EIC

Embedded Real-Time systems

Assignment 3 - Modeling and implementaion of a system behavior using UML patterns.

Name	Initials	student number
Jakob Bjerregaard Olsen	JBO	202109022
Alexander Nygaard Thomsen	AT	201911329
Wouter Oudijk 6	WO	202109059

Lecturer: Jalil Boudjadar, Kim Bjerge

Due date: 01/12/2025

1 Introduction

The main goal is this assignment is to implement a simulated version of an embedded system using state patterns, more specifically a concurrent state machine, where the handling of event are processed, by using *GOF State pattern*, Singletong pattern and the *Active object pattern*

In this assignment, the application has been implemented by adhering to these requirements, and following the provided state diagram given with the assignment. Some liberty has been taken in terms of implementation, regarding how to interact with the system, but also how the states and classes has been used and implemented. The goal of this report is to explain the major outlines of the decisions made along, with visual descriptions and depictions. This includes large code snippets and UML diagrams which will serve as context.

Included with this report is the full code implementation, an executeable file, all UML diagrams and code to recreate/recompile the diagrams.

1.1 tools used

To complete this assignment some tools have been used.

Tool	**Description**
VS Code	IDE, for programming the assignemt
C++	the programming language required to implement the assignment and a compiler for C++14++ is required.
LaTeX	using MIKTEX
plantUML	UML tool of choice, converting it to a programming language, requires Java JDK
Git	for version control, using GITHUB to control the repo

Table 1: Tools used to complete the assignment

1.2 Goals

- **GOF Singleton pattern** To ensure the correct behavior of the statemachine, the Singleton pattern must be used, it ensures that only one instance of every object class can exist at any given time.
- **GOF state pattern** This pattern ensures that an object changes behaviour when it changes state, but to the outside world it looks like a new object. Instead of enumeration, the state and its behaviour is encapsulated in multiple classes, where our object can delegate to the current state class.
- **Active Object pattern** The main purpose of the active object pattern, is it separate method invocation from method execution, by decoupling. This lets us asynchronously invoke methods while not tying up valuable time to execute those methods, by having separate threads do both scheduling and execution.
  - Active object pattern must implement the provided interface
  - not be blocking
- **follow the provided state pattern** With the assignment, the *EmbeddedSystemX* class was provided, which outlined the overall requirement for the application.

2 Solution

2.1 Overall architecture

- **EmbeddedSystemX** Is the overall class which implementes all functionality of the application any user interaction is done through a simple text based user interface written in main.

To avoid unncessary C++ semantics, the entire composition of EmbeddedSystemX and all its subcomponents is written in one large *.cpp* file, this is has mainly been done to reduce compiler complexity and and practical implementation difficulties, which is bound to arise, due to things such as *circular-referencing*.

2.1.1 State Diagram

The overall structure follows the provided model of the system

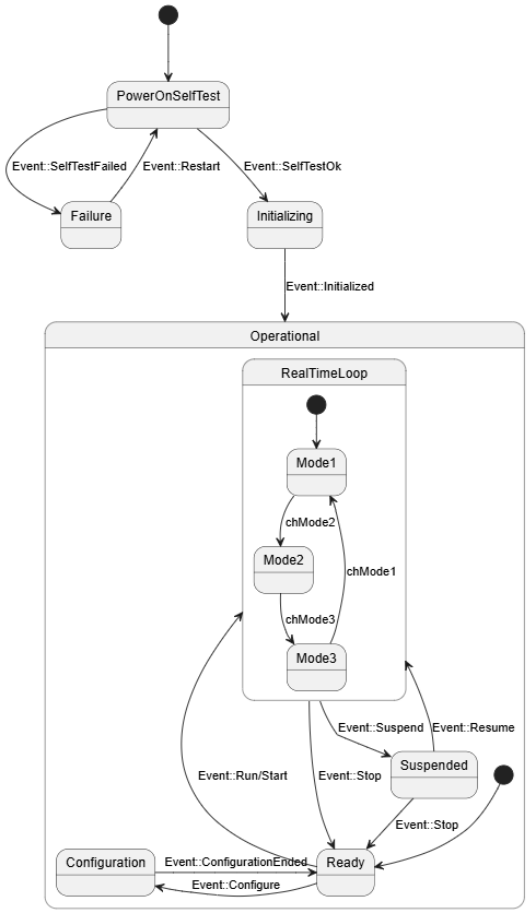

Figure 1: Overall statemachine of EmbeddedSystemX

There is a focus on the inner statemachine inside of the *Operational* class, which contains a nested statemachien In this case referred to as *opFsm - operational state machine*.

2.1.2 Class Diagram

To comform with the requirement of the GOF state machine, all of the states are implemented as classes, which are inheriting from the *State*-class

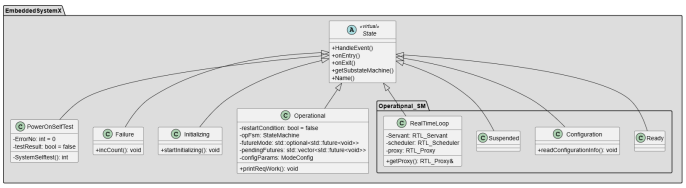

Figure 2: The State class, is a pure virtual class, which will be explained in a later section

The states are implemented this way, so that the statemachine trusts the state class with all the behavior of the implemented classes, this means that they all have the some core functions: *eventHandler(), onEntry(), onExit()* and so on. It only fits together with the the singleton pattern. So matter what iteration of the state class, there only exists one instance of that class, that has the same public interface as the previous instance.

2.1.3 Sequence Diagram

the sequence of execution in the overall state machine is as follows

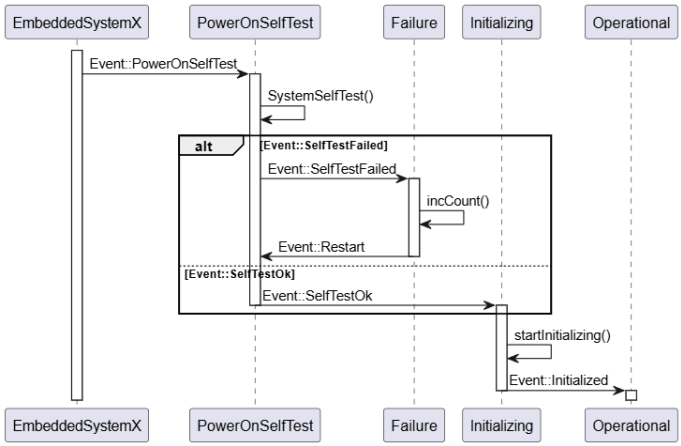

Figure 3: The initial phase of the EmbeddedSystemX's statemachine, here it automatically executes through the initial states. A small chance has been added so that the SelfTest might fail. Else it justs continues to the ready state. It also shows the lifetime of the states as objects.

The operational class is where the internal statemachine gets activated by the *onEntry()* function inside the operational class. That signifies the start of the operational sequence diagram

The important thing in this sequence diagram here is, that the *RealTimeLoop* is an active object, meaning that its method invocation has been decoupled from its exection, this is also shown in figure: 1. Where to overall mechanism follow the following class diagram.

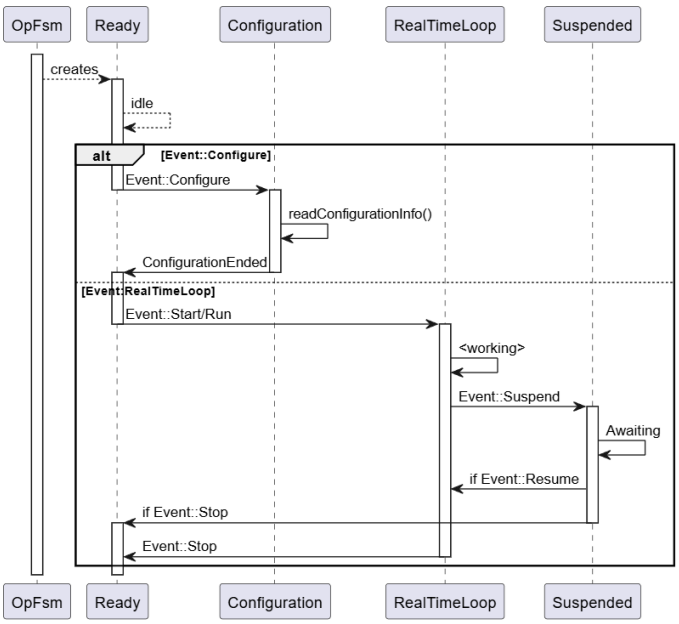

Figure 4: The internal statemachine *opFsm* inside operational, this is a concurrent statemachine to the on outside, however the two don't have any interplay. As with the outside statemachine, this also have classes as states, implemented by the state/eventHandler mechanism, all are singletons aswell. All data passed through opFsm, are referenced to the outside machine.

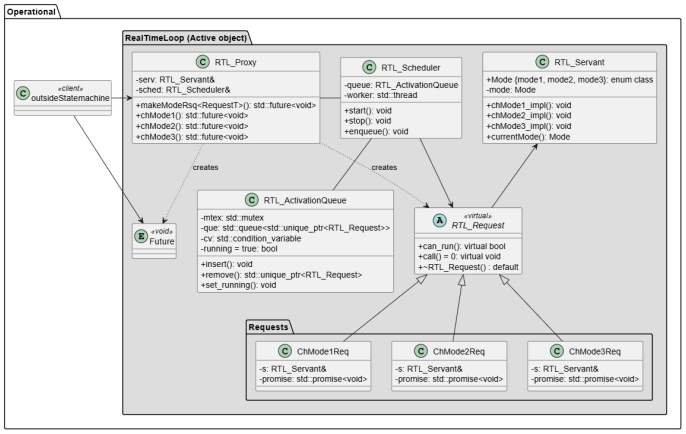

Figure 5: The internal class diagram of the active object, which resides inside the RealTimeLoop class

3 Implementation

```
#include <iostream>
int main() {
    std::cout << "Hello␣World!";
}
```