GLM: Analisi del problema

Impostare il gioco come il risultato delle attività di celle logicamente disposte in una griglia, che possono interagire solo con scambio di messaggi, implica dover affrontare nuovi problemi. E' quindi

necessaria una accurata fase di analisi

per individuare e discutere i nuovi problemi che si pongono.

Iniziamo ricordando che disponiamo già di un *Actor* (*griddisplay*) con cui rendere visibile su una GUI una rappresentazione dello stato corrente di una cella.

Abbiamo anche stabilito (regola *R_1*) che il dispositivo *griddisplay* deve fornire solo funzioni di I/O, lasciando a *gamelife* la responsabilità della logica applicativa.

Problematiche

P_1

- La transizione dello stato di una cella (da **true** a **false** o viceversa) implica che la cella debba acquisire conoscenza sullo stato corrente delle sue celle adiacenti.

  P_1) come fa una cella a conoscere quali sono le sue celle adiacenti in assenza di una struttura globale?

  Abbiamo già osservato che la *Denominazione delle celle* implica che una cella, pur non avendo riferimenti concreti (puntatori o altro) alle celle vicine, può *conosere i nomi delle sue celle logicamente vicine*, restituiti dalla funzione *genNeighborsNames*.

P_2

- L'assenza di una matrice globale di celle, come *grid* del *caso concentrato* implica che una cella possa venire a conoscenza dello stato corrente di una sua vicina solo attraverso una comunicazione esplicita a messaggi.

P_2) quale è la forma di comunicazione più opportuna tra le celle?

Strategie di comunicazione tra le celle

Si possono individuare diverse strategie. Una cella **C**, per *conoscere lo stato* di una sua cella adicente **A**, può:

1. inviare ad **A** una *request* per conoscere il valore corrente dello stato di **A**, che si impegna a inviare tale valore come risposta (*reply*) a **C**;
2. elaborare un *dispatch* inviato da **A** a **C**, con il valore del suo stato corrente;
3. operare come un *observer* della cella **A**. Ad ogni cambio di stato, **A** *emette informazione di aggiornamento* a favore di a **C** e di tutte le altre celle-observer (quelle a lei logicamante adiacenti).

Se una cella opera come un ente osservabile, l'informazione di aggiornamento può essere propagata in modi diversi:

Interazione a eventi

la cella **A** emette un *event generico*, che viene propagato a tutti i componenti del sistema.

Interazione a eventi-stream

la cella **A** emette un *evento-stream*, che viene propagato solo alle celle (locali, vicine) che si sono sottoscritte a **C** mediante la primitiva **qak** *subscribeTo*.

Interazione publish-subscribe

la cella **A** *pubblica* l'informazione su un *Message Broker*, usando ad esempio un protocollo *publish-subscribe* come *MQTT*

A queste forme aggiungiamo, per completare l'elenco, la già menzionata:

Interazione a dispatch

la cella **A** invia a ogni sua cella vicina **C** un *dispatch* con il valore del suo stato corrente

Ciascuna di queste forme di interazione ha i suoi pro e contro ed andrebbe sperimentata concretamente. Noi procederemo in questo modo:

- Escludiamo la *Interazione a eventi* generici, in quanto un evento **qak** viene propagato a tutti gli attori del sistema (escluso l'emettitore); quindi la riteniamo inutilmente onerosa.
- Lasciamo la interazione tramite **request** a una discussione finale.

Per quanto riguarda le rimanenti forme di interazione tra le celle, osserviamo che. concettualmente, sono forme diverse di realizzazione di una unica operazione logica: l'invio di informazione alle celle vicine. Questa operazione logica può essere espressa da una nuova funzione.

sendToNeighbors: operazione logica di alto livello (definita nel file *GMLSupport.kt*), con cui una cella può inviare il valore del suo stato corrente alle celle vicine.

Pianificazione degli ulteriori Sprint di sviluppo

1. Iniziamo pensando che tutte le celle siano locali a uno stesso **Context** e che possano quindi usare la *Interazione a eventi-stream*. Questa forma verrà sperimentata nello *Sprint3*
2. Visto che l'ipotesi di località non è ammissibile se il sistema è distribuito su più nodi computazionali, sperimentiamo nello *Sprint4* una *Interazione publish-subscribe* che dovrebbe costituire una evoluzione semplice della versione precedente.
3. Per verificare come l'uso dei modelli di alto livello faciliti la modifica degli schemi di interazione, adotteremo una *Interazione a dispatch* nello *Sprint5*.

La funzione *sendToNeighbors* può quindi essere impostata come segue:

sendToNeighbors

La cella rappresentata dall'*Actor* **a** invia informazione **MyInfo** sullo suo stato corrente alle celle vicine, usando forme di

```
suspend fun sendToNeighbors(
   a: ActorBasic, MyInfo: String,
   mode:String="evstream",
   NbNameslist: Vector<Term>? = null ){
   when( mode ){
   "evstream"->
     emitLocalStreamToNeighbors(a,MyInfo )
   "mqtt"    ->
```

comunicazione diverse a seconda dell'argomento **mode**.

La comunicazione a *dispatch* richiede di ricevere in ingresso la lista dei nomi delle celle vicine (**NbNameslist**) che alle altre forme non serve.

```
      emitEvstreamWithmqtt(a,MyInfo)
   "dispatch"->forwardToNeighbors(a,MyInfo,NbNameslist!!)
   else -> CommUtils.outred(
       "sendToNeighbors mode $mode not found")
   }
}
```

In questo Sprint, useremo il valore di default (*evstream*) di **mode**, usando la funzione *emitLocalStreamEvent* che ogni *Actor* posside

emitLocalStreamToNeighbors

```
@JvmStatic
suspend fun emitLocalStreamToNeighbors(a: ActorBasic, MyInfo:String ){
    a.emitLocalStreamEvent("curstate", "curstate($MyInfo)" )
}
```

emitLocalStreamEvent

Questa funzione è definita in **ActorBasic.kt** e permette a un *Actor* di inviare un evento-stream a tutti i suoi *subscriber*.

```
suspend fun emitLocalStreamEvent(ev: String, evc: String ){
   emitLocalStreamEvent(MsgUtil.buildEvent(name, ev, evc))
}

suspend fun emitLocalStreamEvent(v: IApplMessage){
   subscribers.forEach { it.actor.send(v) }
}
```

P_3

- La logica del comportamento di una cella deve essere identico a quello di ogni altra cella.

  P_3) come definire il comportamento di una cella?

  Sappiamo che ogni cella opera logicamente in due fasi:

1. Acquisizione del numero di celle vicine con stato **live**
2. Calcolo del nuovo stato e propagazione della informazione relativa a tale cambiamento alle celle vicine.

P_4

- L'autonomia delle celle implica **asincronismo** nelle loro attività.

  P_4) come coordinare i comportamenti delle celle?

  In linea generale, osserviamo che la presenza di un **'orchestratore'** facilita la possibilità che le celle aggiornino il loro valore in modo coerente e coordinato con quello delle altre celle.

  Impostare un soluzione priva di orchestratore (anche detta **coreografata**) è possibile, ma risulta (molto) più difficile da realizzare. Si veda *OrchestrazionevsCoreografia*

P_5

- Il campo di gioco può essere composto da molte celle. Nel caso di una **griglia 10x10**, il sistema applicativo dovrà essere composto da **100** celle-attori

  P_5) chi ha la responsabilità di creare/configurare il sistema delle celle?

Come principio generale, può essere opportuno tenere conto del *Principio di singola responsabilità*.

P_6

- Aspetti rilevanti sono anche legati problematica dell'*User exeperience* (**UX**), cioè della relazione tra una persona-utente e il gioco.

  P_6) quali funzioni introdurre per una accettabile/buona UX?

In questa fase saremo alquanto minimalisti, ipotizzando solo che l'utente finale si aspetta di:

1. P_6_0: poter usare la GUI solo a gioco completamente inizializzato;

2. P_6_1: poter sospendere il gioco;
3. P_6_2: poter ripredere il gioco da dove l'aveva sospeso;
4. P_6_3: poter sospendere il gioco, *ripulire* la griglia astratta (cioè porre a **off** lo stato di tutte le celle) e ripredere dopo avere impostato sulla griglia una nuova configurazione iniziale di celle **on**.

Notiamo che *Una GUI in JavaFX* fornisce già un pulsante **start** per attivare il gioco, (che poi commuta in un pulsante **stop** per poterlo sospendere) e un pulsante **clear** per poter ripulire la griglia astratta (in una fase di gioco sospeso).

A questo punto, procediamo con *GLM-Sprint3: interazioni a eventi-stream* in cui realizzare e sperimentare interazioni tra celle locali a una stessa JVM medianie *eventi-stream*.

GLM: un quadro di riferimento proposto dall'analista

1. Il sistema è composto da un insieme di attori (celle) che operano (al momento) in un contesto condiviso.
2. Le celle sono create da un attore :blue:`gridcreatorc che si occupa anche di configurarle, una volta che sono state create tutte. in modo che ciascuna cella possa conoscere il numero delle sue celle vicine.
3. Ogni cella è un *observer* delle sue celle vicine e produce un evento-stream *curstate* che contiene il suo stato corrente.
4. Ogni cella elabora gli eventi *curstate* provenienti dalle sue celle vicine, per determinare il numero di vicini con stato **true** operando in tre macro-fasi:
    1. Emissione dell'evento *curstate* con il proprio stato corrente
    2. Ricezione degli eventi *curstate* delle celle vicine
    3. Calcolo del nuovo stato
    4. Attesa di un segnale di coordinamento (da parte del gestore del gioco *gamelife*) per iniziare un nuovo ciclo di calcolo.
5. L'attore *gridcreator* avvisa il gestore del gioco *gamelife* quando tutte le celle sono pronte per iniziare il gioco ed emette informazione osservata da *griddisplay*
6. L'attore *griddisplay* è un *observer* delle celle e visualizza il gioco.
7. L'attore *gamelife* è l'orchestratore del gioco e coordina le attività delle celle, gestendo anche i comandi dell'utente.

Indice: *Indice GLM*