Architecture and Design of Embedded Real-Time Systems

Journal on assignment 3

Group 5

Authors: Christian Marius Lillelund and Daniel Ejnar Larsen

Supervisor:
Jalil Boudjadar

Contents

Revision History

Revision	Date/Authors	Description
1.0	28.02.1991/All	Journal created, written and finalized.

1 Introduction

This journal describes the work done in assignment 3 of the course TIAREM. In this assignment, we use the gang of four (GoF) state and singleton patterns combined to implement a finite state machine with several nested states in a hierarchical state structure. GoF state is a behavioural design pattern that define a clean way for an object to change its behavioural after being created without using a switch-case structure that can easily be polluted when new states are added. With GoF state, we can easily add new states without having to recompile the user of our state machine. First we will show an initial state diagram, then our implementation in C++ using Visual Studio and finally the completed class diagram of the solution.

1.1 Intro to requirements for the exercises

R1: The client must be able to invoke an event on the context class (EmbeddedSystemX) and the object should behave accordingly.

R2: All states should act on their own events as well as events of the super state, if any.

R3: All public operations on the context class must be accessible and respond to the client.

1.2 Patterns used in the solution

GoF State and GoF Singleton.

2 Solution

2.1 Introduction to architecture and decisions

The architecture of this solution follows the GoF state pattern design, as seen below. The client uses the context to call actions that change the state of the system. *State* is an abstract base class with pure virtual functions for each action. The derived states must implement all actions from the superclass, in our solution. When a new action is called by the client, Context redirects the call to the current state of Context to handle the action accordingly. It is the responsibility of each state to handle state changes.

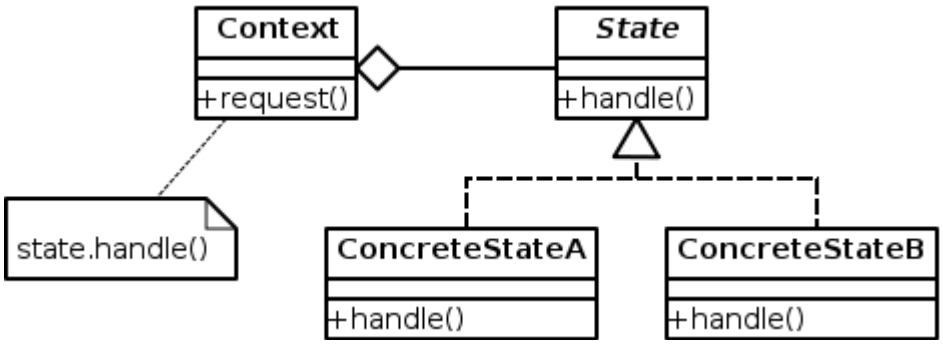

Figure 1. Architecture of the application [GoF State, Wikipedia.org].

2.2 Use Case View

From the assignment description, a user interacts with the EmbeddedSystemX class and calls events on this class.

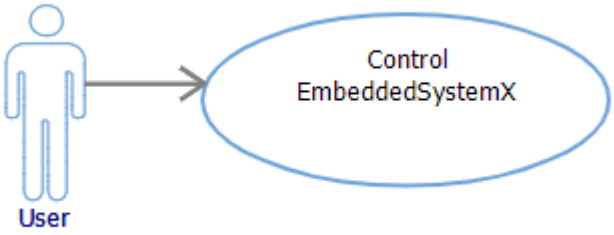

Figure 2. Use case diagram [Assignment3.pdf]

2.3 Logical View

2.3.1 Class diagram(s)

An UML class diagram has been created to show all classes, their dependencies and associations. A full version of this diagram is attached to the journal. Worth noting is how we handle the nested state aspect. The State, Operational and RealTimeLoop classes are all abstract. These cannot be instantiated. This means all subclasses must implement all functions of the superclass, in our solution.

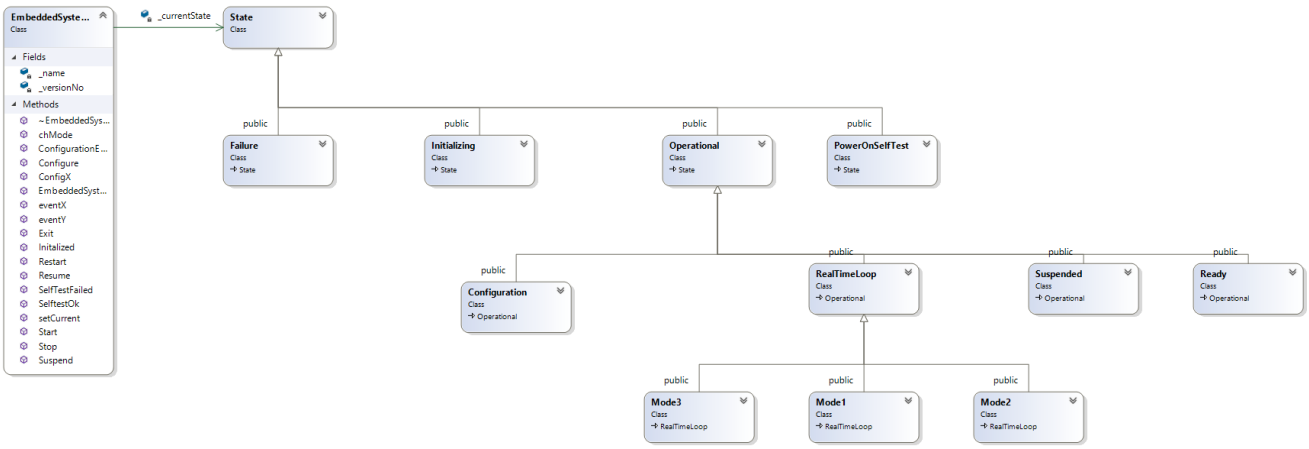

Figure 3. Class diagram of our solution.

2.3.2 Sequence diagram(s)

A sequence diagram has been created to show how actions are delegated as operations on the current state set in the Context. This way the client need not know the implementation details of the states nor how to change to them. Figure 4 exhibits this.

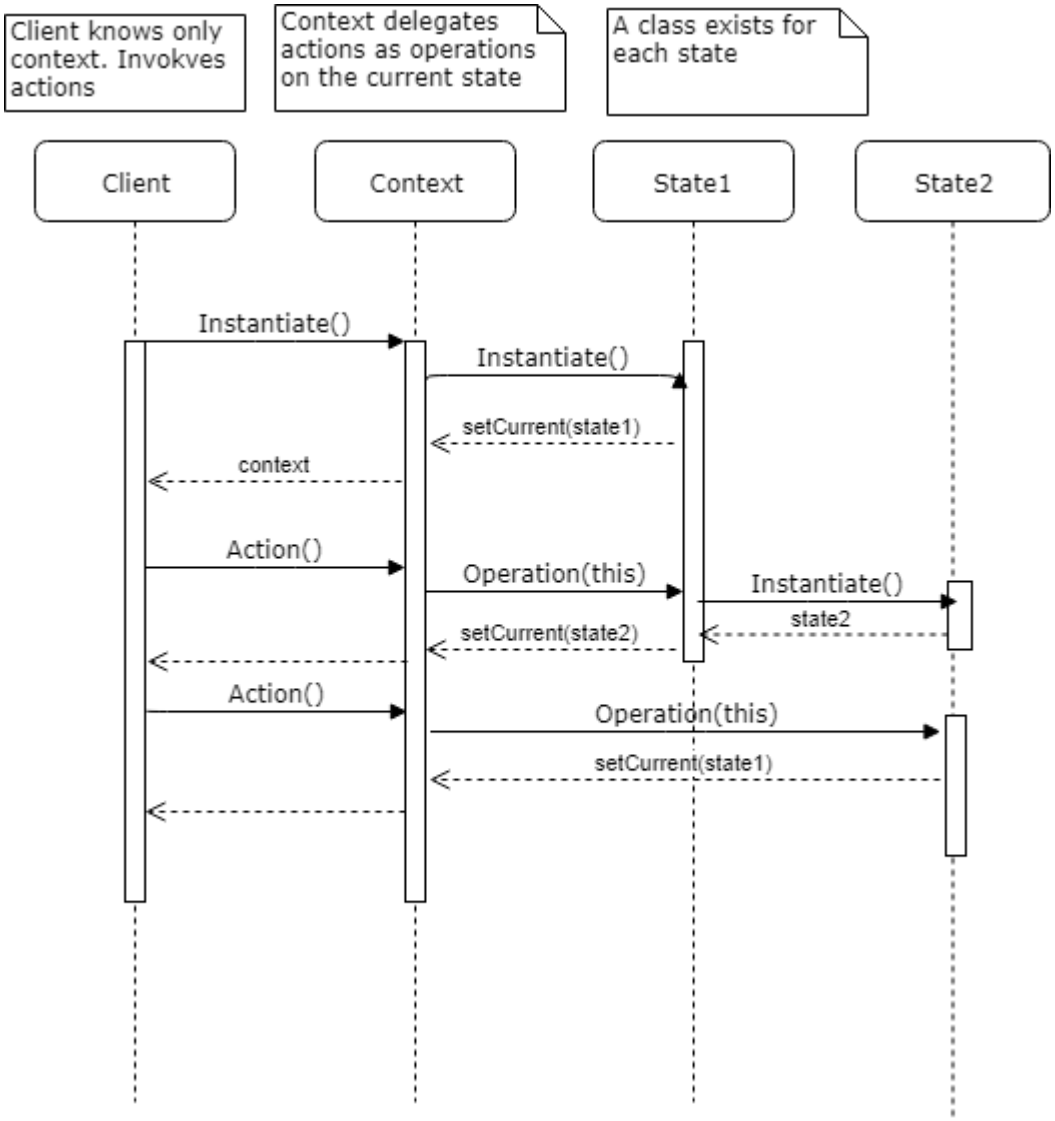

Figure 4. A sequence diagrams that conceptually shows how the state pattern works in our solution.

2.3.3 State Diagram(s)

Figure 5 and 6 show the state diagram of EmbeddedSystemX. They are taken from the description of assignment 3.

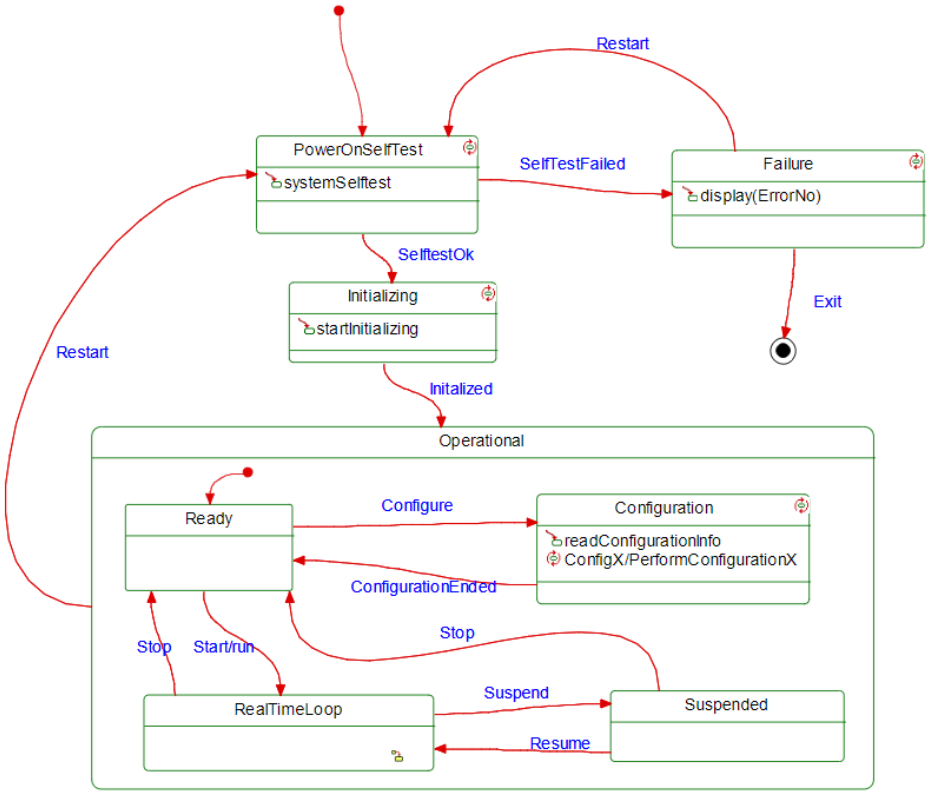

Figure 5. State diagram of the general State and the Operational state [Assignment3.pdf].

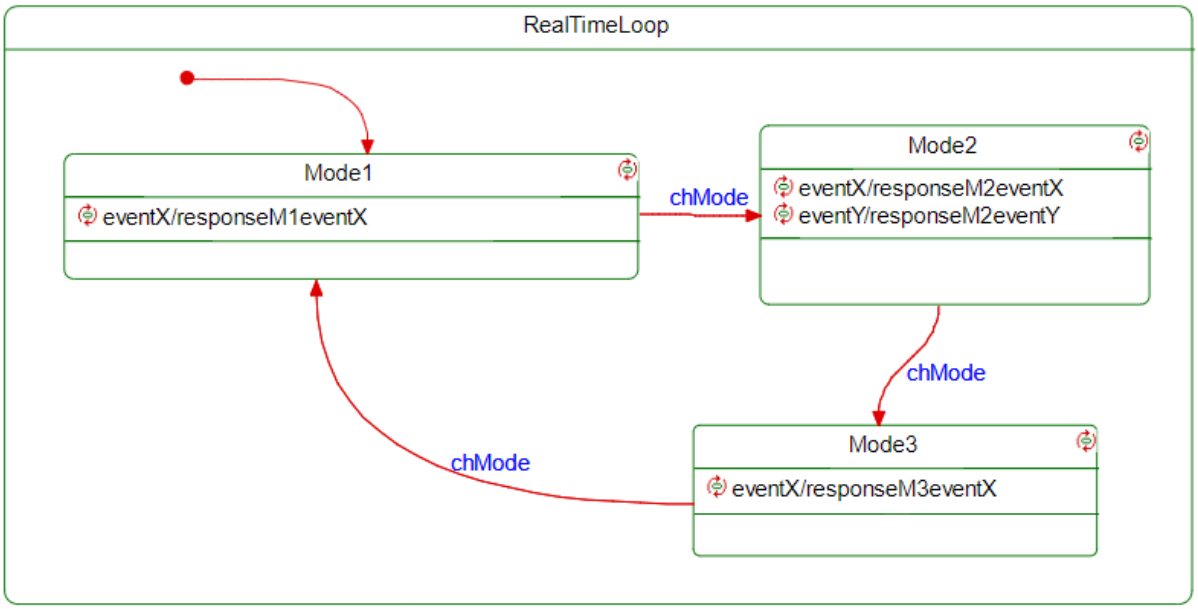

Figure 6. State diagram of the nested state RealTimeLoop [Assignment3.pdf].

2.4 Implementation View

2.4.1 Implementation details

The implementation details of classes of interest are showed here. In figure 7 we see how the context class (EmbeddedSystemX) saves the current state in a variable. This a pointer to state. All available actions are defined here.

```
1.  #pragma once
2.  #include "PowerOnSelfTest.h"
3.  class State;
4.
5.  class EmbeddedSystemX
6.  {
7.  public:
8.      EmbeddedSystemX();
9.      ~EmbeddedSystemX();
10.     void SelftestOk(EmbeddedSystemX* context);
11.     void Initalized(EmbeddedSystemX* context);
12.     void Restart(EmbeddedSystemX* context);
13.     void Configure(EmbeddedSystemX* context);
14.     void ConfigurationEnded(EmbeddedSystemX* context);
15.     void Exit(EmbeddedSystemX* context);
16.     void Stop(EmbeddedSystemX* context);
17.     void Start(EmbeddedSystemX* context);
18.     void Suspend(EmbeddedSystemX* context);
19.     void SelfTestFailed(EmbeddedSystemX* context, int errorNo);
20.     void ConfigX(EmbeddedSystemX* context);
21.     void chMode(EmbeddedSystemX* context);
22.     void eventX(EmbeddedSystemX* context);
23.     void eventY(EmbeddedSystemX* context);
24.     void Resume(EmbeddedSystemX* context);
25.     void setCurrent(State *s);
26. private:
27.     int _versionNo;
28.     char* _name;
29.     // Pointer which holds the current state
30.     State* _currentState = PowerOnSelfTest::GetInstance();
31. };
```

Figure 7. Class EmbeddedSystemX.h.

Figure 8. Class EmbeddedSystem.cpp. Trivial details omitted. Notice the current state handle actions.

```
1.  #include "EmbeddedSystemX.h"
2.  #include "State.h"
3.
4.  EmbeddedSystemX::EmbeddedSystemX()
5.  {}
6.
7.  EmbeddedSystemX::~EmbeddedSystemX()
8.  {}
9.
10. void EmbeddedSystemX::SelftestOk(EmbeddedSystemX* context) {
11.     _currentState->SelftestOk(this); }
```

```
1.  class EmbeddedSystemX;
2.
3.  class State
4.  {
5.  public:
6.      State();
7.      virtual ~State();
8.      virtual void SelftestOk(EmbeddedSystemX* context) = 0;
9.      virtual void Initialized(EmbeddedSystemX* context) = 0;
10.     virtual void Restart(EmbeddedSystemX* context) = 0;
11.     virtual void Configure(EmbeddedSystemX* context) = 0;
12.     virtual void ConfigurationEnded(EmbeddedSystemX* context) = 0;
13.     virtual void Exit(EmbeddedSystemX* context) = 0;
14.     virtual void Stop(EmbeddedSystemX* context) = 0;
15.     virtual void Start(EmbeddedSystemX* context) = 0;
16.     virtual void Suspend(EmbeddedSystemX* context) = 0;
17.     virtual void Resume(EmbeddedSystemX* context) = 0;
18.     virtual void SelfTestFailed(EmbeddedSystemX* context, int errorNo) = 0;
19.     virtual void ConfigX(EmbeddedSystemX* context) = 0;
20.     virtual void chMode(EmbeddedSystemX* context) = 0;
21.     virtual void eventX(EmbeddedSystemX* context) = 0;
22.     virtual void eventY(EmbeddedSystemX* context) = 0;
23. };
```

Figure 9. State.h. Made as an abstract class with pure virtual functions. All defined states must override and implement these.

Figure 10. PowerOnSelfTest.h. Implementes State. Static function GetInstance() returns a singleton instance, as the constructor and destructor are private. Notice how we override the functions from State.

```
1.  #pragma once
2.  #include "State.h"
3.
4.  class PowerOnSelfTest : public State
5.  {
6.  public:
7.      static PowerOnSelfTest* GetInstance();
8.      void SelftestOk(EmbeddedSystemX* context);
9.      void SelfTestFailed(EmbeddedSystemX* context, int errorNo);
10.     static void systemSelftest();
11. private:
12.     static PowerOnSelfTest* _instance;
13.     PowerOnSelfTest();
14.     ~PowerOnSelfTest();
15. public:
16.     void Initialized(EmbeddedSystemX* context) override;
17.     void Restart(EmbeddedSystemX* context) override;
18.     void Configure(EmbeddedSystemX* context) override;
19.     void ConfigurationEnded(EmbeddedSystemX* context) override;
20.     void Exit(EmbeddedSystemX* context) override;
21.     void Stop(EmbeddedSystemX* context) override;
22.     void Start(EmbeddedSystemX* context) override;
23.     void Suspend(EmbeddedSystemX* context) override;
24.     void Resume(EmbeddedSystemX* context) override;
25.     void ConfigX(EmbeddedSystemX* context) override;
26.     void chMode(EmbeddedSystemX* context) override;
27.     void eventX(EmbeddedSystemX* context) override;
28.     void eventY(EmbeddedSystemX* context) override; };
```

```
1.  #include "PowerOnSelfTest.h"
2.  #include <iostream>
3.  #include "Failure.h"
4.  #include "EmbeddedSystemX.h"
5.  #include "Initializing.h"
6.
7.  PowerOnSelfTest* PowerOnSelfTest::_instance = 0;
8.
9.  PowerOnSelfTest::PowerOnSelfTest()
10. {
11. }
12.
13.
14. PowerOnSelfTest::~PowerOnSelfTest()
15. {
16. }
17.
18. PowerOnSelfTest* PowerOnSelfTest::GetInstance()
19. {
20.     return (!_instance) ? _instance = new PowerOnSelfTest : _instance;
21. }
22.
23. void PowerOnSelfTest::SelftestOk(EmbeddedSystemX* context)
24. {
25.     std::cout << "SelfTestOk. Changing to Initializing\n";
26.     Initializing* state = Initializing::GetInstance();
27.     state->Initialized(context);
28.     context->setCurrent(state);
29. }
30.
31. void PowerOnSelfTest::SelfTestFailed(EmbeddedSystemX* context, int errorNo)
32. {
33.     std::cout << "SelfTestFailed. Changing to Failure\n";
34.     Failure* state = Failure::GetInstance();
35.     state->display(errorNo);
36.     context->setCurrent(state);
37. }
38.
39. void PowerOnSelfTest::systemSelftest()
40. {
41.     std::cout << "Performing system self test!\n";
42. }
```

Figure 11. PowerOnSelfTest.cpp. Trivial details omitted. Here we implement the functions that belong to this state. Each take a pointer to the context class, so we can change its state with setCurrent(State). Output messages are printed to the client.

```
#pragma once
#include "State.h"
class Operational :
    public State
{
public:
    Operational();
    virtual ~Operational();
    void SelftestOk(EmbeddedSystemX* context) override = 0;
    void Initialized(EmbeddedSystemX* context) override = 0;
    void Restart(EmbeddedSystemX* context) override = 0;
    void Configure(EmbeddedSystemX* context) override = 0;
    void ConfigurationEnded(EmbeddedSystemX* context) override = 0;
    void Exit(EmbeddedSystemX* context) override = 0;
    void Stop(EmbeddedSystemX* context) override = 0;
    void Start(EmbeddedSystemX* context) override = 0;
    void Suspend(EmbeddedSystemX* context) override = 0;
    void Resume(EmbeddedSystemX* context) override = 0;
    void SelfTestFailed(EmbeddedSystemX* context, int errorNo) override = 0;
    void ConfigX(EmbeddedSystemX* context) override = 0;
    void chMode(EmbeddedSystemX* context) override = 0;
    void eventX(EmbeddedSystemX* context) override = 0;
    void eventY(EmbeddedSystemX* context) override = 0;
};
```

Figure 12. Operational.h. Our second “State” class.

Figure 13. We jump to show how state Mode1 is implemented in the second-nested class. Trivial details omitted.

```
Mode1* Mode1::_instance = 0;

void Mode1::Suspend(EmbeddedSystemX* context)
{
    std::cout << "Suspend. Changing to Suspended\n";
    Suspended* state = Suspended::GetInstance();
    context->setCurrent(state);
}

Mode1* Mode1::GetInstance()
{
    return (!_instance) ? _instance = new Mode1 : _instance;
}

void Mode1::chMode(EmbeddedSystemX* context)
{
  std::cout << "chMode Mode1. Changing to Mode2\n";
  RealTimeLoop* state = Mode2::GetInstance();
  context->setCurrent(state);
}

void Mode1::Restart(EmbeddedSystemX* context)
{
    std::cout << "Restart. Changing to PowerOnSelfTest\n";
    PowerOnSelfTest* state = PowerOnSelfTest::GetInstance();
    context->setCurrent(state);
}

void Mode1::responseM1EventX()
{
    std::cout << "Event X happened. Responding with a :)\n"; }
```

The full code is attached to this journal.

3 Discussion of results

A main.cpp was created to test the application. It shows how the object transits through the different states. We test all actions in this code shown in figure 14. The output is shown in figure 15.

```
1.  #include "EmbeddedSystemX.h"
2.  #include "PowerOnSelfTest.h"
3.  #include <iostream>
4.  #include <cstdio>
5.
6.  int main()
7.  {
8.      std::cout << "Booting ...\n";
9.      EmbeddedSystemX* context = new EmbeddedSystemX();
10.     context->SelfTestFailed(context, 99);
11.     context->Restart(context);
12.     context->SelftestOk(context);
13.     context->Initalized(context);
14.     context->Configure(context);
15.     context->ConfigurationEnded(context);
16.     context->Start(context);
17.     context->Suspend(context);
18.     context->Resume(context);
19.     context->Suspend(context);
20.     context->Stop(context);
21.     context->Start(context);
22.     context->chMode(context);
23.     context->eventX(context);
24.     context->chMode(context);
25.     context->eventX(context);
26.     context->eventY(context);
27.     context->chMode(context);
28.     context->eventX(context);
29.     context->Restart(context);
30.     std::cout << "Ending ...\n";
31.     getchar();
32.     return 0;
33. }
```

Figure 14. Main.cpp.

```
1.  Booting ...
2.  SelfTestFailed. Changing to Failure
3.  ErrorNo: 99
4.  Restart. Changing to PowerOnSelfTest
5.  SelfTestOk. Changing to Initializing
6.  Initialized. Changing to Ready
7.  Initialized. Changing to Ready
8.  Configure. Changing to Configuration
9.  Reading configuration...
10. ConfigurationEnded. Changing to Ready
11. Start. Changing to RealTimeLoop
12. Suspend. Changing to Suspended
13. Resume. Changing to Mode1
14. Suspend. Changing to Suspended
15. Stop. Changing to Ready
16. Start. Changing to RealTimeLoop
17. chMode Mode1. Changing to Mode2
18. chMode Mode2. Changing to Mode3
19. chMode Mode3. Changing to Mode1
20. Restart. Changing to PowerOnSelfTest
21. Ending ...
```

Figure 15. Output from main.cpp when executed.

4 Conclusion

In this journal we have shown how we have designed and implemented assignment 3 using GoF state and GoF singleton pattern. The state pattern provides a sound decoupling of the client and each state, while providing extensibility as it is easy to extend the design with new states, as the context class depend only on the interface of the State class.