Architecture and Design of Embedded Real-Time Systems

Journal on Exercises 3

Group 10

Authors:
Mads Krabsen, 201507805
Jonas K. Rask, 201507306
Frederik P. Madsen, 201504477

Supervisor:
Jalil Boudjadar

Contents

Revision History

Revision	Date/Authors	Description
1.0	23.11.2019/MK	Document created
1.1	23.11.2019/MK	Introduction, Requirements, Patterns and UC view initiated

1 Introduction

This Journal is made as an assignment for the Embedded Real-Time Systems course at Aarhus University. The journal will consist of a short description of the requirements of the system, then an identification of the design patterns used to realise the system and then a short description of the architecture and design using the 4+1 software engineering model.

1.1 Intro to requirements for the exercises

The requirements of the exercise will be stated here:

1) The EmbeddedSystemX must be implemented using GoF State Pattern
2) Each state from the GoF State Pattern must implemented using Singleton pattern
3) The command pattern must be used to implement the processing of the sub states within state Operational

1.2 Patterns used in the solution

The solution for the system relies on three design patterns. To realise the state machine in the exercise the GoF state pattern is used. The state pattern as described by the GoF is like the class diagram on Figure 1 where each concrete state inherits from the state 'TCPState' that has the prototype for each of the events possible in the system.

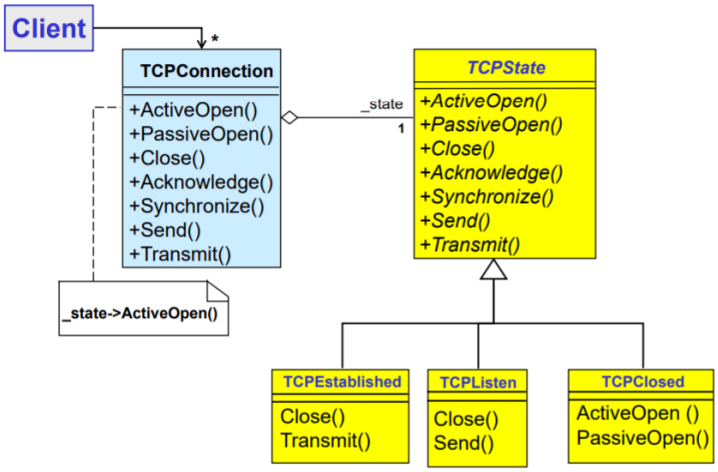

Figure 1 State Pattern Example

To ensure that each state object of the state patterns is not created and destroyed every time a new state is entered and exited each inheritor of the state class is implemented using the Singleton pattern. This ensures as described by GoF that a static instance of the class is created the first time with a call to the static function Instance(). An illustration of the singleton pattern can be found on Figure 2.

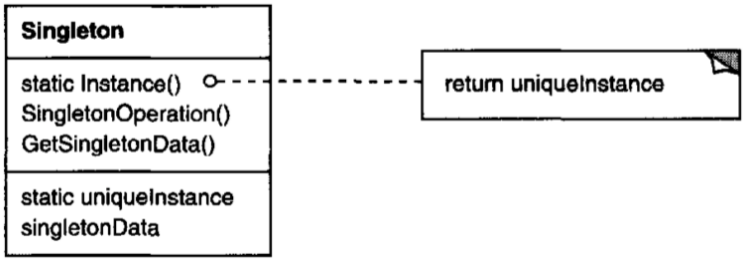

Figure 2 Generic Structure of Singleton Pattern

To abstract the user interface from the internal workings of the EmbeddedSystemX a command pattern is implemented so that each action performed by the user is abstracted away from the implementation of the action.

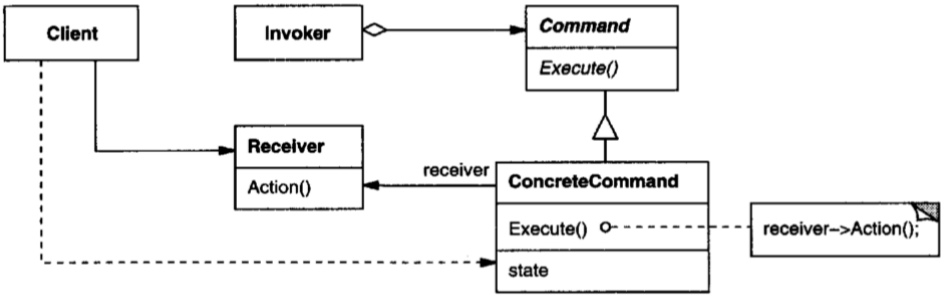

Figure 3 Generic Structure of Command Pattern

2 Solution

2.1 Introduction to architecture and decisions

To implement the state machine, the State Pattern described in section 1.2 is used. The in the abstract state superclass all event operations have a default implementation. This way it is only necessary to implement event operations for the actual state.

To implement the event handling we use a solution where all classes has a public function for each event, as seen on Figure 4.

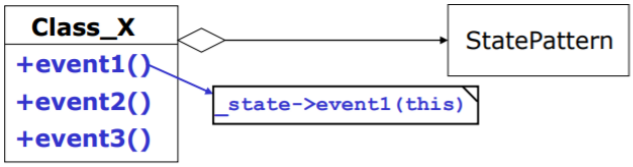

Figure 4 Event handling solution example

To make the hierarchic state machine of state Operation, the sub states inherit from Operation that is then implemented as an abstract class.

To include the command pattern, we chose to create an implementation where all events are handled as commands not just the ones in state Operation.

2.2 Use Case View

The EmbeddedSystemX offers three main use cases for the user to interact with, these can be seen on the use case diagram on Figure 5. The use cases are derived from the system behaviour described in state machine diagram on Figure 11. Here is can be identified that the user is responsible for triggering the state transitions:

State transition	Description
PowerOnSelfTest → Failure	If the initial PowerOnSelfTest is failed the user must trigger either a system exit or a restart.
Ready → Configuration	When the system is in operational ready state the user must trigger transition to configure state.
Ready → RealTimeLoop	When the system is in operational ready state the user must trigger transition to RealTimeLoop state. This could be a sensor read, actuator actuate operational loop
RealTimeLoop →Ready	The user must stop the system from executing
RealTimeLoop → Suspended	When the system is executing the real time loop the user must trigger the suspend
Suspended → RealTimeLoop	When the system is suspended the user must trigger the resume.
Operational → PowerOnSelfTest	The user must trigger a restart so that the system can go from operational state to PowerOnSelfTest state

The functionality described in the table above is summed up in the Actor context diagram below

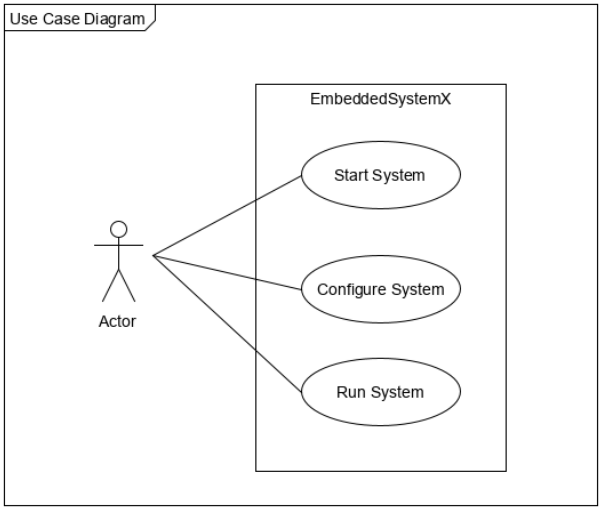

Figure 5 Actor-context diagram for the EmbeddedSystemX

2.2.1 Start System

The Start System use case follows the general use case description below:

UC 1: Start System	**Description**
Actor	User
Post Condition	System is in Operational Ready state
Trigger	User power on system
Main flow	
1.	User power on system
2.	System performs PowerOnSelfTest [Exception: SelfTestFailed]
3.	System performs Initializing
4.	System enters Operational Ready state
5.	User is informed of System state
Exceptions	
[SelfTestFailed]	
1.	System informs user that system self-test fails
2.	User restarts system
3.	UC continues form step 2.

2.2.2 Configure System

UC 1: Start System	**Description**
Actor	User
Precondition	System is in Operational Ready state
Post Condition	System has updated configuration
Trigger	User trigger configuration
Main flow	
1.	User enters configuration
2.	System enters Configuration state
3.	System reads configuration information
4.	System informs user that configuration is done
5.	System returns to Operational ready state

2.2.3 Run System

UC 1: Start System	**Description**
Actor	User
Precondition	System is in Operational Ready state
PostCondition	System returns to Operational Ready state
Trigger	User trigger start/run
Main flow	

1.	User trigger start/run
2.	System enters RealTimeLoop state
3.	System runs real time loop indefinitely [Exception: Suspend]
4.	User stops system
Exceptions	
[Suspend]	User suspends system
1.	User trigger suspend real-time loop
2.	System enters Suspended state
3.	User trigger Resume real-time loop
4.	UC continues from step 2

2.3 Logical View

The main goal of the logical view is to define the components that make up the system and to define the interfaces through which they will communicate and interact with each other.

2.3.1 Class diagram(s)

For the implementation only using the GoF state patteren, i.e. not using the command pattern. The class diagram seen on Figure 6 is realized. The user can then access the functions of EmbeddedSystemX through a simple UI created in the main file. Note that for this implementation to be viable the classes 'EmbeddedSystemState' and 'EmbeddedSystemX' must have each other as *friend*.

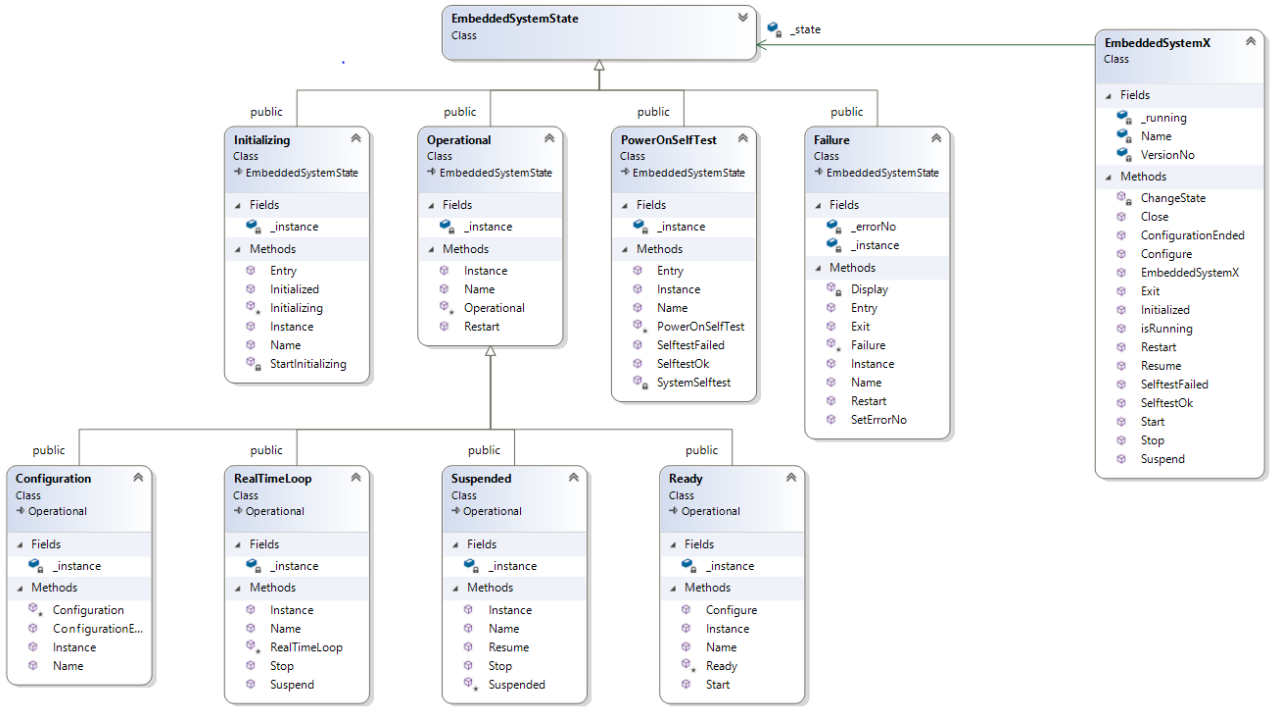

Figure 6 Class diagram for EmbeddedSystemX using the GoF state pattern

For the implementation using command pattern the class diagram seen on Figure 7 and Figure 8 are implemented.

Figure 7 shows how the command pattern integrates with the state machine depicted on Figure 6 by the association between "EmbeddedSystemX" and "command". Figure 8 shows the implementation of the command pattern with concrete commands inheriting from the abstract base class "command".

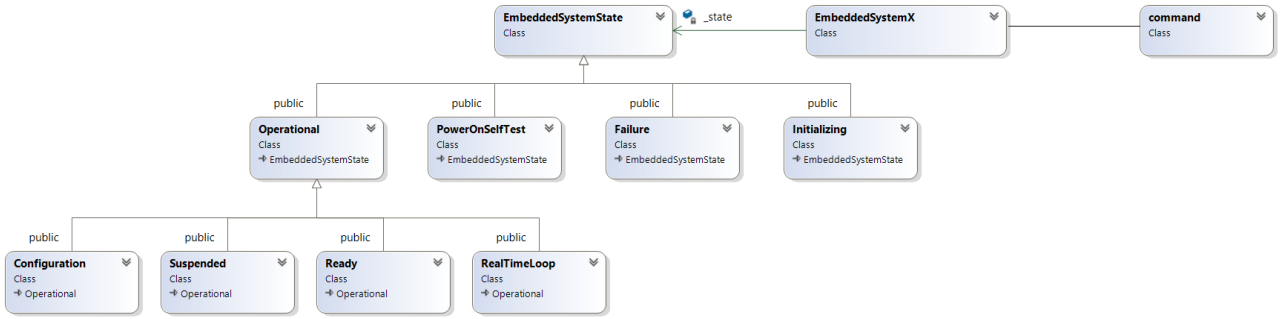

Figure 7 Class diagram for EmbeddedSystemX using the GoF state pattern in concert with command pattern, part 1

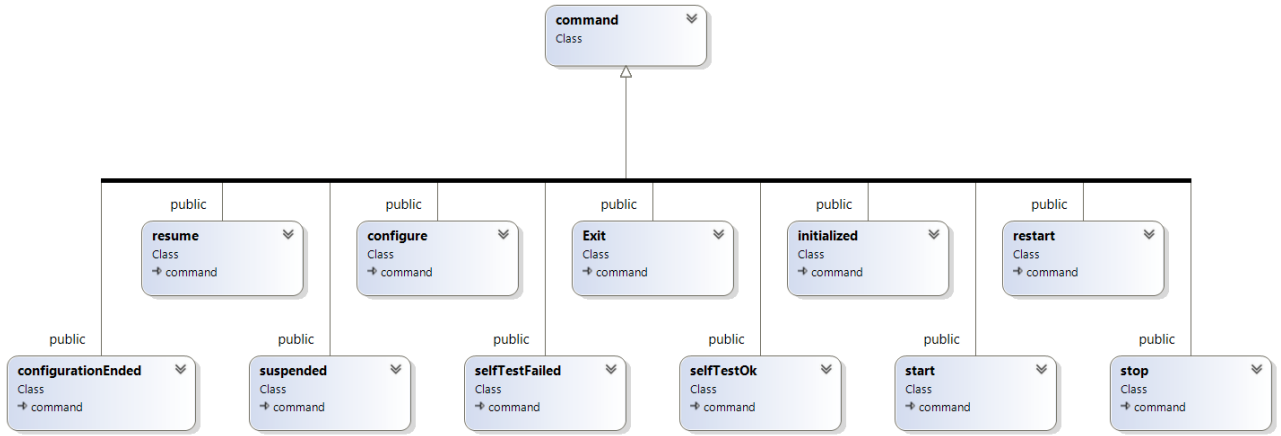

Figure 8 Class diagram for EmbeddedSystemX using the GoF state pattern in concert with command pattern, part 2

2.3.2 Sequence diagram(s)

An example of a call sequence not using the command pattern can be found on Figure 9. On the figure it is seen how a call to the EmbeddedSystemX's SelfTestOk() function executes the first time in the state PowerOnSelfTest. Here the state PowerOnSelfTest creates an instance of state Initializing that is saved for later, following the Singleton pattern. The state in EmbeddedSystemX is then changed to Initializing by PowerOnSelfTest that knows that this is the response to a SelftestOk event. After this the ChangeState actives the Entry() function in the new state, that for Initializing will call startInitializing().

All the other states follow a similar sequence when they execute an event that triggers a state change.

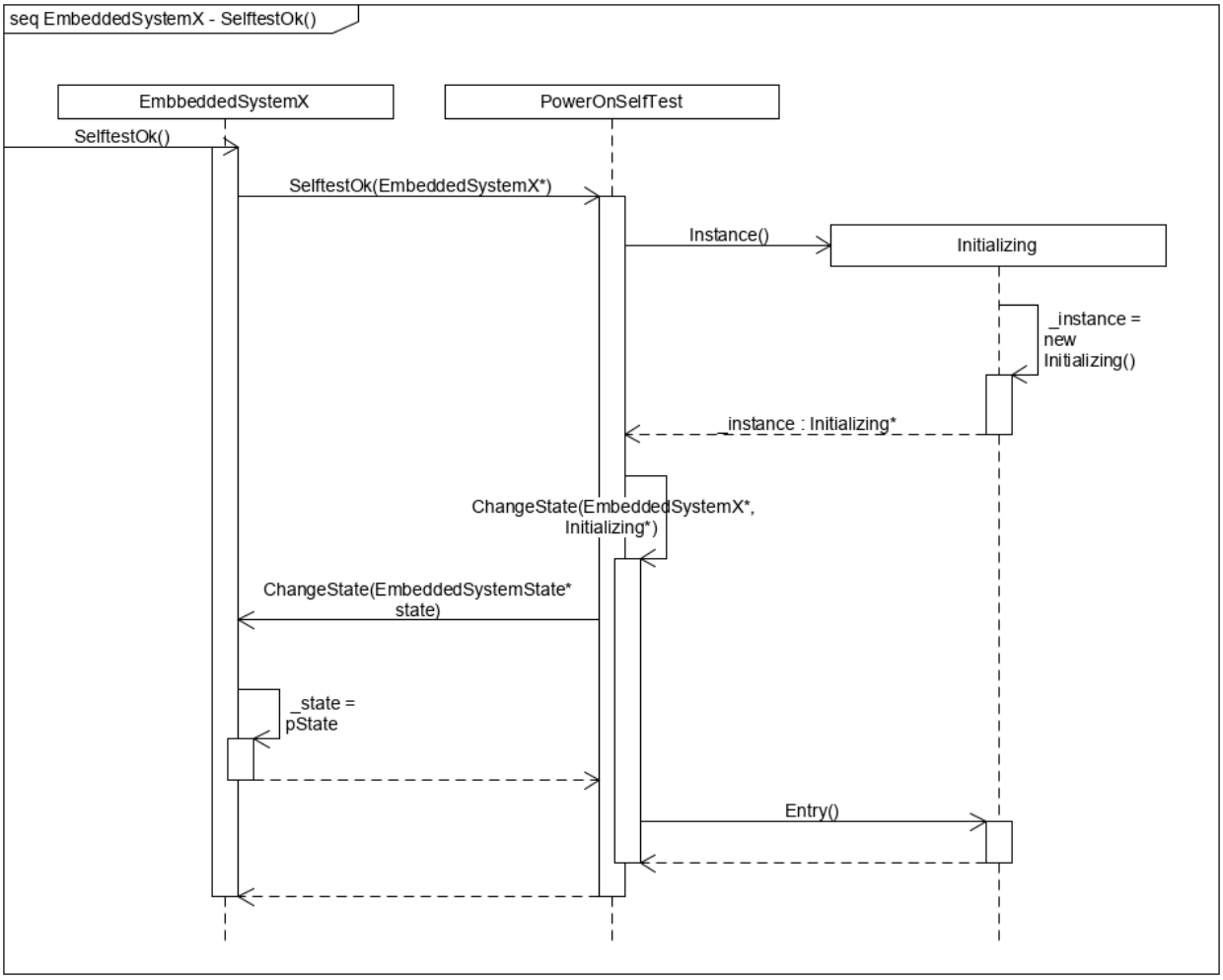

Figure 9 Sequence diagram for PowerOnSelfTest() not using the command pattern

The command pattern implementation is seen on Figure 10 deviates from the previous implementation in how the state machine functions are invoked. The Client, which interacts has the UI creates a command object, in this example the SelftestOk object. When the Client invokes the handleCommand() function in the EmbeddedSystemX class. This invocation could be handles by a separate invoker class which holds a queue of pending commands. This implementation would be necessary if a concurrent version of the system should be implemented. The Commands have an execute() function which invokes the state functions and from there the flow in identical to the normal state pattern.

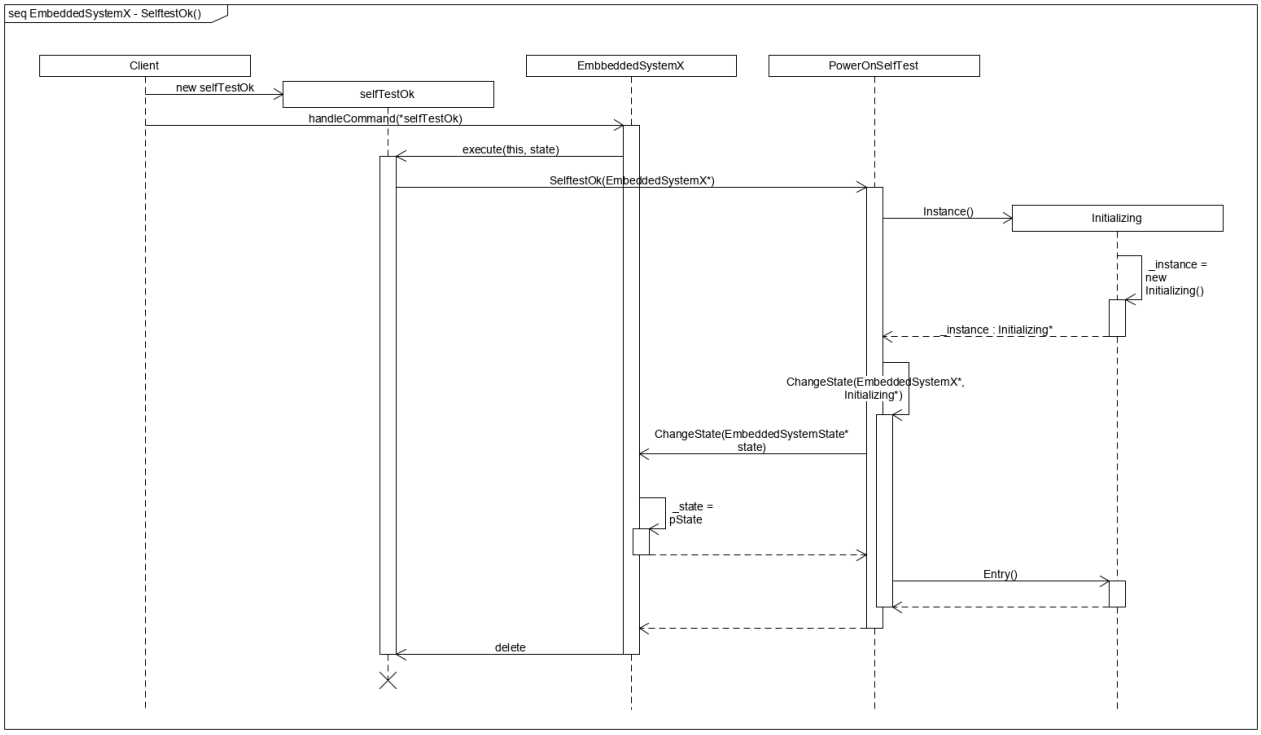

Figure 10 Sequence Diagram with command pattern

2.3.3 State Diagram(s)

Figure 11 shows the state diagram for EmbeddedSystemX. Each event/transition is given by an arrow with an event name. The initial state is PowerOnSelfTest and the only way the state machine exits is if the self-test has failed, i.e. the system is in state Failure and an Exit event is triggered.
In state Operational the system has four different sub states, thus any event triggered while in state Operational will propagate into the sub state machine. If the system is in state Operational a Restart event will change state to PowerOnSelfTest no matter which sub state is active.

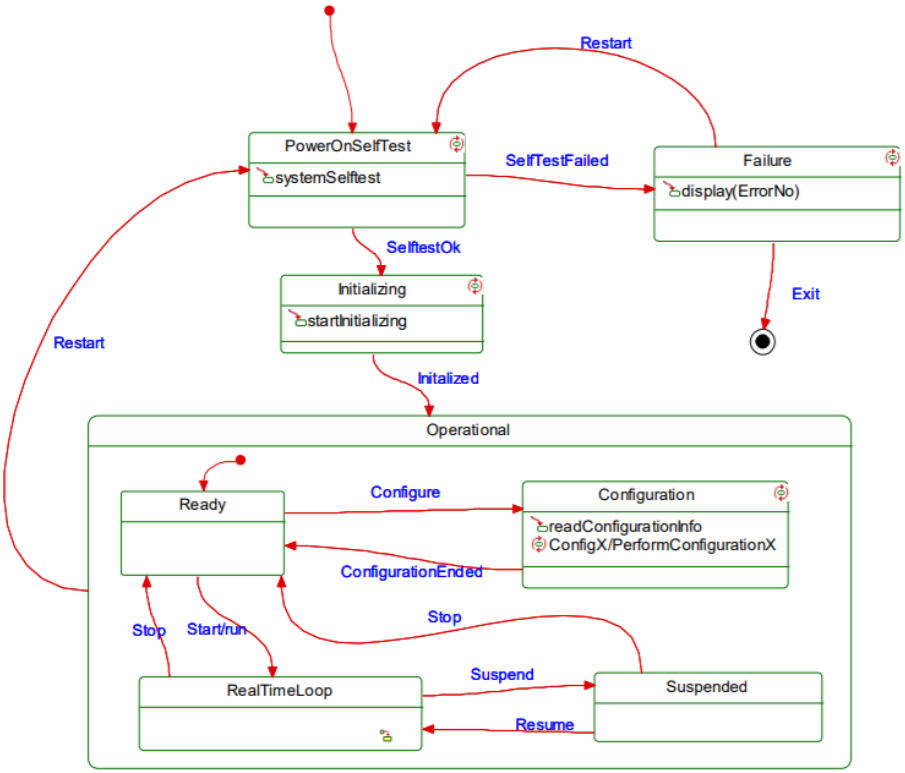

Figure 11 State Diagram of EmbeddedSystemX

2.4 Implementation View

In this section some of the implementation details regarding the specific patterns will be shown and code snippets added to illustrate.

2.4.1 Implementation details

State pattern with Singleton states

To implement the specific states om the state pattern a singleton was implemented for each state as seen on the example in Figure 12. The Instance() function ensures that only one instance of the state object will be created. The constructor of each state is declared protected so that only the Instance() function is able to call it.

```
5     PowerOnSelfTest* PowerOnSelfTest::_instance = 0;
6
7     PowerOnSelfTest* PowerOnSelfTest::Instance()
8     {
9         if (_instance == 0)
10            _instance = new PowerOnSelfTest;
11
12        return _instance;
13    }
```

Figure 12 Singleton pattern snippet from PowerOnSelfTest

Changing state

The responsibility of changing state is handed to the specific states, therefore each states state function, in Figure 13 the Initialized function, the this pointer of EmbeddedSystemX is used by the states to call the changeState() function and set the next state.

```
3   void EmbeddedSystemState::ChangeState(EmbeddedSystemX* pESX, EmbeddedSystemState* pState)
4   {
5       std::cout << "Changing state to: " << pState->Name() << std::endl;
6       pESX->ChangeState(pState);
7       pState->Entry();
8   }
```

Figure 13 changeState function of EmbeddedSystemState

Command pattern

The command pattern is implemented such that each function of the state machine is converted into a command object. Each object is then able to trigger the function. On Figure 14 the Execute() function of the configure command is shown. It is implemented as a sort of wrapper that trickers the states function and gives the EmbeddedSystemX pointer so that the state changes can happen as before with no change.

```
3   void configure::Execute(EmbeddedSystemX* context, EmbeddedSystemState* state)
4   {
5       state->Configure(context);
6   }
```

Figure 14 execute function for the configure command

The EmbeddedSystemX has a handleCommand() function that receives the command object from the client. The client which have the user interface is seen on Figure 16 here the implementation is made in main. The user can create commands and as seen in Figure 16 these are passed to the EmbeddedSystemX.

```
66  void EmbeddedSystemX::handleCommand(command* cmd)
67  {
68      cmd->Execute(this, _state);
69      delete cmd;
70  }
```

Figure 15 handleCommand function of EmbeddedSystemX

```
34      EmbeddedSystemX system;
35      command* pCmd = nullptr;
36      while (system.isRunning())
37      {
38          char keypress;
39          std::cin >> keypress;
40          switch (keypress)
41          {
42          case '1':
43              pCmd = new selfTestOk;
44              break;
45
46          case '2':
47              pCmd = new initialized;
48              break;
```

Figure 16 UI code snippet, implemented in main

3 Discussion of results

The EmbeddedSystemX has been implemented using two different approaches to event handling in the state pattern used to implement the state machine depicted on Figure 11.

The first solution uses event handling as depicted on Figure 4. Using this solution, the context class (EmbeddedSystemX) has a tight coupling to the actual state changing event, as each of the events are represented as a concrete method in the context class. Thus, adding a new state change event would result in changes in all classes of the system, thereby increasing the overall coupling and decreasing the coherency which increases the overall system complexity.

As an alternative implementation of the events in the state pattern the command pattern can be used, as done in the second implementation of the state machine on Figure 11. In this implementation the context class (EmbeddedSystemX) has a low coupling to the actual state changes as these are done through commands. Therefore, the context class needs only one function that takes a command as input, removing the need for any coupling between the context class and concrete implementations of state changes. If a new state change event is to be added, the system needs only to be extended with a new concrete state class and a concrete command class. This lowers the overall system coupling while increasing coherency which in turn lowers the overall complexity of the system.

Looking at performance differences between the two implementations, using the command pattern in concert with the state pattern, the resulting decoupling enables implementation of the system on two threads. One thread running executing context/client functions while the other executes states and state changes. This would not be possible using only the simple event structure as depicted on Figure 4.

4 Conclusion

In conclusion the state pattern allows the implementation of a state machine in a maintainable fashion that allows state changes to be simplified. The addition of the command patterns improves the scalability of the system as it decreases the coupling in the system.