SWE 261P Project Part 2

Project Overview

This report explores the application of finite state machines (FSMs) in software testing, using the Lifecycle component of an Elasticsearch server as a case study. Finite models, like FSMs, offer a structured approach to understanding and testing the behavior of software components, ensuring reliability and robustness.

Finite models

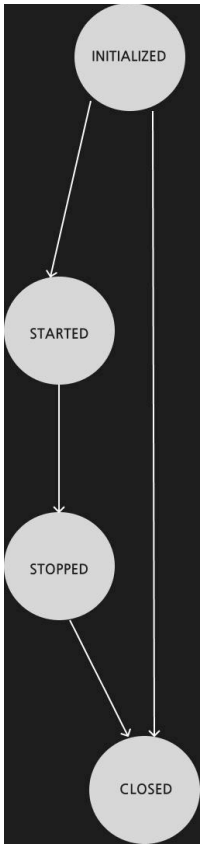
The power of finite models lies in their simplicity and precision. By breaking down the behavior of a component like Lifecycle into discrete states and allowable transitions, developers can create a clear roadmap of how the component should operate. This not only aids in the initial development phase but becomes invaluable during testing. Each transition between states can be tested individually, ensuring that the component behaves as expected in every conceivable scenario.

The Lifecycle component in Elasticsearch

The component we chose is:
server/src/main/java/org/elasticsearch/common/component/Lifecycle.java

state/function	moveToStarted	moveToStopped	moveToClosed	
Initialized	started	x	closed	
stopped	x	x	closed	
started	x	stopped	x	
closed	x	x	x	

```
 * <ul>
 * <li>INITIALIZED -> STARTED, CLOSED</li>
 * <li>STARTED     -> STOPPED</li>
 * <li>STOPPED     -> CLOSED</li>
 * <li>CLOSED      </li>
 * </ul>
 * <p>
```



The purpose of a lifecycle model is to manage the state of a component in a controlled and predictable manner. It ensures that the component goes through a proper sequence of states, and it can help prevent issues like memory leaks or inconsistent behavior due to the component being in an unexpected state. This type of model is common in resource management, where clean-up and proper handling of state transitions are crucial.

Upon its instantiation, a Lifecycle object is established in an 'Initialized' state. It is from this initial juncture that it may proceed to either a 'Started' or 'Closed' state. This progression is facilitated by the invocation of the moveToStarted function, transitioning the state to 'Started', or the moveToClosed function, leading to a 'Closed' state. Notably, once the Lifecycle object exits the 'Initialized' state, it is precluded from reverting to it.

The transition into the 'Started' state occurs after the moveToStarted function call, signifying that the object is now actively engaged in its designated tasks. The 'Started' state is designed to be transient, with the singular permitted transition being to the 'Stopped' state, which is effectuated by calling the moveToStopped function.

On the invocation of the moveToStopped function, the object transitions to the 'Stopped' state, indicating a cessation of active duties without reaching a conclusive end—it is essentially in a state of suspension. The progression from the 'Stopped' state to the 'Closed' state is unidirectional and is executed through the moveToClosed function.

The culmination of the object's lifecycle is the 'Closed' state, attainable either directly from the 'Initialized' state or following the 'Stopped' state. When an object arrives at the 'Closed' state, it marks a definitive endpoint; subsequent transitions are disallowed. At this stage, the

object is primed for garbage collection or must undergo recreation for future utilization. In the 'Closed' state, the object stands deactivated, concluding its operational lifecycle.

Test Cases

We added unit tests covering all the edges representing all the possible transitions in the Finite state machine, such as moving from started to stopped, from initialized to started, from stopped to closed, etc. We also verified exceptions are thrown when we call an illegal transition function on a state, for example when someone calls moveToClosed when the state is started.