

Midway delivery

Embedded Real Time Systems

Spring 2010

Michalis Voliotis mvoliotis@gmail.com
José Antonio Esparza jaesparza@gmail.com
Torben Helligsø thel@gmx.com
Amos Hoste a_hoste@hotmail.com

Table of contents

| | |
|---|-----------|
| 1. Exercise 1: Implementation of Generic State Machine for an Embedded System | 3 |
| 2. Exercise 2: Implementation of a Command Pattern in concert with State Pattern | 5 |
| 3. Exercise 3: Factory method and Template method pattern | 6 |
| 4. Exercise 4: Dynamic change of algorithm for the real time loop using the strategy pattern and the real time abstraction | 9 |
| 5. Exercise 5: AND-States, Five-Layer Architecture Pattern and target platform | 11 |

1. Exercise 1: Implementation of Generic State Machine for an Embedded System

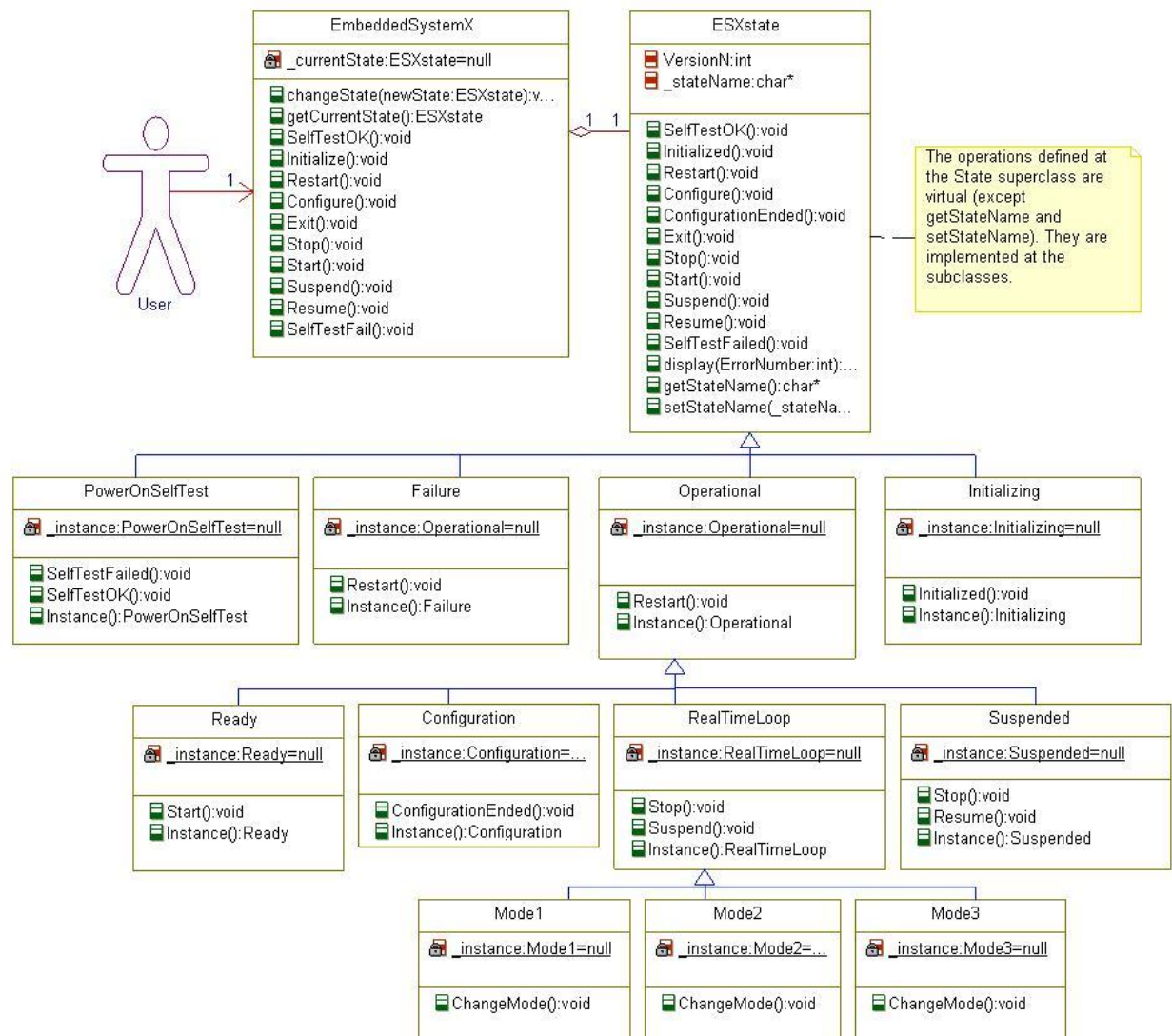


Figure 1: State pattern applied to the given state machine

1. State pattern

The GoF state pattern is used to manage the transitions of a state machine. It is a way to make sure that system is in only one state at a time, and that only transitions that are valid in accordance to the state machine are possible.

The participants of the state pattern are given below

- General state class (ESXState). Parent class that contains virtual (empty) operations for all of the transitions in the system.
- Concrete State subclasses: a class for each state in the system. Here the relevant transitions are overridden.
- Context class (EmbeddedSystemX) with the responsibility of defining an interface through which state transitions can be triggered from the outside. The context class has operations for each state transition in the system that delegates to the state classes.

2. Singleton pattern

By using the singleton pattern it is ensured that only one instance of a given class exists, and only one access point is defined.

- In embedded system X, the concrete state subclasses are implemented as singletons. The classes has a Instance method which is used to access the one instance of the class. The constructor is made private to prevent instantiation by other means than the Instance method.

2. Exercise 2: Implementation of a Command Pattern in concert with State Pattern

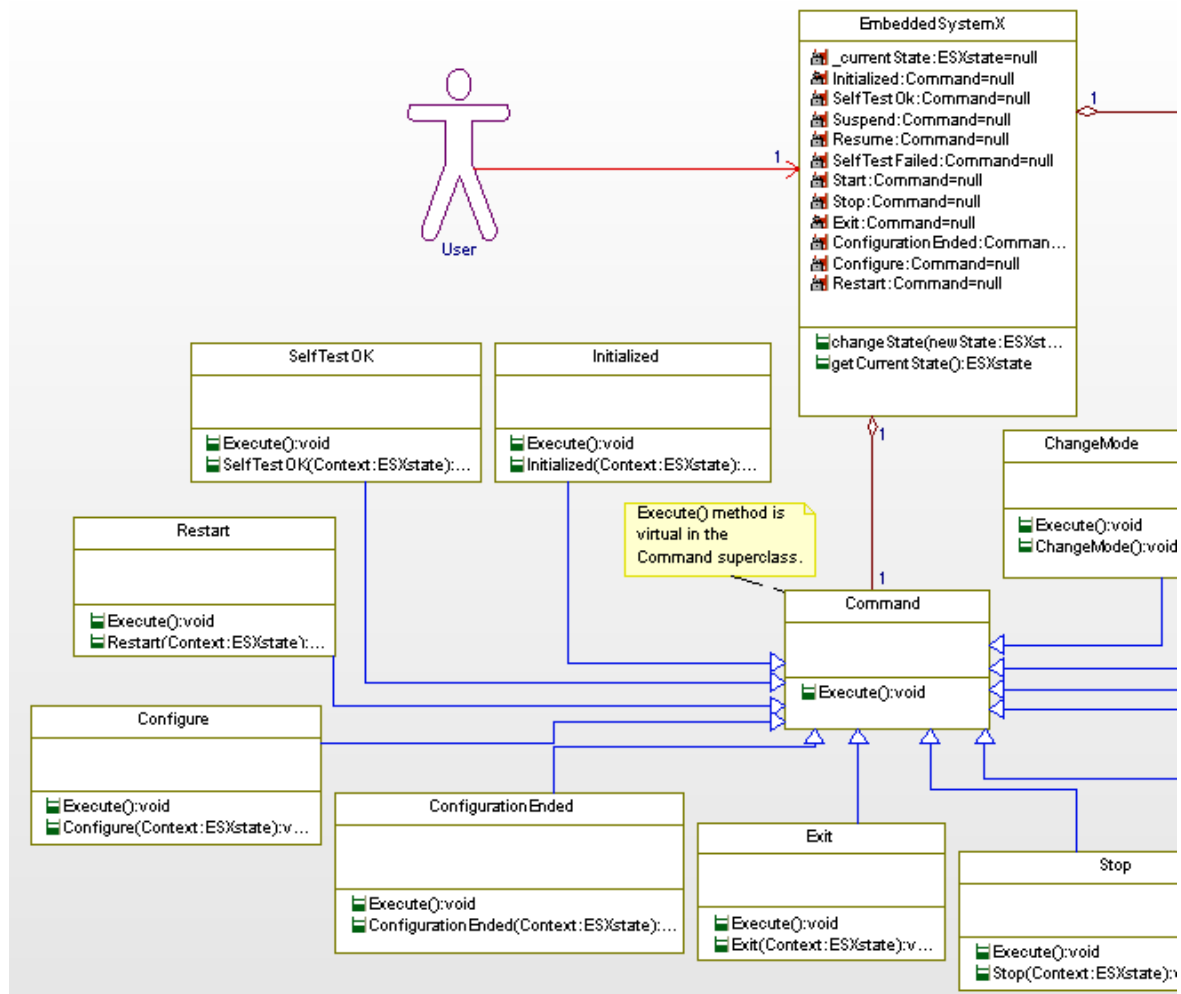


Figure 2: Part of the command pattern application

The command pattern decouples the invocation of an operation from the object that knows how to perform it.

The participants of the command pattern are given below

- Command: base class that defines the virtual Execute operation
- Concrete commands. Their constructors take a pointer to the command's receiver as a parameter. The overridden Execute method uses this pointer to perform the desired command.
- Client (EmbeddedSystemX). The client is the class that instantiates the concrete commands
- Invoker (EmbeddedSystemX). The invoker invokes the commands.
- Receiver (EmbeddedSystemX). The receiver is the class that knows how to carry out a command request.

3. Exercise 3: Factory method and Template method pattern

3.1. The factory method

The factory method defines an interface for creating an object, letting the subclasses decide which class should be instantiated.

The method could be useful in this case because the command creation cannot be anticipated, since they represent external events. A possible alternative could be to create all the command instances at the beginning of the application, but this could lead to the situation in which commands instances are created but never used. By using the factory pattern the command classes can be created dynamically in an easier way.

The factory method has four major components, the product, the concrete products, the creator and the concrete creators, which are mapped to this situation in the following way:

- Product: The command class, superclass that define the structure for the concrete commands.
- Concrete Product: The different commands that can be created. These commands are specific implementations of the command superclass.
- Creator: The creator class, that defines the general structure of the concrete creators. It provides an interface to the context class so the concrete command creation can be triggered from it.
- Concrete Creator: The command creators, responsible for creating the associated concrete commands.

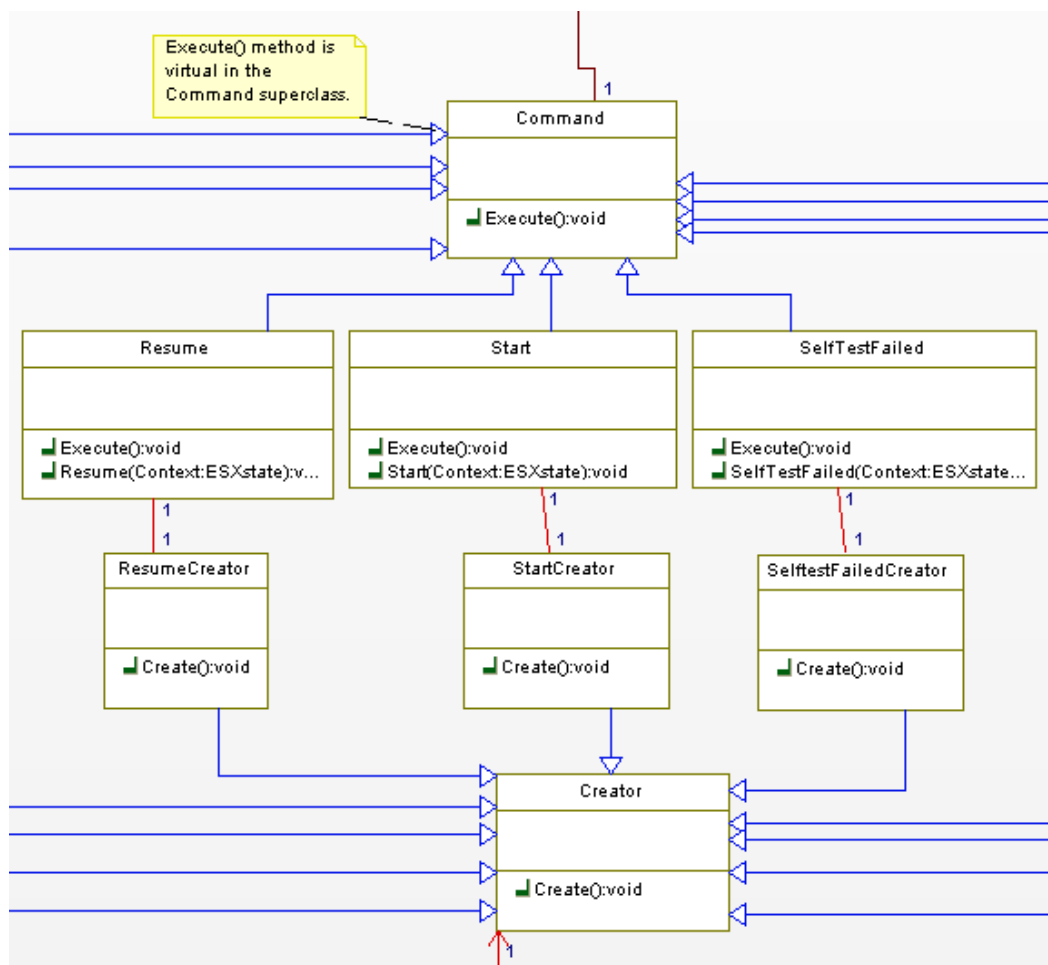


Figure 3: The factory method applied to the command creation

In the previous UML diagram (Figure 3) can be see how the factory method has been applied to manage the creation of the concrete commands Resume, Start and SelfTestFailed.

3.2. The template method

The template method can be used to create different types of classes that execute a set of operations defined by an upper class.

This pattern can be applied to define a concrete sequence of actions in the creation of the different commands that we have specified in the previous sections. For instance, in figure number four is illustrated how a generic class called CommandCreation is extended by the classes SelfTestOKCreation and StartCreation. Both classes are responsible for creating the concrete commands SelfTestOK and Start, but they are able to display as well some information by using the method DisplayInformation.

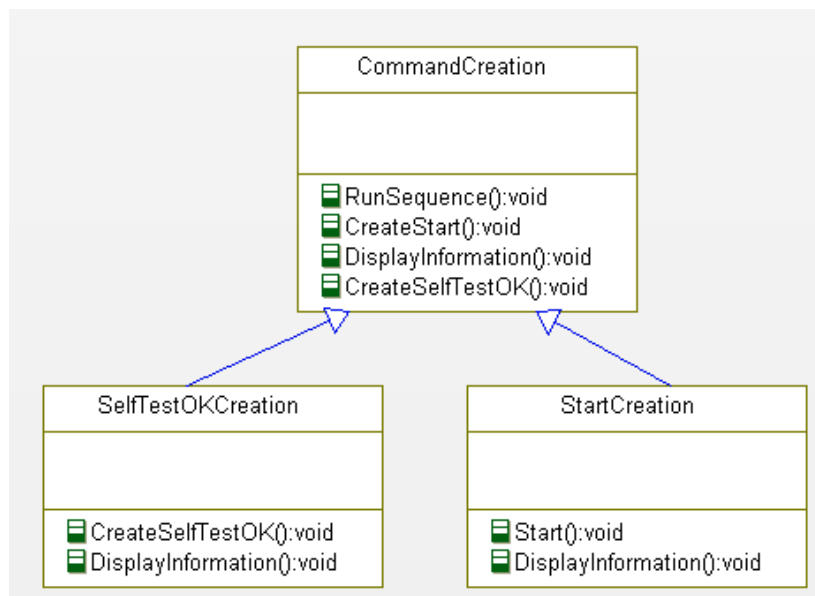


Figure 4: A possible application of the template method to the command creation

As it can be seen, by applying the template method it is easy to define new classes implementing specific behaviour based upon the already defined methods in the superclass.

Taking advantage on this feature, another possible application example could be the creation of test scenarios. This is illustrated in the class diagram showed at figure 5. A super class called SystemInput is defining a set of commands in the method section. These methods are defined as virtual methods in the superclass and delegating the creation responsibility to the subclasses.

As it can be seen in the class diagram, three test scenarios have been created. In Test1, the commands sent to the state machine are the required events to go through the PowerOnSelfTest, Initializing, Operational and Exit. In the Test2 scenario, the state machine will go through the states PowerOnSelfTest, Failure and finally Exit. In the third test, events to enter the RealTimeLoop and Suspended states will be sent, after that the operation will stop.

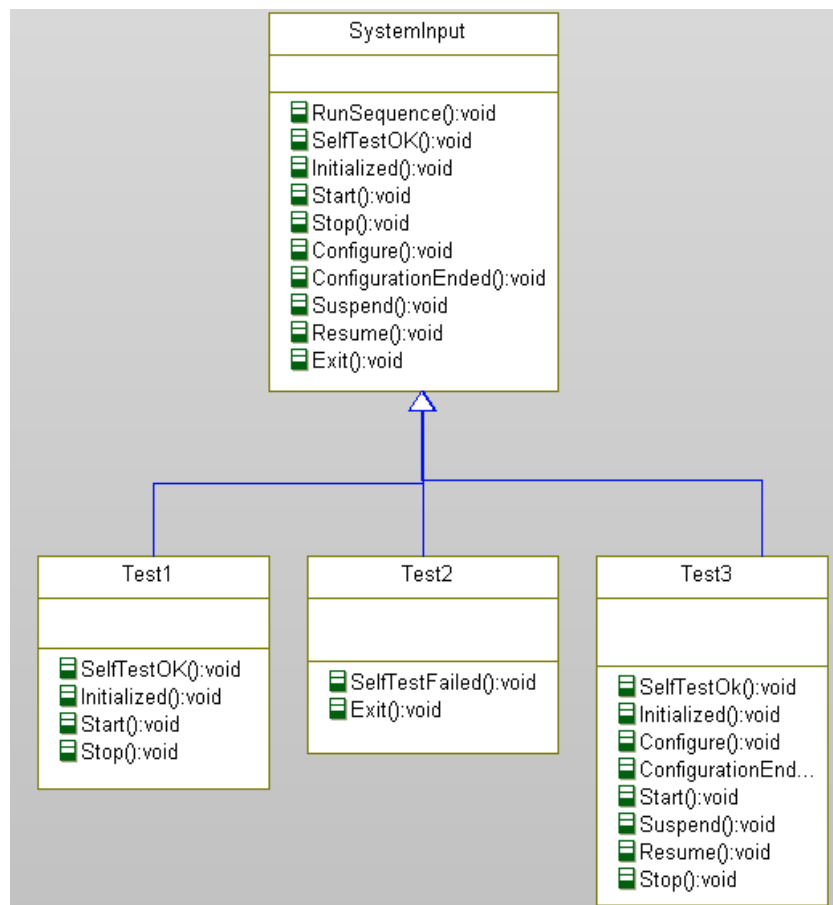


Figure 5: A possible application of the template method on the creation of test scenarios

4. Exercise 4: Dynamic change of algorithm for the real time loop using the strategy pattern and the real time abstraction

4.1. The strategy pattern

In the strategy pattern a family of algorithms are defined and can be easily exchanged during the execution time depending on the conditions. The different algorithms can be exchanged by interacting with the context class.

The elements that are present in the Strategy pattern are the Context, the Strategy, and the concrete strategies, that are applied in the our system's real time loop in the following manner:

- Context: RealTimeLoop class. It acts as an intermediate point, and interface that can be used in order to select the different strategies.
- Strategy: RealTimeStrategy. The general strategy, this superclass defines the methods that can be used by the concrete strategies as virtual methods.
- Concrete strategies: Mode1Strategy, Mode2Strategy and Mode3Strategy. This concrete strategies represent the different modes that can be used in the real time loop. All of them presents a similar structure, and implements the methods defined in the RealTimeStrategy or part of them.

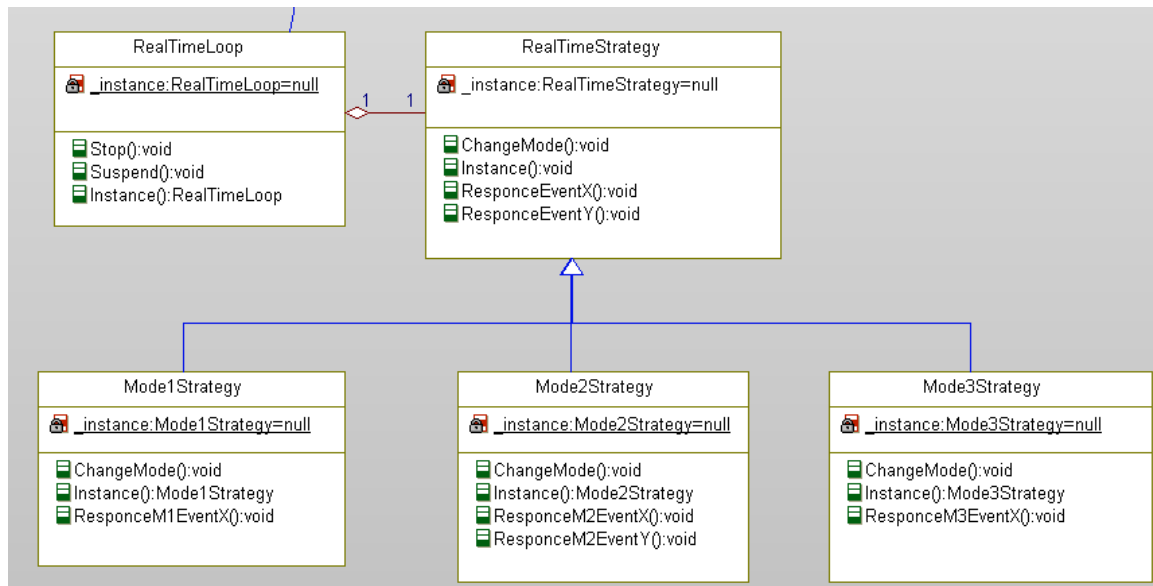


Figure 6: The strategy pattern applied to the real time loop

4.2. The two layer architecture

The two layer architecture allows to separate the logic for the discrete processing from the continuous processing. That means that the system architecture can be organized in a manner in which the part that should react to the incoming events can be running in a thread and the part of the system that should run continuously can run in a separate one.

This architecture applied to our system result on running the real time loop in a single thread and the rest of the state machine in another. In this sense, the event driven part can be reacting to the environment while the continuous processing is preformed separately. The communication between this two parts can be carried out by using an interface, that should be implemented in both packages.

The interfaces in this particular case will be an abstraction from the operative system, allowing us to deal with the concurrency by using semaphores and with the data exchange by using mail boxes, shared memory segments etc. These mechanism are known as IPC (Inter Process Communication).

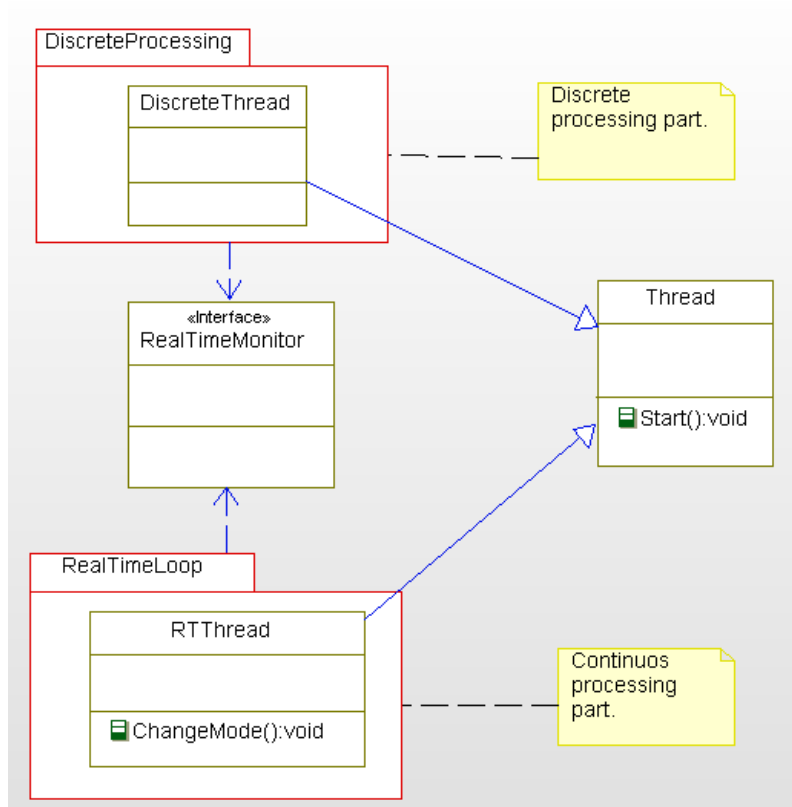


Figure 7: Two part architecture style.

5. Exercise 5: AND-States, Five-Layer Architecture Pattern and target platform

5.1. AND states

In this step we introduce an AND state in the realtime loop so that the user has the ability to choose if the real or a simulated realtime loop should be used. Regardless of run mode the 3 substates of the realtime state still exist. We implement the AND state by replacing the three subclasses of RealTimeStrategy with two sets of subclasses – one for the realtime run mode and one for simulate run mode.

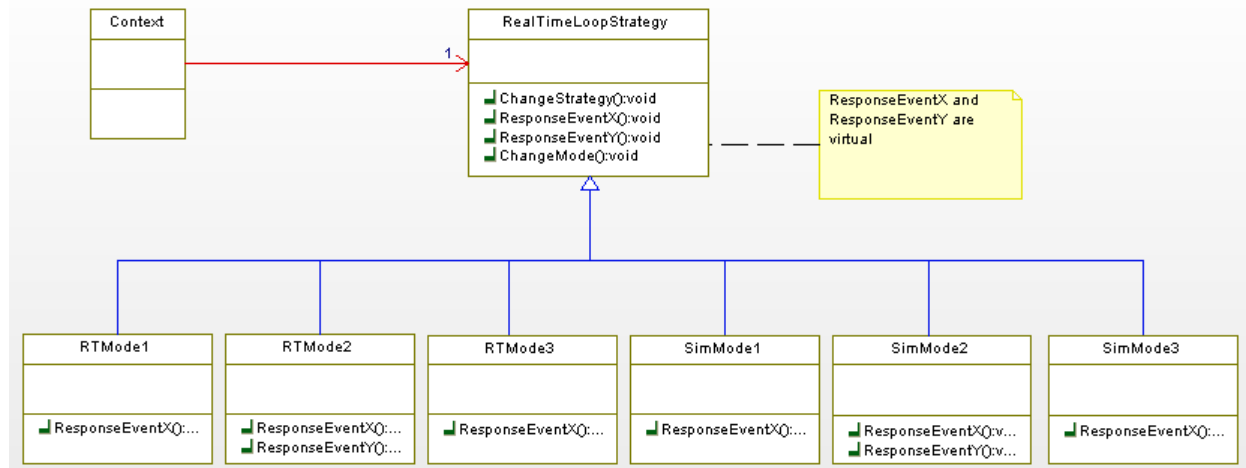


Figure 8: AND states

This change also introduces changes to the state diagram of the realtime loop, for instance, transitions from RTMode1 now goes to SimMode1, RTMode2 and SimMode2

5.2. The five layer architecture

This architecture is useful to create a structure with different communicated layers, that will contain the classes depending on its responsibility.

This architecture can be applied in our system by grouping the classes responsible for the user communication in a layer called User Interface and doing the same with the resting classes, keeping them together at the Application layer.

Since we still have not introduced any classes interacting with hardware, communication or operative system interaction, these layers will remain empty at the moment.

For more details the included rational diagram should be consulted.

6. References

- [1] Overgaard Hansen, Fin. Notes on the course Embedded Real Time Systems. Spring 2010.
- [2] Gamma, Erich, Helm, Richard et al. "Design patterns: Elements of Reusable Object-Oriented Software". Addison-Wesley Longman 1998.
- [3] Powel Douglas, Bruce. "Real Time Design Patterns. Robust Scalable Architecture for Real-Time Systems". Pearson Education. 2003.