QActor Extensions:
Additional mechanisms to define the model

Luca Marchegiani

May 30, 2022

Contents

Abstract

In this paper we discuss new mechanisms to define an executable *actor model* using the custom infrastructure and modeling language given by the course of *Ingegneria dei Sistemi Software* of the Bologna university.

The main mechanism that we present is based on `Java Annotations`.

1 Introduction

QActor (or ***QAK***) is a modeling language to define meta models using *actors*. The language is well explained in the official web page [link].

The *QActor* does not only provide a custom `DSL`, but also an entire infrastructure that let the developer to design and build basic *actor* systems. Actually, there are two ways to write systems based on actor modeling using `QAK`:

1. *the custom DSL* made using `Eclipse` and `Xtext`;
2. *manually*, writing the description of a system in a `.pl` file and extending some class of the infrastructure like `ActorBasic` or `ActorBasicFsm`.

Unfortunately, these two mechanisms have some problems:

1. the DSL is strongly dependent from `Eclipse` because it has not an own IDE. This can be a problem because the `QAK` is written in `Kotlin` and `Eclipse` is not fully compatible with this language.
2. writing all things *manually* should be very *uncomfortable*.

So, in this report we analyze some alternatives to define the actor model according with the `QAK` infrastructure. We will not go into the details of the `QActor` implementations because they are fully described into the official web page but we emphasize that the main way to create an actor in this system is:

> **Creates a class** that extends `ActorBasic` or `ActorBasicFsm` with the body of the actor and **add a proper description** of it into a file `.pl`. This file must also contains all the information about the context of the actor and the other actors that can be also remote.[1]

Indeed, the `DSL` does not do magic: it does nothing more than auto-generate code that follows the mechanism that we have just described. As we have already said, we want to extend this in order to have a new mechanism based on `Java Annotation`.

But before doing this, we also want to find a way to strongly **separate the actor system description from its runtime implementation**. In fact, if we consider a single actor, actually both of its description and its runtime context are enclosed into the `ActorBasic` class (or `ActorBasicFsm`) and its subclass that contains the body.

Then we want to provide a way to define *passive entities* that only contain the description of the actor system you want to define. As these entities will only be used to describe the system, we will call them ***transient***.

[1]See the documentation for more details.

2 The Transient model

2.1 Package `it.unibo.kaktor.model`

As we have already said, the *transient* model consists in a series of entities that represent the description of the actor system that the application designer want to define. In a first approximation these entities will then be wrapped into `ActorBasic` instances that will be used as regular.

We remember that the main entities defined into the `QA-System` are:

- **actors** active components that are able to receive messages and handle them in a proper way;
- **body** as the main behavior of each actor;
- **messages** as the communication unit for the actors;
- **states** (for finite state machine actors);
- **transitions** (for finite state machine actors);
- **contexts**;

Slightly abusing the `UML` notation, the figure 1 shows the diagram of the transient model package. We have provided these classes:

- `TransientActorBasic`:
  The class represents the description of an actor (e.g. its name, its scope, its channel size and other options), particularly **its body** that will be described in a few lines; this is the central class of the model that will be *wrapped* into the `ActorBasic` class used from the system for the runtime implementation.
- `TransientActorBasicFsm`:
  This class represents the description of an actor that is a *finite state machine*. So it extends the `TransientActorBasic` class but has a *finite state body* instead of the normal body of its super-class.
- `TransientState`:
  This class represents the description of a state of a finite state machine actor. It has a *name* and a *state body* that will be called when a *FSM* actor enters the state.
- `TransientTransition`:
  This class represents the description of a transition from one state to another into a *FSM* actor. So it has an *edge name* used to identify uniquely the transition, a *target state* and a *type*. Based on the type it also has additional fields used be the specific type.
- `TransientContext`:
  This class represents the description of a context that contains a collection of actors. In addition to this, a context also have some field that describes some of its characteristics (like its name, its address and so on).
- `TransientSystem`:
  This class represents the description of the entire system that will be executed. As a `TransientContext` it also has a `ImmutableParameterMap` that is an object defined

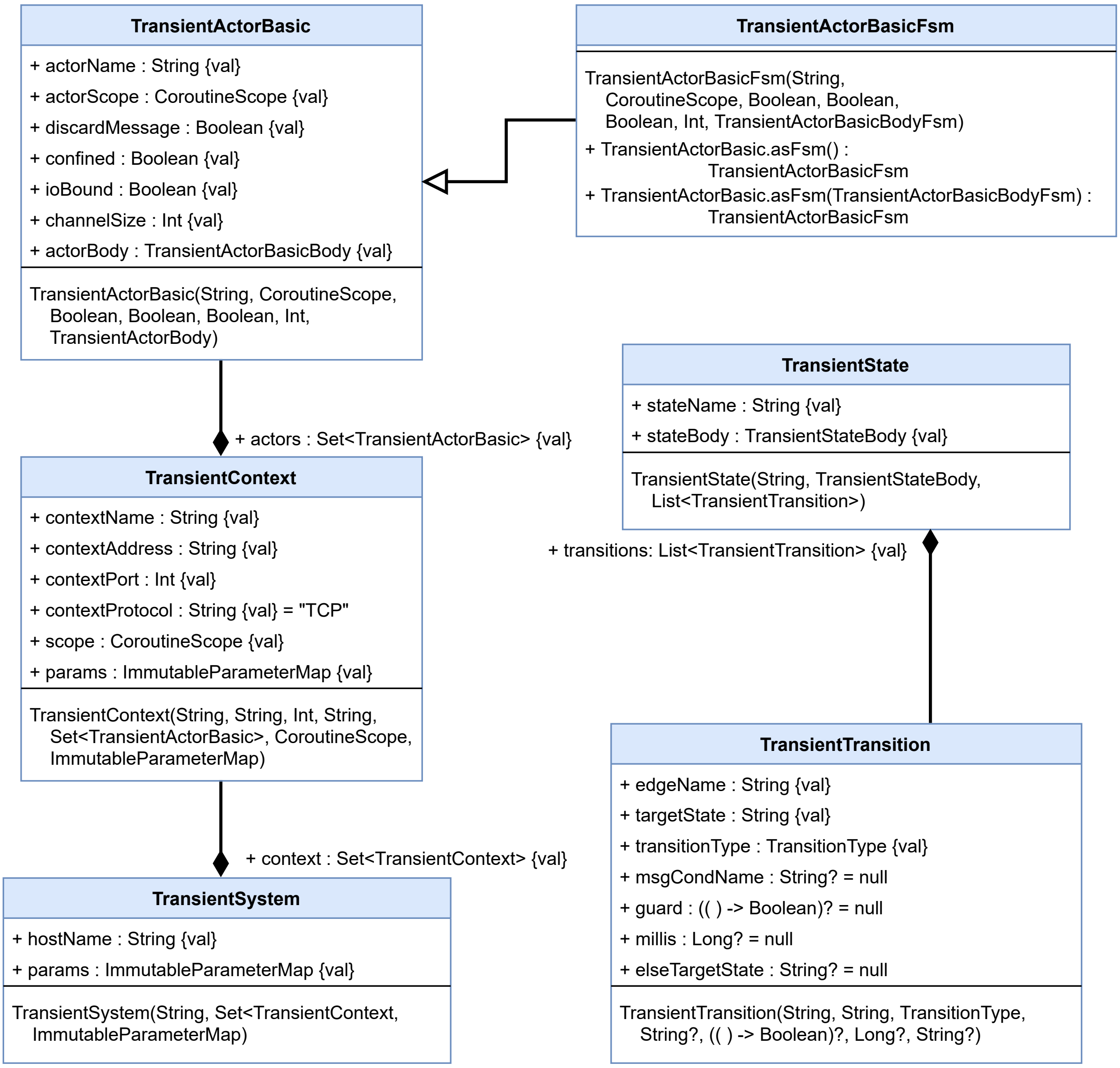

Figure 1: UML diagram for the Transient Model

in another package that maintains a series of key-object pairs for re-usability. **This is the end class of the *transient model* that will be passed to method that load and build the entire system.**

2.2 Package `it.unibo.kaktor.model.actorbody`

We have shown that the `TransientActorBasic` class maintains an object that represents the actor body. Same for the `TransientActorBasicFsm` class in which the difference is that the body has the behavior of a finite state machine.

The figure 2 summarizes the package that contains the class for the actor body. The main classes of this package are:

- `TransientActorBasicBody`:
  This classes that implements this symbolic interface are actor basic bodies.
- `TransientLambdaActorBasicBody`:

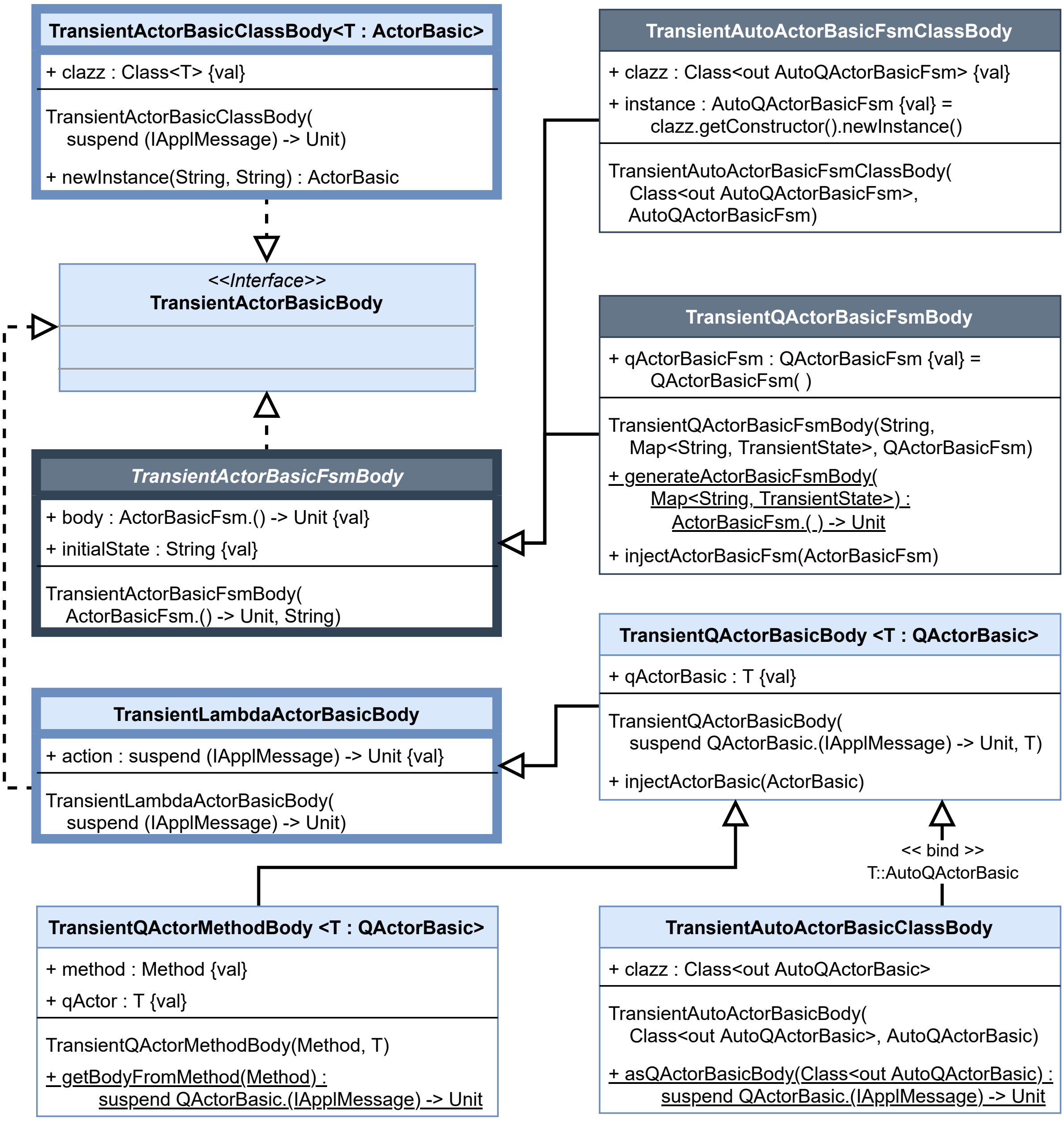

Figure 2: UML diagram for the Transient Model of the actor body

This is the main class for a body of an `ActorBasic` instance. It maintains a *lambda function* that describes the actions the actor will have to perform when receives a message.

- `TransientActorBasicFsmBody`:
  This is the main class for a body of an `ActorBasicFsm` instance. It maintains a *lambda function with closure* that contains the actions to be create an instance of the `ActorBasicFsm` class[2] and also the name of the initial state.

The other classes of this package are useful in order to easily create instances of these main superclasses and will be clarified soon.

The figure 3 shows the classes describing the body of a state. It contains:

[2]See the official `QAK` documentations for details about the creation of a finite state machine actor.

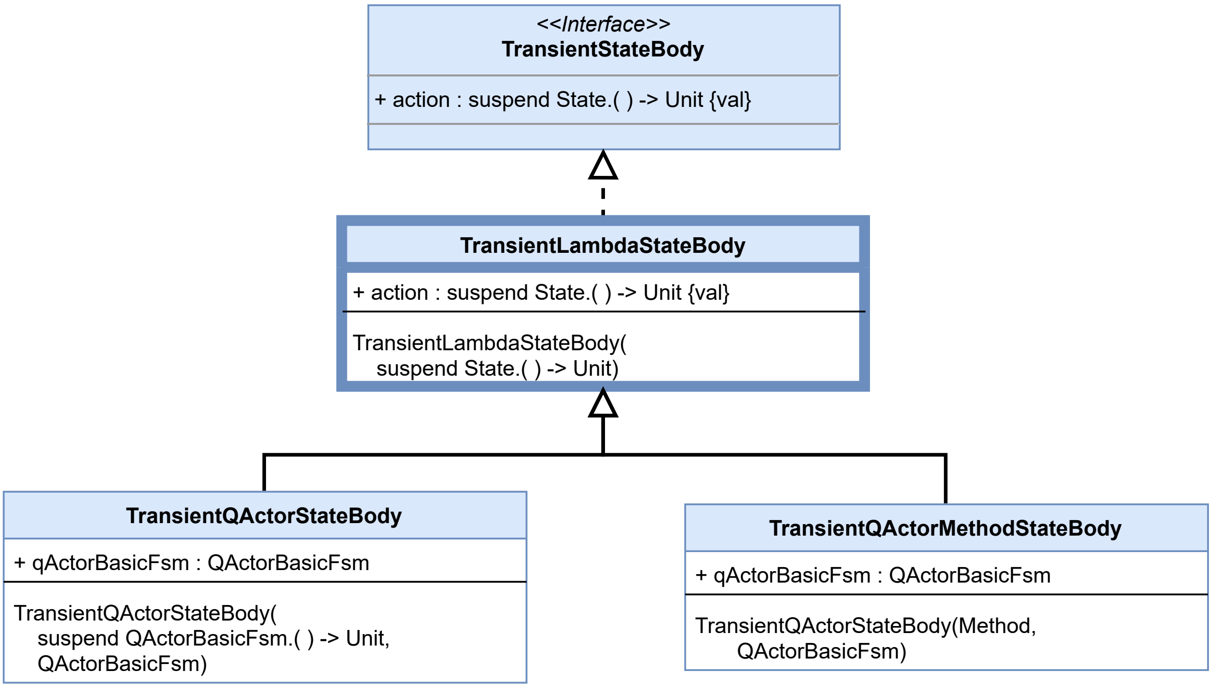

Figure 3: UML diagram for the Transient Model of the actor body

- `TransientStateBody`:
  This classes that implements this symbolic interface are finite state machines bodies.
- `TransientLambdaStateBody`:
  This is the main class for a body of a finite state machine. It maintains a *lambda function* that describes the actions the actor will have to perform when enters the state owning this body.

The other two subclasses will be used to easily create instance of lambda state body and will be explained in the next sections.

3 The builder mechanism

3.1 Overview of the builder package `it.unibo.kaktor.builders`

In addition to the transient model, we want to provide a sort of *standard mechanism* that must be reliable and reusable to create the transient entities.

So, we decided to use the builder pattern.

The figure 4 shows the main builder components for the transient system. They are:

- `ActorBasicBuilder`:
  This component let to create a `TransientActorBasic` using the builder pattern. It is easy possible to set the actor body by calling the `addActorBoby(TransientActorBody)` method. There are others additional methods that can be used to quickly add more complex body that the normal lambda body (the classes not already explained of the transient body model).
- `ActorBasiFsmcBuilder`:

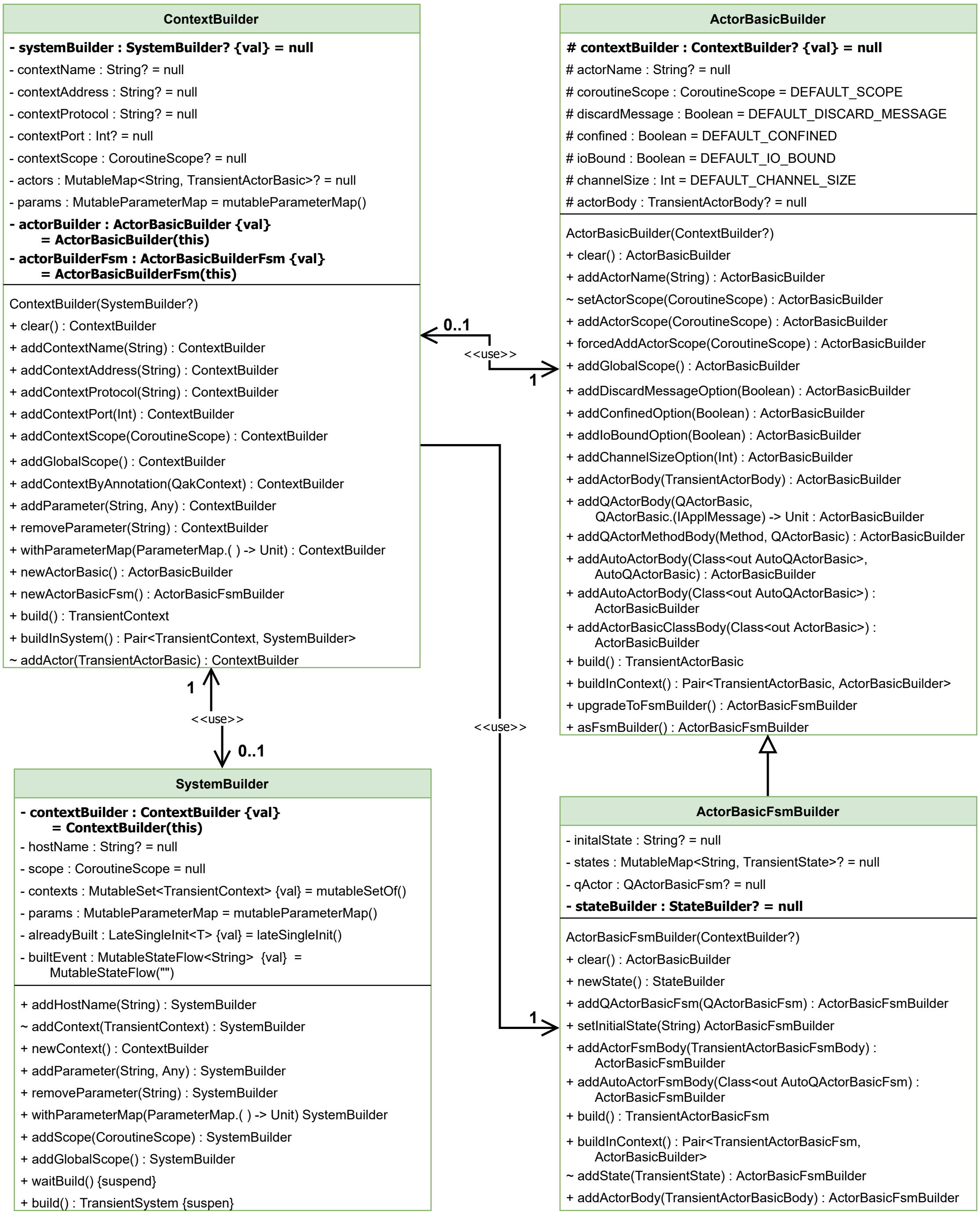

Figure 4: UML diagram for the actor, context and system builders

This component let to create a `TransientActorBasicFsm` using the builder pattern. This class extends the `ActorBasicBuilder` then add others additional method to its in order to create a finite state machine actor. It is easy possible to add a state to the actor that is building by calling `newState()` method that returns a `StateBuilder` for the new state.

- `ContextBuilder`:
  This component let to create a `TransientContext`. It is easy possible to add an actor to the context that is building by calling `newActorBasic()` or `newActorBasicFsm()` methods that return a builder for the new actor.

- SystemBuilder:
  This component let to create a `TransientSystem`. It is easy possible to add a context to the system that is building by calling `newContext()` method that returns a `ContextBuilder`. When the creation of the transient system is completed so it is needed to invoke the `buil()` method that returns the `TransientSystem`. Notice that **a `SystemBuilder` cannot be reused then once the system is created it not possible to clear the builder and start again the creation**. In addition to this, after the build method invocation, there are no possibilities to add other contexts or to build again.

In addition to all things we have just explained, the builders can throw a `BuildException` if something goes wrong or if the developer has not passed all the needed information to it before invoking `build()`, for example if the developer invoke it without calling the `addActorName(String)` before.

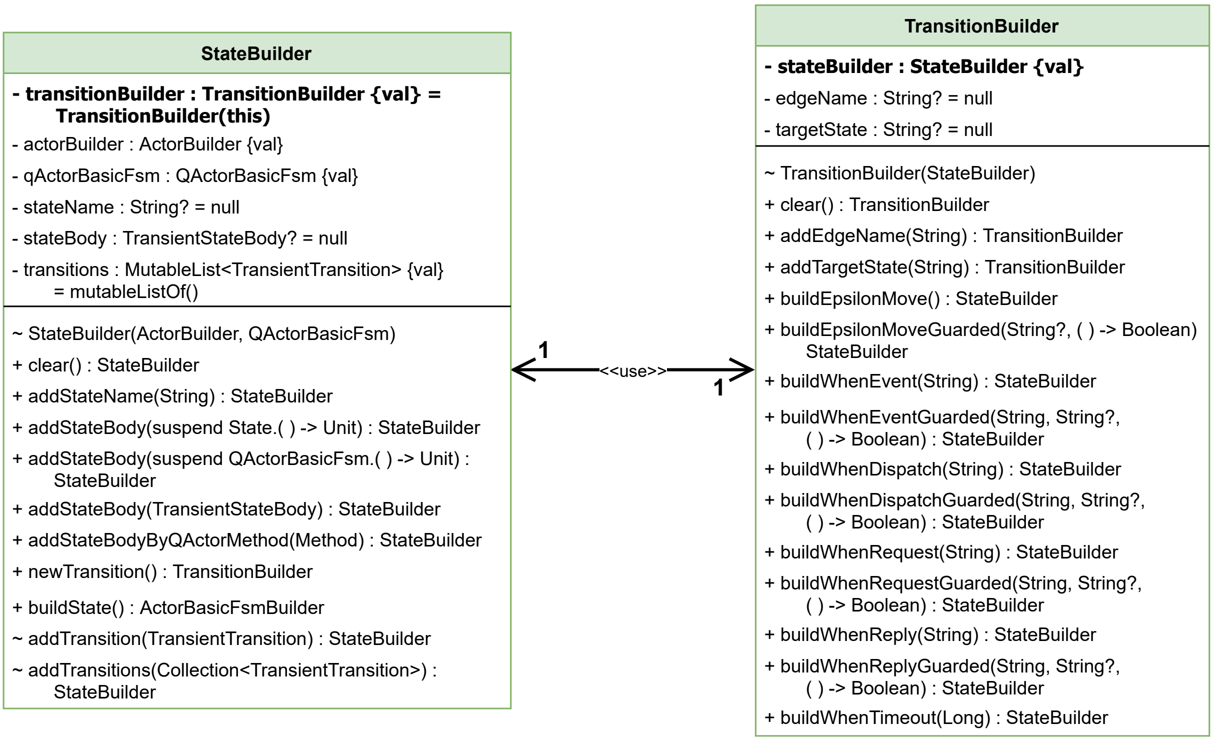

Figure 5: UML diagram for the for the state and transition builders

As anticipated, for finite state machine actors we also provide some additional builders shown in the figure 5:

- StateBuilder:
  The component for building states. If we have an `ActorBasicFsmBuilder` we can call the `newState()` method that returns an instance of the `StateBuilder` class that can be used to add states. When all of the states are added then it is possible to invoke the `buildState()` method that return the original actor builder. **Notice that it not possible to create a `StateBuilder` because it can only be obtained from an actor builder**.

- TransitionBuilder:
  The component for building transitions. It can be obtained using the `newTransition()`

method of the `StateBuilder` class with the same mechanism by which the state builder can be obtained from the actor builder. In addition, this component has more than one `build` method for each type of transition supported by the infrastructure.

3.2 The wrappers

As we have already said, the transient entities of the model are only a **passive description** of the system that will have to run. So this description must be transformed into the **executable units** that are present in the `QA` infrastructure: `ActorBasic` and `ActorBasicFsm`.

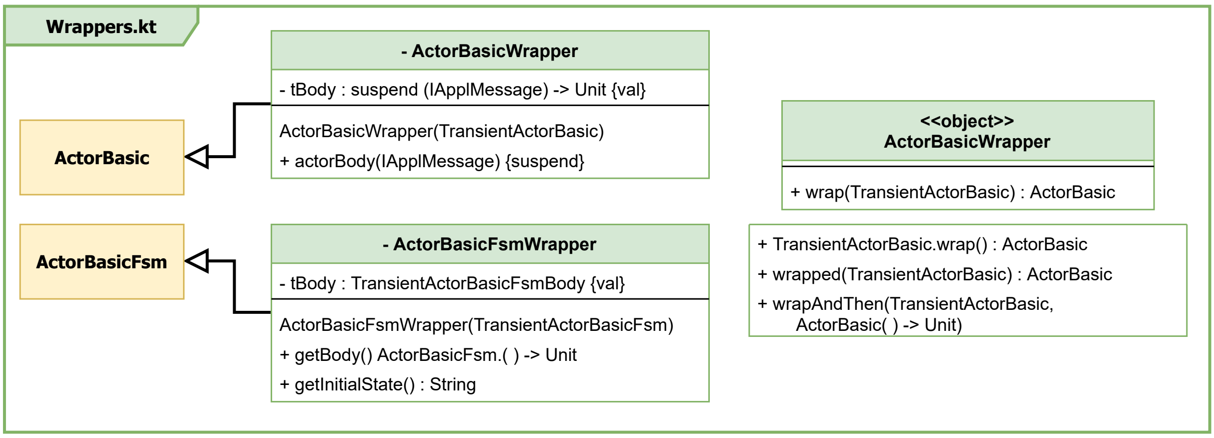

Figure 6: UML diagram for the wrappers

The `Wrappers.kt` file contains the classes to **wrap** the `TransientActorBasic` and the `TransientActorBasicFsm` entities into the active entities of the `QA-System`. This file also contains some extensions method for the `TransientActorBasic` class to quickly wrap it into an `ActorBasic` instance.

For the details about wrappers and their work, please see the source code.

3.3 Example of system creation using builders

Suppose to have a system with a context that contains an actor called *echoactor* with this behavior:

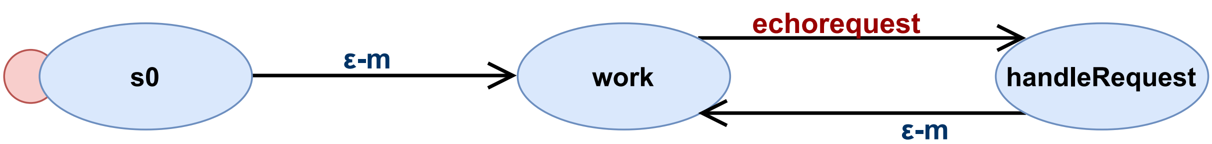

Figure 7: Behavior of the *echoactor*

This simple actor is able to handle a request called `echorequest` by answering with an `echoresponse` reply containing the same contents of the request. Then, in order to define the system using the builders, the procedure is:

Listing 1: Example of builders use

```
1 /* BODIES OF THE STATES FOR echoactor ***************************** */
2 val s0Body : suspend QActorBasicFsm.()-> Unit =
3  { println("started") }
4 val workBody : suspend QActorBasicFsm.() -> Unit =
5  { println("idle") }
6 val handleRequestBody : suspend QActorBasicFsm.() -> Unit =
7  { answer("echorequest", "echoreply", currentMsg.msgContent()) }
8 
9 /* SYSTEM BUILDER ********************************************** */
10 val sysBuilder = SystemBuilder()
11 
12 /* SYSTEM CREATION ********************************************* */
13 val system = runBlocking {
14  sysBuilder.addHostname("localhost").addScope(this)
15  //Context: "ctxecho"
16  .newContext()
17  .addContextName("ctxecho")
18  .addContextAddress("localhost").addContextPort(9000)
19  .addContextProtocol("TCP")
20  //Actor: "echoactor"
21  .newActorBasic().addActorName("echoactor")
22  .upgrateToFsmBuilder().addQActorBasicFsm(QActorBasicFsm())
23     //State: "s0"
24  .newState().addStateName("s0").addStateBody(s0Body)
25  .newTransition()
26  .addEdgeName("t0").addTargetState("work")
27  .buildEpsilonMove().buildState()
28  .setInitialState("s0")
29     //State: "work"
30  .newState().addStateName("work").addStateBody(workBody)
31  .newTransition()
32  .addEdgeName("t1").addTargetState("handleRequest")
33  .buildWhenRequest("echorequest").buildState()
34     //State: "handleRequest
35  .newState().addStateName("handleRequest")
36  .addStateBody(handleRequestBody)
37  .newTransition()
38  .addEdgeName("t2").addTargetState("work").buildEpsilonMove()
39  .buildState()
40  .buildInContext().second.buildInSystem().second.build()
41 }
```

The `.kt` source code is available here.

At the end of the execution of this snippet, the `system` variable contains the `OOP` description of the actor system with the `echoactor` described in the figure 7.

The motivations for the line 22 (`addQActorBasicFsm(QActorBasicFsm())`) will be clarified in the next section.

3.4 The last step for builders: adding support for make `TransientSystem` runnable

In the previous example we have created a complete description of the system contained into the `system` variable. But what do we do with this now? How we can run the

`TransientSystem`?

In order to do it, **we have to modify the launching methods of the `QA-System`**. Without going into details, we have created a new method into the QakContext.kt that has this signature:

```
fun createSystem(transientSystem : TransientSystem)
```

This method **creates and run the system** starting from a **`TransientSystem`** instance. In addition to this, we have created lots of method into the `sysUtil.kt` utility that helps the `createSystem()` to do its work such as:

```
fun createSystem(tSystem: TransientSystem, start : Boolean = true)
fun createContext(tCtx : TransientContext, hostName: String) : QakContext?
fun addTheActors(ctx: TransientContext, qakCtx : QakContext )
```

which follows the methods that were used by the old mechanism.

In order to conclude the example of the 1, we must add this line to run the system:

```
QakContext.createSystem(system)
```

4 Annotations

All the source code of this section is available here.

4.1 The example case: `LedSonar` system

Before talking about annotations, we show a small example that will be used to demonstrate some usages and to make some first approximated tests.

We will consider a simple case that is used in the course of *Ingegneria Dei Sistemi Software*: a system with a led and a sonar connected to a single board computer like Raspberry. **When the sonar detects a distance less than a threshold, the system must turn on the led. If the distance detected goes over this threshold, the led must be powered off.**

The figure 8 shows the diagram of the `ledsonar` system that will be used fot the examples. The legend of the used notation can be found here, but we will not go into the details of the logic.

In summary:

- `SonarActor`:
  This actor hold a sonar that it can use to read the value from it. The actor can receive request and answer to it but it also do polling emitting `sonarDistance` events with the current value of the distance read from the sonar.

- `LedActor`:
  This actor hold a led that it can command. The actor can receive dispatch `ledCmd(CMD)` with two possible value: `ON` for power on the led and `OFF` for turn it off.

- `DistanceActor`:
  This actor continuously monitors the distance emitted by the `SonarActor`: if the value is less then the threshold then emits a `distanceAlarm("CRITICAL")` event, otherwise if the the distance returns to a value greater than the threshold then it fires the event `distanceAlarm("NORMAL")`.

- `AppActor`:
  This actor realizes the business logic of the example system. When a `distanceAlarm` event is detected then it command the led in the proper way following the logic we have already shown.

We also need a *virtual environment* in order to test our example system. Then we have created a small *web based* architecture that simulates a *sonar* and a *led* using `WebSocket`. The source code of this small system made in `Kotlin` using `Ktor` can be found here.

When this system is started it possible to go at localhost:8000/index.html in order to get a small web based GUI that is shown in the figure 9. The GUI use `WebSocket` in order to send new distances when the slider of the sonar is moved or to receive the command to power on/off the led. In particular, the GUI offers:

- a **slider** to modify the distance read from the sonar;
- a **box** in which to type the new value of the threshold that can be sent over the WebSocket when the `Set` button is pressed;
- a **led image** that shows when the led is powered on or off.

For this reason the actor system in the figure 8 contains an additional actor called `ThresholdActor`: it listens the commands to update the threshold over the WebSocket and when a new value is received emits the proper event `newTreshold`.

In order to start this system it possible to call the `startLedSonarSystem()` function that is present in the file LedSonarSystem.kt.

If you want to use a real sonar or a real led you only have to extends properly the two `POJO` owned by the `SonarActor` and the `LedActor`.

4.2 Descriptive annotations [ledonarsystem0]

The focus of our discussion is **to introduce a new mechanism do let the developer able to easily describe the system using annotations**. The use of the annotation can be very useful:

- they do not depend on an IDE and are included in Java so there are no compatibility problems;
- they can completely describe all the aspects of the actor system;
- the use of the annotations is very easy and intuitive.

So the first step is **to develop some annotations to eliminate the need for the `.pl` file**. This annotations let the system able to find all the information about the contexts and the actors that are defined.

- QakContext:
  This annotation is applicable to a class and give to the system the information about a context (name, address, protocol and port).

```
annotation class QakContext(
 val contextName : String,
 val contextAddress : String,
 val contextProtocol : String,
 val contextPort : Int,
)
```

- QActor:
  This annotation is applicable to a class that is an actor and then ***it must extends*** ***`ActorBasic`****. It maintains all the information of the actor (name, context, ecc. . .)*

```
annotation class QActor(
 val contextName : String,
 val actorName : String = "",
 val discardMessage : Boolean = false,
 val confined : Boolean = false,
 val ioBound : Boolean = false,
 val channelSize : Int = 50
)
```

- Hostname:
  This annotation is applicable to a class and indicates the hostname of the system. This annotation is not strictly necessary but at the moment it's mandatory to identify the context that have to run locally.

```
annotation class HostName(
 val hostname : String
)
```

Actually, there is no full support to the description of remote actors but in future development it is easy to add it.

Then, as explained the *location* of the entities of the system can be describing using these annotations. The ledonarsystem0 example shows how it is possible to use them. In this paper we just show the SonarActor and the LedActor:

Listing 2: `SonarActor` (`ledsonarsystem0`)

```
@QActor("ctxLedSonarAbFsmDemo")
class SonarActor(name : String, scope : CoroutineScope) :
 ActorBasicFsm(name, scope, autoStart = false) {

 var distance = -1
 var prevDistance = distance
 val sonar = SYSTEM_SONAR

 override fun getBody(): ActorBasicFsm.() -> Unit {
  return {
   state("begin") {
    transition(edgeName = "t0", targetState = "work", cond = doswitch())
   }

   state("work") {
```

```
    action {
     stateTimer = TimerActor("timer_begin",
     scope, context!!, "local_tout_${name}_$stateName", 2000 )
    }
    transition(edgeName = "t1", targetState = "readSonar",
    cond = whenTimeout("local_tout_${name}_$stateName"))
    transition(edgeName = "t2", targetState = "answareWithActual",
    cond = whenRequest("readDistance"))
   }

   state("readSonar") {
    action {
     prevDistance = distance
     distance = sonar.read()
     if(prevDistance != distance) {
      emit("sonarDistance", "sonarDistance($distance)")
     }
    }
    transition(edgeName = "t3", targetState = "work", cond = doswitch())
   }

   state("answareWithActual") {
    action {
     replyToCaller("readDistance", "readDistance($distance)")
    }
    transition(edgeName = "t4", targetState = "work", cond = doswitch())
   }
  }
 }

 override fun getInitialState(): String {
  return "begin"
 }

}
```

Listing 3: LedActor (ledsonarsystem0)

```
@QActor("ctxLedSonarAbFsmDemo")
class LedActor(name : String, scope : CoroutineScope) :
 ActorBasicFsm(name, scope, autoStart = false) {

 val led = SYSTEM_LED

 override fun getBody(): ActorBasicFsm.() -> Unit {
  return {

   state("begin") {
    action {}
    transition(edgeName = "t0", targetState = "work", cond = doswitch())
   }

   state("work") {
    action {}
    transition(edgeName = "t1", targetState = "handleLedCmd",
     cond = whenDispatch("ledCmd"))
   }

```

```
    state("handleLedCmd") {
     action {
      try {
       if(checkMsgContent(
        Term.createTerm("ledCmd(CMD)"),
        Term.createTerm("ledCmd(CMD)"),
        currentMsg.msgContent()))
       when(val ledCmdArg = payloadArg(0)) {
        "OFF", "off" -> {
         if(led.isPoweredOn()) {led.powerOff()}
        }
        "ON", "on" -> {
         if(led.isPoweredOff()) {led.powerOn()}
        }
       }
      } catch (e : Exception) {/* ... */}
     }
     transition(edgeName = "t2", targetState = "work", cond = doswitch())
    }
   }
  }

  override fun getInitialState(): String {
   return "begin"
  }
}
```

With the others class that we have not reported here, the description of the actors of the system is completed. But the system can not be started yet because there are nothing able to make it run reading the annotations. It is needed a component that **scans all the classes and load the annotated one**. Indeed, we have developed a new component called `AnnotationLoader` with the method

```
fun loadSystemByAnnotations(params : ReadableParameterMap
    = immutableParameterMap()) : SystemBuilder
```

This method **scan all classes of the application and search for annotated** loading them depending on the annotation. We have also added a component called `QakLauncher.kt` with proper methods that launch the system calling the `AnnotationLoader`. So the `Kotlin` script that launches the system is:

Listing 4: `App.kt` (`ledsonarsystem0`)

```
@QakContext("ctxLedSonarAbFsmDemo", "localhost", "TCP",9000)
@HostName("localhost")
class ContextConfiguration

fun main(args: Array<String>) {
 startLedSonarSystem()

 runBlocking {/*this:CoroutineScope*/
  launchQak(this)
 }
}
```

Notice that this mechanism is added to the infrastructure as an extension and without invalidating the old `.pl` definition.

4.3 New classes for injection

We remember that ActorBasic and ActorBasicFsm classes are *abstract* so it is not possible to directly create instances of this classes because of the *abstract* definition. For this reason we have created the two classes shown in the previous section and that are used from the builders: `ActorBasicWrapper` and `ActorBasicFsmWrapper` that takes the transient actor definitions and wrap them into `ActorBasic` or `ActorBasicFsm` instances.

Now it is clear that a level of strong separation has been introduced between the ***definition*** of the actor system and the ***runnable implementation*** of it.

In addition to this, the new `@QActor` annotation allow the developer to specify the actor's name directly into the proper annotation field. So, if the developer is forced to directly extends the `ActorBasic` class, he must always specify a constructor with the two parameter needed by the superclass as done in the previous listings.

We want not only to avoid this but also to introduce **a new stronger level of separation** between the runnable part of the actor (represented by the `ActorBasic` inherited type) and the description (represented by the override of the proper methods).

Then, first, we have created new classes in order to realize the separation: `QActorBasic` and `QActorBasicFsm`. These two classes are illustrated in the figure 10 can be used in order to define the behaviour of the actors (*basic* or finite state machine) by extending them and adding other mechanisms such as some proper annotations that will be shown.

However, the application designer can decide to insert some operations inside the behaviour that requires to be invoked on the *runnable instance* of the actor he is developing. For this reason, the `QActorBasic` must anyway maintain the *runnable part* that is the `ActorBasic` instance (and the correspondent `ActorBasicFsm` for the `QActorBasicFsm`) that unfortunately should be available only after the instantiation of the `Q` classes.

As shown in figure 10, the main operations that require the executable instance are related with sending messages and events or performing `CoAP` updates or `Mqtt` interactions.

The `QActorBasic` and `QActorBasicFsm` classes are very useful in order to define new methods for describing the behaviour of the actors, but they do not provide a full and complete mechanism for this. What the developer needs to put inside these classes? And above all, how it is possible to initialize the `ActorBasic` variable inside?

We will answer to the first question in the next subsection. Indeed, for the second, we can say that the most easy and useful way to put the `ActorBasic` instance is by using reflection. So, the application designer writes the code to create an actor using `QActorBasic` and then the component able to load the class instantiates the related `ActorBasic` instance and inject it to the `Q` class.

4.4 `AutoQ` classes [ledonarsystem1]

In the figure 10 are present two new classes that we have not clarified: `AutoQActorBasic` and `AutoQActorBasicFsm`. They are very simple because their only work is to let the application designer to use the *legacy* actor definition with the *descriptive annotation* without forcing him to use the classical constructor of the `ActorBasic` or `ActorBasicFsm` classes.

So, **it is possible to use the `@QActor` annotation with a class that extends** `AutoQActorBasic` without specifying any parameter in the superclass constructor (and in this sense it is *auto*).

For example, consider the `SonarActor` in the ledonarsystem1. With these new classes the code becomes:

Listing 5: `SonarActor` (`ledsonarsystem1`)

```
@QActor("ctxLedSonarAutoQAbFsmDemo")
class SonarActor : AutoQActorBasicFsm() {

 /* ... [same definition od variables of ledsonarsystem0] ... */

 override fun getBody(): ActorBasicFsm.() -> Unit {
  /* ... [ same code of ledsonarsystem0 ] ... */
 }

 override fun getInitialState(): String {
  /* ... [ same code of ledsonarsystem0 ] ... */
 }

}
```

As we can see there are no `name : String` and `scope : CoroutineScope` parameters in the constructor like the previous version. Nothing else to say about these two classes.

4.5 Behavioural annotations [ledonarsystem2]

At this point we have some annotations the developer can use to describe and *identify* actors and contexts in the system instead of using the old `.pl` files. In addition to this, we have provided the new classes `QActorBasic` and `QActorBasicFsm` that open the way to new mechanisms to define the behaviour.

Indeed, we have created new annotations that can be used in `QActorBasic` and `QActorBasicFsm`:

- `ActorBody`:
  This annotation is applicable to a method of a class that extends `QActorBasic` and represents the classical body of `ActorBasic`. ***The signature of the marked method must be the same of the `ActorBasic#actorBody()`, so it must return `Unit` and take one and only one `IApplMessage` input parameter.***

  ```
  annotation class ActorBody()
  ```

- `State`:
  This annotation is applicable to a method of a class that extends `QActorBasicFsm`) and represents a state with a name and a body that is the method that is marked. The name can also be not present and in this case it is taken from the name of the method the annotation is marking. ***The marked method must has no input parameter while the return type is ignored.***

  ```
  annotation class State(
   val name : String = ""
  )
  ```

- Initial:
This simple annotation is applicable to a method of a class that extends `QActorBasicFsm`) and marks the state that is the initial for a finite state actor.

```
annotation class Initial()
```

- EpsilonMove, WhenDispatch, WhenReply, WhenEvent, WhenInvitation, WhenTime:
*This group of annotations represents the transitions that can be performed by a finite state machine after the execution of the body of a state. So, **all of these annotations have sense if put to a method that has the `@State` annotation** and define the transition to be invoked after the referred state body is executed. Notice that these annotations are `Repeatable` so it is possible to link more than one transition of the same type to the same state (particularly useful for guarded transitions). All of them has an `edgeName` and a `targetState` and others parameters based on the type of the transition.*

```
annotation class EpsilonMove(
  val edgeName : String,
  val targetState : String
)

annotation class WhenDispatch(
  val edgeName : String,
  val targetState : String,
  val messageName : String
)

annotation class WhenRequest(
  val edgeName : String,
  val targetState : String,
  val messageName : String
)

annotation class WhenReply(
  val edgeName : String,
  val targetState : String,
  val messageName : String
)

annotation class WhenInvitation(
  val edgeName : String,
  val targetState : String,
  val messageName : String
)

annotation class WhenEvent(
  val edgeName : String,
  val targetState : String,
  val eventName : String
)

annotation class WhenTime(
  val edgeName : String,
  val targetState : String,
  val millis : Long
)
```

- GuardFor:
This simple annotation is applicable to a method of a class that extends `QActorBasicFsm` and marks a method that is the guard for an existing transition. ***The method marked with this annotation must imperatively have no input parameters and return a*** ***`Boolean`*** *that is* ***`true`*** *if the referred transition has to be performed depending on the guard or* ***`false`*** *otherwise. The association between the guard and the transition is made by the* ***`transitionEdgeName`*** *parameter of this annotation and the state to reach if the guard is* ***`false`*** *is maintained by the* ***`elseTarget`*** *variable that can be empty if no state has to be reached.*

```
annotation class GuardFor(
 val transitionEdgeName : String,
 val elseTarget : String = ""
)
```

So the `AnnotationLoader` has been extended in order to support these new annotations letting the software developer to use them in order to define not only the actors and the contexts that are present but also their behaviour, their states and their actions.

The ledonarsystem2 example shows how it is possible to use these new annotations. In this paper we just show the SonarActor and the LedActor as previously done:

Listing 6: `SonarActor` (`ledsonarsystem2`)

```
@QActor("ctxLedSonarQAbFsmDemo")
class SonarActor : QActorBasicFsm() {

 var distance = -1
 var prevDistance = distance
 val sonar = SYSTEM_SONAR

 @Initial
 @State
 @EpsilonMove("t0", "work")
 suspend fun begin() {
 }


 @State
 @WhenTime("t1", "readSonar", 2000)
 @WhenRequest("t2", "answareWithActual", "readDistance")
 suspend fun work() {
 }

 @State
 @EpsilonMove("t2", "work")
 suspend fun readSonar() {
  prevDistance = distance
  distance = sonar.read()
  if(prevDistance != distance) {
   emit("sonarDistance", "sonarDistance($distance)")
  }
 }

 @State
 @EpsilonMove("t3", "work")
 suspend fun answareWithActual() {
```

```
    answer("readDistance", "readedDistance", "readedDistance($distance)")
  }

}
```

Listing 7: `LedActor (ledsonarsystem2)`

```
@QActor("ctxLedSonarQAbFsmDemo")
class LedActor : QActorBasicFsm() {

  val led = SYSTEM_LED

  @Initial
  @State
  @EpsilonMove("t0", "work")
  suspend fun begin() {
  }

  @State
  @WhenDispatch("t1", "handleLedCmd", "ledCmd")
  suspend fun work() {
  }

  @State
  @EpsilonMove("t2", "work")
  suspend fun handleLedCmd() {
    actorPrintln("Current command: $currentMsg")
    try {
      when(val ledCmdArg = currentMessageArgs[0]) {
        "OFF", "off" -> {
          if(led.isPoweredOn()) {
            led.powerOff()
            actorPrintln("Led powered off")
          }
        }
        "ON", "on" -> {
          if(led.isPoweredOff()) {
            led.powerOn()
            actorPrintln("Led powered on")
          }
        }
        else -> {}
      }
    } catch (e : Exception) {
      e.printStackTrace()
    }
  }

}
```

Notice that nothing has changed in the `App.kt` file but simply the `AnnotationLoader` has been extended as we already said.

4.6 Infix Functions

`Kotlin` let the developer write functions with the **infix notation**.

Thanks to this possibility we can make the code smarter defining some infix function for sending messages, emitting events and updating `CoAP` states. For more details about that, see the implementations of `QActorBasic` and `QActorBasicFsm` classes.

We only show some *smart* way to do these operations:

```
/* Emits the event "eventName(arg0,arg1,arg2)" */
emit event "eventName" withArgs "arg0,arg1,arg2"

/* Emits the event "eventName(arg0,arg1,arg2)" (smart syntax for args)
   This syntax can be use with all 'withArg' keywords */
emit event "eventName" withArgs arg ["arg0", "arg1", "arg2"]

/* Sends the dispatch "dispatchName(arg0,arg1)" to "destActor" */
send dispatch "dispatchName" to "destActor" withArgs "arg0,arg1"

/* Sends the request "requestName(arg0,arg1,arg2,arg3)" to "destActor" */
send request "requestName" to "destActor" withArgs "arg0,arg1,arg2,arg3"

/* Answers to request "requestName" with the reply "replyName(arg0)" */
replyTo request "requestName" with "replyName" withArgs "arg0"

/* Updates the CoAP resource */
update resource "new coap update"
```

We underline that are present two types of primitives for the argument of messages:

- by using `withArgs` keyword that adds the arguments to the message using the same syntax of the classical `QAK` DSL;
- by using `withContext` keyword that directly put the raw contents into the message.

5 Conclusions

By summarizing, we started our work by **defining** passive entities that only describe all the components of an actor based system (in terms of contexts, actors, states and transitions). After, we said that we used the *builder pattern* in order to create these entities. Then, the builders can be used from others component in order to load the system with a certain mechanism and after wrap them into `ActorBasic` instances thanks to some *wrappers*. The mechanism we have presented is based on **annotations**, and we have shown all those available.

Now, the application designer can use three ways to define his own actor system:

1. **using only the *descriptive annotation*** with the legacy `ActorBasic` and `ActorBasicFsm` classes [ledonarsystem0]
2. **using only the *descriptive annotation*** with the new `AutoQActorBasic` and `AutoQActorBasicFsm` classes that let to use the empty constructor [ledonarsystem1]
3. **using the full annotation support (*descriptive* and *behavioural*)** with the new `QActorBasic` and `QActorBasicFsm` classes [ledonarsystem2]

Surely, the third mechanism is the most effective and practical with the smartest code even if it has not the same *language support* of the original `DSL`. However, it has the great advantage **to be completely independent of any IDE** or third-party tools because the annotations are directly supported from `Java` and `Kotlin`.

Appendices

A Additional annotations

In addition to the *descriptive annotations* that have already been presented, we develop some other annotations in order to configure the system for *logging* or `Mqtt`:

- `MqttBroker`:
  This annotation is applicable to a class (preferably the same of the `@QakContext` *annotation) and specify the* `Mqtt Broker` *of the system like the original* `DSL`*.*

```
annotation class MqttBroker(
  val address : String,
  val port : Int,
  val topic : String
)
```

- `Tracing`:
  This annotation is applicable to a class (preferably the same of the `@QakContext` *annotation) and enable the* `trace` *option of the legacy* `DSL`*.*

```
annotation class Tracing(
  val active : Boolean = true
)
```

- `MsgLogging`:
  This annotation is applicable to a class (preferably the same of the `@QakContext` *annotation) and enable the* `msglogging` *option of the legacy* `DSL`*.*

```
annotation class Msglogging(
  val active : Boolean = true
)
```

B Invocation of suspend methods

As we said, in the methods marked with `@State` annotation, the application designer can call some methods like `emit(...)`, `send(...)` or `answer(...)` but according to the `QAK` specifications these operations are `suspend fun`.

So, when the `AnnotationLoader` load all the classes marked with `@Actor`, it has to consider that some *state methods* can be suspendable, so they must be called from a `Coroutine`. For this reason, we have developed `MethodUtils.kt`, a small `.kt` file with some utility methods, in particular we have the function:

```
suspend fun Method.invokeSuspend(obj : Any, vararg param : Any?) : Any
```

This method uses the `suspendCoroutineUninterceptedOrReturn` built-in function that obtain the current coroutine `Continuation` and call be `block` passed as parameter inside this continuation. Summarizing, the extension function `invokeSuspend` we have defined is able to obtain the current continuation and invoke the method using the instance which calls the function by using reflection.

Let us make a simple example (see `InvokeSuspendDemo.kt`):

Listing 8: Example for `invokeSuspend`

```
class ExampleClazz(val exampleName : String) {

 suspend fun suspendWelcome(name : String) {
  println("[${exampleName}] Welcome from suspend $name")
 }
}

fun main(args : Array<String>) {
 val suspendWelcomeMethod = ExampleClazz::class.java.methods
 .find { it.name == "suspendWelcome" }
 val exampleInst = ExampleClazz("EXAMPLE NAME")
 runBlocking {
  suspendWelcomeMethod?.invokeSuspend(exampleInst, "main")
 }
}
```

The example shows how it is possible to use our `invokeSuspend` function in order to invoke a method using an instance of `Method` class (by reflection).

Notice that **at the moment a method marked with `@State` annotation must be `suspend`**. This is a *small* limitation because the developer is forced to make all *state methods* `suspend`, but it is not a problem and in future developments this mechanism can be improved very rapidly.

C Suspendable actors

In addition to our work, we also want to resolve a small problem that is: how to *pause* an actor? And what does *paused* means for an actor?

When an actor receives a *pause* command in a certain S_x state, from this moment it will not handle any messages until he receives the *resume* command, then it regularly returns to S_x state at the same conditions it was before pausing.

So, what appends to the messages that are received while the actor is paused? There are two possibilities:

1. the messages are **ignored** and discarded;
2. the messages are **restored** and handled by the actor when it receives the `resume` command.

Obviously, the `pause` and `resume` command are **messages** with a dedicated and fixed id. We choose to use `sys_suspend_actor` and `sys_resume_actor` as the ids for `pause` and `resume`. We also decide to let the application designer to choose how to manage the messages while an actor is in pause.

The figure 11 shows the components for the suspension mechanism. The main classes are `SuspendableActorBasic` and `SuspendableActorBasicFsm` and their names suggest their use.

Please notice the class `SuspendableCore` that is developed in order to make the code very reusable and to limit code repetitions. In addition to all of these classes we also provide `EchoSuspendableActor` that show some examples of this pausing mechanism.

Without going deeply into details, we can say that this mechanism is based on the `actorBody` function of the `ActorBasic` class that is extended by `SuspendableActorBasic` and `SuspendableActorBasicFsm`. As you can see in the source code, the overriding methods call the function `handleMessage` defined into `SuspendableCore` that is able to decide to handle the message if not pausing or to do something else if the actor is in pause mode. Then, the `handleMessage` is a sort of *filter* that decides if the message has to be handled.

So, in order to define actors with the ability to go into suspension, the application designer must extend `SuspendableActorBasic` implementing `onMessageWhenNotSuspended` or `SuspendableActorBasicFsm` with the normal procedure used also for `ActorBasicFsm`. All the job is done by the already implemented `actorBody` function.

In order to decide if the messages have to be saved into a buffer and restored when resumed or to be discarded, the developer can pass a proper boolean inside the constructor of the suspendable actor classes. If the messages are stored into a buffer, when the actor receives the `resume` command, then all the saved messages are restored **in order** and the actor pass into a *meta-state* that is called `RESUMING` (see the `SuspensionState` enumeration). After all messages have been restored, the actor returns at the normal work in the state it was before the suspension.

Notice that as anticipated, we have added a new *layer of state* to the actors that concerns only the *suspension state of the actor* and that is defined by the `SuspensionState` enumeration. This kind of state of a `SuspendableActorBasic` (or `SuspendableActorBasicFsm`) is observable thanks to the new Kotlin StateFlow that can be obtained from the owned `SuspendableCore` of these actors.

If a suspendable actor is resuming, then also the messages that arrives in this state are enqueued into the buffer. In addition to this, please notice that a finite state machine actor does not really change his *fsm* state during the pause, but it is not able to have the normal work because it is locked by the `SuspendableCore`.

In future development, it is easy to add a new annotation like `@Suspendable` that marks an actor that can be suspended. It is only required to add the proper support for this kind of annotation to the `AnnotationLoader` that can inject a `SuspendableActorBasic(Fsm)` to a `QActorBasic` instance.

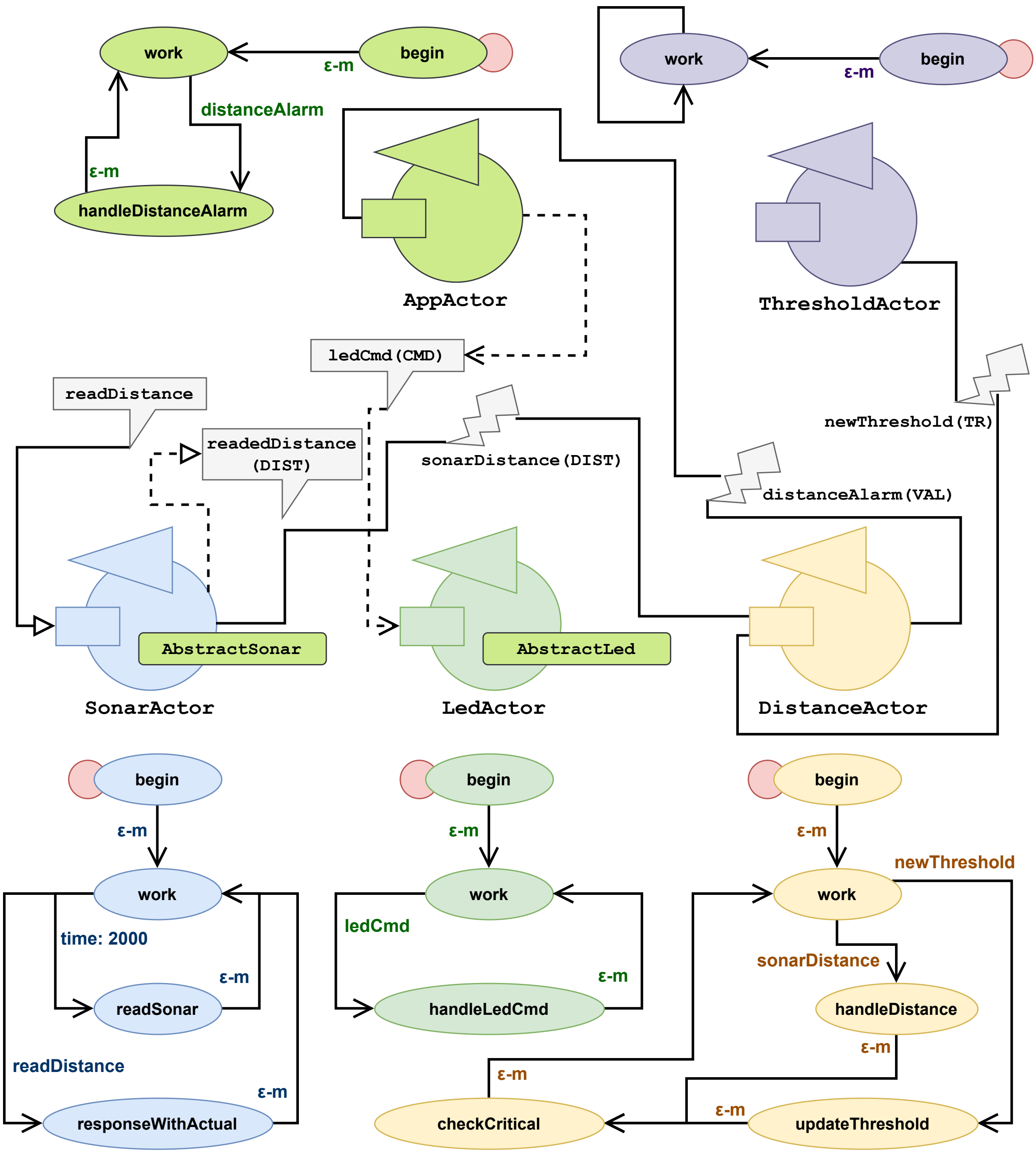

Figure 8: Diagram of the `ledsonar` system

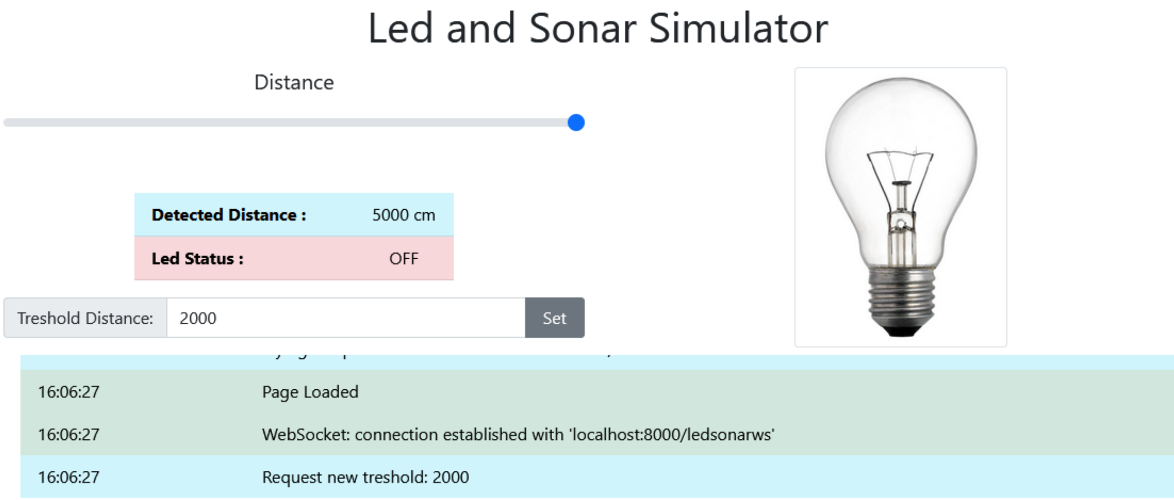

Figure 9: Web GUI of the `ledsonar` system

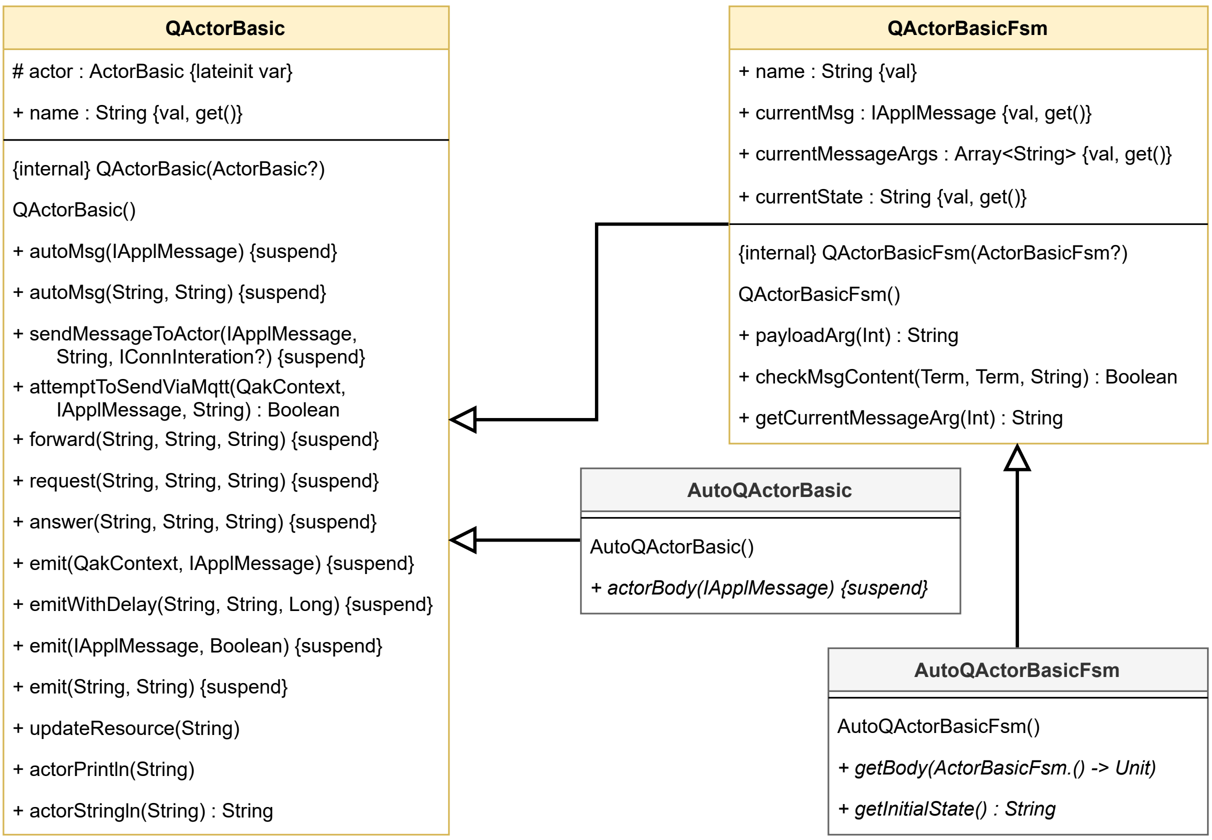

Figure 10: `Q` and `AutoQ` classes

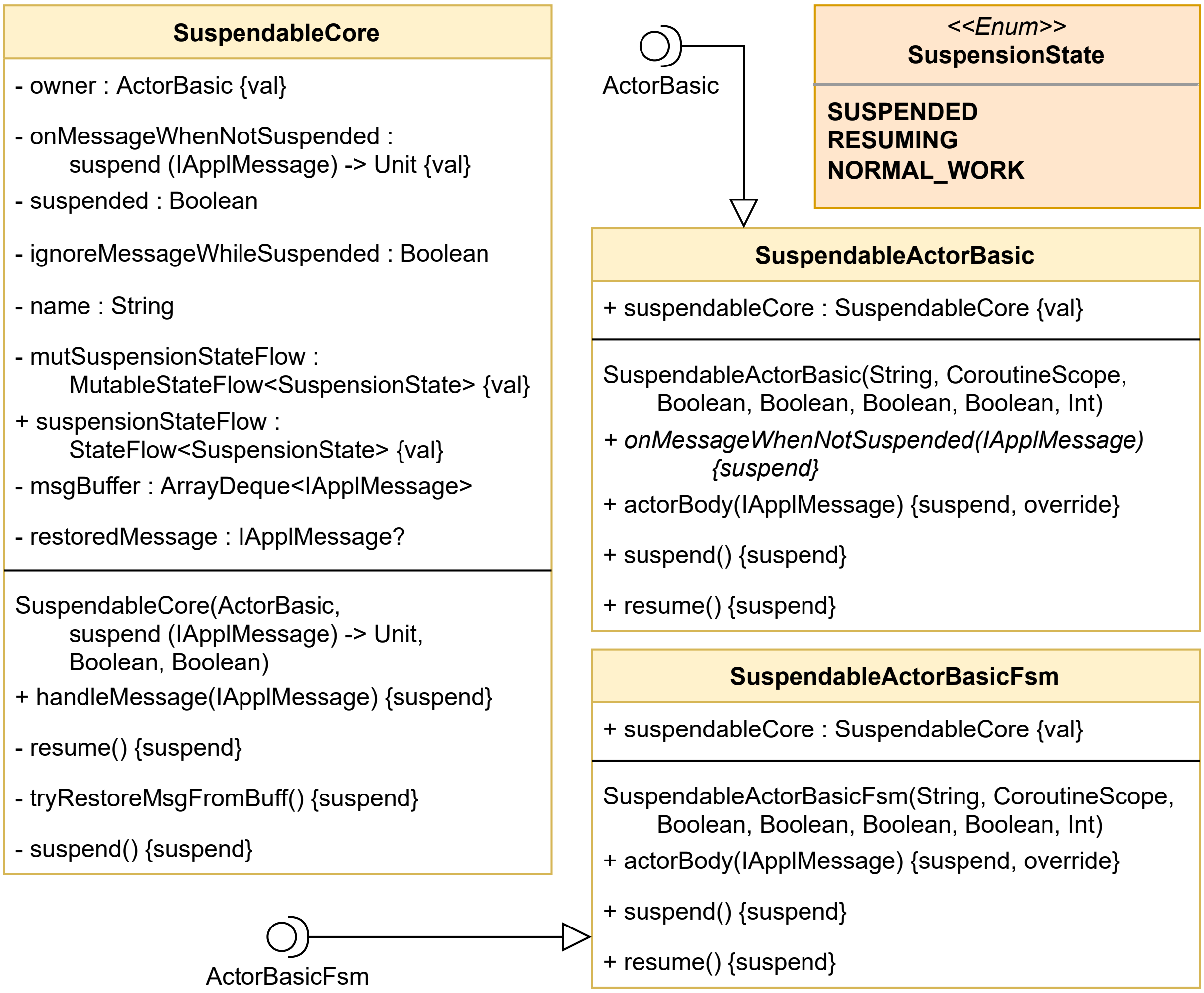

Figure 11: Suspendable classes