# QakActors24

Secondo Carl *Hewitt* -> (uno dei padri fondatori) il modello computazionale ad attori è stato ispirato, a differenza dei precedenti modelli di calcolo, dalla fisica, inclusa la relatività generale e la meccanica quantistica.

Proviamo a *chiedere a ChatGpt*

Vi è oggi una ampia gamma di proposte di linguaggi / librerie ad attori, tra cui:

- *Akka* ->: ispirato a *Modello computazionale ad attori* -> di Hewitt. Per le motivazioni si veda *Akka actors* ->.
- *GO* ->: ispirato a *CSP*->, propone *goroutine* e *CanaliGO*. Per la documentazione si veda *GO doc*->.
- *Kotlin Actors* -> : propone *croutines* e *channels* (si veda *Kotlin channel*->)

Un motto di riferimento alquanto significativo per questo modello è il seguente:

- **Do not communicate by sharing memory ...**
- **... instead, share memory by communicating.**

# QakActors24: Introduzione

La *Q/q* nella parola *QActor*, significa “quasi” poiché il linguaggio non è inteso come un linguaggio di programmazione generico, ma piuttosto un linguaggio di modellazione eseguibile, da utilizzare durante l’analisi del problema e il progetto di protitpi di sistemi distribuiti, i cui componenti sono attori che si comportano come un *Automa a stati finiti*, in stretta relazione con l’idea di sistemi basati su *Microservizi*.

L’aggiunta di *k* al prefisso (es **qak**, **Qak**) significa che stiamo facendo riferimento alla versione implementata in *Kotlin* ->, senza utilizzare i supporti Akka (come fatto nella prima versione del linguaggio).

Per una Introduzione all’uso di Kotlin si veda: si veda: *KotlinNotes*.

## Quadro generale

Un attore qak specializza la classe astratta *it.unibo.kactor.ActorBasicFsm.kt* che a sua volta specializza la classe astratta *it.unibo.kactor.ActorBasic.kt*, entrambe definite nella *Qak infrastructure*.

E’ possibile costruire un sistena software basato su attori qak semplicemente usando queste librerie; per un esempio, si veda **TODO**.

Tuttavia, l’uso della *Qak software factory* e del connesso *Linguaggio qak* rende lo sviluppo dei sistemi molto più rapido, comprensibile e gestibile.

## Qak software factory

Il *Linguaggio qak* è definito utilizzando il framework *Xtext* ->, che permette di costruire un insieme di

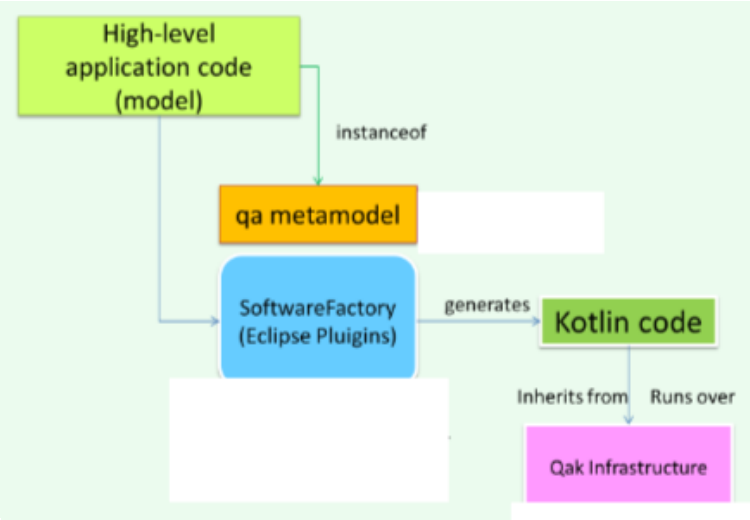
plugin per l'ecosistema Eclipse che, una volta installati, permettono ad un application designer de realizzare in tempi rapidi un modello eseguibile del sistema.

### I plugin della Qak factory

I plugin che, installati in Eclipse, realizzano la *Qak software factory* sono:

- **it.unibo.Qactork.ide_1.5.3.jar**
- **it.unibo.Qactork.ui_1.5.3.jar**
- **it.unibo.Qactork_1.5.3.jar**

Essi sono disponibili in: **issLab24/iss24Material/plugins**.




L'application designer usa l'editor guidato dalla sintassi per scrivere un *modello* del sistema che definisce struttura, interazione e comportamento di un sistema distribuito.

Il *modello* è una istanza de *Il metamodello Qak*, sulla base del quale è costruita la Factory.

Una volta salvato il modello, la factory produce *codice e risorse*.

---

## Qak codice e risorse generate

La *Qak software factory* costruisce vari prodotti indispensabili o utili, tra cui:

- un file che contiene la descrizione del sistema, in sintassi Prolog
- il file **build2024.gradle** e altre risorse
- il codice di raccordo con la *Qak infrastructure* (la *parte sommersa* di ogni sistema Qak)
- il codice Python per la produzione di una rappresentazione grafica del sistema

## Qak infrastructure

La libreria **unibo.qakactor23-5.0.jar** è prodotta nel progetto *unibo.qakactor23* e costituisce la *qak-infrastructure*, che si appoggia al supporto **unibo.basicomm23-1.0.jar** introdotto nel progetto *unibo.basicomm23*, che implementa il concetto astratto di *Interaction* per diversi protocolli (TCP, UDP, CoAP, etc.).

La classe **sysUtil** della infrastruttura offre un insieme di metodi di utilità:

### sysUtil for users

1. **curThread() : String**
2. **aboutThreads(info: String)**


1. **curThread() : String**
2. **aboutThreads(info: String)**

3. strRepToList( liststrRep: String ) : List<String>
4. strCleaned( s : String) : String
5. showOutput(proc: Process)
6. waitUser(prompt: String,tout: Long=2000)
7. createFilefname:String,dir:String ="logs" )
8. deleteFile( fname : String, dir : String )
9. updateLogfile(fname:String,msg:String,dir:String="logs")
10. getMqttEventTopic() : String

3. strRepToList( liststrRep: String ) : List<String>
4. strCleaned( s : String) : String
5. showOutput(proc: Process)
6. waitUser(prompt: String,tout: Long=2000)
7. createFilefname:String,dir:String ="logs" )
8. deleteFile( fname : String, dir : String )
9. updateLogfile(fname:String,msg:String,dir:String="logs")
10. getMqttEventTopic() : String

### sysUtil for kb

1. getPrologEngine() : Prolog
2. solve(goal:String, resVar:String):String?
3. loadTheory( path: String )
4. loadTheoryFromDistribution( path: String )

1. getPrologEngine() : Prolog
2. solve(goal:String, resVar:String):String?
3. loadTheory( path: String )
4. loadTheoryFromDistribution( path: String )

### sysUtil for system

1. getActorNames(ctxName:String):List<String>
2. getAllActorNames(ctxName: String) : List<String>
3. getAllActorNames( )
4. getNonlocalActorNames(ctx:String ):List<String>
5. getActor( actorName : String ) : ActorBasic?
6. getContext(ctxName : String) : QakContext?
7. getContextNames(): MutableSet<String>
8. getActorContextName(actorName:String):String?
9. getActorContext(actorName : String):QakContext?
10. getCtxCommonobjClass(ctxName:String): String

1. getActorNames(ctxName:String):List<String>
2. getAllActorNames(ctxName: String) : List<String>
3. getAllActorNames( )
4. getNonlocalActorNames(ctx:String ):List<String>
5. getActor( actorName : String ) : ActorBasic?
6. getContext(ctxName : String) : QakContext?
7. getContextNames(): MutableSet<String>
8. getActorContextName(actorName:String):String?
9. getActorContext(actorName : String):QakContext?
10. getCtxCommonobjClass(ctxName:String): String

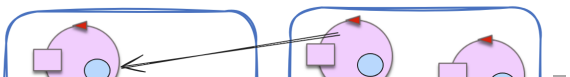
# Il metamodello Qak

Il *Linguaggio qak* reso disponibile dalla *Qak software factory* intende fornire un linguaggio per la definizione di modelli eseguibili di un sistema, basati su un insieme di concetti volti a cattuare l'idea che un sistema software (distribuito):

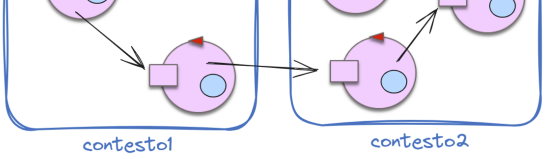
- è formato da una insieme di attori che si comportano come *Automi a stati finiti*
- che interagiscono scambiandosi messaggi
- raggruppati in contesti che li abilitano a interazioni via rete
- contesti che possono essere allocati (deployed) su uno o più nodi computazionali

## QakActors24: il sistema

Un sistema ad attori qak è composto da una collezioni di attori, attivati in uno o più contesti, allocati in uno o piò



nodi di elaborazione.




---

Un sistema ad attori qak è configurato in modo automatico a partire da una descrizione espressa in forma di *base di conoscenza*, in sintassi Prolog.

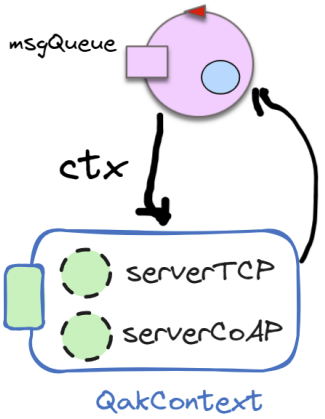
```
context(ctx1, "localhost",  "TCP", "8923").
context(ctx2, "localhost",  "TCP", "8925").
qactor( producer1, ctx1, <className>).
qactor( consumer,  ctx2, <className>).
qactor( producer2, ctx3, <className>).
...
```

---

## QakActors24: l'attore

Un attore qak è un componente attivo che:

- nasce, vive e muore in un *contesto* che può essere comune a (molti) altri attori;
- ha un **nome univoco** nell'ambito di tutto il sistema;
- è logicamente attivo, cioè dotato di flusso di controllo autonomo;
- è capace di inviare messaggi ad un altro attore, di cui conosce il **nome**, incluso sè stesso;
- è capace di eseguire elaborazioni autonome e/o elaborazioni di messaggi;
- è dotato di una sua coda locale (**msgQueue**) in cui sono depositati i messaggi a lui inviati




- Elabora i messaggi secondo quanto riportato in *La gestione dei messaggi*.

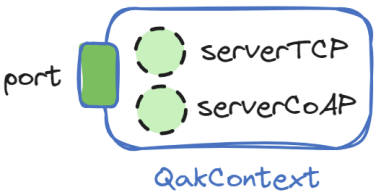
---

## QakActors24: il contesto

Un contesto è un componente software che gestisce **N>0** actor qak, **abilitandoli** alla ricezione e trasmissione di messaggi via rete.

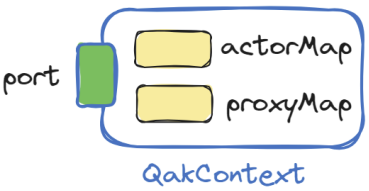
Un contesto rappresenta un nodo logico di elaborazione dotato di un server e di un porta di ingresso, su cui altri contesti possono stabilire una *Interconnessione*, di solito basata su **TCP**, **CoAP** e **MQTT**.

Un contesto deve essere allocato su un computer fisico o su un virtual macìhine / container.




---

Un contesto mantiene una tabella (*actorMap*) con i riferimento agli attori locali e una tabella (*proxyMap*) con i riferimenti ai Proxy che mantengono una *Interconnessione* con gli altri contesti del sistema.

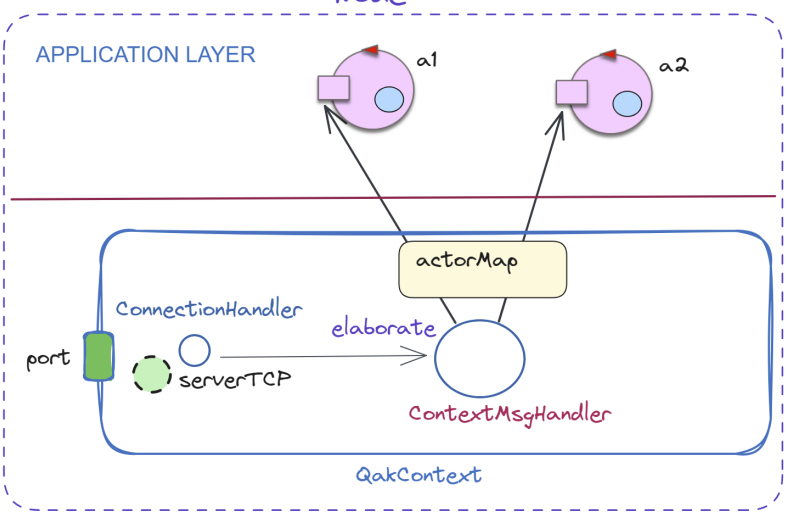



---

Il Server di contesto depone i mes-

saggi *IApplMessage* ricevuti su una *Interconnessione* sulla **msgQueue** dell'attore destinatario.

Per questo scopo, il Sever si avvale di un unico *gestore di messaggi di sistema*: il **ContextMsgHandler**.




La figura mostra il caso di attori locali ad un nodo di elaborazione che possono inviare/ricevere messaggi tra loro oppure elaborare messaggi inviati da componenti remoti.

---

# Linguaggio qak

Il linguaggio si pone nel solco dei *Domain Specific Languages* e permette di esprimere un insieme dei concetti che forma *Il metamodello Qak*.

Il ruolo 'strategico' dei linguaggi in informatica si comprende subito considerando che **ogni computazione** (ogni sistema software) può essere espressa usando un insieme molto limitato di 'mosse' (istruzioni) studiate dalla teoria come *macchine astratte elementari*.

In sintesi, possiamo dire che l'uso du un linguaggio comporta descrizioni di un sistema software:

più compatte, più esplicie e semanticamente più ricche

Il linguaggio qak intende promuovere la definizione in tempi brevi di *prototipi di sistemi distribuiti*, utilizzabili nelle fasi preliminari di un progetto di sviluppo software, al fine di **interagire con il committente**, per chiarire e stabilizzare i requisiti.

In molti casi, la formalizzazione dei requsiti e della analisi del problema in ternini di modelli eseguibili qak costituisce anche un passo pragmaticamente utile per la costruzione effettiva del prodotto finale.

## Qak syntax

La sintassi del linguaggio è riportata in *Qak syntax*.

### Scopo della grammatica

Lo 'scopo' della grammatica è la produzione relativa alla specifica del sistema.

```
QActorSystemSpec:
  name=ID                              //(1)
  ( mqttBroker = BrokerSpec)?          //(2)
  ( libs       = UserLibs  )?          //(3)
```

**Specifica del sistema**

1. nome del sistema

```
( message   += Message   )*          //(4)
( context   += Context   )*          //(5)
( actor     += QActorDeclaration )* //(6)
(display    = DisplayDecl)?          //(7)
(facade     = FacadeDecl)?           //(8)
```

( ... )? signitifca **opzionale**

( ... )* signitifca **zero o più volte**

2. indirizzo di un broker MQTT
3. dichiarazione di una o più librerie applicative
4. *Dichiarazione dei messaggi*
5. *Dichiarazione dei contesti*
6. *Dichiarazione degli attori*
7. Dichiarazione di un Display di sistema
8. Dichiarazione di una Facade di sistema

---

Le regole sintattche del linguaggio impongono che un modello Qak venga definito organizzando la sua descrizione in una **sequenza di dichiarazioni**.

## Dichiarazione dei messaggi

I diversi *Tipi di messaggi* sono dichiarati usando una *sintassi* Prolog-like (si veda *tuProlog* ->):

```
Message      :  BasicMessage | Event | OtherMsg;
BasicMessage :  Dispatch | Request;
OtherMsg     :  Reply;

Event:    "Event"     name=ID  ":" msg = PHead  (cmt=STRING)?;
Dispatch: "Dispatch"  name=ID  ":" msg = PHead  (cmt=STRING)?;
Request:  "Request"   name=ID  ":" msg = PHead  (cmt=STRING)?;
Reply:    "Reply"     name=ID  ":" msg = PHead  ( "for" reqqq = [Request] )? (cmt=STRING)?;

PHead :       PAtom | PStruct | PStructRef ;  //sintassi Prolog
...
```

## Dichiarazione dei contesti

```
Context : "Context"  name=ID "ip"  ip = ComponentIP   ( "commonObj" commonObj = STRING)?  ;
ComponentIP : {ComponentIP} "[" "host=" host=STRING "port=" port=INT "]" ;
```

Un contesto può introdurre un oggetto accessibile a tutti gli attori.

## Dichiarazione degli attori

```
QActorDeclaration: QActorInternal |
                   QActorExternal;

QActorInternal:  QActor |
                 QActorCoded;
```

La sintassi indica che vi sono tre tipi di attori.

1. *Attori normali*
2. *Attori coded*
3. *Attori external*

---

Gli attori possono inoltre comportarsi come generatori di stream

### Attori normali

Gli attori 'normali' sono descritti come *Automi a stati finiti*.

1. Nome dell'attore
2. Riferimento al Contesto
3. Oggetto locale usato dall'attore
4. Attore creato solo dinamicamente
5. Libreirie importate

```
QActor: "QActor"
/*1*/ name=ID
/*2*/"context" context = [ Context ]
/*3*/("withobj" withobj = WithObject)?
/*4*/( dynamic ?= "dynamicOnly")?
  "{"
/*5*/( imports += UserImport )*
/*6*/( start   = AnyAction )?
/*7*/( states += State )*
  "}";
```

6. Azioni iniziali dell'attore
7. Stati dell'attore
8. Dichiarazione dell'oggetto locale
9. Dichiarazione delle librerie importate

```
/*8*/WithObject: name=ID
       "using" method=STRING;

/*9*/UserImport:
    "import" file=STRING ;
```

Si veda *AnyAction*

---

### Stati di un attore normale

1. Nome dello stato
2. Stato iniziale. Il tag **initial** deve essere presente in un **unico stato**.
3. Azioni locali allo stato
4. Transizioni verso lo stato futuro

```
/*1*/State: "State" name=ID
/*2*/( normal?="initial")?
  "{"
/*3*/( actions += StateAction )*
  "}"
/*4*/( transition = Transition )?
```

---

### Attori coded

Un **CodedQActor** è un attore scritto direttamente in codice (kotlin, Java o altro) che si comporta come fli attori qak.

```
/*1*/QActorCoded : "CodedQActor"    name=ID
    "context" context = [ Context ]
    "classname" className = STRING
    ( dynamic ?= "dynamicOnly")?;
```

**Es->** *democodedqactor.qak*

---

### Attori external

Un attore dichiearato **external** è un attore cha fa parte del sistema ma senza essere definito nel modello corrente, in quanto parte di un altro contesto.

```
/*1*/QActorExternal: "ExternalQActor" name=ID
    "context" context = [ Context ] ;
```

**Es->** *demoaddtocore.qak*

---

### Attori streamer

La *Reactive programming* è una combinazione di idee riconducibili al modello Observer, al modello Iterator e al modello di programmazione funzionale.

In questo stile di programmazione, un servizio-consumatore reagisce ai dati non appena arrivano, con la capacità anche di propagare le modifiche come eventi agli osservatori registrati.

Un sistema Qak può essere impostato seguendo questo stile programmazione emettendo eventi usando la primitiva *subscribeTo*. Per un esempio si veda: **Es->** *demostreams.qak*

## Transizioni di stato

```
Transition          :  EmptyTransition | NonEmptyTransition  ;
```

La transizone da uno stato a uno stato successivo può avvenire senza attesa di alcun messaggio (*EmptyTransition*) oppure (*NonEmptyTransition*) in relazione alla disponibilità di un messaggio tra

quelli definiti in *Dichiarazione dei messaggi*.

## EmptyTransition

```
EmptyTransition:
/*1*/    "Goto" targetState=[State]
/*2*/    ("if" eguard=AnyAction
/*3*/    "else" othertargetState=[State] )?  ;
```

1. Riferimento allo stato futuro (nel caso di guardia **true**)
2. Specifica (opzionale) di una *Guardia*
3. Stato futuro nel caso di *Guardia* **false**

**Es->** *demoguards.qak*.

---

## Guardia

- Una guardia è una espressione scritta in Kotlin, che può essere valutata come **true** o **false**.
- Una transizione associata a una guardia, viene attivata solo se la valutazione della condizione espressa dalla guardia produce il valore **true**.

## NonEmptyTransition

```
NonEmptyTransition:
/*1*/  "Transition" name=ID
/*2*/  (duration=Timeout)?
/*3*/  (trans+=InputTransition)*
```

1. Nome della transizione
2. Specifica di un tempo massimo di attesa: si veda *Timeout per transizioni*
3. Transizione relativa a un messaggio

**Es->** *demoguards.qak*.

---

Una *NonEmptyTransition* associata alla disponibilità di un messaggio distingue tra i diversi *tipi di messaggio*:

```
InputTransition    : MsgTransSwitch | RequestTransSwitch |  ReplyTransSwitch |
                     EventTransSwitch | InterruptTranSwitch ;

MsgTransSwitch     : "whenMsg"     message=[Dispatch]
                      ("and"  guard=AnyAction  )?  "->"  targetState=[State]  ;
RequestTransSwitch : "whenRequest" message=[Request]
                      ("and"  guard=AnyAction  )?  "->"  targetState=[State]  ;
ReplyTransSwitch   : "whenReply"   message=[Reply]
                      ("and"  guard=AnyAction  )?  "->"  targetState=[State]  ;
EventTransSwitch   : "whenEvent"   message=[Event]
                      ("and"  guard=AnyAction  )?  "->"  targetState=[State]  ;
InterruptTranSwitch: "whenInterrupt"   message=[Dispatch]
                      ("and"  guard=AnyAction  )?  "->"  targetState=[State]  ;
```

## whenInterrupt

Esegue la transzione da uno stato **SA** a uno stato **SB**, con ritorno allo stato **SA**, quando **SB** esegue l'istruzione qak *returnFromInterrupt*.

**Es->** *demointerrupt.qak*.

## Timeout per transizioni

Per evitare una attesa indefinita di messaggi in uno stato, è poosibile associare alla transizione un *timeout* (come un numero naturale in *msec*) scaduto il quale l'automa transita nello stato specificato.

```
Timeout             : TimeoutInt | TimeoutVar | TimeoutSol | TimeoutVarRef;
TimeoutInt          : "whenTime"     msec=INT   "->" targetState = [State]  ;
TimeoutVar          : "whenTimeVar"     variable    = Variable     "->" targetState = [State]  ;
TimeoutVarRef       : "whenTimeVarRef" refvar      = VarRef       "->" targetState = [State]  ;
TimeoutSol          : "whenTimeSol"     refsoltime = VarSolRef    "->" targetState = [State]  ;
```

Lo scadere del tempo indicato in *whenTime* (regola **TimeoutInt**) provoca l'emissione di un **evento**, con indentificatore **local_tout_actorname_state** ove **actorname** è il nome dell'attore e **state** è il nome dello stato corrente. **Es->** *demo0 perceiver*

Le forme che si aggiungono a **TimeoutInt** sono utili in situazioni in cui il tempo non sia noto a priori, ma derivi da elaborazioni. **Es->** *demo0 sender*

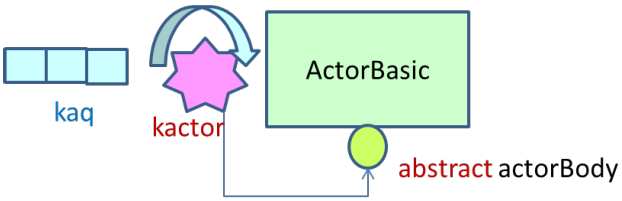
# Comportamento di un attore

Prima di illustrare cosa un attore qak può fare, è importante sottolienare che:

- un attore qak non dispone di una operazione receive bloccante

La ragione è dovuta al comportamento *message-driven* dell'attore.

Infatti, il **comportamento di base** di un attore qak è definito dalla classe *it.unibo.kactor.ActorBasic.kt* che gestisce i messaggi disponibili sulla *msgQueue dell'attore* in modo FIFO; l'attore qak di base opera quindi in modo **message-driven** utilizzando un *canale Kotlin*.




L'attore che specializza *it.unibo.kactor.ActorBasicFsm.kt* opera invece come un **automa di Moore** *a stati finiti* , gestendo i messaggi ricevuti sul *canale Kotlin* ereditato da *ActorBasic* non in modo FIFO, ma in funzione dello suo stato corrente.

Un attore qak ha un comportamento autonomo e quindi, una volta attivato dalla *Qak infrastructure* con un messaggio iniziale di 'start', può eseguire azioni anche ignorando eventuali altri messaggi sulla sua coda di input.

Normalmente però, un attore qak può entrare in specifici stati elaborativi se sulla sua coda di input sono presenti messaggi di un certo tipo. Vediamo come.

## La gestione dei messaggi

1. ogni attore possiede, oltre alla coda dei messaggi principale *msgQueue* , una seconda coda msgQueueStore, in cui memorizza messaggi (di tipo **request** e **dispatch**) non elaborati;
2. uno stato è di norma associato a un insieme di transizioni **TSET**, ciascuna delle quali speciifca lo stato futuro, in corrispondenza a un messaggio con uno specifico identificatore **msgId**;
3. al termine delle sue azioni, lo stato corrente dell'attore qak consulta, **nell'ordine**, le sue code

**msgQueueStore** e **msgQueue**, ciascuna in modo FIFO;

4. se l'identificatore del messaggio prelevato da una coda è uguale al **msgId** di una qualche transizione in **TSET**, quella transizione è attivabile; in caso contrario, il messaggio 'esaminato' viene lasciato dove è se era nella coda **msgQueueStore**, oppure, se è un messaggio di tipo **request** o **dispatch** prelevato dalla coda principale **msgQueue**, viene depositato in fondo alla coda **msgQueueStore**.

   Un messaggio di tipo event il cui identificatore non compare in **TSET**, viene **scartato** (e quindi ignorato e dimenticato);

5. appena lo stato corrente trova una transizione attivabile,; passa il controllo allo stato futuro specificato da questa transizione;

6. se nessuna transizione è attivabile, l'attore qak rimane nello stato corrente; all'arrivo di un nuovo messaggio, si riprende ad eseguire il punto **3**.

## Variabili e riferimenti

```
Variable:  varName= VARID ;

//USING vars (from solve or from code)
VarRef       : "$" varName= VARID ;
VarRefInStr  : "#" varName= VARID ;
VarSolRef    : "@" varName= VARID ;
```

I simboli **$ # @** sono *Notazioni Shortcut* per l'accesso al valore di variabili i cui nomi iniziano con una **lettera MAIUSCOLA**

---

### Notazioni Shortcut

Notazione **$** Kotlin-like per accesso al valore di una variabile entro una String

```
VarRef : "$" varName= VARID ;
```

```
// Esempio:
[# var N = 0 #]        //Instruzione Kotlin
println("Valore di N=$N")   //Frase qak
```

---

Notazione **#** per accesso al valore in forma di String al vslore di una variabile della soluzione di una dimostrazione logica

```
VarRefInStr : "#" varName= VARID ;
```

```
//Equivale a:
${ getCurSol("<VARID>").toString() }

// Esempio:
solve(move(M));println( #M )
```

---

Notazione **@** per accesso al vslore di una variabile della soluzione di una dimostrazione logica

```
VarSolRef    : "@" varName= VARID ;
```

```
//Equivale a:
getCurSol("<VARID>").toString()

// Esempio:
solve(move(M)); doMove( @M )
```

---

## Azioni di un attore

Una volta entrato in un particolare stato computazionale, un attore qak può eseguire una sequenza di azioni di 'alto livello' espresse in linguaggio qak oppure di 'basso livello' espresse direttemante in

Kotlin.

```
StateAction:
/*1*/ AnyAction |
/*2*/ Forward|Demand|Answer|ReplyReq|AutoMsg|AutoRequest|
/*3*/ MsgCond | GuardedStateAction |     IfSolvedAction |
/*4*/ MqttConnect | Publish | Subscribe | SubscribeTopic|
/*5*/ Emit |  EmitLocal |  EmitLocalStream |
/*6*/ UpdateResource |  ObserveResource  |
/*7*/ Delegate |  DelegateCurrent        |
/*8*/ SolveGoal |
/*9*/ CreateQActor | ExecResult |
/*10*/ ReturnFromInterrupt |
/*11*/ CodeRunSimple | CodeRunActor | MachineExec |
/*12*/ Print | PrintCurMsg | DiscardMsg |
/*13*/  DelayInt  | MemoTime | Duration |
/*14*/ EndActor |
```



---

## AnyAction

Le azioni esprimibili nel linguaggio qak non danno un linguaggio computazionalemte completo.

Pertanto, volendo rendere eseguibile un modello qak, si introduce la possibilità che un attore qak possa esprimere una qualunque sequenza di azioni scritte in Kotlin.

```
AnyAction : "[" body=KCODE "]"  ;

terminal KCODE  : '#' ( . )* '#'  ;
```

**[# ... #]**: Specifica di codice Kotlin.

---

## Operazioni di messaggista punto a punto

Le operazioni di invio messaggio sono le seguenti:

```
Forward: "forward" dest=[QActorDeclaration]
          "-m" msgref=[Dispatch] ":" val=PHead;
```

**forward**: Invio di Dispatch. **Es->** *demo0.qak*

---

```
Demand: "request" dest=[QActorDeclaration]
          "-m" msgref=[Request] ":" val=PHead;
```

**request**. Invio di Request. **Es->** *demorequest caller*

---

```
Answer: "replyTo" reqref=[Request]
       ("ofsender" sender=VarRef)?
       "with"  msgref=[Reply]
       ":" val=PHead
       ("caller==" dest=[QActorDeclaration])?;
```

**replyTo**. Invio di Reply a una Request. **Es->** *demorequest called*

---

```
ReplyReq  : "ask"  reqref=[Request]
  ":" val = PHead
  "forrequest" msgref=[Request]
```

**ask**. Invio di Request a un attore che ha fatto una Request. **Es->** *demoasktocaller caller*

```
  ( "caller==" dest=[QActorDeclaration])?;
```

---

```
AutoMsg: "autodispatch" msgref=[Dispatch]
  ":" val = PHead ;

AutoRequest: "autorequest"  msgref=[Request]
  ":" val = PHead ;
```

**autodispatch**. Invio di Dispatch di un attore a sè stesso.

**autorequest**. Invio di Request di un attore a sè stesso.

---

## Accesso al contenuto dei messaggi

```
MsgCond: "onMsg" "(" message=[Message]
     ":" msg = PHead ")"
     "{" ( condactions += StateAction )*  "}"
     ("else" ifnot = NoMsgCond )? ;

NoMsgCond:
"{" (  notcondactions += StateAction )*  "}";
```

**onMsg**: esegue il body *condactions* solo se il *messaggio corrente* ha msgId di <*Message*> e può essere **unificato in Prolog** con il template di messaggio definito nella dichiarazione **e** con il template <*msg*> specificato in *onMsg*.

**Es->** *QActor demo0*

---

## Azioni condizionali

```
GuardedStateAction :
  "if" guard = AnyAction "{"
   ( okactions += StateAction )*
  "}"
                                ("else" "{"
   ( koactions += StateAction )*
   "}" )?;
```

**if else**

---

```
IfSolvedAction: "ifSolved" "{"
 ( solvedactions += StateAction )*  "}"
                               ("else"  "{"
  ( notsolvedactions += StateAction )*
  "}")?;
```

**ifSolved**

---

Supponiamo di avere un messaggio dichiarato come segue:

```
Dispatch m : m(X,Y,Z)
```

## payloadArg

Lo stato relativo alla elaborazione di tale messaggio potrebbe voler accedere a un argomento specifico del suo payload.

In tal caso si può usare la primitiva *onMsg* e la funzione **payloadArg(N)**:

```
onMsg( m : m(X,Y,Z) ){
  println("$payloadArg(1)")  //stampa Y
}
```

*payloadArg(N)*

Restituisce l'argomento di ordine N (convertito in String) del payload di un messaggio.

---

## Operazioni di messaggista publisch-subscribe

```
MqttConnect: "connectToMqttBroker" brokerAddr=STRING;
```

**connectToMqttBroker**. Connessione a Broker MQTT.

---

```
Publish: "publish"   topic=STRING
           "-m" msgref=[Event] ":" val=PHead;
```

**publish**. Pubblicazione su topic MQTT.

---

```
SubscribeTopic: "subscribe" topic=STRING;
```

**subscribe**. Sottoscrizione a topic MQTT.

---

```
Subscribe : "subscribeTo" localactor=[QActor]
             ("_" suffix=STRING)? "for" event=[Event];;
```

**subscribeTo**. Sottoscrizione a eventi emessi con la primitiva *emitlocalstream*. Si veda *Attori streamer*.

---

## Operazioni relative agli eventi

```
Emit: "emit" msgref=[Event] ":" val=PHead;
```

**emit**. Emissione di un evento globale. **Es->** *demo0.qak*

---

```
EmitDelayed: "emitdelayed" msgref=[Event]
              ":" val=PHead
             delay=Delay;
```

**emitdelayed**. Emissione di un evento dopo un dato tempo.

---

```
EmitLocal: "emitlocal"
               msgref=[Event] ":" val=PHead;
```

**emitlocal**. Emissione di un evento locale .

---

```
EmitLocalStream: "emitlocalstream"
        msgref=[Event] ":" val = PHead;
```

**EmitLocalStream**. Emissione di un evento stream. **Es->** *demostreams.qak*

---

## Operazioni relative alla osservabilità

```
UpdateResource: "updateResource"   val=AnyAction;
```

**updateResource**. **Es->** *helloworld4*.

---

```
ObserveResource: "observeResource"
        resource=[QActorDeclaration]
        ("_" suffix=STRING)?
        ("msgid" msgid=[Dispatch] )?;
```

**observeResource**. **Es->** *helloworld4*.

---

- L'informazione emessa un *observable* mediante **updateResource** sono gestite dalla *Qak infra-structure* inviando a ciascun *observer* il dispatch che l'*observer* stesso ha dichiarato (campo opzionale **msgid**) di voler usare per ricevere l'informazione.
- Se l'*observer* non dichiara **alcun dispatch**,** il nome usato dalla *Qak infrastructure* è **coapinfo**.

## Operazioni di delegazione

```
Delegate : "delegate"
```

**delegate**

```
msg=[BasicMessage] "to" localactor=[QActor];
```

Delega la gestione del *BasicMessage* a un attore locale.

**Es->** *demodelegate.qak*.

---

**delegateCurrentMsgTo**.

```
DelegateCurrent:
   "delegateCurrentMsgTo" localactor=[QActor];
```

Delega la gestione del *BasicMessage* a un attore locale creato dinamicamente.

**Es->** *democreate.qak*

---

## Operazioni per le basi di conoscenza

Dimostrazione goal Prolog

- **solve**.

Si veda *PrologOps* (html)

---

## Creazione dinamica di attori

**create**:

```
CreateQActor: "create"
/*1*/  executor=[QActorDeclaration]
/*2*/  ("_" suffix=STRING)?
/*3*/  (confined="confined")?
  (outinforeply=OutInforReply)?
```

1. Riferimento all'*attore* da creare
2. Suffisso opzionale per il nome dell'attore creato
3. Attore creato attivato in modo confinato (vedi *confined*)

**Es->** *democreate.qak*

---

```
OutInforReply: "requestbycreator"
  msgref=[Request]  ":" val = PHead ;
```

**requestbycreator**: richiesta inviata all'attore creato

---

```
ExecResult: "execresultReplyTo"
  reqref=[Request]  "with"  msgref=[Reply]
  ":" val = PHead ;
```

**execresultReplyTo**: invio di Reply a una Request associata alla creazione di un Attore Questa operazione è superata dalla primitiva *delegateCurrentMsgTo*.

---

## Operazioni di ritorno da interruzione

```
ReturnFromInterrupt:
    "returnFromInterrupt" memo=STRING?;
```

- restituisce il controllo allo stato precedente (interrotto), senza eseguirne le azioni, ma solo le transizioni.
- interrupt innestati non sono supportati

**Es->** *demointerrupt.qak*

---

## Operazioni per esecuzione di codice

```
CodeRunSimple : "run" bitem=QualifiedName
   "("
      (args+=PHead ("," args+=PHead)* )?
```

**run**: **run ccc.xxx()**

invoca il metodo *static* **xxx** della classe **ccc**.

```
  ")";
```

Esecuzione codice esterno

---

```
MachineExec: "machineExec" action=STRING ;
```

**machineExec**: **machineExec(cmd:string)**

Esecuzione codice di sistema locale

---

```
CodeRunActor: "qrun"
  aitem=QualifiedName
  "("
      "myself" ( "," args+=PHead ("," args+=PHead)* )?
  ")";
```

**qrun** : **qrun ccc.xxx()**

invoca il metodo *static* **xxx** della classe **ccc**. Il metodo deve avere come primo argomento un riferimento all'attore corrente (**myself**).

---

## Operazioni con il tempo

```
Delay: DelayInt | DelayVar | DelayVref | DelaySol ;

DelayInt : "delay" time=INT ;
```

**delay**

---

```
MemoTime: "memoCurrentTime" store=VARID ;
```

**memoCurrentTime** **Es->** *corecaller* in *demoaddto-core.qak*

---

```
Duration: "setDuration"
     store=VARID "from" start=VARID;
```

**setDuration**

---

## Operazioni di terminazione

```
EndActor: "terminate" arg=INT;
```

**terminate**

---

## Operazioni di utilità

```
PrintCurMsg: "printCurrentMessage"
          ("color" color=PCOLOR )? ;
```

**printCurrentMessage**

---

```
Print:"println"
  "(" args=PHead ")"
   ("color" color=PCOLOR )?;
```

**println**

---

# Parti ereditate

Ogni attore è una specializzazione della classe **it.unibo.kactor.ActorBasicFsm** (che specializza **it.unibo.kactor.ActorBasic** del progetto *unibo.qakactor23*) da cui eredita un insieme di variabili e operazioni, tra cui quelle qui di seguito riportate.

## Variabili interne importanti

| | |
|---|---|
| Sostituto di *this* | • **myself**. |

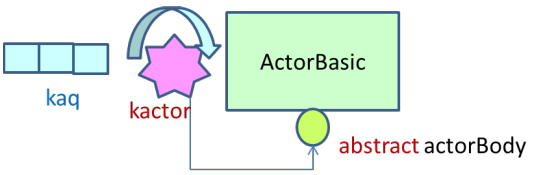
| | |
|---|---|
| Riferimento allo stato corrente | • **currentState**. |
| Riferimento al messaggio corrente | • **currentMsg**. |
| Memorizzazione messaggi non attesi | • **discardMsg On/Off**. **Es->** *QActor demo0* |

# Note sulla implementazione

## it.unibo.kactor.ActorBasic.kt

Realizza il concetto di un ente computazionale dotato di flusso di controllo autonomo, capace di ricevere e gestire messaggi in modo FIFO, sfruttando un *Kotlin actor* incapsulato:

```
/*1*/ abstract class  ActorBasic(
/*2*/    name: String,
/*3*/    val scope:CoroutineScope=GlobalScope,
/*4*/    var discardMessages Boolean=false,
/*5*/    val confined :     Boolean = false,
/*6*/    val ioBound :      Boolean = false,
/*7*/    val channelSize : Int = 50
       ) :
/*8*/      CoapResource(name),
/*9*/       MqttCallback {
      ...
    //To be overridden by the application
/*10*/ abstract suspend fun actorBody(
           msg:IApplMessage)
  }
```




1. **class ActorBasic** Si veda *Oggetti e classi* in *KotlinNotes*.
2. **name** Nome (**univoco** nel sistema) dell'attore
3. **scope** Si veda *Le coroutines* in *KotlinNotes* e *kotlinUniboCoroutinesIntro* in *kotlinUnibo*.
4. **discardMessages** scarta o meno i messaggi non attesi. Usato principalmente in *ActorBasicFsm*
5. **confined** Si veda *Confinamento* in *KotlinNotes*.
6. **ioBound** Si veda *Confinamento* in *KotlinNotes*.
7. **channelSize** Si veda *I canali* in *KotlinNotes*.
8. **CoapResource** Si veda *Estende CoapResource*
9. **MqttCallback** Si veda *Implementa MqttCallback*
10. **actorBody** codice per la gestione dei messaggi *IApplMessage* ricevuti dall'attore.

Si veda: *actor channel*

La notazione:

```
class  ActorBasic( ... ) : CoapResource(name), MqttCallback
```

esprime in forma compatta che *ActorBasic* eredita dalla classe *CoapResource* e implementa l'interfaccia *MqttCallback* (si veda *kotlinInheritance*).

## Estende CoapResource

Ogni attore è anche una risorsa CoAP, specializzazione della classe definita nella libreria *https://www.eclipse.org/californium/*.

## Implementa MqttCallback

Ogni attore implementa anche l'interfaccia **org.eclipse.paho.client.mqttv3.MqttCallback**. Pertanto ogni attore può gestire notifiche emesse da un MQTT client, attraverso il metodo **messageArrived**. (TODO REF)

## actor channel

```
val actor = scope.actor<IApplMessage>( dispatcher, capacity=channelSize ) {
  for( msg in channel ) {
                       if( msg.msgContent() == "stopTheActor") {  channel.close() }
    else actorBody( msg )
  }
}
```

Si veda: *Kotlin actor* in *KotlinNotes*.

## sendMessageToActor

Il metodo **sendMessageToActor** realizza l'invio di un messaggio ad un attore di cui è noto il nome o la connessione.

```
suspend fun sendMessageToActor(msg : IApplMessage,
       destName: String, conn : Interaction? = null ) {
  //realizza l'invio di msg all'attore di nome destName
  //usando conn se conn!=null (destname è un 'alieno')
  val destactor = context!!.hasActor(destName)
  /*
    se destactor  è locale: destactor.kactor.send( msg )
    altrimenti usa il proxy verso il contesto di destactor
}
```

## it.unibo.kactor.ActorBasicFsm.kt

```
abstract class ActorBasicFsm(  qafsmname:  String,
                        fsmscope: CoroutineScope = GlobalScope,
              discardMessages : Boolean = false,
                        confined :     Boolean = false,
                        ioBound :      Boolean = false,
                        channelSize : Int = 50
): ActorBasic(qafsmname,fsmscope,discardMessages,confined,ioBound,channelSize) { ... }
```

- Un attore che specializza questa classe opera come un automa a stati finiti.
- Il codice Kotlin viene generato dalla *Qak software factory*
- I messaggi ricevuti sul canale Kotlin (ereditato da *ActorBasic*) sono gestiti in relazione alle specifiche sulle transizioni associate allo stato corrrente dell'automa.

NEXT: *QakActors24Demo*