Architecture and Design of Embedded Real-Time Systems

Journal on Exercises 12/11/2018

Group 9

Authors:
David Jensen
Henrik Bagger Jensen

Supervisor:
Jalil Boudjadar

Contents

1 Introduction

In this exercise we will implement a State Machine with the GoF State Pattern and the GoF Singleton Pattern. This will show two patterns, that can work together.

1.1 Patterns used in the solution

To solve this exercise, it has been a demand to use the GoF State and Singleton patterns.

2 Solution7

2.1 Introduction to architecture and decisions

By design we are using the Gof State to model our system states, and to make sure we keep track of what state we are in, the Singleton helps us. This makes sure what even if we revisit a state we will use the same instance.

2.2 Use Case View

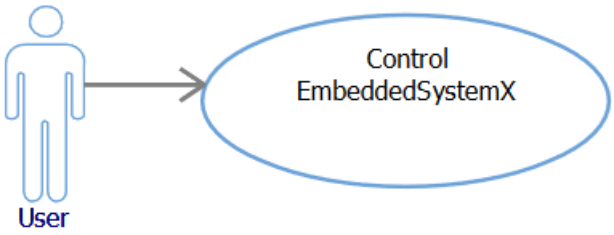

Figure 1 - Use Case Diagram

The user interacts with Control EmbeddedSystemX. This assignment gave the use case diagram seen on Figure 1.

2.3 Logical View

2.3.1 Class diagram(s)

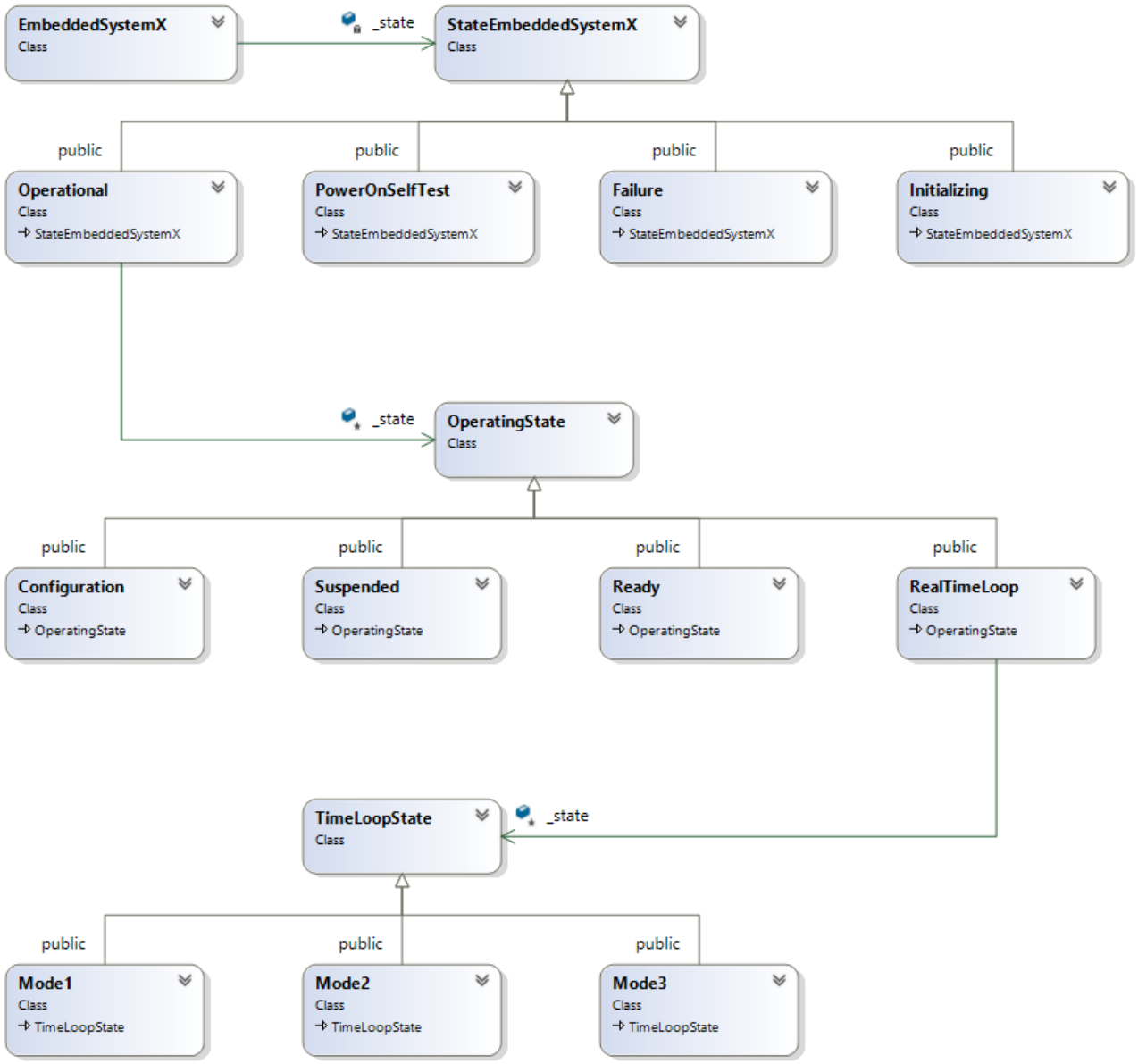

Figure 2 - Overview Class Diagram

All the classes in the solution is seen on Figure 2. The _state is controlled by different classes depending on which, states are need. The top state everything that can be used by the entire system, but as we decent down the class hierarchy. The specific states are controlled. An example of this is when the code is in the Operational state, it then has a state that control the states within it Configuration, Suspended Ready and RealTimeLoop. This allows the code to a clear separation of concerns, because the restart action is handled here for everything. On Figure 3, Figure 4, Figure 5 and Figure 6 more detailed class diagrams of the different classes are given.

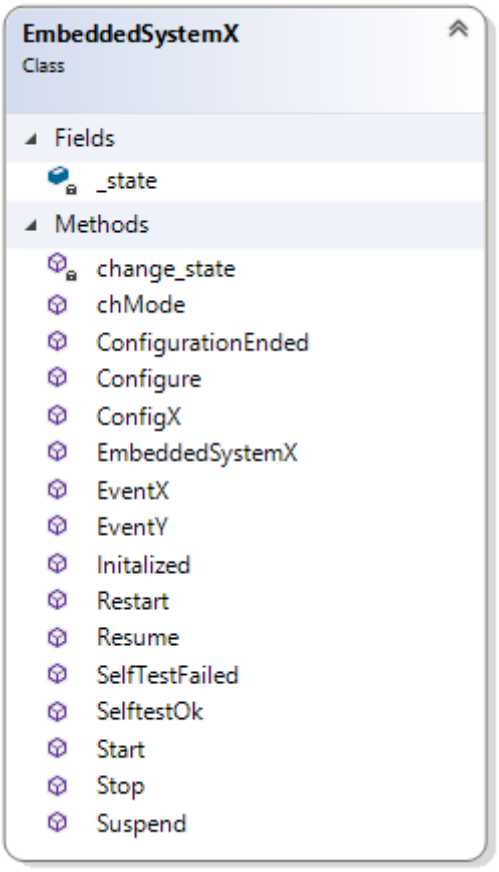

Figure 3 - EmbeddedSystemX Class Diagram

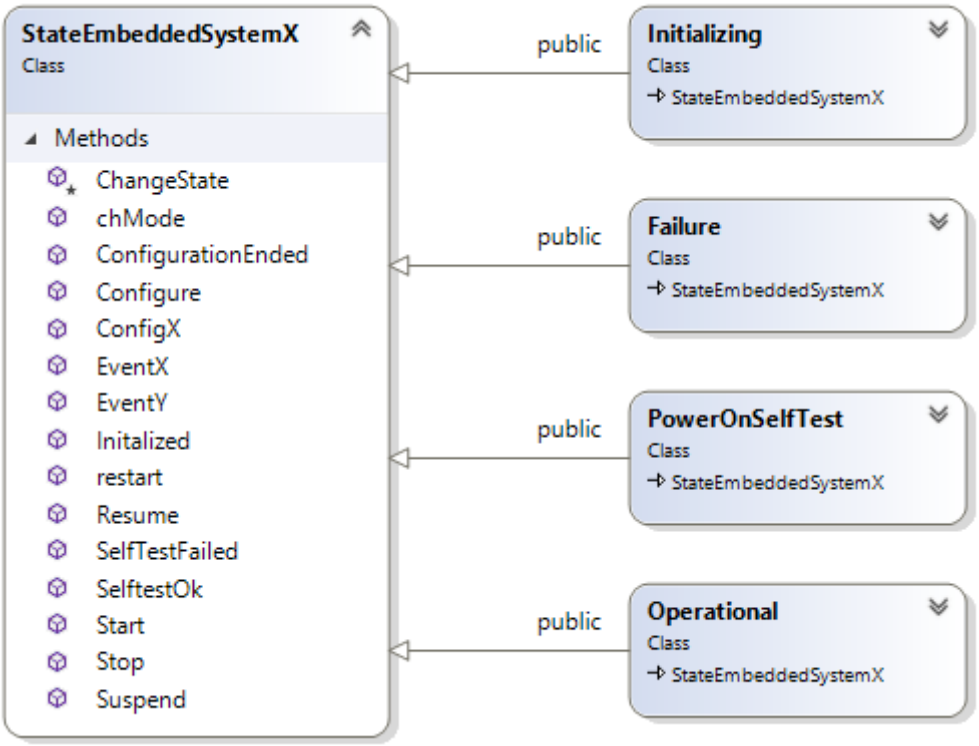

Figure 4 - StateEmbeddedSystemX Class Diagram

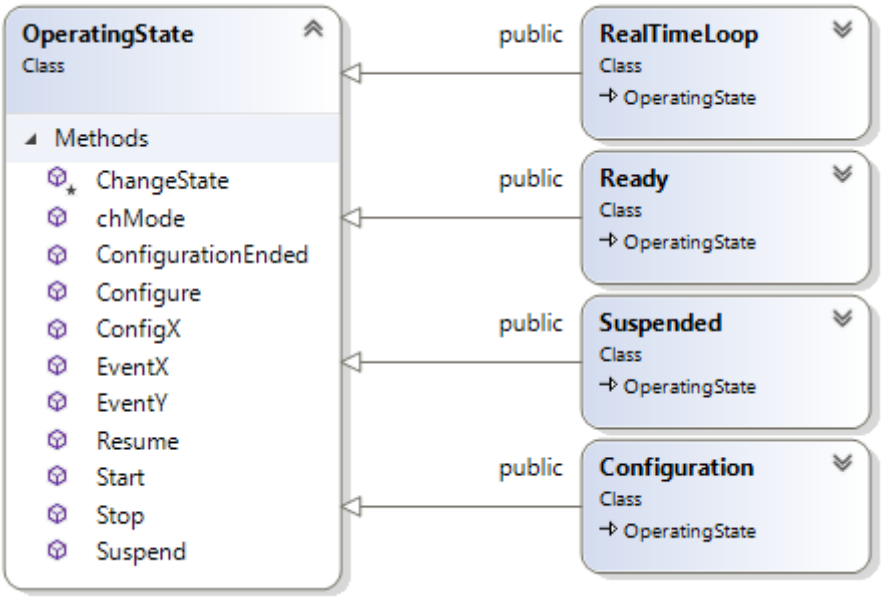

Figure 5 - OperatingState Class Diagram

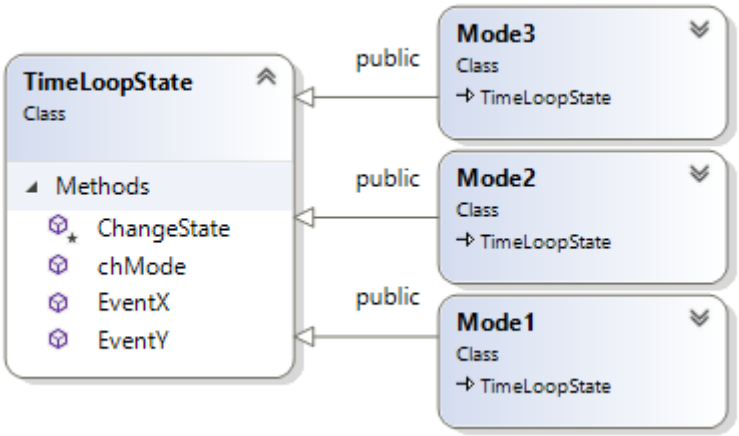

Figure 6 - TimeLoopState Class Diagram

2.3.2 State Diagrams

In this assignment the state diagrams on Figure 7 and Figure 8 where given as part of the assignment to implement.

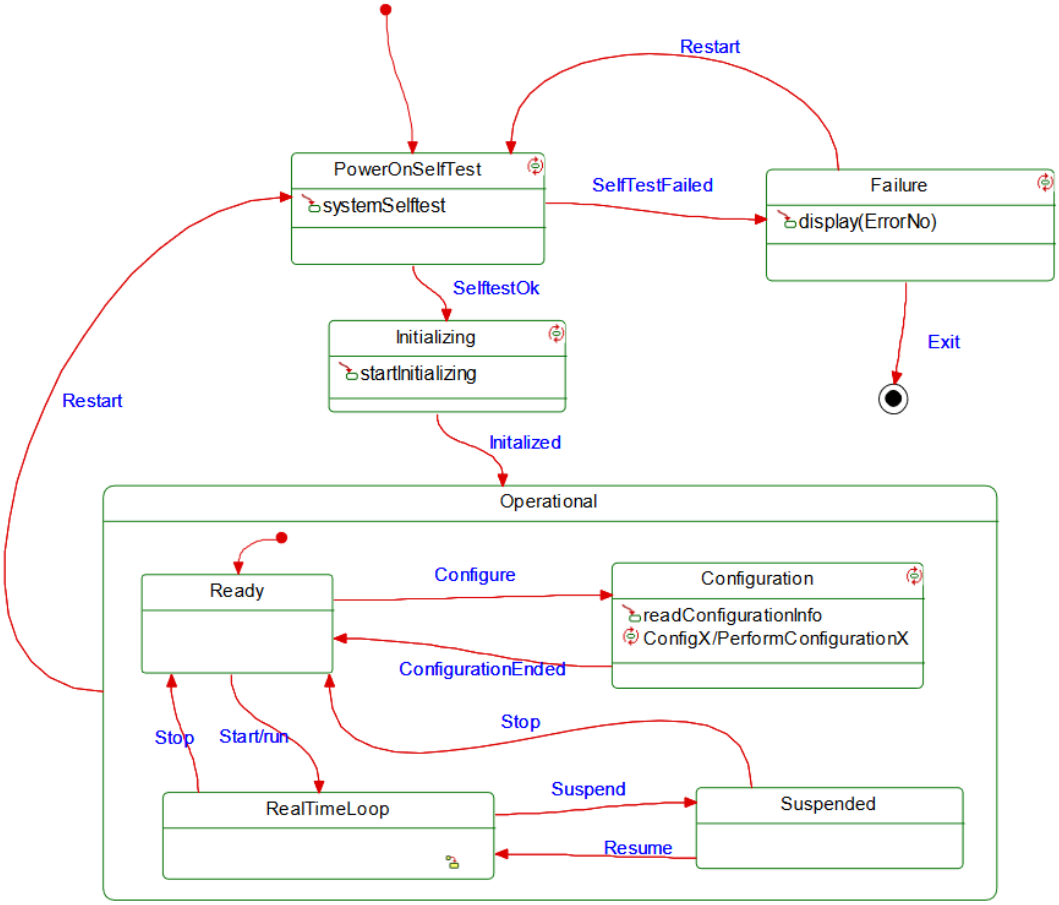

Figure 7 - EmbeddedSystemX State Diagram

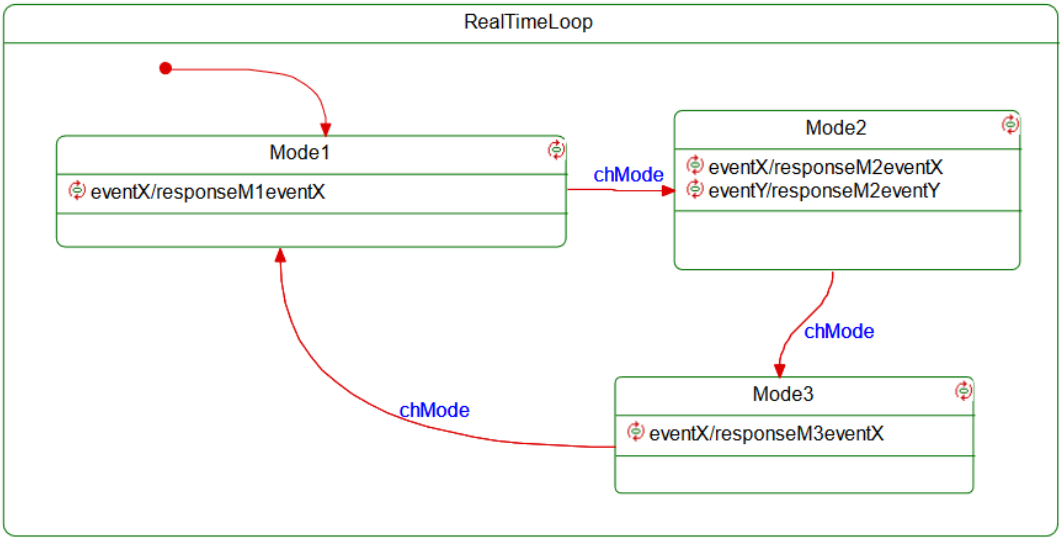

Figure 8 - RealTimeLoop State Diagram

2.4 Implementation View

2.4.1 Implementation details

In every state we have a singleton pattern. There is an example of it in the PowerOnSelfTest.
The header file, has a public Instance that others can get, but it's own internal instance is private.

```
private:
	static PowerOnSelfTest* _instance;
public:
	static PowerOnSelfTest* Instance();
```

The cpp file. Note that the private instance is 0 at first, hence uninitialized. But when it is needed it is created and there after the same instance is returned

```
//Singleton
PowerOnSelfTest* PowerOnSelfTest::_instance = 0;

PowerOnSelfTest* PowerOnSelfTest::Instance()
{
	if (_instance == 0) {
		_instance = new PowerOnSelfTest();
	}
	_instance->systemSelfTest();
	return _instance;
}
//Singleton
```

The state pattern, makes use of a context called EmbeddedSystemX here, it's used to interact with the state machine.

The header file can be seen here, and all the methods that it has to manipulate the state machine.

```
class EmbeddedSystemX {
public:
	EmbeddedSystemX();
	void Restart();
	void SelfTestFailed(int);
	void SelftestOk();
	void Initalized();
	void Configure();
	void ConfigurationEnded();
	void Start();
	void Stop();
	void Suspend();
	void Resume();
	void chMode();
	void ConfigX();
	void EventX();
	void EventY();

	//
private:
	friend class StateEmbeddedSystemX;
	void change_state(StateEmbeddedSystemX*);

	StateEmbeddedSystemX* _state;
};
```

The StateEmbeddedSystemX has all the actions that can be called. Also note it has a method named ChangeState, that takes a pointer to EmbeddedSystemX and StateEmbeddedSystemX.

```
class StateEmbeddedSystemX {
	public:
		virtual void SelfTestFailed(EmbeddedSystemX*, int ErrorNo);
		virtual void restart(EmbeddedSystemX*);
		virtual void SelftestOk(EmbeddedSystemX*);
	    virtual void Initalized(EmbeddedSystemX*);
		virtual void Configure(EmbeddedSystemX*);
		virtual void ConfigurationEnded(EmbeddedSystemX*);
		virtual void Start(EmbeddedSystemX*);
		virtual void Stop(EmbeddedSystemX*);
		virtual void Suspend(EmbeddedSystemX*);
		virtual void Resume(EmbeddedSystemX*);
		virtual void chMode(EmbeddedSystemX*);
		virtual void ConfigX();
		virtual void EventX();
		virtual void EventY();
	protected:
		static void ChangeState(EmbeddedSystemX*, StateEmbeddedSystemX*);
	};
```

An example of a StateChange. Here we can see PowerOnSelfTest change to Initializing.

```
void PowerOnSelfTest::SelftestOk(EmbeddedSystemX* t)
{
	std::cout << "Change to Initializing by input SelftestOk" << std::endl;
	ChangeState(t, Initializing::Instance());
}
```

3 Discussion of results

As seen in the screenshots, we can move around in the state machine and get the wanted output. If we do a command at a point where the command is not usable, we get a respond telling us, we are not able to use that command here, as seen in Cmd 4.3.2 where we call the event command EventY while in the Mode3 state as seen on Figure 9.

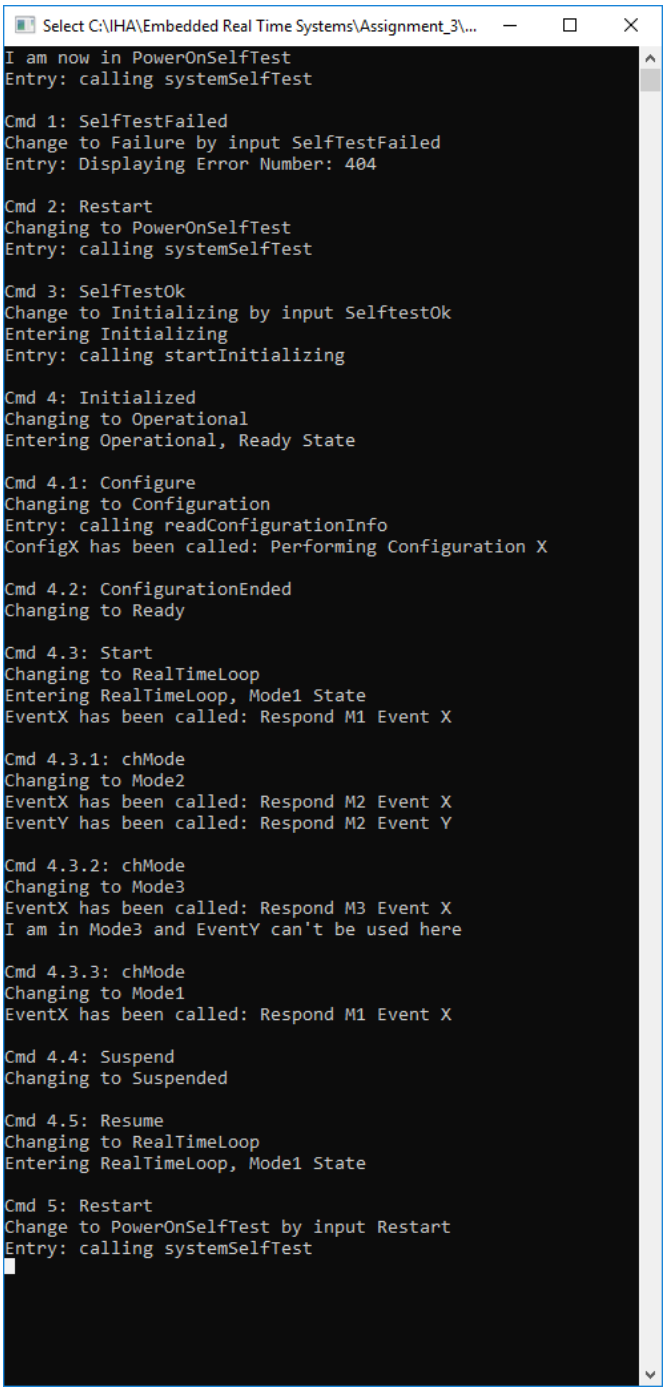

Figure 9 - Output

4 Conclusion

Throughout this assignment a combination of the GoF State and Singleton patterns have been used, to create the functionality of a specified state machine. This has been demonstrated and shows that it is indeed possible, to combine these two software patterns.