We used a state design pattern in our code to replicate the functionality of a simple expression evaluator. The code incorporates 8 states as children of an abstract state class to replicate the functionality demonstrated in our state diagram. Each state is implemented following the singleton object pattern to help with memory management. Without the implementation of a singleton duplicate instances of a particular state may negatively affect memory for long expressions with many state transitions.

The benefit of using the state design pattern is that it separates the transitioning between states (behavior) from the logic of the calculator (context) operations, updating of operand and updating of result variables.

Below is an explanation of the state mappings in the state diagram to the state classes in the program.

* S\_0 = InitialState Class: This state is the first state which accepts any digit from 1-9. This state is needed specifically to handle cases where a leading zero is erroneously entered on the first operand in the expression.
* S\_1 = FirstOperandState Class: This state follows the InitialState after entry of a digit from 1-9. The purpose of this state is to gather the rest of the digits in the first operand. This state directly precedes any operators that may exist in the entire expression. The program may exit successfully from this state if the user does not enter an operator.
* S\_2 = SubtractionOperatorState Class: This state follows the FirstOperandState, SecondSubtractionOperandState, or the SecondAdditionOperandState after entry of a subtraction operator (-). This state only accepts digits from 1-9. This state is needed to handle cases where a leading zero is erroneously entered for the operand that comes after a subtraction operator.
* S\_3 = SecondSubtractionOperandState Class: This state follows the SubtractionOperatorState. The purpose of this state is to gather the rest of the digits in the operand that comes after a subtraction operator. This state also accepts the addition operator, subtraction operator, and newline character as valid inputs, and each of these inputs will transition the context to a separate state.
* S\_4 = AdditionOperatorState Class: this state follows the FirstOperandState, SecondSubtractionOperandState, or the SecondAdditionOperandState after entry of an addition operator (+). This state only accepts digits from 1-9. This state is needed to handle cases where a leading zero is erroneously entered for the operand that comes after an addition operator.
* S\_5 = SecondAdditionOperandState Class: This state follows the AdditionOperatorState. The purpose of this state is to gather the rest of the digits in the operand that comes after an addition operator. This state also accepts the addition operator, subtraction operator, and newline character as valid inputs, and each of these inputs will transition the context to a separate state.
* Exit Successfully State = FinalSuccessState Class: This state is the final state for the program and is reached when the program reaches the end of a valid expression and displays the result of that expression. The program exits after reaching this state.
* Exit With Error State = FinalFailureState Class: This state is the final state for the program and is reached when the program encounters an invalid input character for the current state of the context. The program throws an exception before transitioning to this state which lets the user know why the expression is invalid and then exits after entering this state. Possible exceptions that could be thrown prior to transitioning to this state include IncompleteExpressionException, InvalidCharacterException, LeadingZeroException, and UnexpectedOperatorException.