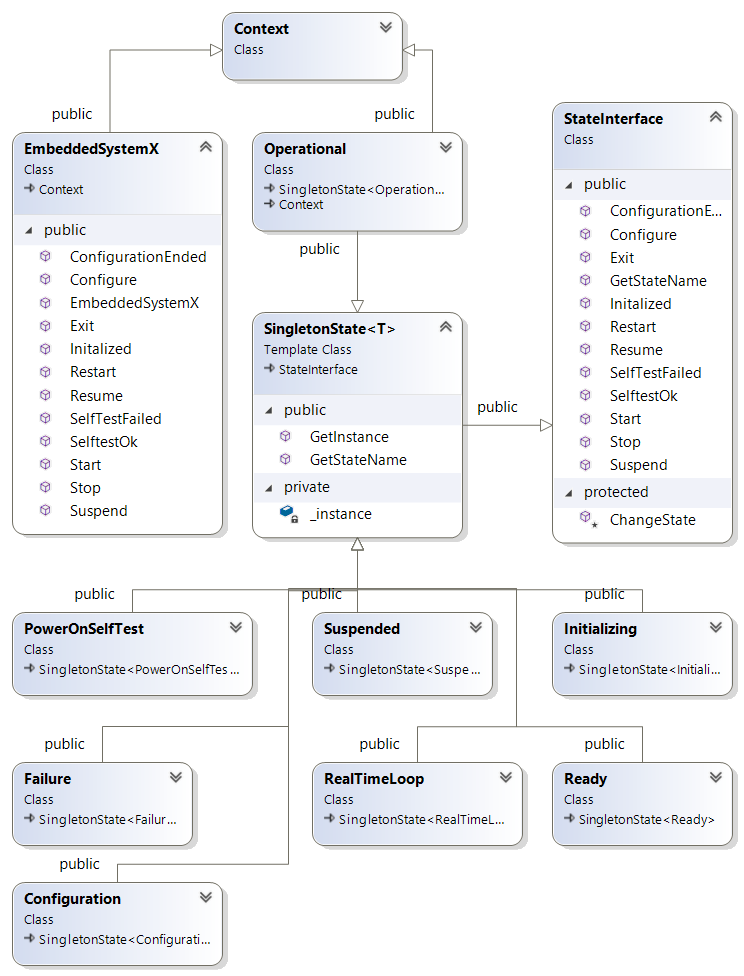
# Solution design for implementing GoF state pattern

Following is the class diagram for a solution implementing GoF state pattern. In this design shared logic of the states is encapsulated in the **SingletonState** and **StateInterface** classes. Moreover, **Operational** class is both a state and a **Context** because it is a super state that can be in different states.



# Implementation in C++

Implementation was done based on the design presented in the previous section. Following are snippets of more important code.

Figure 1 shows state interface. State interface provides a method for each possible event in the system with a default empty implementation. Classes implementing the interface can optionally provide new implementation handling specific events.

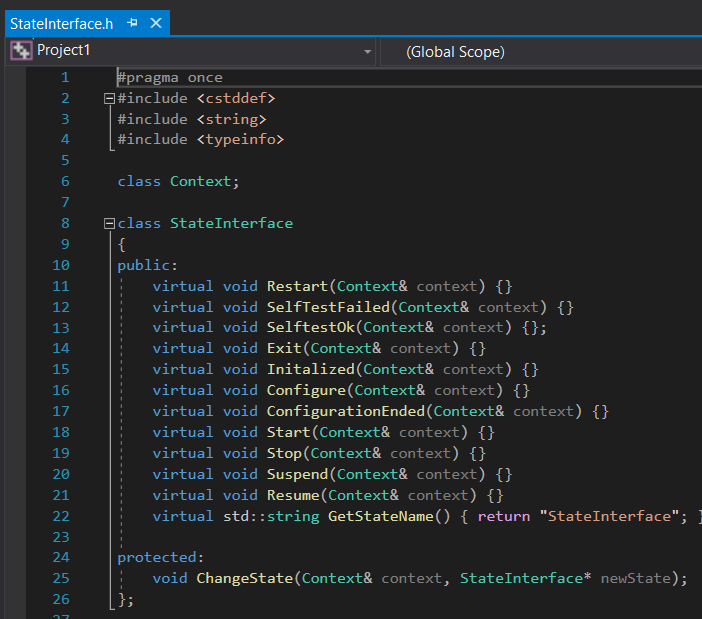


Figure 1 State Interface Definition

There should be only one instance of a state at any point in time. In order to accommodate this requirement **SingletonState** was implemented. As shown in Figure 2, this implementation provides access to the same single instance of a state. Each specific state extends **SingletonState**.

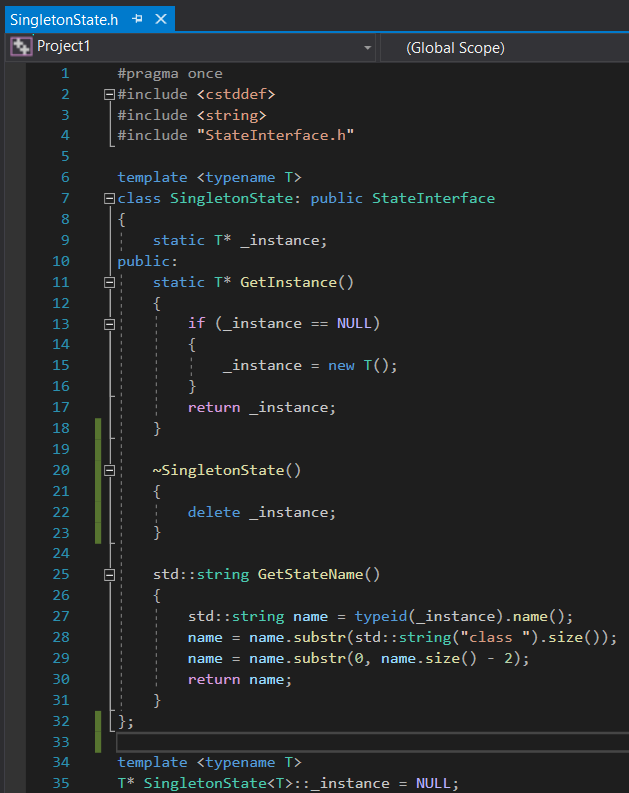


Figure 2 SingletonState template

The required system has two state machines – the global one and the inner one in the **Operational** state. Therefore, a **Context** class has been implemented as shown in Figure 3. It provides the functionality of changing states and providing current state name. Changing state contains a dummy delay to simulate a workload associated with changing states.

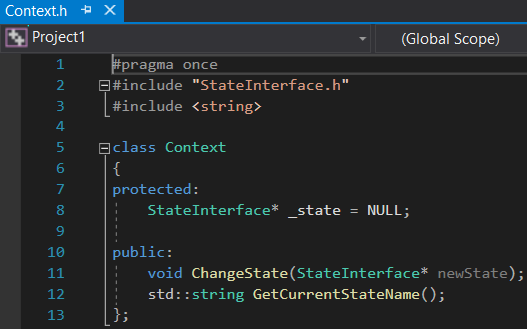


Figure 3 Context Class

The Operational state is a super state (a state and a context at the same time).

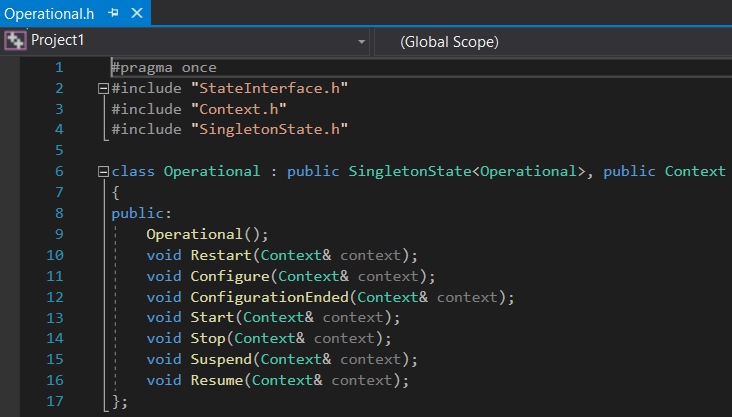


Figure 4 Operational State Declaration

A C++ console application was implemented in order to test the state machine. Part of the implementation of the main loop is shown in the Figure 5. The control loop first accepts instructions from the console invoking specific events. Next, as shown in Figure 6, it measures the available time for execution until the state of **EmbeddedSystemX** changes. Once the state change is observed, a new instruction from the user is requested. The time until the state of the system changes can be considered as available for additional computation tasks to be performed in the main control loop.

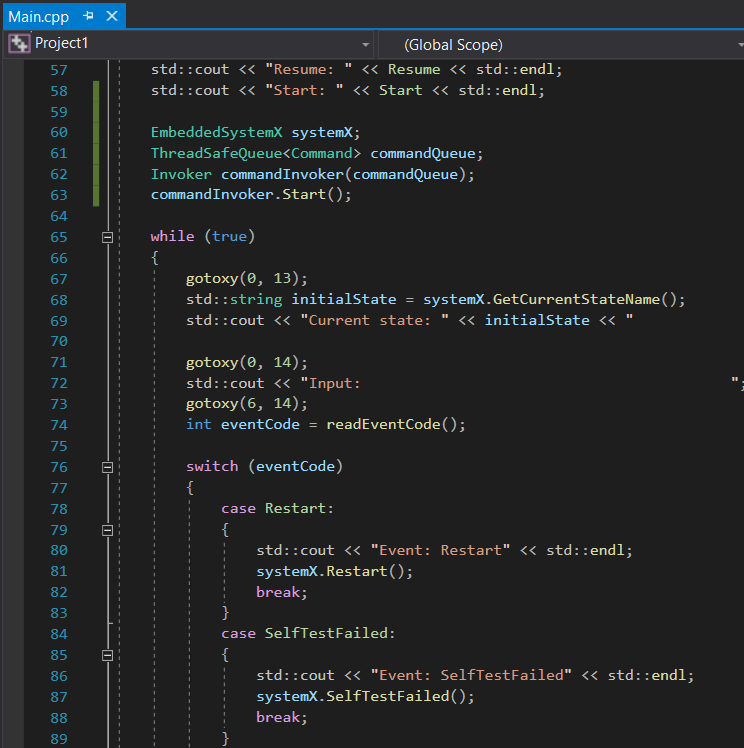


Figure 5 Part 1 of the main loop

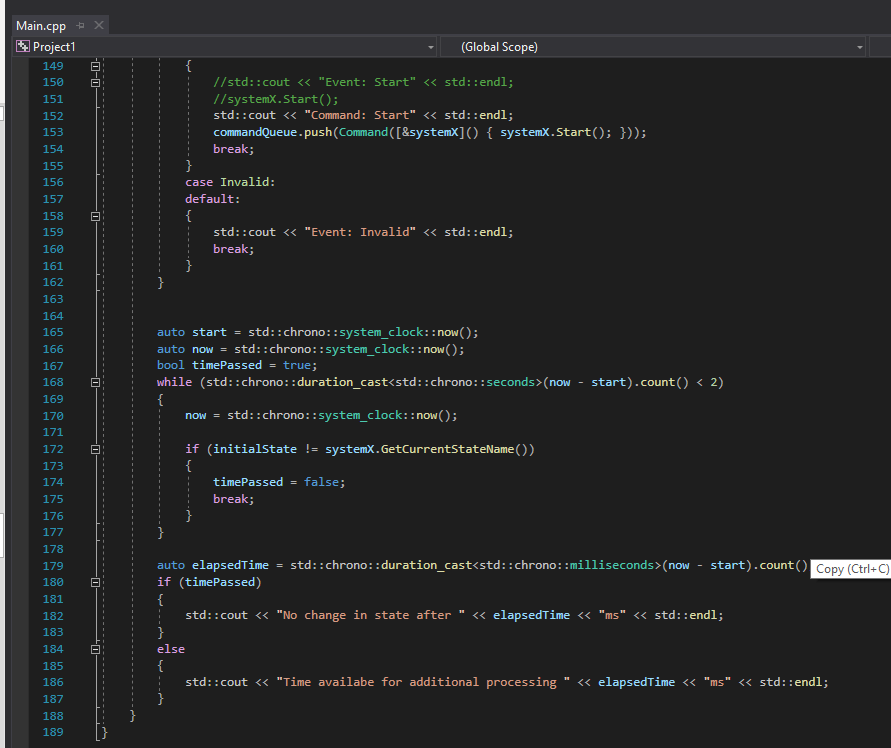


Figure . Part 2 of the main loop

In the initial implementation there is no additional time for computation because the main control loop thread is responsible for executing the change of the state – as soon as it is done the state is already changed and there is no time to perform other tasks. This is illustrated in Figure 7, Figure 8 and Figure 9. In the following section the execution of the main control loop and state change will be decoupled using the command pattern.

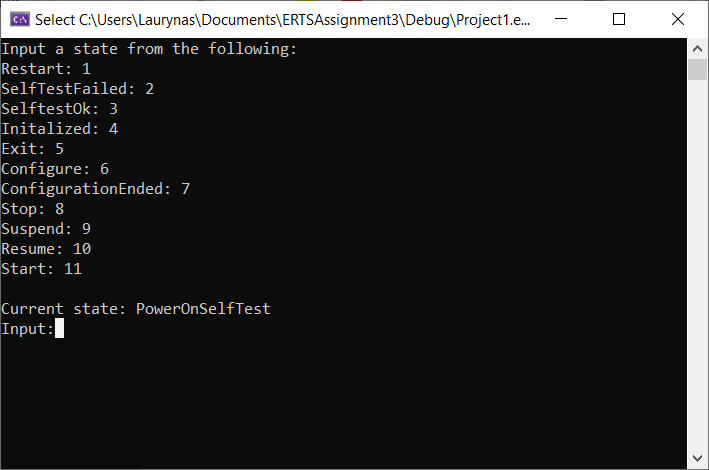


Figure . Initial system state

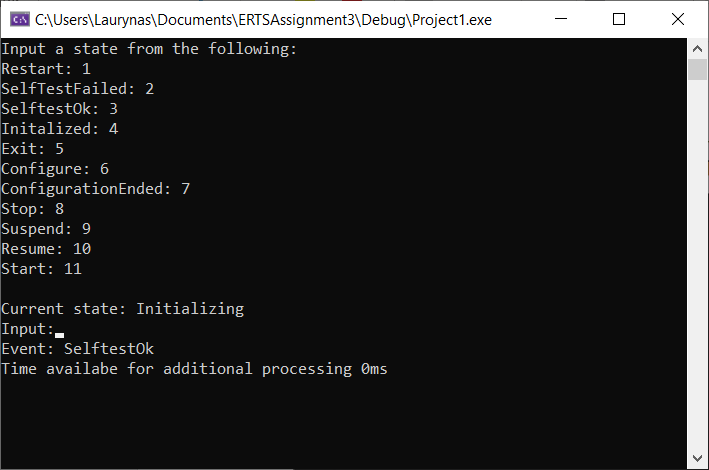


Figure . Initializing system state

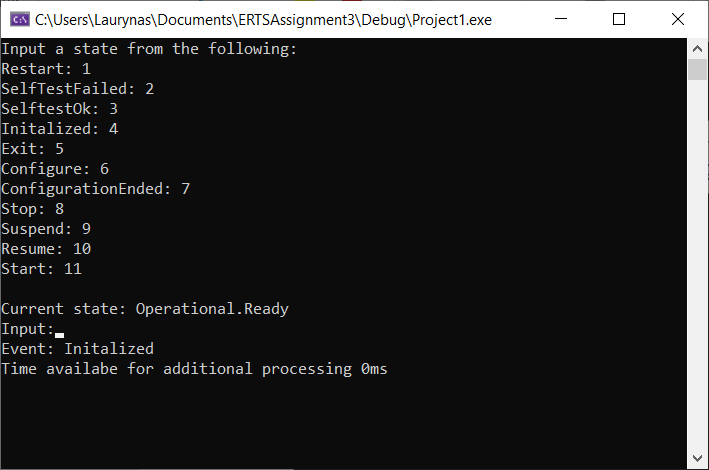


Figure . Operational system state with Ready sub-state

# 1.3 Event processing using command pattern

Design was extended with the classes illustrated in the Figure 10 in order to implement the command pattern. In this iteration, the main initializes a command queue and an **Invoker**. The latter will start its own thread in which it will constantly check the queue in order to execute any available commands. The main control loop will now process some of the user instructions by placing new commands in the queue as shown in Figure 11. Each command holds a function object. This function is called when the respective command is executed.

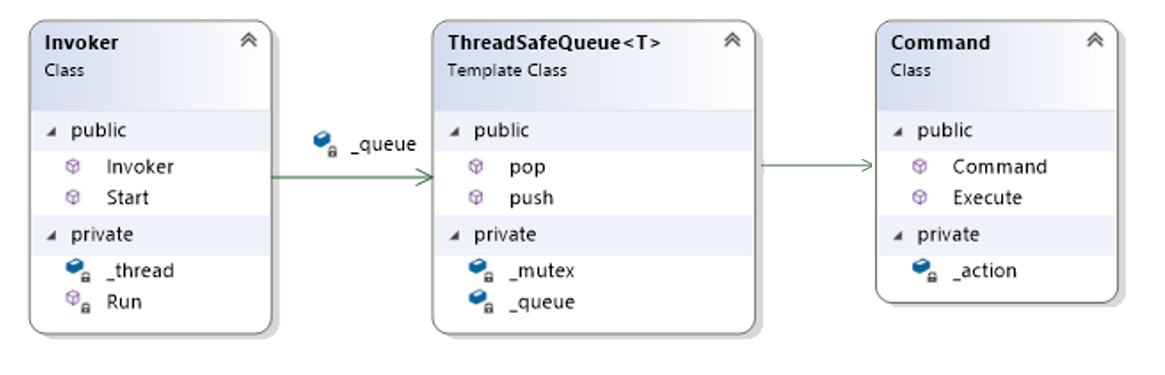


Figure 10. Command pattern classes

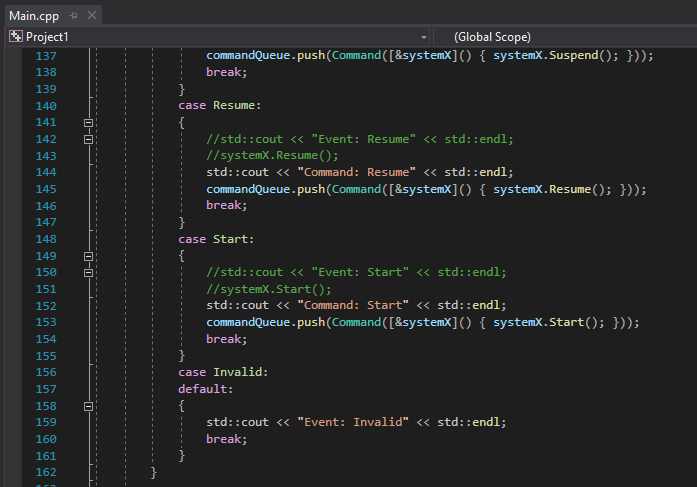


Figure . Command pattern use

Using this design pattern allows to decouple the main control loop process from the process responsible for changing states. In this example, it frees up execution time in the main control loop as illustrated in the Figure 12.

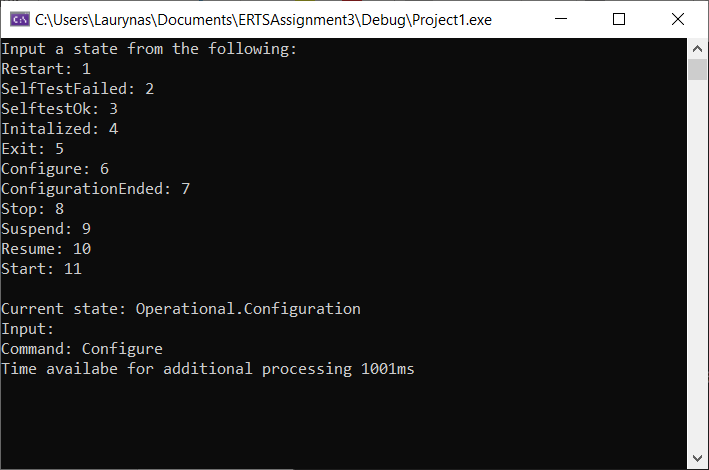


Figure . Operational system state with Configuration sub-state

# Differences between the design iterations

As explained and illustrated in the above sections, introduction of command pattern allows decoupling the execution of the main control loop from the process of changing several states. If state change requires additional processing, this frees up execution time on the main control process. Such approach, however, somewhat increases code complexity.