https://github.com/roxanazachman01/FLCD

Documentation:

Statement:

Write a program that:

1. Reads the elements of a FA (from file).
2. Displays its elements, using a menu: the set of states, the alphabet, all the transitions, the initial state and the set of final states.
3. For a DFA, verifies if a sequence is accepted by the FA.

Input: FA.in, sequence

Output: whether the sequence is accepted by the FA or not, if the given FA is valid

Deliverables:

1. FA.in - input file (on Github)
2. Source code (on Github)
3. Documentation. It should also include in BNF or EBNF format the form in which the FA.in file should be written (on Moodle and Github)

Details:

Max grade = 10: Use FA to detect tokens <identifier> and <integer constant> in the scanner program

Analysis: The Finite automata should store all the necessary information, like the set of states, the alphabet, the set of transitions, the initial state and the set of final states. This information should be read from a file and also validated. It should support verifying if a sequence is accepted by it, and whether the FA is a DFA or a NFA. The finite automata should be integrated in the scanner to detect identifiers and integer constants, so we would also need 2 input files with FA for identifier and integer constants.

Implementation: The input file format should be created according to the following EBNF rules:

state = letter{letter|digit}

accepted = letter|digit|”’”|””””|”_”|” +”|”-”|” ”

accepted_sequence = accepted {“,” accepted}

transition = state “ “ accepted_sequence “ “ state transition_sequence = transition {"\n" transition}

state_sequence = state " " {state}

(denote by "\n" the new line char)

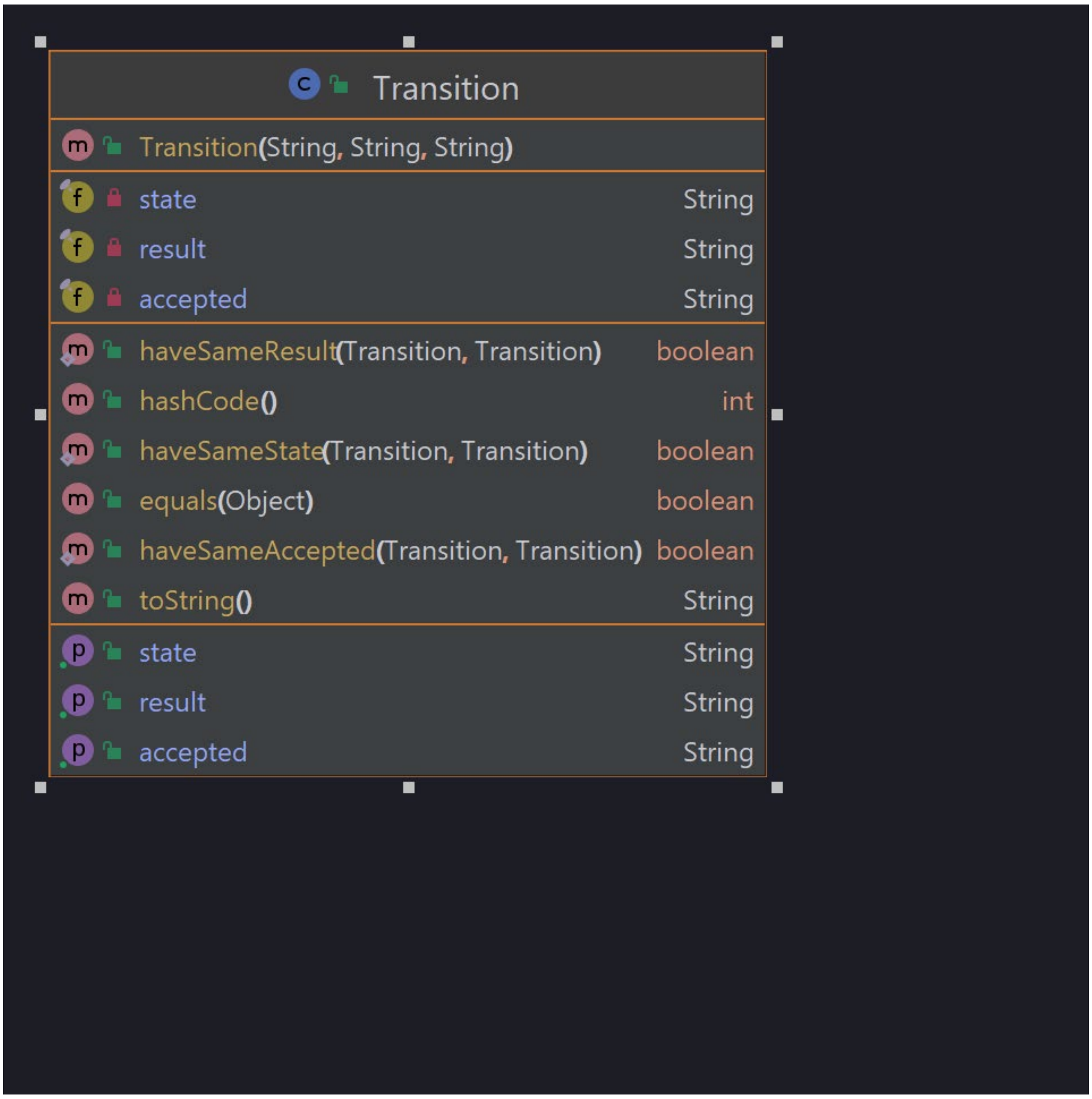
file_format = state_sequence "\n" accepted_sequence "\n" integer "\n" transition_sequence "\n" state "\n" state_sequence

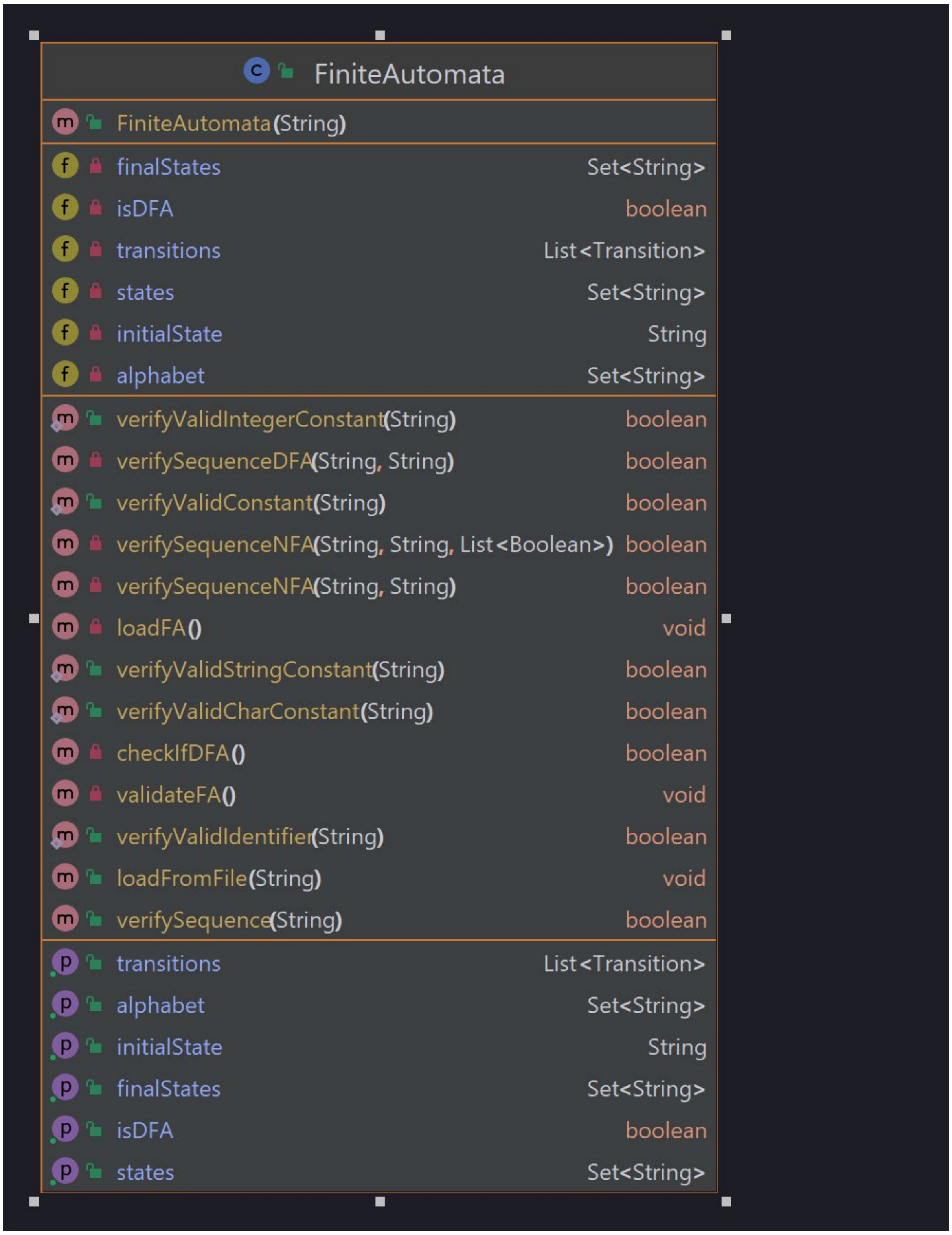
The input file is read in the constructor of the FA, then the loaded elements are validated. (throws exception if transitions contain accepted terms which are not present in the alphabet, or states which are not in the set of states, of if the initial state is not initial in the transition or if one of the final states is never final and so on; validation also prints warnings in case of redundancy, like when the set of states contains states that are never used in transitions, so inaccessible states).

The method that checks if the FA is a DFA or not takes every pair of transitions d(p1,a1)=q1 and d(p2,a2)=q2, and if it finds at least a pair that respects the condition p1=p2, a1=a2 and q1!=q2, returns false. If no such pair is found, then it returns true. (DFA = at most one state can be obtained as a result).

The method that implements the verifying if a sequence is accepted is implemented as follows: it first checks whether the given sequence is empty; if it is, then it returns whether the set of final states contain the initial state or not. (checks whether the FA accepts epsilon or not).

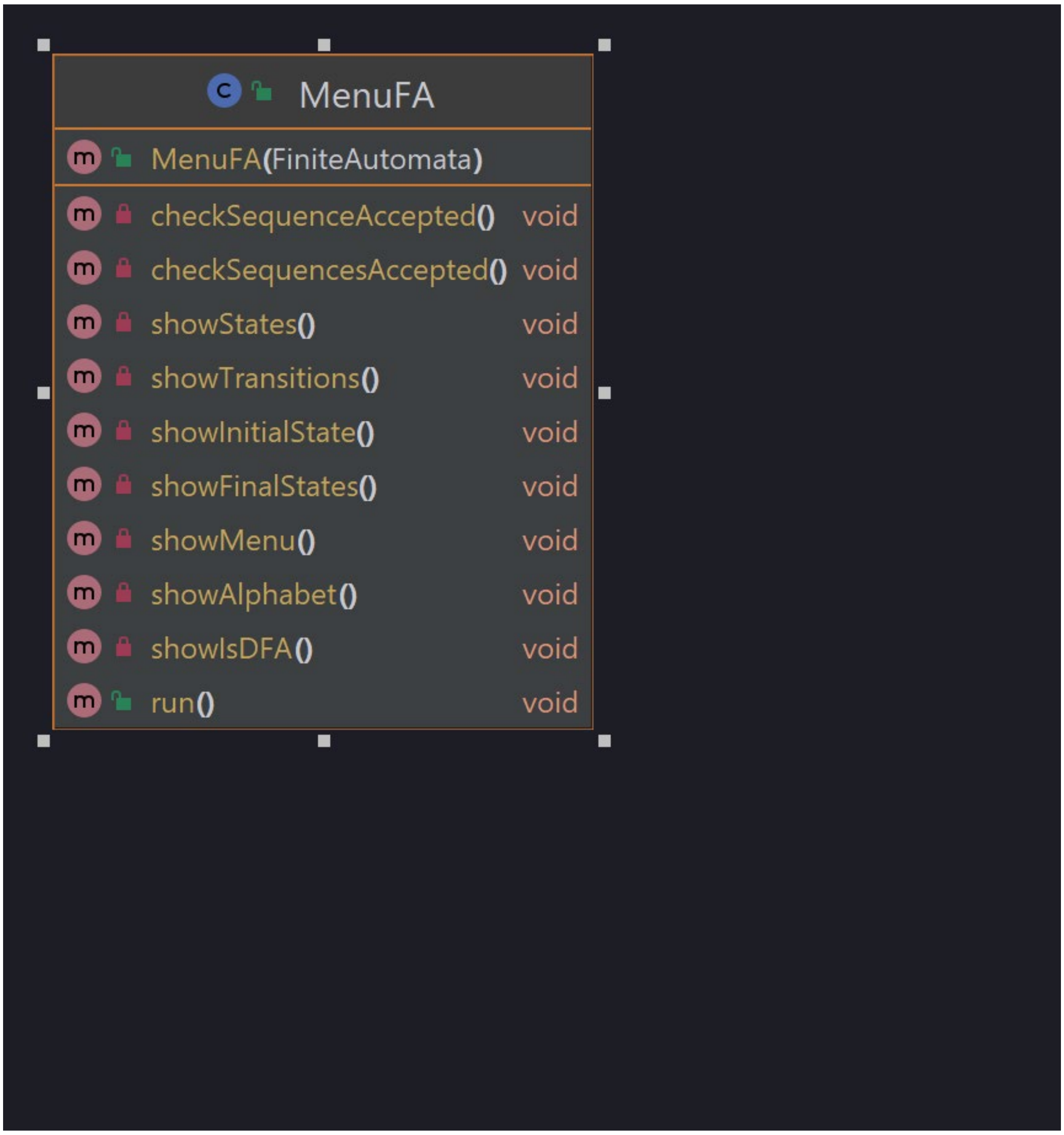
If the sequence is not empty, then it enters a recursive function for checking, where in each step it accepts one element and appends to a list of Boolean whether the sequence was accepted at that point or not. After all cases are checked, it performs a logical OR on the list of Boolean. This function also works in the case of NFA, because it doesn't stop when the sequence was emptied and we haven't reached the final state, it just appends false to that list of checks and continues checking for other 'paths' in the FA.


Transition
Transition(String, String, String)
state String
result String
accepted String
haveSameResult(Transition, Transition) boolean
hashCode() int
haveSameState(Transition, Transition) boolean
equals(Object) boolean
haveSameAccepted(Transition, Transition) boolean
toString() String
state String
result String
accepted String



FiniteAutomata

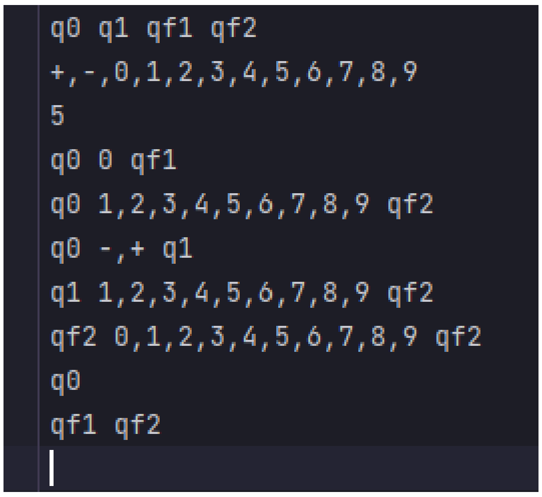
FiniteAutomata(String)	
finalStates	Set<String>
isDFA	boolean
transitions	List<Transition>
states	Set<String>
initialState	String
alphabet	Set<String>
verifyValidIntegerConstant(String)	boolean
verifySequenceDFA(String, String)	boolean
verifyValidConstant(String)	boolean
verifySequenceNFA(String, String, List<Boolean>)	boolean
verifySequenceNFA(String, String)	boolean
loadFA()	void
verifyValidStringConstant(String)	boolean
verifyValidCharConstant(String)	boolean
checkIfDFA()	boolean
validateFA()	void
verifyValidIdentifier(String)	boolean
loadFromFile(String)	void
verifySequence(String)	boolean
transitions	List<Transition>
alphabet	Set<String>
initialState	String
finalStates	Set<String>
isDFA	boolean
states	Set<String>



```
MenuFA
MenuFA(FiniteAutomata)
checkSequenceAccepted()   void
checkSequencesAccepted()  void
showStates()              void
showTransitions()         void
showInitialState()        void
showFinalStates()         void
showMenu()                void
showAlphabet()            void
showIsDFA()               void
run()                     void
```

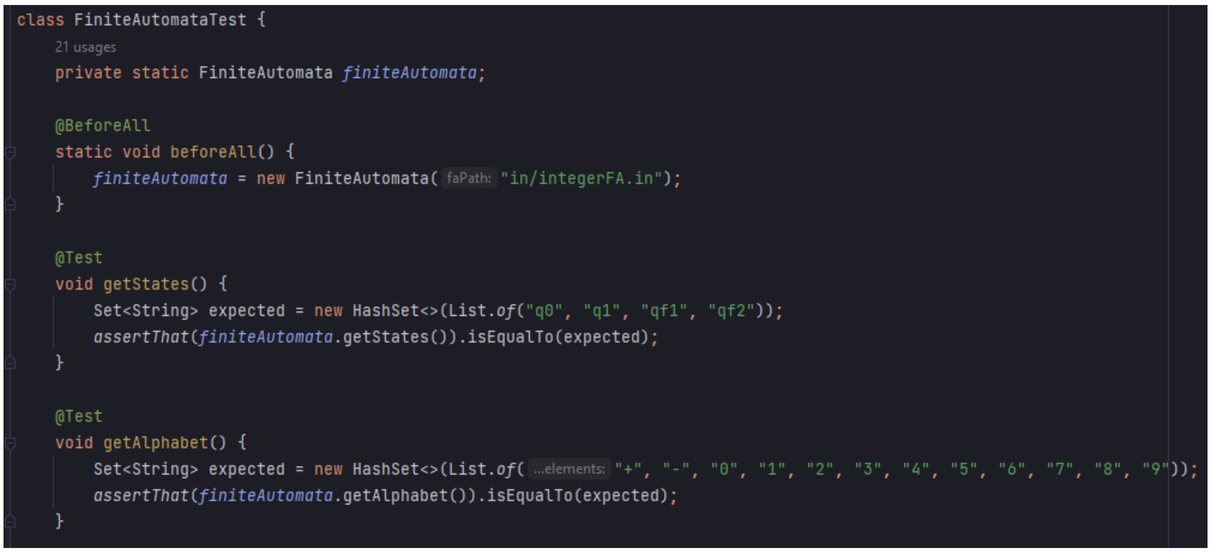
Testing: tests the FA loads correctly the input file for the integer FA and also that it accepts or not sequences

Input file:

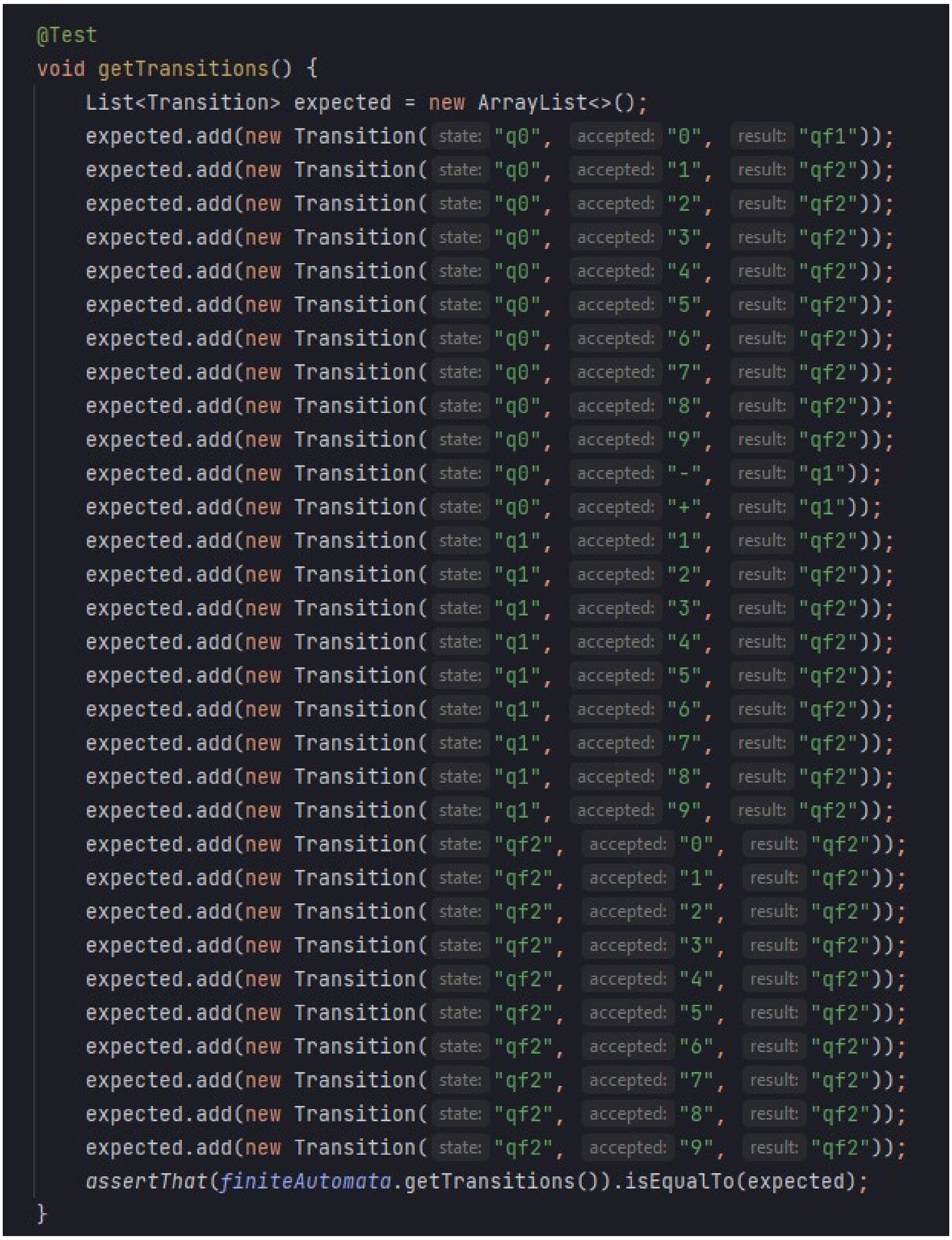


```
q0 q1 qf1 qf2
+,-,0,1,2,3,4,5,6,7,8,9
5
q0 0 qf1
q0 1,2,3,4,5,6,7,8,9 qf2
q0 -,+ q1
q1 1,2,3,4,5,6,7,8,9 qf2
qf2 0,1,2,3,4,5,6,7,8,9 qf2
q0
qf1 qf2
```

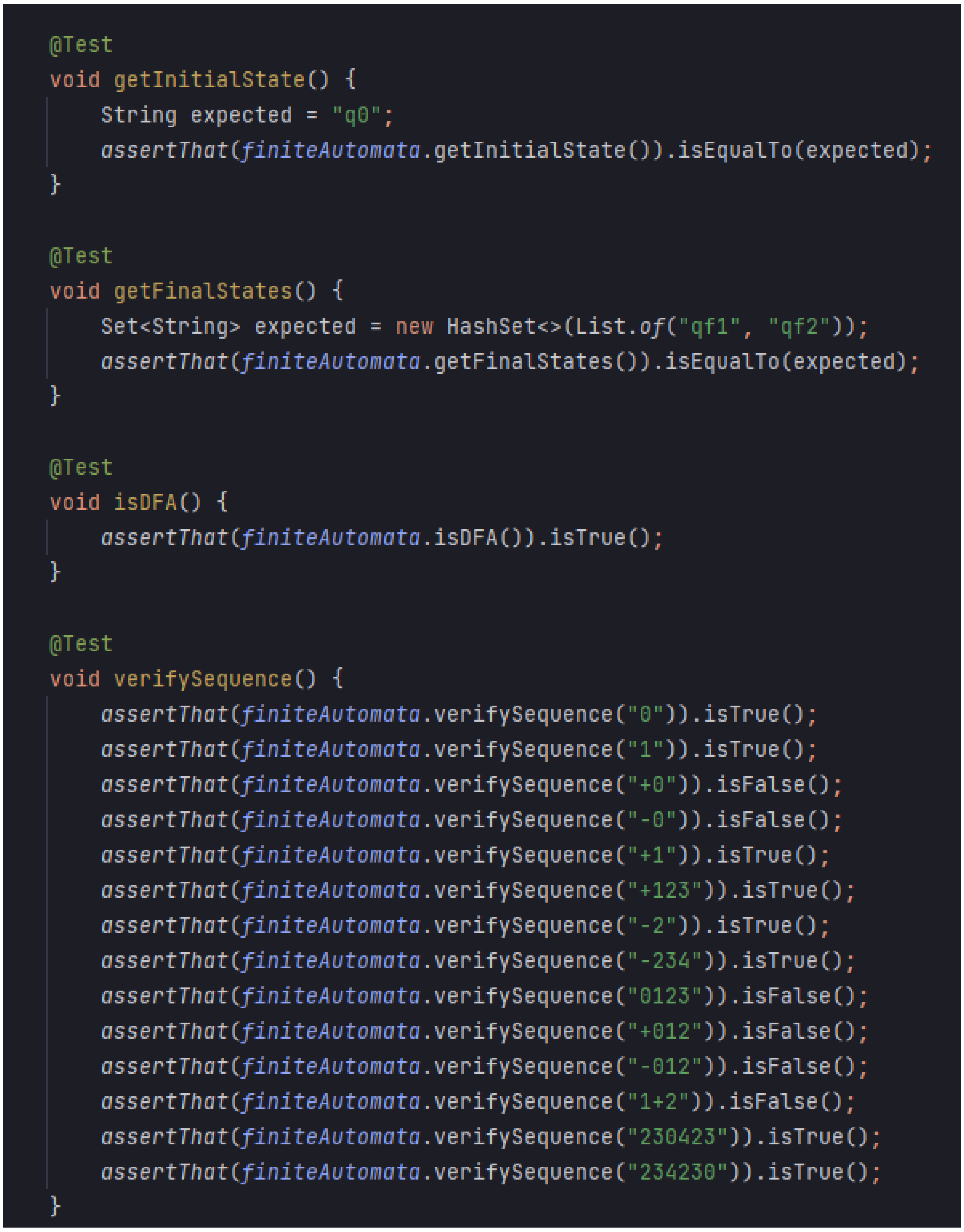
Tests:



```
class FiniteAutomataTest {
    21 usages
    private static FiniteAutomata finiteAutomata;

    @BeforeAll
    static void beforeAll() {
        finiteAutomata = new FiniteAutomata( faPath: "in/integerFA.in");
    }

    @Test
    void getStates() {
        Set<String> expected = new HashSet<>(List.of("q0", "q1", "qf1", "qf2"));
        assertThat(finiteAutomata.getStates()).isEqualTo(expected);
    }

    @Test
    void getAlphabet() {
        Set<String> expected = new HashSet<>(List.of( ...elements: "+", "-", "0", "1", "2", "3", "4", "5", "6", "7", "8", "9"));
        assertThat(finiteAutomata.getAlphabet()).isEqualTo(expected);
    }
```



```java
@Test
void getTransitions() {
    List<Transition> expected = new ArrayList<>();
    expected.add(new Transition( state: "q0",  accepted: "0",  result: "qf1"));
    expected.add(new Transition( state: "q0",  accepted: "1",  result: "qf2"));
    expected.add(new Transition( state: "q0",  accepted: "2",  result: "qf2"));
    expected.add(new Transition( state: "q0",  accepted: "3",  result: "qf2"));
    expected.add(new Transition( state: "q0",  accepted: "4",  result: "qf2"));
    expected.add(new Transition( state: "q0",  accepted: "5",  result: "qf2"));
    expected.add(new Transition( state: "q0",  accepted: "6",  result: "qf2"));
    expected.add(new Transition( state: "q0",  accepted: "7",  result: "qf2"));
    expected.add(new Transition( state: "q0",  accepted: "8",  result: "qf2"));
    expected.add(new Transition( state: "q0",  accepted: "9",  result: "qf2"));
    expected.add(new Transition( state: "q0",  accepted: "-",  result: "q1"));
    expected.add(new Transition( state: "q0",  accepted: "+",  result: "q1"));
    expected.add(new Transition( state: "q1",  accepted: "1",  result: "qf2"));
    expected.add(new Transition( state: "q1",  accepted: "2",  result: "qf2"));
    expected.add(new Transition( state: "q1",  accepted: "3",  result: "qf2"));
    expected.add(new Transition( state: "q1",  accepted: "4",  result: "qf2"));
    expected.add(new Transition( state: "q1",  accepted: "5",  result: "qf2"));
    expected.add(new Transition( state: "q1",  accepted: "6",  result: "qf2"));
    expected.add(new Transition( state: "q1",  accepted: "7",  result: "qf2"));
    expected.add(new Transition( state: "q1",  accepted: "8",  result: "qf2"));
    expected.add(new Transition( state: "q1",  accepted: "9",  result: "qf2"));
    expected.add(new Transition( state: "qf2",  accepted: "0",  result: "qf2"));
    expected.add(new Transition( state: "qf2",  accepted: "1",  result: "qf2"));
    expected.add(new Transition( state: "qf2",  accepted: "2",  result: "qf2"));
    expected.add(new Transition( state: "qf2",  accepted: "3",  result: "qf2"));
    expected.add(new Transition( state: "qf2",  accepted: "4",  result: "qf2"));
    expected.add(new Transition( state: "qf2",  accepted: "5",  result: "qf2"));
    expected.add(new Transition( state: "qf2",  accepted: "6",  result: "qf2"));
    expected.add(new Transition( state: "qf2",  accepted: "7",  result: "qf2"));
    expected.add(new Transition( state: "qf2",  accepted: "8",  result: "qf2"));
    expected.add(new Transition( state: "qf2",  accepted: "9",  result: "qf2"));
    assertThat(finiteAutomata.getTransitions()).isEqualTo(expected);
}
```



```java
@Test
void getInitialState() {
    String expected = "q0";
    assertThat(finiteAutomata.getInitialState()).isEqualTo(expected);
}

@Test
void getFinalStates() {
    Set<String> expected = new HashSet<>(List.of("qf1", "qf2"));
    assertThat(finiteAutomata.getFinalStates()).isEqualTo(expected);
}

@Test
void isDFA() {
    assertThat(finiteAutomata.isDFA()).isTrue();
}

@Test
void verifySequence() {
    assertThat(finiteAutomata.verifySequence("0")).isTrue();
    assertThat(finiteAutomata.verifySequence("1")).isTrue();
    assertThat(finiteAutomata.verifySequence("+0")).isFalse();
    assertThat(finiteAutomata.verifySequence("-0")).isFalse();
    assertThat(finiteAutomata.verifySequence("+1")).isTrue();
    assertThat(finiteAutomata.verifySequence("+123")).isTrue();
    assertThat(finiteAutomata.verifySequence("-2")).isTrue();
    assertThat(finiteAutomata.verifySequence("-234")).isTrue();
    assertThat(finiteAutomata.verifySequence("0123")).isFalse();
    assertThat(finiteAutomata.verifySequence("+012")).isFalse();
    assertThat(finiteAutomata.verifySequence("-012")).isFalse();
    assertThat(finiteAutomata.verifySequence("1+2")).isFalse();
    assertThat(finiteAutomata.verifySequence("230423")).isTrue();
    assertThat(finiteAutomata.verifySequence("234230")).isTrue();
}
```