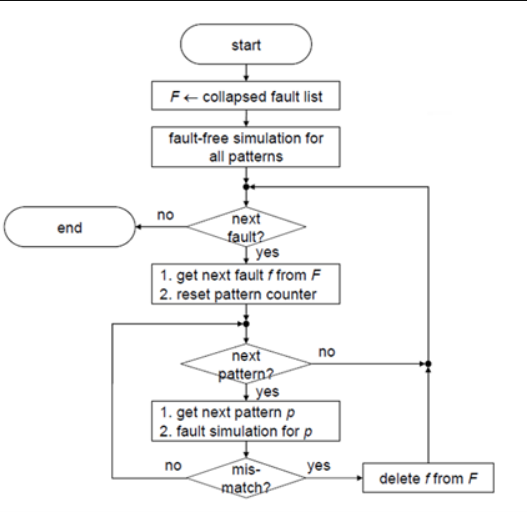


Figure : Serial Fault Simulator algorithm.

We did a lot of options for that where the user can select any of those options. The first option would be to select a certain stuck-at fault and check if the specific input test vector detects it. The second option would be the number of faults detected for each test vector pattern to know which test vector pattern, but these 2 options are extra analysis for the user. As for the 3rd option, it does the serial fault simulator as explained in the flowchart where it is as if we are finding the test vector pattern which detects a certain fault by comparing it with the true-value simulation of this test pattern. If a specific test pattern detects it, move on to the next fault without the need to detect it again using another test vector pattern as we already achieved that. If the test pattern doesn’t detect it, move on to the next test pattern until all test pattern vectors are exhausted. If all of test vector patterns are exhausted and not a single one detects the fault, the fault is undetectable.

In the end, the timing, total fault coverage and fault efficiency were calculated and the results showed which test vector pattern detected the specific stuck-at fault.

The codes for options 1 and 2 are shown below:

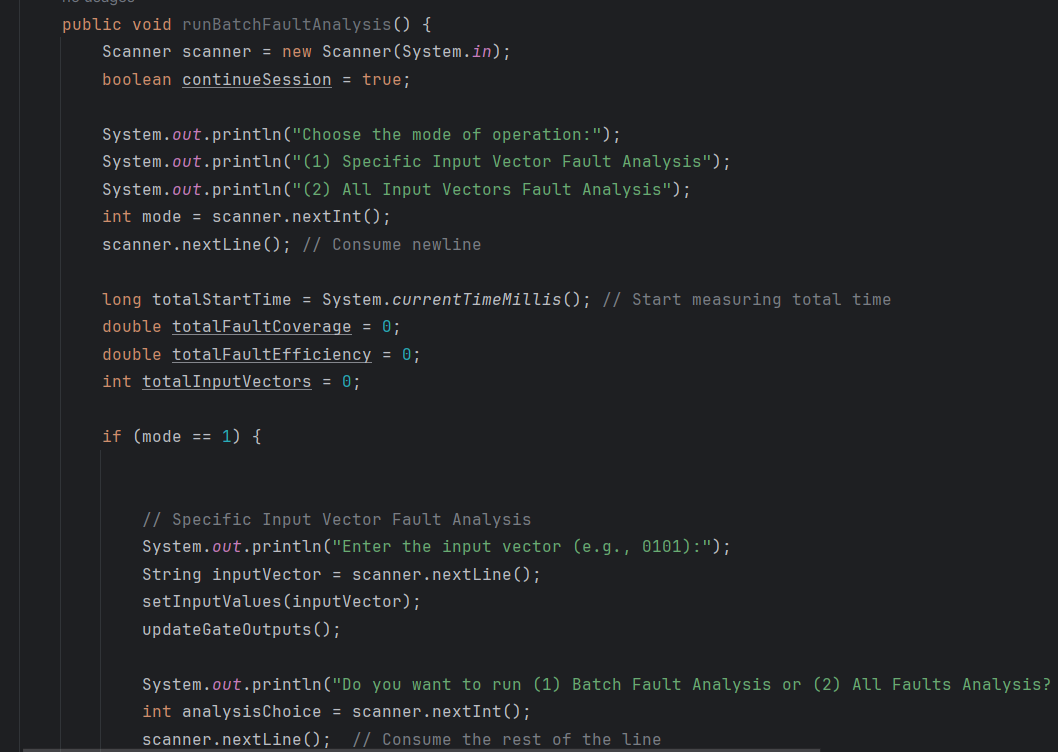


Figure : Options 1 and 2 code.

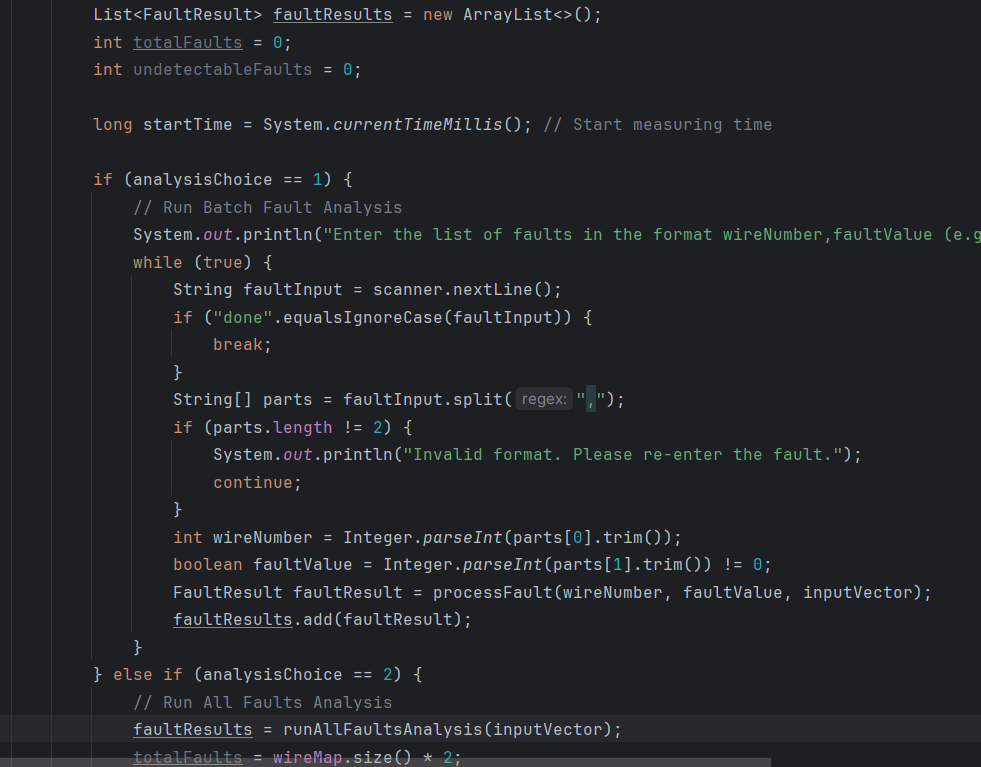


Figure : Options 1 and 2 code part 2.

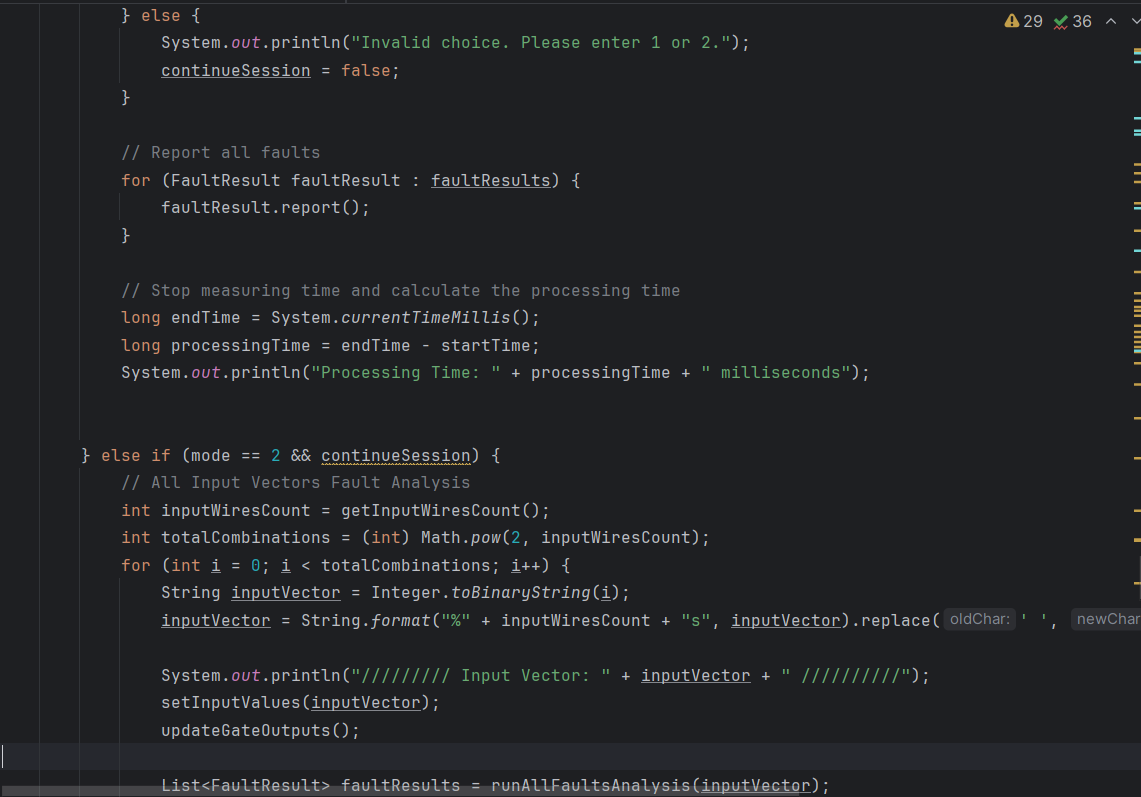


Figure : Options 1 and 2 code part 3.



Figure : Options 1 and 2 code part 4.

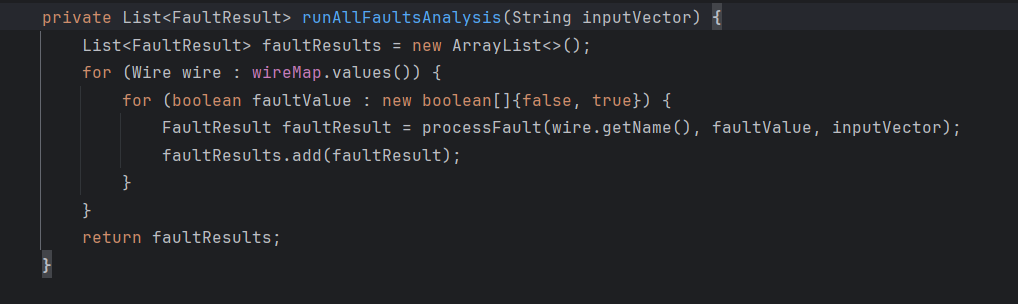


Figure : Options 1 and 2 code part 5.

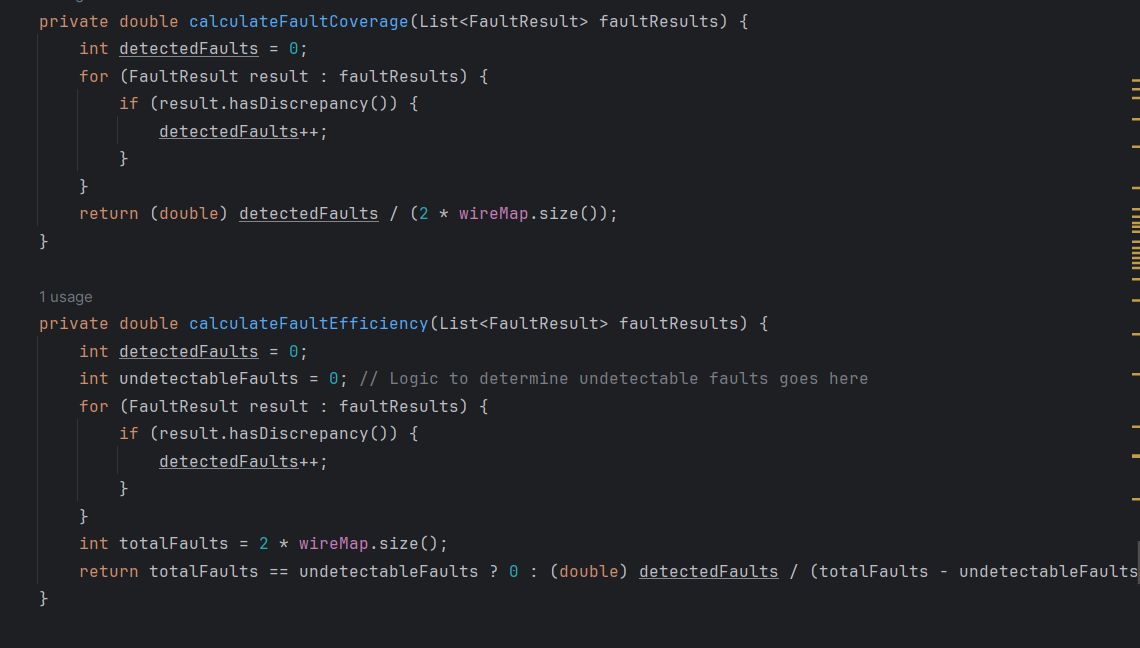
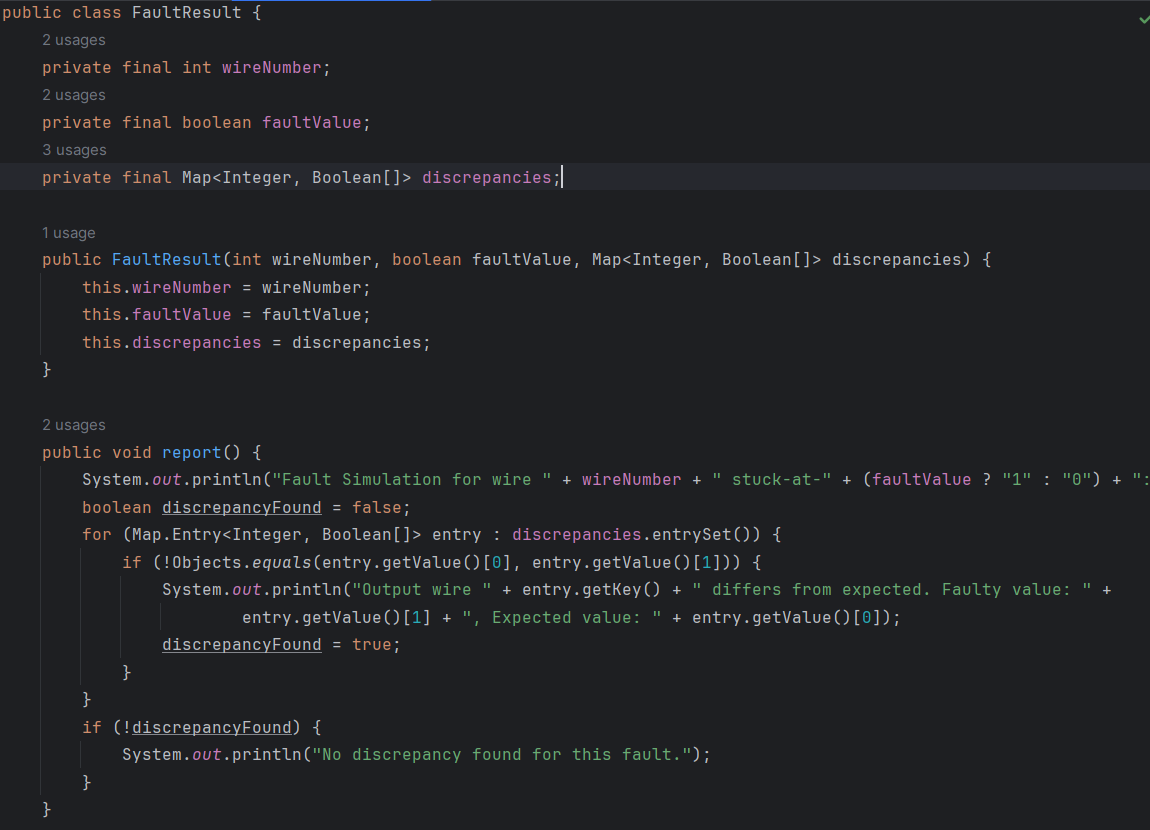
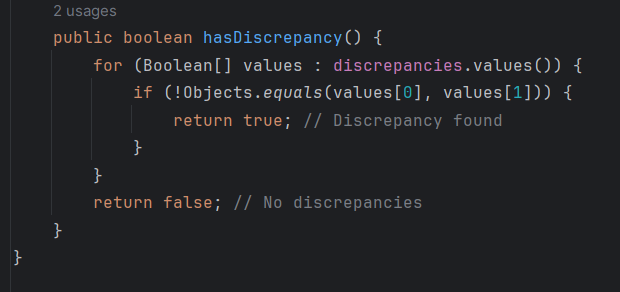


Figure : Options 1 and 2 code part 6.

It should be noted that a new Object class called FaultResult was also implemented to execute this phase of the project. This class aims to take the wire number and its fault value and then will be used in order to compare the outputs of the true-value simulation and the faulty value.





As for the actual serial fault simulator, the code is shown below:

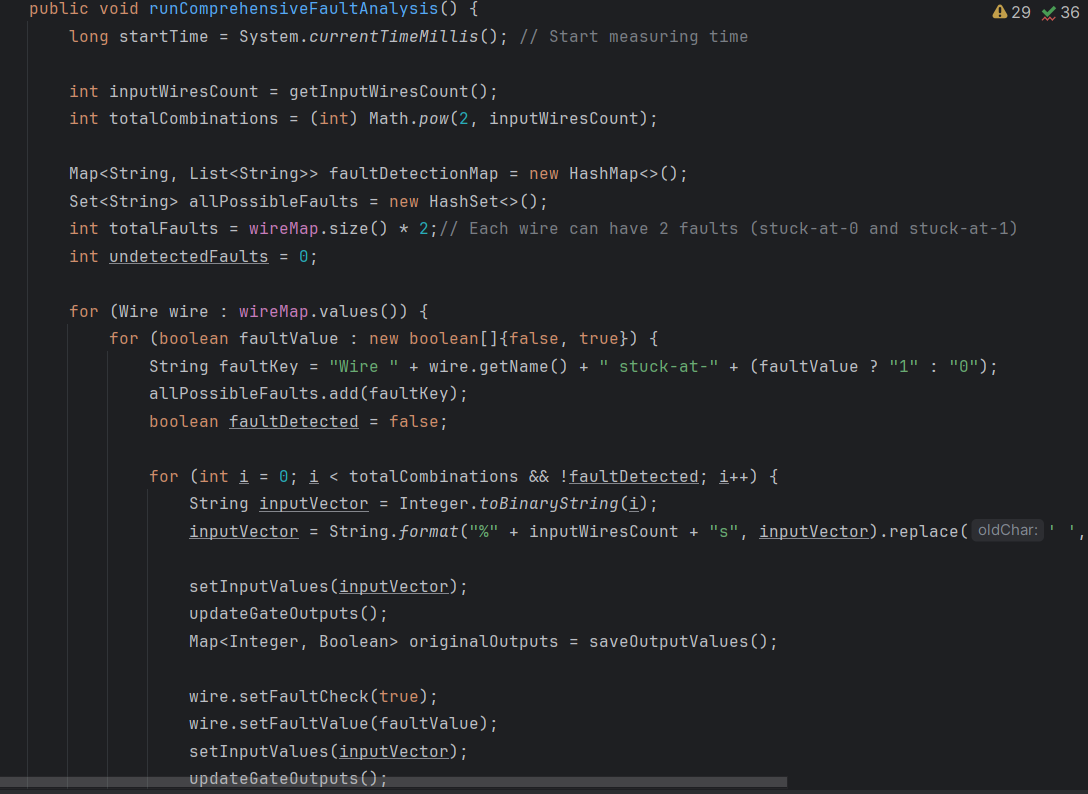


Figure : Serial fault simulator code part 1.

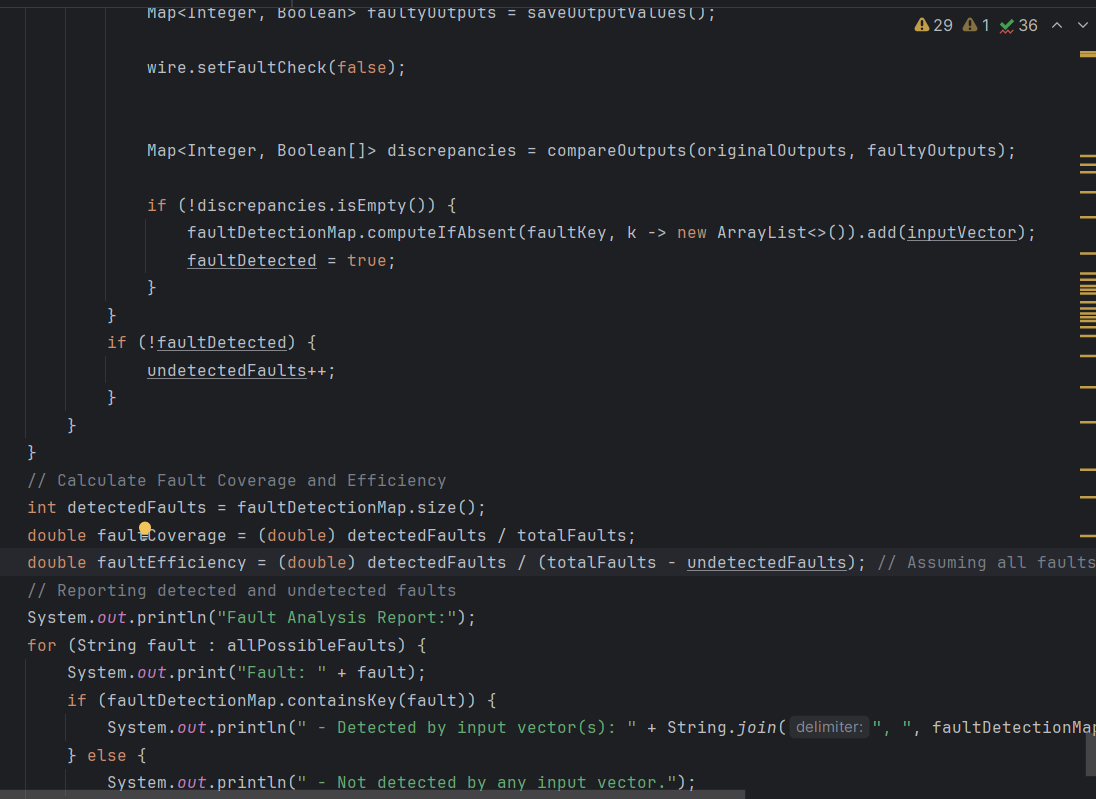


Figure : Serial fault simulator code part 2.

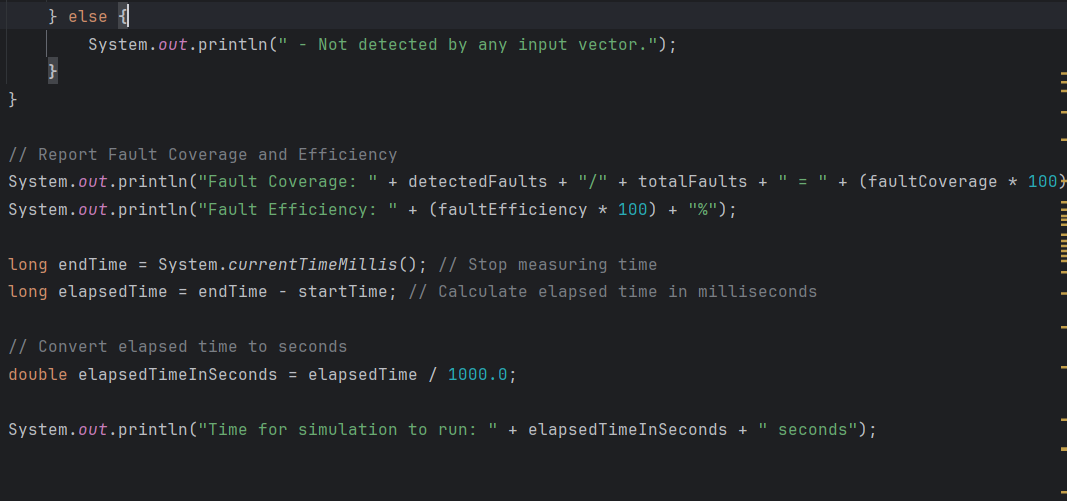


Figure : Serial fault simulator code part 3.

It can be seen from the code that the algorithm used is to iterate over the wires first, take each wire and its 2 stuck-at faults and try to find a test vector pattern which detects the fault at the outputs. If no test vector patter found, fault is redundant. If a test vector pattern is found, then fault is detected and move to next fault. It can be seen from the most inner for loop that the true-value simulation and faulty simulations are computed and then their outputs are saved in maps called originalOutputs and faultyOutputs respectively and then they are compared to check if fault is detectable or undetectable for the test vector pattern. In the end, each fault is printed with the test vector pattern that detected it (or a statement that says it is undetected if no test vector pattern found), and the simulation time, fault coverage, and fault efficiency are calculated. It should be noted that a discrepancy at any of the outputs would lead to a fault being detected

Some methods that helped us get the results:

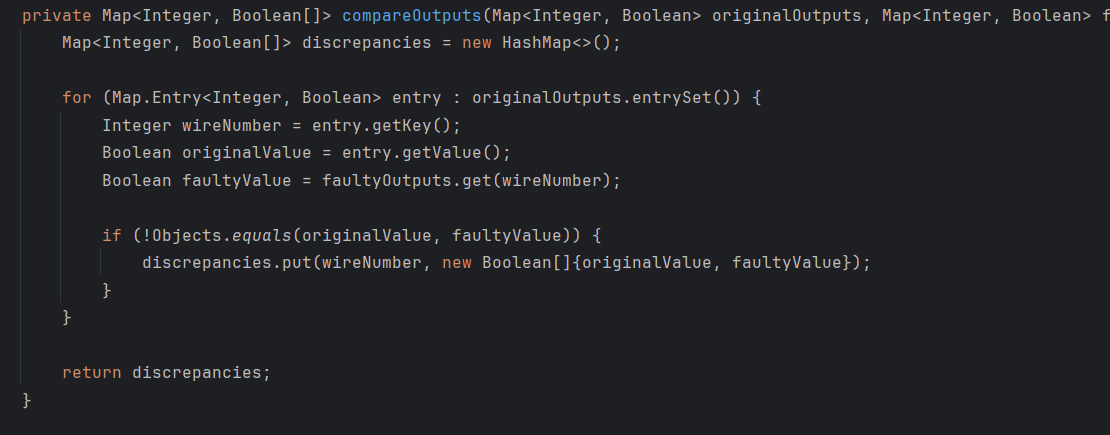


Figure : compareOutputs method.

This method helped us compare the outputs of the true-value simulation and faulty simulation to determine whether the fault is detected at any of the outputs for a specific test vector pattern. If there is discrepancy at any of the outputs, add it to the discrepancy map.

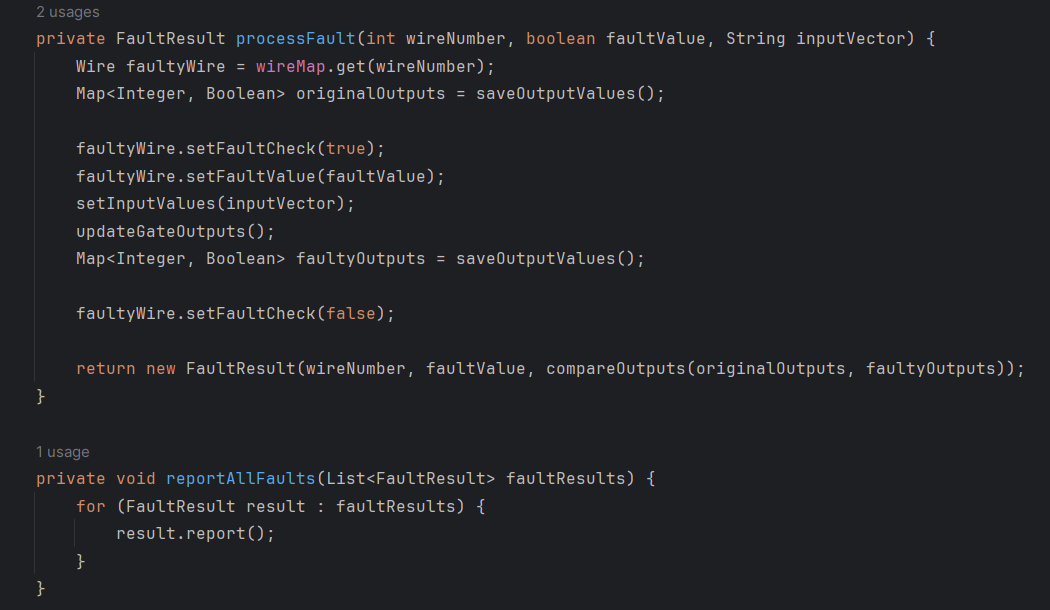


Figure : processFault and reportAllFaults methods.

The processFault method helps us execute the faulty simulation, but was used for options 1 and 2 in our case. The reportAllFaults method reports all faults of the circuit.

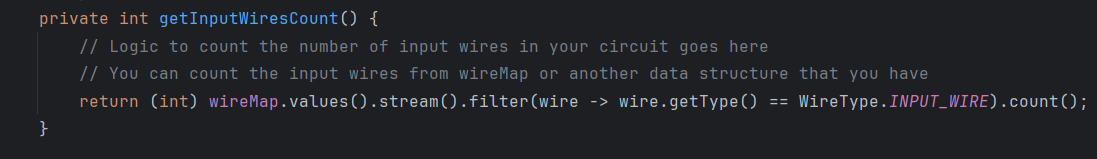


Figure : getInputWiresCount method.

This method counts the number of input wires found in the circuit in order to propagate the test vector patterns in true-value and faulty simulations.

# Results:

For the c17 circuit found in the project manual:

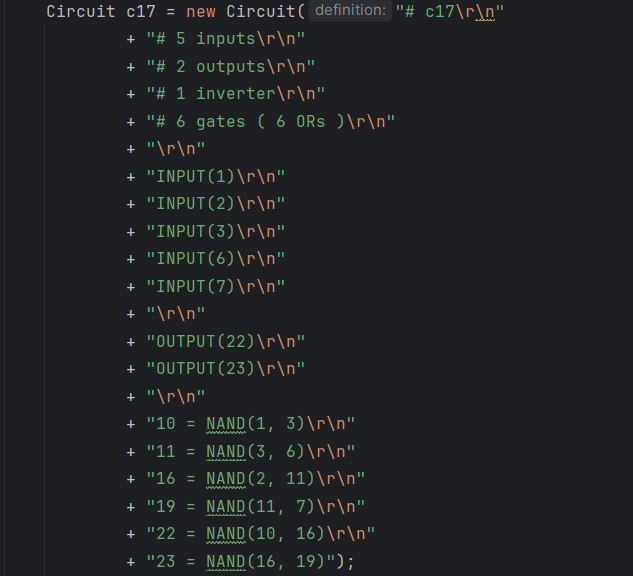


Figure : c17 circuit format.

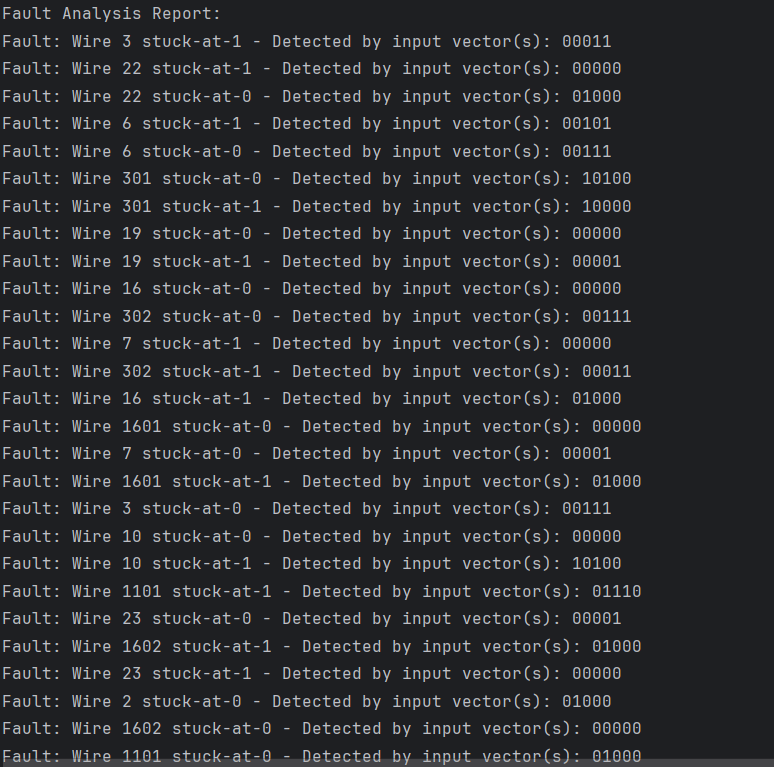


Figure : circuit c17 results part 1.

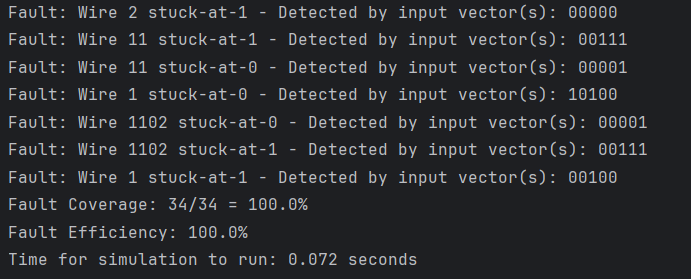


Figure : circuit c17 results part 2.

It can be seen that all of the faults were detected, hence getting the 100% fault coverage. Also, fault efficiency seems to always be 100% as the algorithm exhaustively searches every single fault which means it can know each fault whether it is detected or undetected, so this might be why the fault efficiency is equal to 100% This makes sense according to the formula since in the numerator and denominator, the present values are detectedFaults and (totalFaults – undetectedFaults) respectively, which will always be equal to each other if the algorithm exhausts all possible faults.

Results of another circuit:

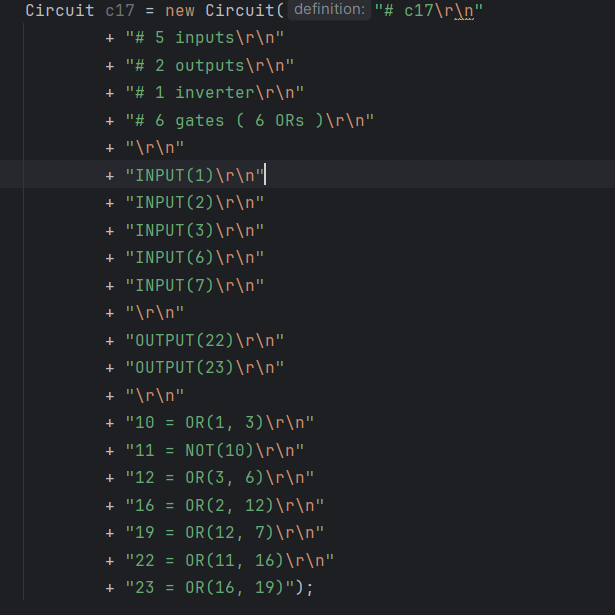


Figure : Another circuit format.

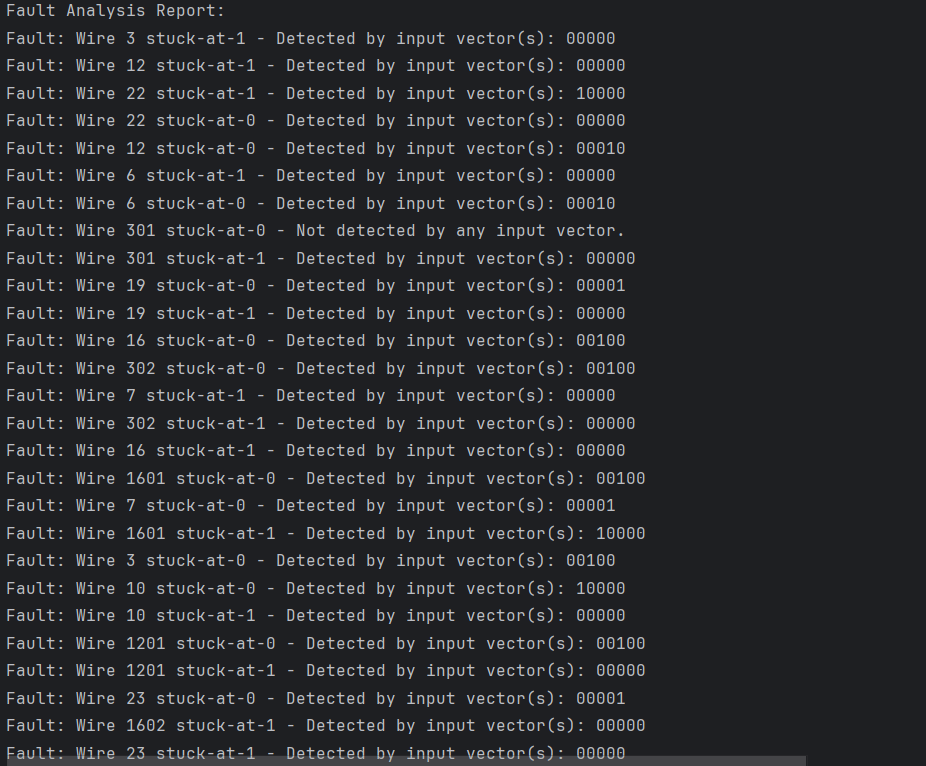


Figure : Circuit results part 1.

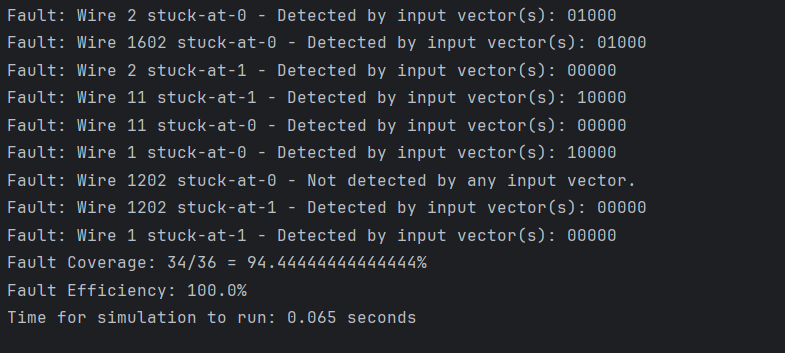


Figure : Circuit results part 2.

Here, we can see that only 2 faults were undetected giving us a fault coverage of 94.444% where the 2 faults undetected are wires 1202 stuck-at 0 and 301 stuck-at zero.

Another circuit will be the following:

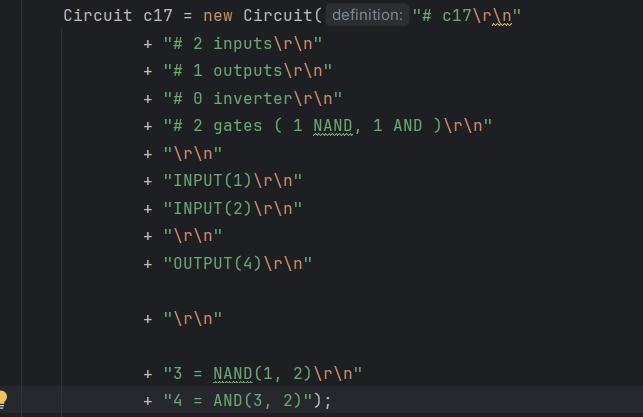


Figure : Last circuit format.

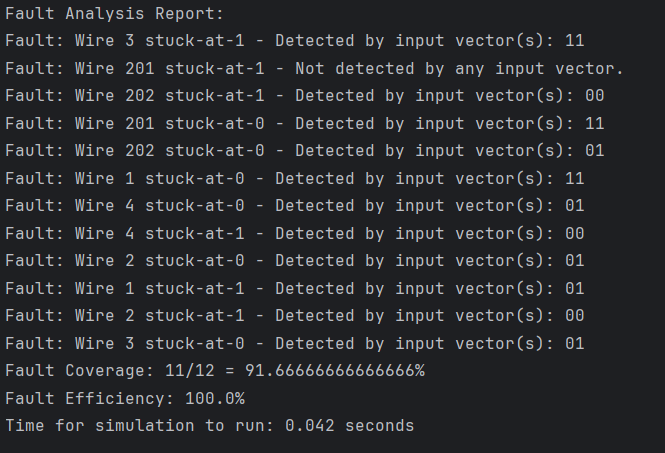


Figure : Last circuit results.

The results show a fault coverage of 91.667% where only 1 undetected fault is found which is the wire 201 stuck-at-1 fault.

# Conclusion:

In this project, we were able to use a high-level programming language Java in order to develop a serial fault simulator. This was done using 3 phases which are the parsing, logic simulation, and fault simulation phases. In the end, results showed each fault whether it is detected by an input vector pattern or not, the timing of simulation, the fault coverage, and the fault efficiency.