Architecture and Design of

Embedded Real-Time

Systems

Journal on Assignment 3

Group 4

**Authors:**

Rasmus B. Langhoff - study number 201506966  
Martin W. Kjær - study number 201509268

Mads M. Beck - study number 201508319

**Supervisor:**

Jalil Boudjadar

Contents

[1 Introduction 2](#_Toc25225001)

[1.1 Intro to requirements for the exercises 2](#_Toc25225002)

[1.2 Patterns used in the solution 2](#_Toc25225003)

[2 Solution 2](#_Toc25225004)

[2.1 Introduction to architecture and decisions 2](#_Toc25225005)

[2.2 Logical View 2](#_Toc25225006)

[2.2.1 State Diagram(s) 2](#_Toc25225007)

[2.3 Implementation View 2](#_Toc25225008)

[2.3.1 Implementation details 2](#_Toc25225009)

[2.4 Process/task View (optional) 2](#_Toc25225010)

[3 Discussion of results 2](#_Toc25225011)

[4 Conclusion 2](#_Toc25225012)

[Appendix A 2](#_Toc25225013)

**Revision History**

|  |  |  |
| --- | --- | --- |
| Revision | Date | Description |
| 1.0 | 25-11-2019 | Initial document - started writing initial work done in the assignment |
| 1.1 | 26-11-2019 | Finished documentation |
|  |  |  |
|  |  |  |

# Introduction

In this assignment, the goal is to gain experience with the GoF State pattern and GoF Singleton pattern by implementing a state machine shown below in figure 1:

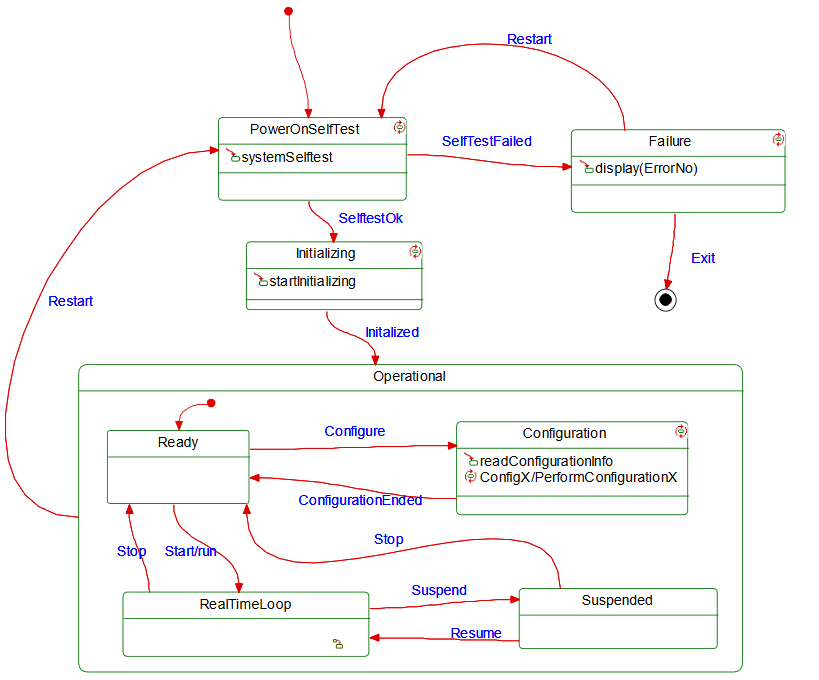


Figure 1 State diagram of EmbeddedSystemX

In the last part of the assignment, the state machine implementation is extended and the GoF Command pattern is to be implemented.

## Patterns used in the solution

In the following paragraphs, the three patterns used in the assignment are described thoroughly. These are the GoF State pattern, the GoF Singleton pattern, and the GoF Command pattern, respectively.

The GoF Sate pattern is a behavioral pattern, which is intended to allow an object to alter its behaviour whenever its internal state changes. Thus, the object will appear to change its class, that is, the class of its state. The main idea behind the pattern is to introduce an abstract class in order to represent the states of the system. This abstract class declares an interface which is mutual for all of the different classes representing each of the various states of the system. The subclasses of the abstract class are used to implement state-specific behaviour of the state classes. Each of the state classes maintains an instance of one of the subclasses of the abstract class. The state classes then use this object to delegate all the state requests to, and use the subclass instances to perform operations which are particular to the given state. Whenever the system changes state, the class state instance changes the state object it should use.

The GoF Singleton pattern is a creational pattern, which is used to ensure that each class only has one instance. It also provides a global point of access to it. This is done by making the classes responsible for keeping track of its sole instance. If the instance already exists, it should return the sole instance. Otherwise, it should create a new instance.

The GoF Command pattern is a behavioral pattern, which is intended to encapsulate a request as an object. This entails, that clients can be parameterized with different request, queues, or log requests. This is done by implementing an abstract command class, which has declared an interfaced for executing operations. The subclasses to the command class then specifies a receiver-action pair by storing the receiver as an instance variable, and by also implementing an abstract execution operation from the command class, which invoke the request. The knowledge to carry out the request is known by the receiver.

# Solution

## Introduction to architecture and decisions

In this assignment a state machine is implemented, and as such, the architecture is based upon an event-driven architecture, as the system only waits for signalling between state transitions, i.e., events. Due to the state machine not having any internal operation, the events are triggered via user input. In a real-world application, this architecture could be extended to a Two Part architecture, if a continuous processing part is needed for the system.

The design decision have been made purely in regard to implementing the three types of pattern described in section 1.1.

## Logical View

### Class Diagram

Below in Figure 2, the overall class diagram of the system is shown:

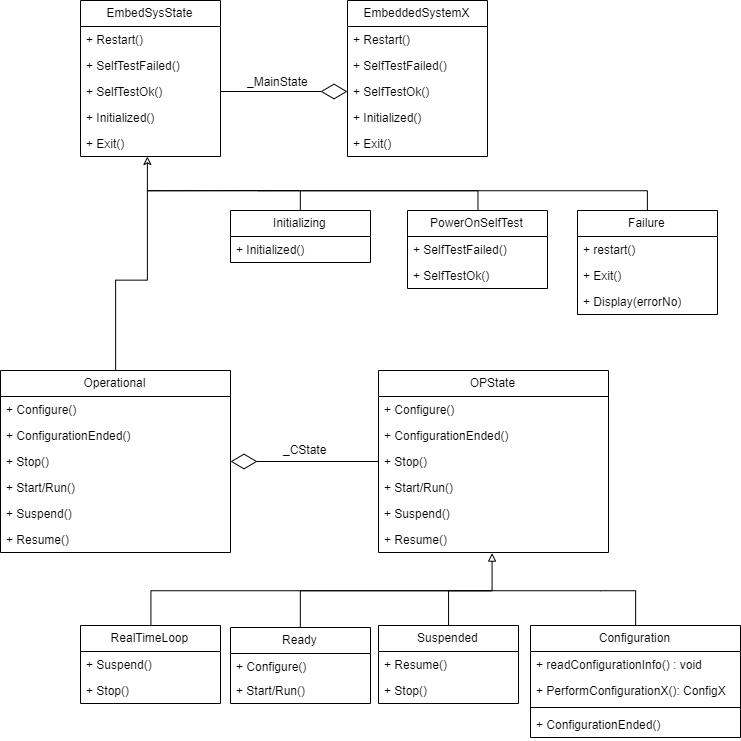


Figure 2 System class diagram

Here the state class EmbeddedSystemX is shown, which contains the different states of the system. It has an object of the abstract class EmbeddedSysState, which provides the interface for the three subclasses, which are responsible for implementing the various events for the system. This conclude the architecture related for the overarching state machine. The internal / nested state machine can be seen connected to the EmbbededSysState virtual class, which also has an object of its own abstract class, OPState, which provides the interface for the four subclasses implementing the events for the internal / nested state machine. As is evident from the description of the GoF State pattern in section 1.1, this class diagram is made in accordance to that.

## Implementation View

### Implementation details

In the first part of the assignment, the goal was to implement the state machine from Figure 1 using the GoF State pattern. In the following sections, code snippets showcasing the overarching state machine design with one of the event subclasses using this pattern is shown. Below in Figure 3, the implementation of the class EmbedSysState is shown:

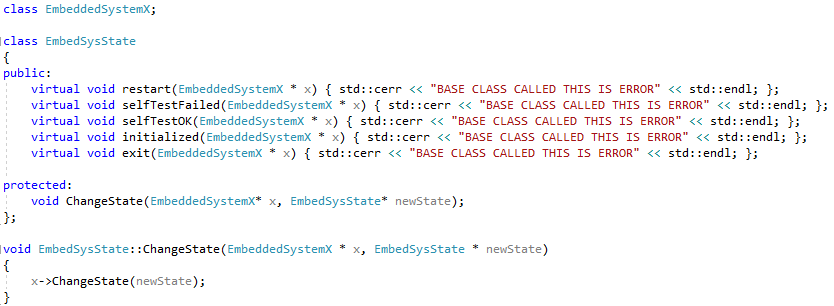


Figure 3 Code snippet of class EmbedSysState

Here the abstract class is shown, with the virtual functions defining the interfaces for the different event subclasses. A method to change the state of the system is also implemented, which has the arguments of a pointer to an EmbeddedSystemX object and a pointer to an EmbedSysState object. These are used to access and use the function ChangeState seen in Figure 4 below.

In figure 4, the class diagram of the context EmbeddedSystemX is shown. Here the top-level state machine is implemented, and the various events are defined by the functions, which are triggered by a user in the main file, as can be seen in Figure 11. As can also be seen, there is an instance of the EmbedSysState class, which is used to switch between the states, by using the function ChangeState. The getObj function is used to trigger the states in the internal / nested state machine in the Operational class, which is seen in Figure 6. The constructor and one of the event function implementations, the restart event, is also visible in Figure 4.



Figure 4 Code snippet of class EmbeddedSystemX

In Figure 5, a snippet of one of the subclasses is shown, which is the PowerOnSelfTest class. Here various functions are implemented as the events of the EmbeddedSystemX state machine, belonging to specific PowerOnSelfTest state. Classes similar to this have been implemented for the remaining states as well, but has been omitted for this journal for practical reasons. As it can also be seen in the instance function, the state is implemented according to the GoF Singleton pattern.



Figure 5 Code snippet of class PowerOnSelfTest

In the latter part of the assignment, the internal / nested state machine Operational was to be implemented using the GoF Command pattern. The implementation can be seen below in Figure 6. The implementation is similar to that of EmbeddedSystemX, but it also implemented as a state in the EmbeddedSystemX state machine. It can also be seen, that a command handler function is implemented in order to comply with the GoF Command pattern. As also was shown in the other state class, PowerOnSelfTest, the Operational class has been implemented using the GoF Singleton pattern as well.



Figure 6 Code snippet of class Operational

Below in Figure 7, a code snippet for the abstract OPState class is shown, which like the former abstract class, EmbedSysState, provides an interface for the various subclasses.

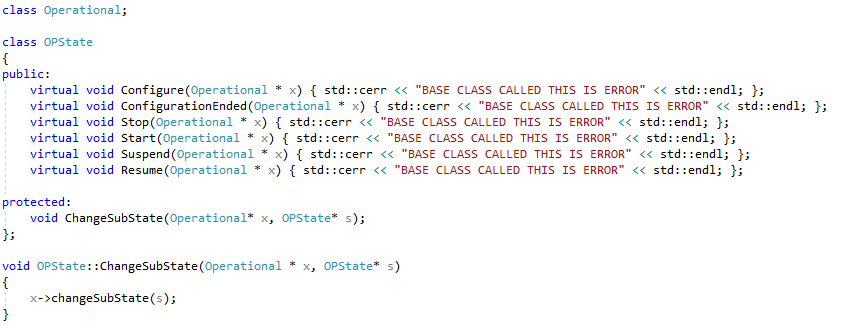


Figure 7 Code snippet of class OPState

In Figure 8 below, the Command class is shown, which is the base class for the state machines various command classes.

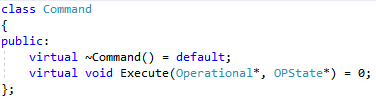


Figure 8 Code snippet of class Command

Below in Figure 9, the command class Start is shown. Here an implementation of the Execute function is shown, which calls the Start function implemented in the Ready class, which is shown in Figure 10 on the next page.

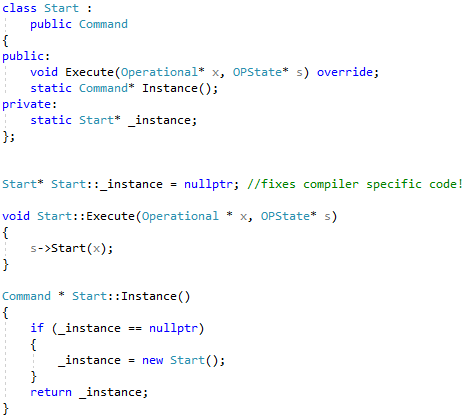


Figure 9 Code snippet of class Start



Figure 10 Code snippet of class Ready

In Figure 10, the Ready state class is shown, where the events are implemented as functions. This is much like the implementation shown earlier in Figure 5.

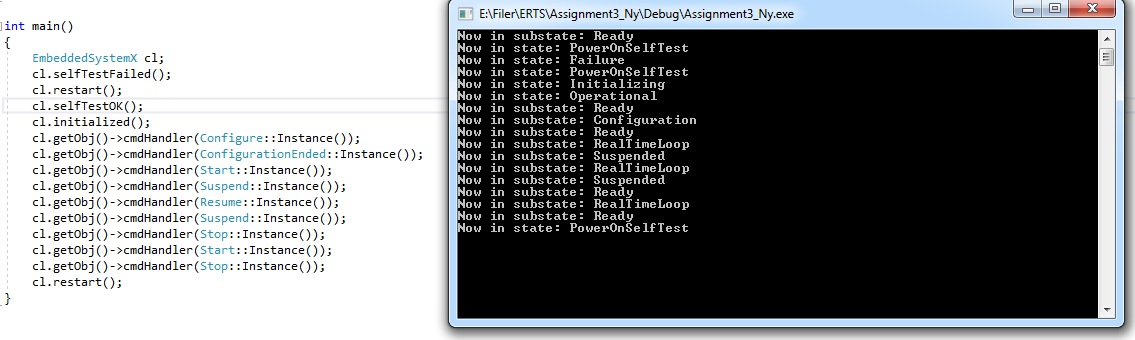


Figure 11 Code snippet of main and terminal output

Lastly, in Figure 11, a test case is shown of the EmbeddedSystemX state machine. Here it can be seen, that a user calls the events based on the GoF State pattern design, when transitioning between the overarching states, and that the user class the commands based on the GoF Command pattern design, when transitioning between the internal / nested states. The figure also shows the console output, which prints the state and sub state, that the state machine has changed to. The console output is as expected.

# Discussion of results

The EmbeddedSystemX state machine with the internal / nested state machine is working as intended, which is confirmed in Figure 11.

When using command patterns the complexity goes up, since there is a need for a whole class to be implemented that represents the command. For the case of state machines, the functionality of the commands is rather simplistic where they just call the changeSubState function of the parent class. This design pattern also implies, that it is easy to add new commands, because there is no need to change existing classes.

In terms of performance, it will always depend on the optimization done by the compiler. However, it does add overhead, e.g., compared to using only the GoF State pattern. Although, when the state machine is large and complex enough, this overhead might be negligible compared to the benefits added by using the GoF Command pattern.

# Conclusion

The GoF State pattern, GoF Singleton pattern, and the GoF Command pattern have all been implemented in the EmbeddedSystemX state machine, working as intended. The use of the GoF State pattern and GoF Command pattern, while a required part of this assignment, is not necessarily the most optimal solution for a state machine like EmbeddedSystemX. That is, the overhead compared to using simple switch-case statements is quite big compared to the benefits, when implementing a state machine which is not very complex like the one implemented in the assignment.