QActor Extensions: Additional mechanisms to define the model

Luca Marchegiani

May 14, 2022

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 a 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 that follows the mechanism the 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***.

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.

[1] See the documentation for more details.

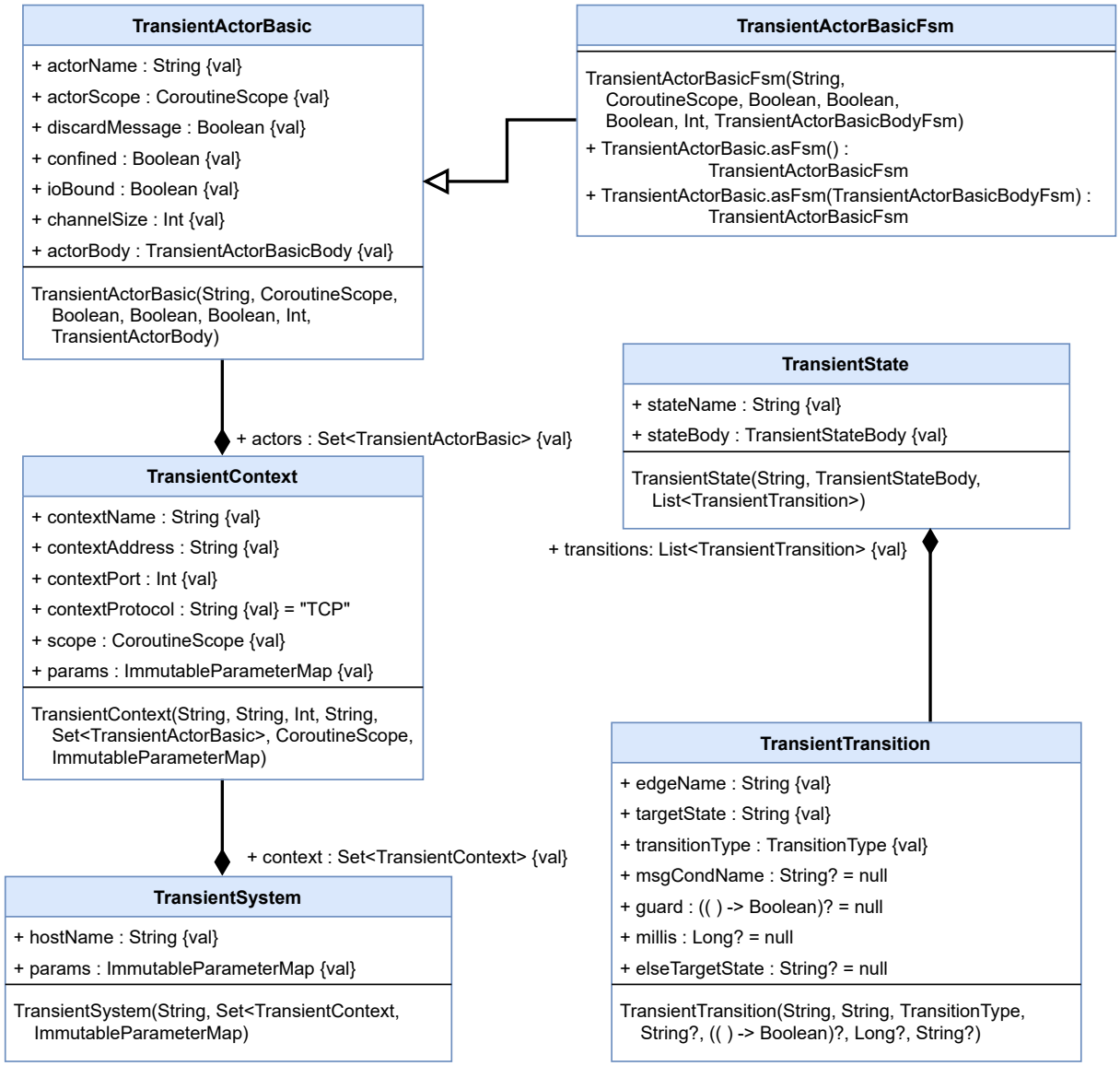

Figure 1: UML diagram for the Transient Model

- `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 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.

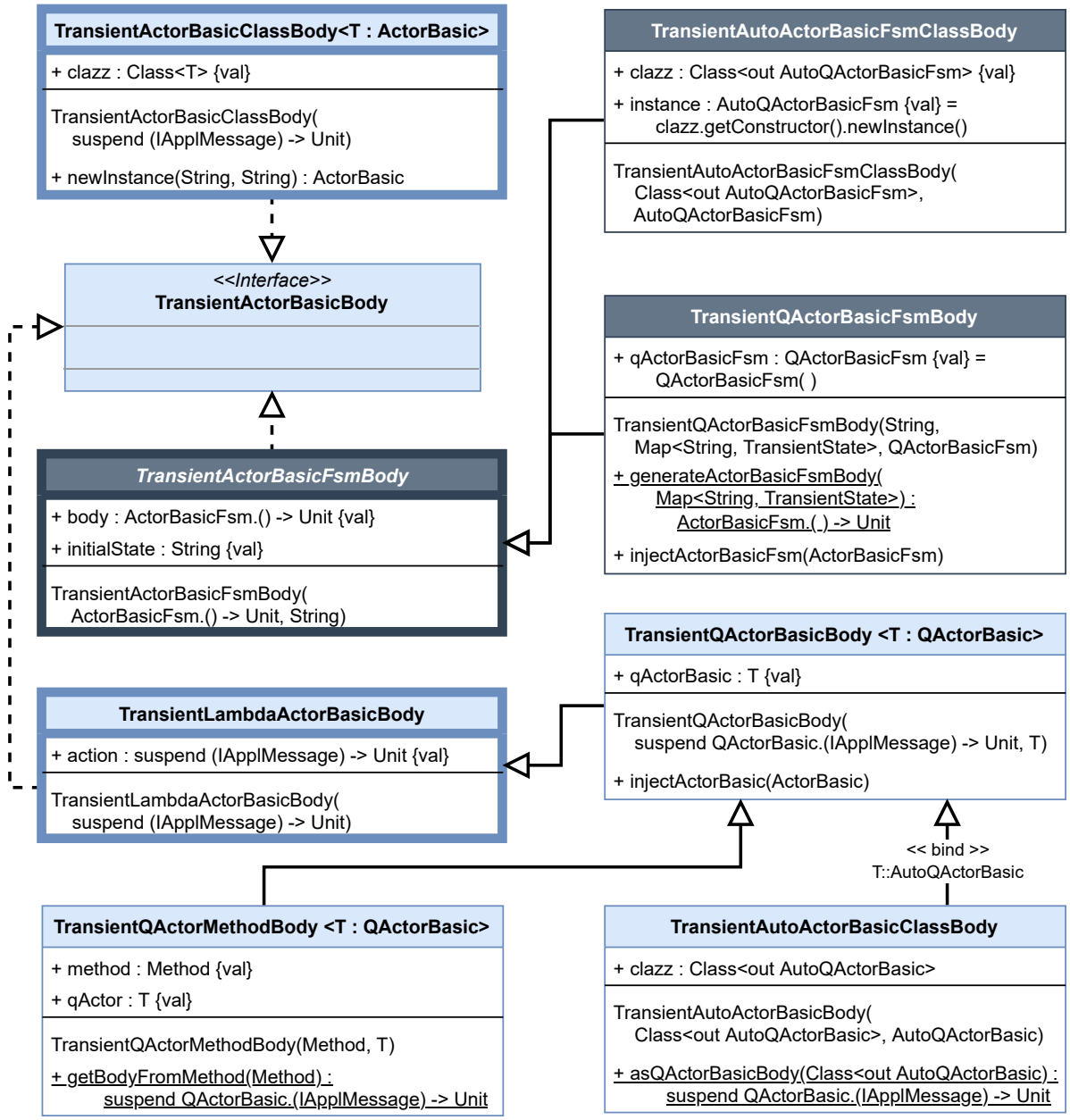

Figure 2: UML diagram for the Transient Model of the actor body

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`:
  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.

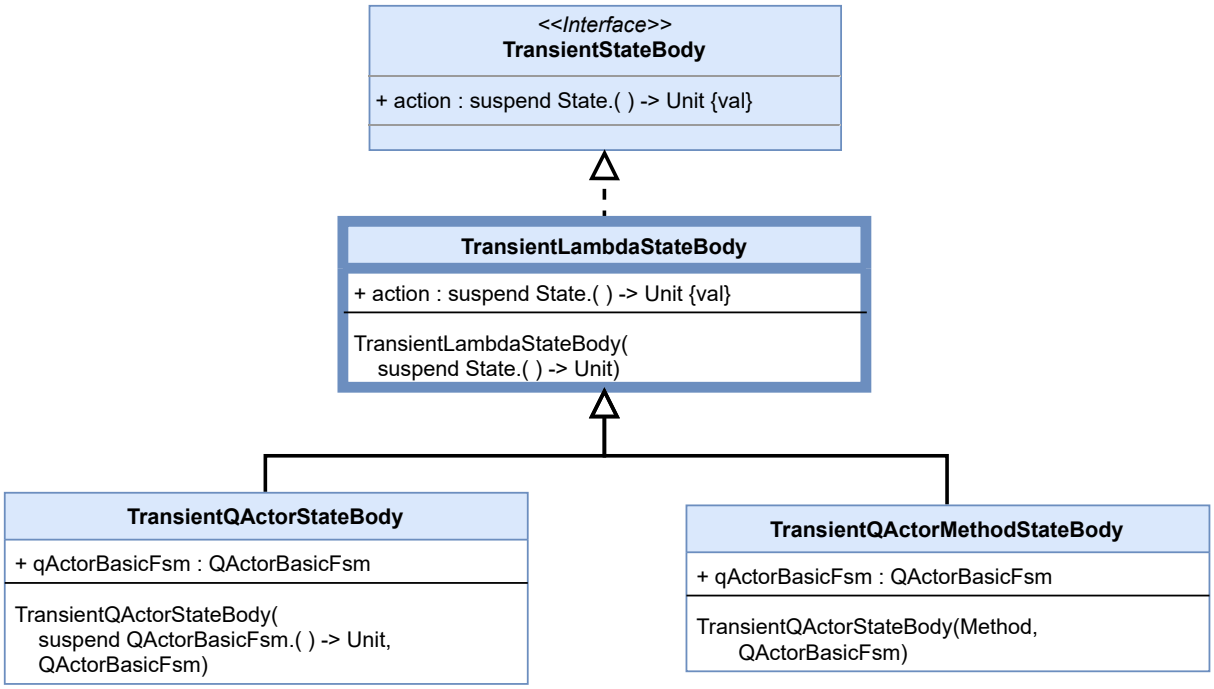

Figure 3: UML diagram for the Transient Model of the actor body

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

- `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.

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

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.

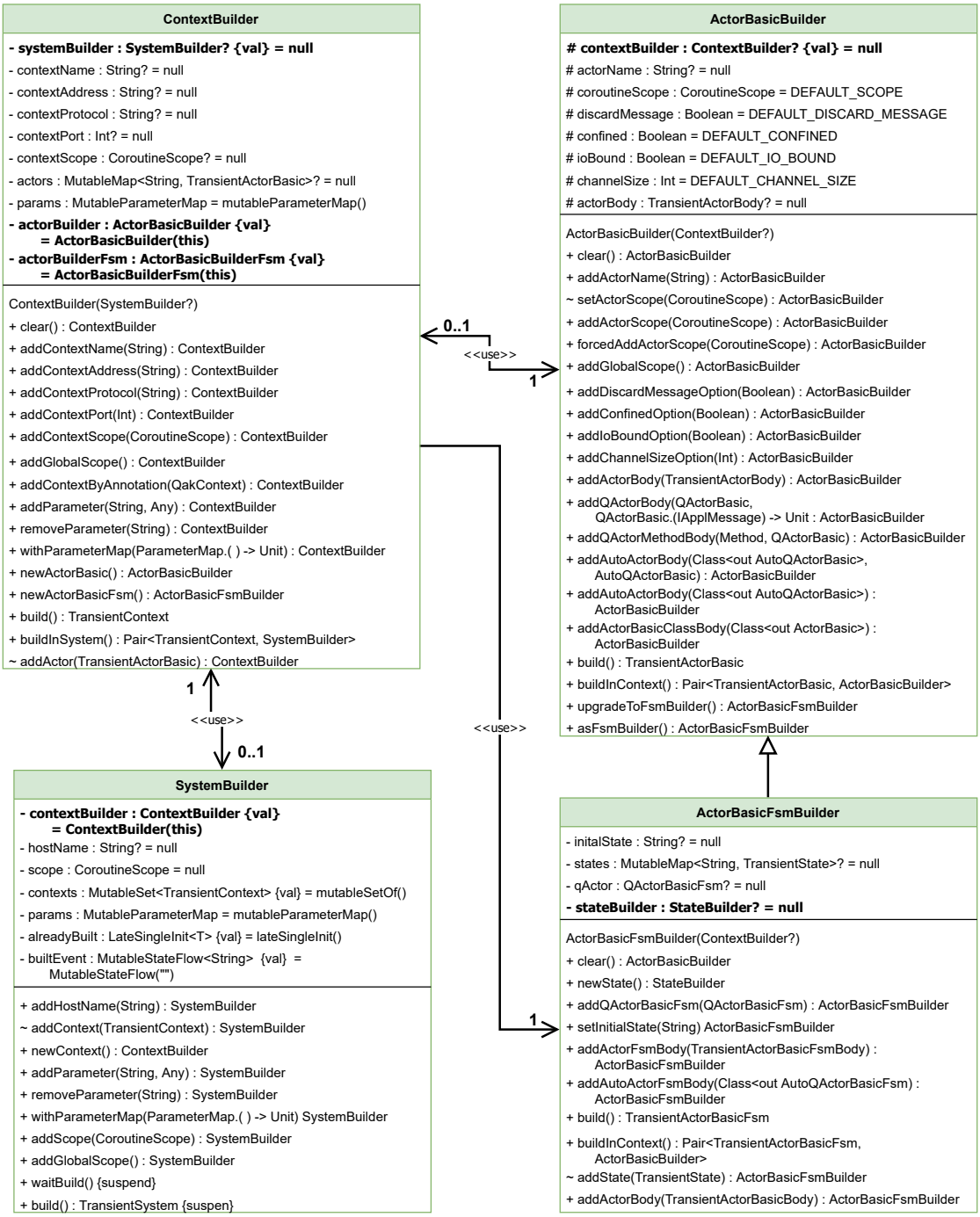

Figure 4: UML diagram for the actor, context and system builders

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`:
  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.

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.

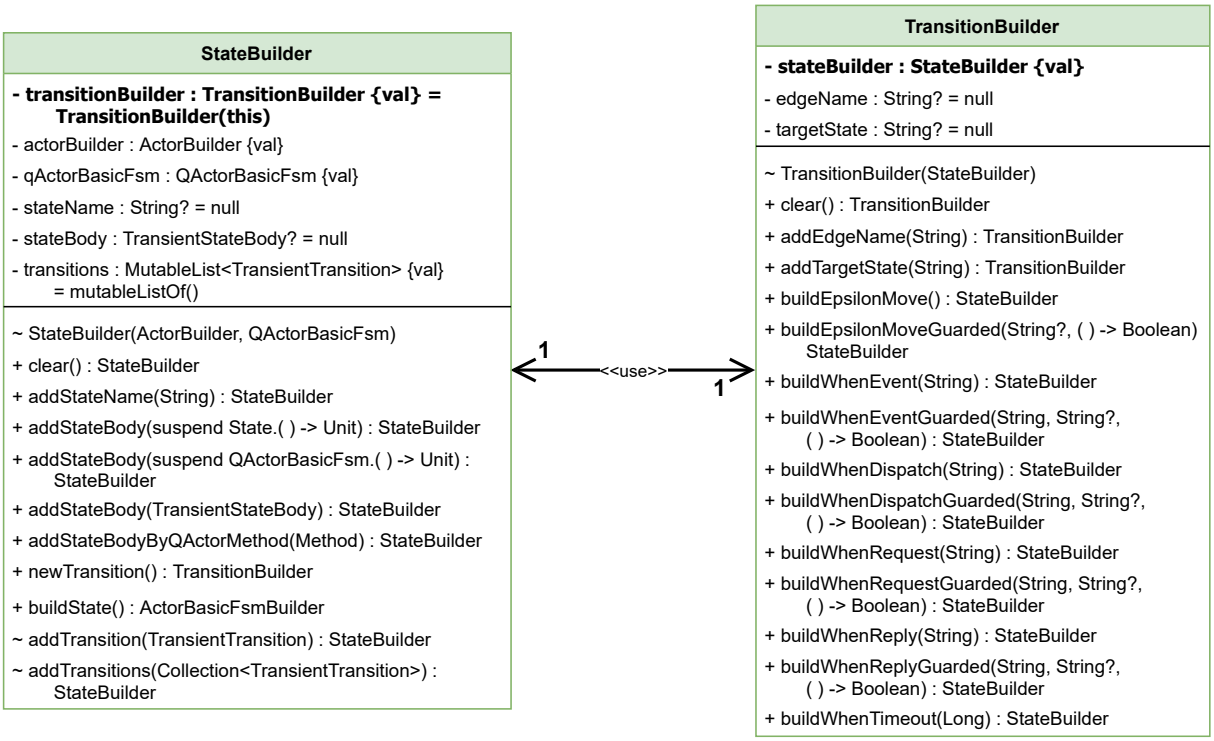

Figure 5: UML diagram for the for the state and transition builders

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`.

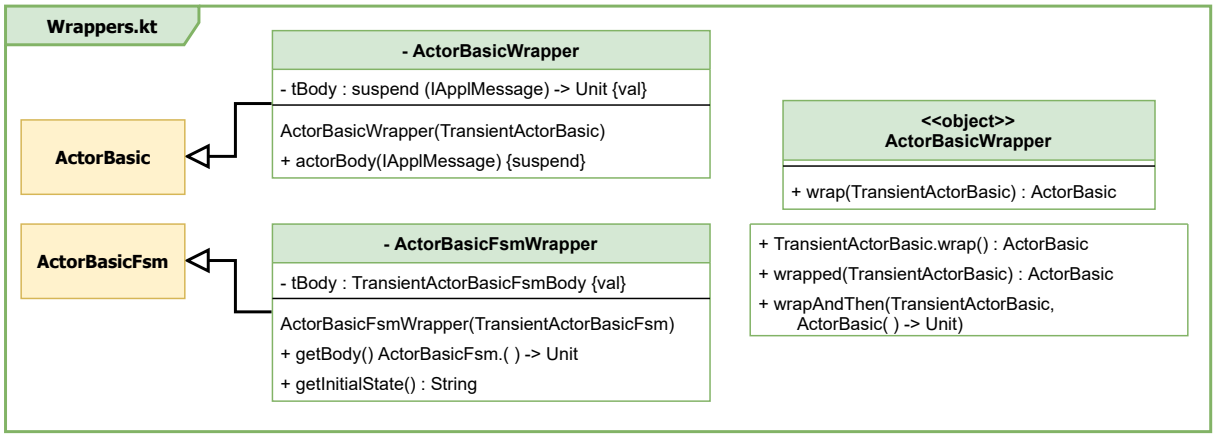

Figure 6: UML diagram for the wrappers

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:

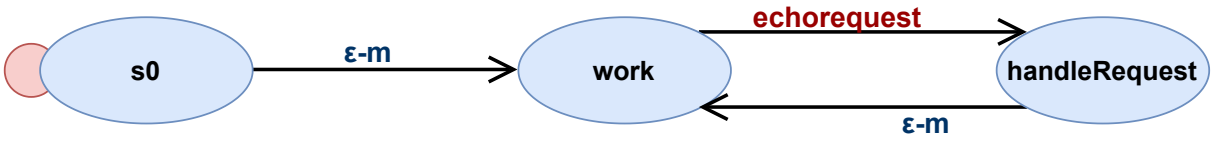

Figure 7: Behavior of the *echoactor*

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

```
/* BODIES OF THE STATES FOR echoactor ****************************** */
val s0Body : suspend QActorBasicFsm.()-> Unit =
 { println("started") }
val workBody : suspend QActorBasicFsm.() -> Unit =
 { println("idle") }
val handleRequestBody : suspend QActorBasicFsm.() -> Unit =
 { answer("echorequest", "echoreply", currentMsg.msgContent()) }

/* SYSTEM BUILDER ************************************************** */
val sysBuilder = SystemBuilder()

/* SYSTEM CREATION ************************************************* */
val system = runBlocking {
 sysBuilder.addHostname("localhost").addScope(this)
 //Context: "ctxecho"
 .newContext()
 .addContextName("ctxecho")
 .addContextAddress("localhost").addContextPort(9000)
 .addContextProtocol("TCP")
 //Actor: "echoactor"
 .newActorBasic().addActorName("echoactor")
 .upgrateToFsmBuilder().addQActorBasicFsm(QActorBasicFsm())
    //State: "s0"
 .newState().addStateName("s0").addStateBody(s0Body)
 .newTransition()
 .addEdgeName("t0").addTargetState("work")
 .buildEpsilonMove().buildState()
 .setInitialState("s0")
    //State: "work"
 .newState().addStateName("work").addStateBody(workBody)
 .newTransition()
 .addEdgeName("t1").addTargetState("handleRequest")
 .buildWhenRequest("echorequest").buildState()
    //State: "handleRequest
 .newState().addStateName("handleRequest")
 .addStateBody(handleRequestBody)
 .newTransition()
```

```
38   .addEdgeName("t2").addTargetState("work").buildEpsilonMove()
39   .buildState()
40   .buildInContext().second.buildInSystem().second.build()
41 }
```

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.