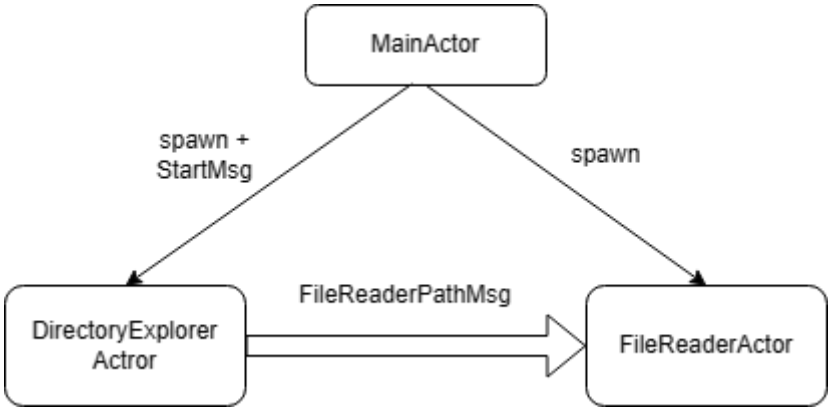
Assignment 03 - Prog. concorrente tramite scambio di messaggi e prog. distribuita

Assignment svolto in gruppo da Federico Pirazzoli (matr. 0001079378) e Gian Luca Nediani (matr. 0001075900)

1. Esplorazione directory con soluzione ad attori

In questa sezione il task di esplorazione ricorsiva di una directory in cerca di file .java e della elaborazione di questi viene risolto con un approccio basato su attori. Il framework utilizzato è Akka per Java. I diversi attori e i relativi comportamenti vengono specificati estendendo dei behavior (Akka tipato). Un attore main (ActorSystem) dà il via alle operazioni creando due attori child: uno per l'esplorazione della directory e uno per l'elaborazione dei file .java trovati. Ogni volta che l'attore per l'esplorazione trova dei file manda un messaggio all'attore di elaborazione che ne elabora il conenuto. Le operazioni sono cominciate da uno StartMsg contenente il path della directory iniziale inviato dal main actor all'actor di esplorazione.



Tempo medio di esecuzione nella stessa configurazione dei precedenti assignment: 823ms.

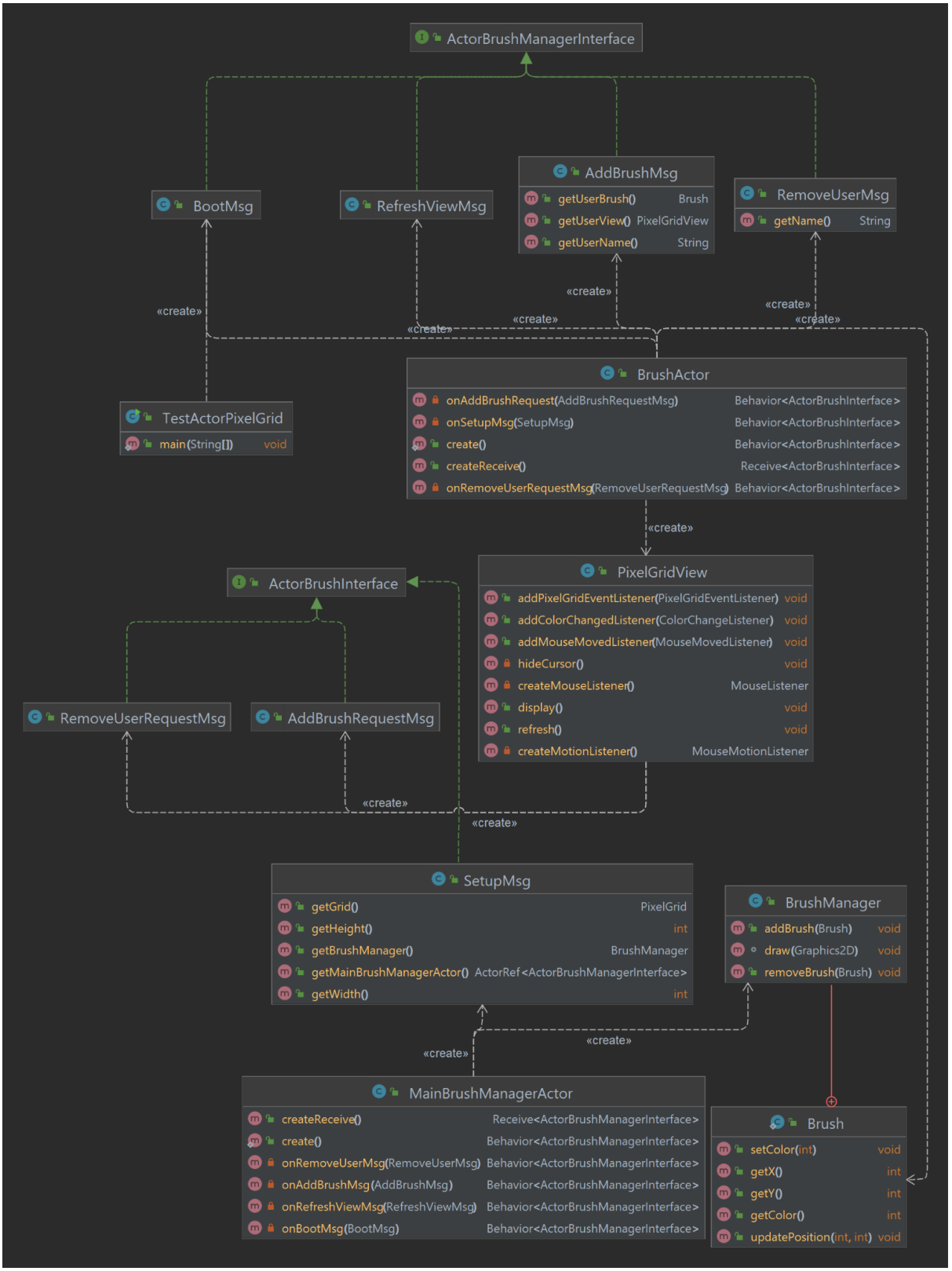
2. Cooperative Pixel Art con async message passing

Anche per questo punto si fa uso di attori con Akka tipato per Java. L'architettura scelta fa uso di un attore di coordinazione (MainBrushManagerActor). Questo attore (ActorySystem) svolge tutti i ruoli di coordinazione centralizzata: avviare l'applicazione facendo lo spawn di un primo client cgh, gestire l'aggiunta e la rimozione di nuovi client e gestire l'aggiornamento di tutte le view client in caso di modifiche alla tela condivisa da parte di uno dei client. Quelli che finora sono qui stati chiamati "client" vengono modellati nell'applicazione come dei BrushActor: una volta creati ognuno di essi avrà un pennello e una view condivisa da modificare con il proprio pennello. Nella view è anche presente la possibilità di aggiungere un nuovo user.
I messaggi che BrushActor può ricevere sono:

- AddBrushRequestMsg: ricezione di richiesta di aggiunta di un nuovo client che viene comunicata al coordinatore (a seguito del click sulla GUI del bottone "Add User")

- RemoveUserRequestMsg: ricezione di richiesta di rimozione di un nuovo client che viene comunicata al coordinatore (a seguito del click sulla GUI del bottone “Remove User”)
- SetupMsg: messaggio con le informazioni per creare la propria view al momento dello spawn

I messaggi che può ricevere il MainBrushManagerActor sono invece:
- AddBrushMsg: alla ricezione effettua lo spawn di un nuovo BrushActor con le informazioni contenute nel messaggio
- BootMsg: messaggio di startup per avviare l’applicazione
- RefreshViewMsg: ricevuto quando un client BrushActor effettua una modifica alla tela condivisa, effettua la refresh di tutte le view
- RemoveUserMsg: alla ricezione rimuove il BrushActor che ha inviato il messaggio



3. Cooperative Pixel Art con oggetti distribuiti

In questa sezione è stato risolto il problema del punto precedente utilizzando però un approccio ad oggetti distribuiti. In particolare è stato utilizzato Java RMI, framework incluso

nella JVM basato sul concetto di invocazione remota di metodi. L'idea di base è quella di poter invocare remotamente metodi di oggetti in altri nodi del sistema distribuito. Inoltre è possibile passare il riferimento a interi oggetti in modo tale da poterli manipolare localmente anche se si trovano in altri nodi.
L'architettura pensata per questo problema è di tipo client-server: un server espone sul registro di Java RMI due oggetti (la griglia e il manager dei pennelli) che fungono da controllo centralizzato della logica dell'applicazione, mentre i client una volta creati ottengono il riferimento a questi oggetti e li utilizzano per il render della loro GUI locale e comunicano il loro riferimento al server in modo che questo possa invocare metodi remotamente su di essi in un momento successivo.
Ogni client ha un pennello con il quale può modificare lo stato della tela condivisa: quando un client effettua una modifica alla tela (spostamento del pennello, colorazione di una cella, selezione di un nuovo colore) esso invoca remotamente un metodo della griglia o del manager, dei quali conserva il riferimento. Il server a sua volta, quando riceve da un client una modifica allo stato della tela, comunica a tutti i client di aggiornare la loro view, sempre facendo uso della griglia e del manager condivisi per riferimento.
Sostanzialmente l'interazione su cui si basa l'architettura prevede due chiamate remote: un client effettua una chiamata remota per comunicare una modifica alla tela cooperativa, dopodiché il server effettua una chiamata remota a tutti i client per aggiornare la loro view.
Di seguito viene mostrato un diagramma di sequenza che spiega il funzionamento dell'applicazione.