Documentazione Progetto LSO Jolie A.A. 2015-2016

Gruppo 4STAGIONI

Composto da: Fausto Carano, Stefano Leggio, Stefania Lusardi, Stefano Pardini

Gestione Stock

Dopo una prima discussione, era emersa l'idea di poter caratterizzare ciascuno Stock mediante un suo specifico file di configurazione, al fine di rendere rapido e modulare il loro ingresso all'interno del sistema.

Inizialmente avevamo vagliato la più semplice delle soluzioni, ovvero impostare direttamente da command line, differenti costanti che caratterizzassero ciascuno Stock in fase di avvio (input port, nome, disponibilità, ecc..). Tuttavia tale modalità faceva decadere la necessità dei file di configurazione ed introduceva almeno due criticità:

- il numero di parametri da dichiarare da linea di comando;
- la comunicazione *uno a molti* tra Market e ciascuno Stock, tuttavia risolvibile mediante dynamic binding sulla output port del Market. Ciascuno Stock, nella fase di registrazione presso il Market avrebbe dovuto accreditare anche la propria input port.

Nella soluzione proposta, invece, se da un lato è parzialmente perduta la possibilità di distribuire i vari servizi Stock, dall'altro è sicuramente acquisita una semplice, e a nostro parere più pulita, gestione degli stessi.

Abbiamo quindi inserito uno Stock Manager (StocksMng.ol di seguito StocksMng).
StocksMng svolge funzionalità di dispatcher / accentratore per le richieste provenienti dal Market e si occupa di inoltrarle ai relativi Stock embeddati.

Dato l'ampio spettro di funzionalità presenti nel Market -il nocciolo dell'intero sistema- egli non deve dunque preoccuparsi di alcun tipo di gestione del dynamic binding sulla output port comunicativa con gli Stock, dato che ne è necessaria una soltanto per l'inoltro delle richieste a StocksMng. E' dunque snellita l'intera procedura comunicativa, di fatto delegata a StocksMng.

Ciascuno Stock può invece comunicare direttamente con il Market poichè conosce le coordinate della sua input port.
Tale soluzione si è resa altresì l'unica percorribile poichè, ad oggi, Jolie non permette il dynamic binding delle input port sulle singole istanze dei servizi dinamicamente allocati / embeddati.

Da un punto di vista figurativo, è come se il mercato comunicasse direttamente con un unico grande magazzino di stoccaggio che si occupa di gestire più risorse e più richieste contemporanee per ciascuna di esse. Ciascuna risorsa può invece inoltrare comunicazioni indipendenti al mercato (nel nostro caso, altro non sono che operazioni di deperimento e produzione):
Market → StocksMng → Stocks
Stocks → Market

Gli Stock sono quindi dinamicamente istanziati all'interno di StocksMng, attraverso la feature di Jolie dynamic embedding. Ciascuno Stock dispone di un proprio file di configurazione XML in cui sono riportate le caratteristiche che lo descrivono. Il servizio *discover@StocksMng* ciclicamente osserva il path *CONFIG_PATH_STOCKS* (config/stocks) e, qualora vi trovi nuovi file XML, avvia

la loro istanziazione come nuovi servizi embeddati:

loadEmbeddedService@Runtime(embedInfo)(StockInstance.location);

All'interno di un'adeguata struttura dati globale è memorizzata la loro location in relazione al nome dello Stock

global.dynamicStockList.(stockName)[0].location = StockInstance.location

E' presente un'unica via comunicativa locale, un'unica output port, per comunicare con tutti i servizi embeddati:

```
outputPort StockInstance {
   Interfaces: StockInstanceInterface
}
```

Sono dunque utilizzate a pieno titolo le feature dynamic embedding e dynamic binding.

Il Market, qualora voglia comunicare con uno specifico Stock, mediante la sua unica output port invia una richiesta all'input port di StocksMng. L'operazione in ascolto, oltrepassati i controlli di routine (di cui discuteremo in seguito), transita per il seguente step:
grazie al dynamic binding (e al dynamic lookup) imposta la location della local port *StockInstance*

StockInstance.location = global.dynamicStockList.(stockName)[0].location;

Quella specifica output port rappresenta adesso una via comunicativa con il relativo Stock sul quale è possibile invocare l'operazione richiesta. L'eventuale response generata dallo Stock transiterà a ritroso sul canale comunicativo.

Durante lo sviluppo sono tuttavia sorti alcuni dubbi. Poichè StocksMng per comunicare con uno specifico Stock deve occupare la output port con la location ad esso associata, non si va così creando un collo di bottiglia in cui le richieste possono esser servite solo sequenzialmente?
E poi. Poichè la output port è considerata alla stregua di una variabile globale, e quindi a tutti gli effetti passibile di race condition, è necessario vincolare il suo accesso in scrittura con una qualche forma di sincronizzazione?

Le risposte sono giunte grazie ad una vivace discussione in classe e ad alcune informazioni indicate nella documentazione.

“Jolie allows output ports to be dynamically bound, i.e., their locations and protocols (called *binding informations*) can change at runtime. Changes to the binding information of an output port is local to a behaviour instance: output ports are considered part of the local state of each instance. Dynamic binding is obtained by treating output ports as variables.”

Poichè ciascuna operazione invocata equivale ad un thread indipendente, l'accesso alla specifica *output port location* non presenta alcun tipo di race condition.

In virtù di quanto appena riportato, all'interno di StocksMng non è necessario alcun costrutto per la gestione della sincronizzazione per l'accesso ad eventuali sezioni critiche dato che, una volta inizializzata, la struttura dati globale è disponibile in sola lettura.

La seguente istruzione :

global.dynamicStockList.(stockName)[0].instantianCompleted = true

posta in calce a *discover@StocksMng*, ed il relativo controllo *is_defined* in testa alle altre

operazioni, garantisce il corretto embedding della Stock instance e la compilazione della struttura dati a supporto.

Ben differente è invece il caso degli Stock in cui in fase di avvio è inizializzata una struttura dati globale, speculare alle informazioni estratte dal file di configurazione XML, passibile di scritture e letture concorrenti.

Le operazioni di produzione e deperimento godono di vita indipendente e sono avviate, quali thread paralleli, mediante due chiamate riflessive alle operazioni :

wasting@Stock e *production@Stock*

solo dopo aver ricevuto una response della corretta registrazione al Market.

L'attributo *availability* traccia la disponibilità residua dello Stock ed è sicuramente passibile di race condition, gestita mediante l'utilizzo del costrutto *synchronized* (in ciascun blocco sono inserite al più una decina di istruzioni).

La casistica di concorrenza è senza dubbio riconducibile al paradigma *readers-writers.* Abbiamo preferito non implementare alcun tipo di priorità sull'accesso alla risorsa condivisa, garantendo un'uniforme equità per scritture e letture concorrenti: un assoluto non determinismo.

Se, come nel classico caso, il primo lettore avesse acquisito il lock per l'accesso alla risorsa e soltanto l'ultimo di essi l'avesse rilasciato, dato il rapido avvicendamento delle operazioni di acquisto, vendita, produzione, deperimento, avremmo potuto innescare casistiche di starvation per gli scrittori in attesa. Stessa problematica, ma con soggetti invertiti, qualora avessimo invece garantito maggior priorità agli scrittori.

L'unico accesso in lettura è previsto dall'operazione *infoStockAvailability*, invocata dal Market durante le operazioni di compravendita o dal Player, mediante l'omonima operazione del Market o attraverso *infoStockPrice* qualora sia richiesto il prezzo corrente.

In scrittura sono invece coinvolti i microservizi *buyStock*, *sellStock, wasting, production*.

Qualora avessimo privilegiato i lettori, nel caso in cui molti Player concorrenti avessero richiesto *infoStockPrice*, avremmo posto eccessivamente in attesa le operazioni *core* del Market, ovvero le compravendite provocando, nel caso limite, casistiche di starvation per i thread scrittori.

Una volta che l'intero applicativo è in fase running è possibile avviare nuovi Stock semplicemente inserendo nuovi file XML all'interno del percorso *config/stocks.* Essi procederanno alla corretta registrazione presso il Market ed i Player inizieranno ad effettuare su di essi le operazioni di compravendita.

Data la loro natura esecutiva (embedding) non è attualmente possibile terminare l'esecuzione di uno Stock. Anche qualora essi fossero stati avviati in forma indipendente, avremmo comunque dovuto prevedere un'articolata procedura di recovery per la propagazione dell'informazione e la loro rimozione.

Gestione Market

Il Market dispone di un'unica output port per la comunicazione con StocksMng e di due input port rispettivamente per ricevere le richieste dei Player e degli Stock.

Come indicato nel precedente paragrafo, egli non deve occuparsi di alcun tipo di dynamic binding per la comunicazione con gli Stock.

Ciascun Stock innesca la registrazione mediante l'operazione:

registerStock@Market

e l'apposita struttura dati globale *registeredStock* (semplicemente accessibile mediante dynamic lookup), si occupa di mantenere l'unica informazione ad essi relativa, ovvero il prezzo totale (*totalPrice*).

Anche i Player debbono transitare per l'operazione di accounting e, in particolare, invocano:

registerPlayer@PlayerToMarketCommunication.

La struttura dati globale *account* mantiene le informazioni di ciascun Player: la liquidità ed il suo *user identification* univoco che, per semplicità, equivale al proprio nome.

Durante la scrittura del codice sono emerse molte problematiche di cooperazione e sincronizzazione tra i vari thread che concorrono per l'accesso alle strutturi dati globali *registeredStock* e *accounts*.

A garanzia di atomicità ed accesso non interferente, ad ogni Stock e ad ogni Player è associato un semaforo binario che, come maggiormente discusso in seguito, non permette l'esecuzione concorrente di acquisto o vendita sullo stesso Stock da parte di Player differenti, o su Stock differenti da parte dello stesso Player. A contorno di tali casistiche debbono inoltre essere considerate eventuali operazioni di deperimento e produzioni provenienti dagli Stock: anche in questo caso i semafori si comporano egregiamente.

Di seguito sono identificati e commentati alcuni possibili scenari.

Si registrano presso il Market due Stock e si accreditano due differenti Player.
Per entrambi le tipologie di soggetti, come precedentemente indicato, sono create specifiche istanze di semaforo, accorpate ai sottonodi delle strutture dati, ed istanziata -per ciascuna- un unico *lock* acquisibile.
Nel codice sono osservabili le due seguenti define:

define createStockSemaphore, define createPlayerSemaphore

Entrambi gli Stock, rispetto alle tempistiche indicate nel file di configurazione, procedono con le normali operazioni di produzione e deperimento invocando, quindi:

addStock@Market e *destroyStock@Market*

che si occupano di intervenire sul prezzo totale, mediante il calcolo della variazione in relazione alla specifica quantità deperita o prodotta.
Una prima situazione di interferenza potrebbe verificarsi qualora lo stesso Stock tenti di intervenire contemporaneamente sulla variabile *totalPrice*. Il semaforo dello Stock e l'unicità del lock acquisibile, prevengono il generarsi di situazioni incongruenti.

Introduciamo adesso le azioni dei Player.
In primis un Player necessita di acquisire informazioni rispetto agli Stock disponibili sul Market. Invoca l'operazione *infoStockList@Market* che accede in sola lettura alla struttura dati globali *registeredStock*.

L'unica situazione incongruente potrebbe verificarsi qualora l'operazione di registrazione di uno Stock non sia perfettamente conclusa, mentre la struttura dati è quindi parzialmente redatta: al Player potrebbero dunque pervenire informazioni incosistenti. Il flag *registrationCompleted* impostato solo al termine dell'operazione di registrazione dello Stock, ed il relativo controllo *is_defined*, consentono di prevenire tale casistica:

```
[ infoStockList( info )( responseInfo ) {
   foreach ( stockName : global.registeredStocks ) {
      if ( is_defined( global.registeredStocks.( stockName ).registrationCompleted )) {
```

Il Player adesso dispone di una lista di Stock sui quali lanciare le varie attività di acquisto e vendita. Supponiamo sia invocata l'operazione *buyStock@Market.* I primi controlli (di cui parleremo in dettaglio nel paragrafo dedicato alle eccezioni) possono tranquillamente essere eseguiti da più thread indipendenti e temporalmente intercalati (qualora avessimo a che fare con più attività di compravendita), poichè richiedono l'accesso in sola lettura alle strutture dati condivise (le cui compilazioni sono avvenute in istanti precedenti e adesso sicuramente concluse).

Successivamente sono acquisiti i due semafori (sullo Stock e sul Player):

```
acquire@SemaphoreUtils( currentStock.semaphore )( response );
acquire@SemaphoreUtils( currentPlayer.semaphore )( response );
```

che, come indicato in precedenza, non permettono l'esecuzione concorrente di acquisto o vendita sullo stesso Stock da parte di Player differenti, o su Stock differenti da parte dello stesso Player.

E' adesso necessario verificare la disponibilità della risorsa richiesta, interrogando direttamente lo Stock:

```
infoStockAvailability@MarketToStockCommunication( stockName )( availability );
if ( availability <= 0) {
   with( exceptionMessage ) { .stockName =stockName };
   throw( StockAvailabilityException, exceptionMessage )
};
```

Successivamente è calcolato il prezzo unitario dello Stock e, qualora il Player non disponga di sufficiente liquidità per procedere con l'acquisto viene interrotta l'intera procedura, rilasciando i semafori acquisiti.
Si procede ora atomicamente con le seguenti operazioni:

```
// (1) l'unità di Stock disponibile viene decrementata di 1
    buyStock@MarketToStockCommunication( stockName )( response );

// (2) si aumenta di 1 la quantità di Stock posseduto dal Player acquirente
    currentPlayer.ownedStock.( stockName ).quantity++;

// (3) si decrementa l'Account del Player dell'attuale prezzo di un'unità di Stock
    currentPlayer.liquidity -= currentPrice;
    toRound = currentPlayer.liquidity; round; currentPlayer.liquidity = roundedValue;

// (4) si aumenta il prezzo totale di quello Stock in maniera corrispondente all'algoritmo di pricing
// (che si avvale delle define timeCalc, priceInc e round)
    timeCalc; priceInc; // algoritmo di pricing
    currentStock.totalPrice += priceVariation;
    toRound = currentStock.totalPrice; round; currentStock.totalPrice = roundedValue;
```

Tuttavia, contemporaneamente alle operazioni indicate, altri thread potrebbero acquisire il controllo del processore e tentare di evolvere nella loro esecuzione.
Lo stesso Player, qualora innescasse un'ulteriore compravendita (ovvero qualora abbia avviato due o più operazioni concorrenti concatenate da un *pipe*), rimarrebbe in attesa sul semaforo ad esso associato. Un Player differente, qualora richiedesse un acquisto o una vendita sullo stesso Stock, anch'egli rimarrebbe in attesa sul semaforo associato allo Stock.
Tale modalità implementativa dei semafori offre, a nostro parere, un alto grado di parallelismo e quindi un'efficiente responsività nell'evasione delle richieste.

Potrebbero altresì presentarsi ulteriori casistiche sensibili in relazione agli Stock.
Qualora il thread associato al deperimento volesse comunicare al Market la variazione della disponibilità, la relativa operazione *wasting@Market* -che interviene in modifica al prezzo totale- dovrebbe poter acquisire il semaforo sullo specifico Stock, ma verrà messo in attesa poichè attualmente in uso dall'operazione di acquisto.
Data la natura OneWay di *wasting@Market*, lo Stock potrà proseguire con il decremento della quantità disponibile (compatibilmente al costrutto synchronized in esso dichiarato) anche senza attendere l'effettivo incremento del prezzo, e quindi la response del Market.

E' altresì vero che il prezzo attuale di acquisto potrebbe adesso non essere congruo rispetto all'effettiva disponibilità dello Stock. Tuttavia dai test effettuati, qualora avessimo optato per una operazione di tipo RequestResponse, ci saremmo sicuramente imbattuti in possibili Deadlock, in cui l'insieme dei thread cooperanti avrebbero ciclicamente atteso la release del semaforo o del token del blocco synchronized.
Data inoltre la necessaria e imprescindibile natura atomica dell'insieme delle operazioni di acquisto (così come quelle di vendita), abbiamo supposto che i Player accedano alle *attuali* informazioni disponibili nel Market, come in una fotografia istantanea. Deperimenti e produzioni sono dunque posti in coda.

Tuttavia, poichè durante le attività di acquisto, è prima richiesta l'effettiva disponibilità dello Stock mediante:

infoStockAvailability@MarketToStockCommunication(stockName)(availability);

e, successivamente, è invocata la l'operazione di decremento della disponibilità sullo Stock

buyStock@MarketToStockCommunication(stockName)(response);

nell'intervallo di tempo tra le due potrebbe tuttavia innescarsi un'operazione di deperimento che porti lo Stock a zero.
In tal caso, la prima operazione è un lasciapassare per giungere al calcolo del prezzo corrente rispetto alla disponibilità effettiva.
La seconda è portata a termine solo se la disponibilità dello Stock è realmente sufficiente.
Adesso possono essere tranquillamente eseguite le successive operazioni (3 e 4), al termine delle quali sono rilasciati entrambi i semafori.

Supponiamo che adesso il controllo sia acquisito dalle operazioni

wasting@Market o *production@Market*

precedentemente poste in attesa al semaforo. Esse potranno evolvere nella loro esecuzione, contestualmente bloccando l'accesso ai singoli Stock e non all'intera struttura dati globale, ponendo dunque in attesa eventuali richieste di compravendita sulla specifica risorsa.

Durante l'insieme delle operazioni concorrenti, un Player potrebbe invocare *infoStockAvailability.*
Tal operazione non richiede alcun tipo di controllo poichè è direttamente inoltrata allo Stock (sarà lui stesso a gestire l'accesso ad *availability* mediante un blocco synchronized).

Differente è invece *infoStockPrice* che richiede il calcolo del prezzo unitario come computo del prezzo totale e dell'effettiva disponibilità.

In generale, così come precedentemente riportato nel paragrafo relativo agli Stock, molte delle attività cooperanti del Market sono riconducibili al paradigma *readers-writers.*
Non abbiamo offerto alcuna priorità nè agli uni nè agli altri poichè, compravendite, deperimenti e produzioni (writers), debbono poter procedere omogeneamente, senza alcuna prelazione da parte dei thread lettori. Il Market è un'unità dinamica in continua evoluzione, in cui eventuali strategie speculative dei Player (a prescindere dall'algoritmo di pricing posto a tutela), qualora individuati i pattern vincenti relativi a eventuali “sincronismi forzati”, potrebbero far deragliare l'intero sistema. Il non determinismo offerto dallo scheduling dei thread concorrenti, le cui sezioni critiche sono ben delineate e tutelate da semafori equivale, a nostro parere, ad un'equa ripartizione delle priorità e ad un corretto avvicendamento delle operazioni.

In ultima istanza vorremmo infine porre risalto a due ulteriori race conditions (poco probabili ma tecnicamente possibili).

Si osservi il seguente blocco di codice (privo di alcun costrutto di sincronizzazione):

```
[ registerStock( newStock )( response ) {
   if ( ! is_defined( global.registeredStocks.( newStock.name )[ 0 ] )) {
      // inizializzazione della struttura dati
      ..
```

Si pensi alla casistica in cui due (o più) Stock con lo stesso nome accedano contemporaneamente all'operazione indicata, il primo Stock oltrepassi con successo la condizione imposta dal costrutto if e, esattamente prima della creazione della struttura dati ad esso relativa, si verifichi un context switch a favore del secondo thread-Stock.
Anche quest'ultimo riuscirà ad oltrepassare il controllo: si potrebbero quindi verificare compromettenti incongruenze all'interno della struttura dati.
StocksMng svolge tuttavia un'attenta gestione delle omonimie dei nomi degli Stock e previene il manifestarsi di tali anomalie. La soluzione adottata ben si confà a contesti distribuiti in cui ciascuno Stock goda di vita indipendente:

```
[ registerStock( newStock )( response ) {
   acquire@SemaphoreUtils( global.semaphores.registerStock )( response );
   if ( ! is_defined( global.registeredStocks.( newStock.name )[ 0 ] )) {
      // inizializzazione della struttura dati
      release@SemaphoreUtils( global.semaphores.registerStock )( response );
      ..
```

La seconda delle casistiche è praticamente analoga alla precedente e coinvolge l'operazione di registrazione di nuovi Player. Non ci dilunghiamo ulteriormente nella sua descrizione, tuttavia ci preme indicare che l'assenza di una corretta gestione delle operazioni di accounting, qualora l'applicativo fosse destinato ad un reale contesto di produzione, potrebbe condurre a pericolose vulnerabilità. Una grossa parte delle seguenti eccezioni è finalizzata al loro contenimento.

Eccezioni

La nostra implementazione di jExchange prevede un articolato sistema di propagazione delle eccezioni tra i vari servizi coinvolti ed alcuni interessanti sistemi di recovery. Di seguito riportiamo le funzionalità più significative.

Qualora sia avviato StocksMng prima dell'avvio del Market, i vari servizi Stock embeddati avviano una procedura ciclica di connessione che, qualora eseguita un numero di volte pari a *MAX_CONNECTION_ATTEMPTS*, innesca una chiusura pulita dell'intero applicativo.

Durante le attività di lettura e verifica dei file di configurazione XML ed il relativo avvio dei servizi embeddati, all'interno di *discover@StocksMng* potrebbero innescarsi varie tipologie di fault. Piuttosto che terminare l'intera esecuzione, la define *discoverFaultMng* si occupa di contabilizzare l file di configurazione “malevoli” all'interno di una exception list, così da prevenire il lancio di nuove eccezioni sulla stessa problematica.
Grazie a *StockDuplicatedException* il Market previene tentativi di registrazione multipla da parte dello stesso Stock, casistica parallelamente gestita in *discover@StocksMng* dal sistema di verifica dei nomi degli Stock all'interno dei file di configurazione XML.

Durante le fasi di sviluppo ci siamo interrogati sulle eventuali vulnerabilità che un Player malevolo avrebbe potuto sfruttare per generare attività illecite e non autorizzate, dati inconsistenti o ben più gravi ed imprevedibili scenari.
Di seguito riportiamo il set di eccezioni che tentano di arginare il manifestarsi di tali casistiche e, per ciascuna, una breve descrizione:

- *PlayerDuplicatedException*: la procedura di accounting deve prevenire la registrazione di due o più Player con lo stesso UserId (nome del Player);
- *PlayerUnknownException*: un Player deve obbligatoriamente essere transitato dalla fase di registrazione prima di poter procedere con qualsiasi operazione;
- *InsufficientLiquidityException*: è innescata qualora un Player tenti di acquistare uno Stock che necessiti di maggior liquidità rispetto a quella effettivamente posseduta;
- *NotOwnedStockException:* il Market è attento all'atteggiamento malevolo in cui un Player tenti di vendere, e dunque di incassare il relativo valore, di uno Stock che non possiede;
- *StockAvailabilityException*: innescata all'interno dello Stock, si propaga sino al Player attraverso StocksMng ed il Market. All'interno di quest'ultimo previene il propagarsi delle attività di compravendita di cui è puntualmente informato il Player.
- *StockUnknownException*: utilizzata anche in altri contesti (comunicazione *Market → StocksMng → Stocks*), previene l'innescarsi di problematiche qualora si tenti di acquistare o vendere uno Stock inesistente.

Intelligenza e funzionalità dei Player

Sono stati previsti tre tipologie di Player:

- Random Player (player1.ol);
- 2 Player Generici con politiche differenti che effettuano operazioni in parallelo (player2.ol);
- Speculative Player (player3.ol).

Tutti i Player gestiscono le eccezioni sollevate dal Market, uniche situazioni che possono portare all'interruzione del player sono:

- *PlayerDuplicatedException:* esiste già un Player con lo stesso nome.
- *IOException*: il maket non è attivo, viene visualizzata per ogni Player la sua situazione

patrimoniale se era già stata effettuata la registrazione al Market, in caso contrario viene solo interrotta l'esecuzione del Player

La gestione delle eccezioni è stata realizzata attraverso l'uso di una interfaccia *LocalInterface* e di un servizio di tipo *OneWay* corredato di *inputPort e outputPort*, per permettere una invocazione *riflessiva* da parte del Player stesso sui propri servizi.

```
interface LocalInterface {
   OneWay: registrationplayer( void ) // registrazione del Player presso il Market
   OneWay: runplayer( void )
}
inputPort LocalInputPort {
   Location: "local"
   Interfaces: LocalInterface
}
outputPort Self { Interfaces: LocalInterface }
```

Si è prevista inoltre la possibilità di rendere parametrizzabili, attraverso l'uso di costanti utilizzabili da line di comando, alcune caratteristiche dei Player quali:

- nome del player (-C Player_Name=\"NOME PLAYER\");
- frequenza con cui i player effettuano le operazioni di acquisto e vendita (-C Frequence=<millisecondi>).

Descrizione delle tre tipologie di Player realizzati:

- Speculative Player: cerca di creare una bolla speculativa, con un'entrata nel mercato controllata (per cercare di far crescere una bolla sileziosamente), ma un'uscita drastica cercando di ottimizzare le vendite.

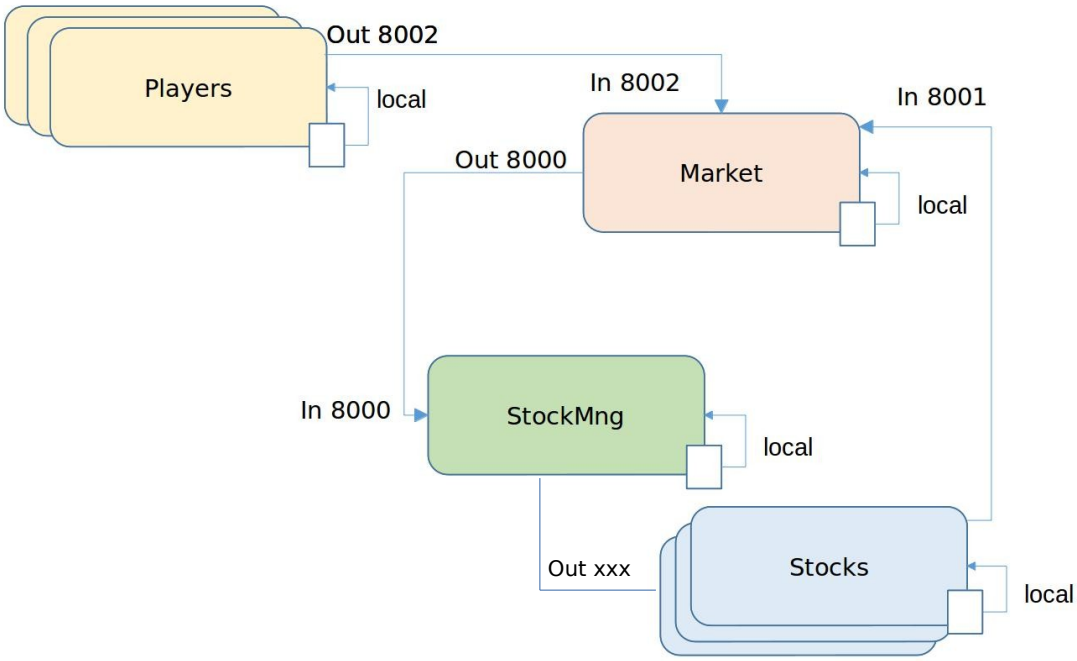
- 2 Player Generici: due player implementano differenti politiche di acquisto e vendita in modo parallero secondo due diverse politiche.
    1. Politica 1:
       Se il prezzo corrente è salito meno del 10% dall'ultima operazione Acquisto;
       se il prezzo corrente è salito più del 30% o è sceso più del 5% dall'ultima operazione Vendo.

    2. Politica 2:
       Se il prezzo corrente è salito meno del 5% dall'ultima operazione Acquisto;
       se il prezzo corrente è salito più del 40% o è sceso più del 2% dall'ultima operazione Vendo.

       La sequenza delle operazioni è parallelizzata e temporizzata.

- Random Player: effettua un modo randomico operazioni di buy e sell sugli stock attivi.

Schema di comunicazione tra i servizi



Interfacce e strutture dati

La suddivisione delle interfacce ed il loro *including* ricalca la logica implementativa delle entità coinvolte.

Per i seguenti flusso comunicativi
Market → StocksMng → Stocks
Stocks → Market
data la promiscuità operativa dei servizi e la conseguente e necessaria condivisione delle strutture dati, si è definito un *iol* comune, comprensivo delle seguenti interfacce i cui nomi sono evocativi della loro funzionalità:

- *StockInstanceInterface:* descrive le operazioni richieste da StocksMng a ciascuna Stock instance dinamicamente allocata ed embeddata.
- *StockToMarketCommunicationInterface:* espone le operazioni invocabili sul Market da parte di ciascuno Stock.
- *MarketToStockCommunicationInterface:* esattamente speculare al precedente. Si ricorda che l'istanza StocksMng svolge le veci di dispatcher / gateway per ciascuna Stock instance.

La struttura dati di maggior rilievo (*Stocks*) prevede in ciascun suo sottonodo – identificato da un valore numerico incrementale – una coppia di strutture dati che descrivo ciascuno Stock, suddiviso in parametri statici (ovvero di puro carattere informativo) e dinamici (variabili soggette a modifica in corso di esecuzione):

```
type StockSubStruct: void {
   .static: StockStaticStruct
   .dynamic: StockDynamicStruct
}
```

E' imporante notare che la struttura dati degli Stock ricalca esattamente i metadati degli XML.
E' invece affidato a *playerInterface.iol* la descrizione del seguente flusso comunicativo:
Player → Market
In *PlayerToMarketCommunicationInterface* sono definite le operazioni pubblicamente esposte nel Market ed invocabili dai Player.

Di minor rilievo è invece *commonInterface.iol* in cui è presente un'interfaccia comune a tutte le entità coinvolte nell'applicativo: è presente un'unica operazione che consente il *check* dello stato del Market.

Come descritto nel precedente paragrafo, per ciascuna operazione sono esplicitate le eventuali eccezioni generabli (sono inoltre implementati specifici *exception data type*).
Ci preme infine far presente che in generale si è optato per una puntuale definizione dei tipi di dato, – coscientemente mai descritti mediante il generico tipo *any* – rispettando la logica cooperativa offerta dal linguaggio.

MonitorX

La visualizzazione dell'output è fornita da un servizio Java embeddato all'interno del Market.
Il Market è l'entità centrale del sistema alla quale tutte le altre componenti richiedono di poter effettuare le loro operazioni. Una volta effettuati i controlli del caso e una volta compiuta la specifica operazione, il Market compone una struttura dati apposita con le informazioni che si è scelto di stampare a video per quell'operazione, e la spedisce al servizio MonitorX in esso embeddato, attraverso un'apposita Output Port.

MonitorX è una classe Java che viene istanziata nel momento dell'esecuzione del Market (chiamata al costruttore nell'init), inizializza tutti i componenti della finestra e la fa apparire. La finestra è divisa in 3 schermi: sul primo viene stampato l'output ricondicibile ai Player (tutti i Player), sul secondo quello riconducibile al Market, sul terzo quello degli Stock (tutti gli Stock).

Il metodo printOut - che riceve in input la struttura dati proveniente dal Market - implementa degli switch per riconoscere su quale schermo va stampato l'output, e a quale operation vada attribuita la singola chiamata. Viene poi preparato un blocco di testo formattato che viene stampato in fondo all'area di testo prescelta. Se l'operazione è ascrivibile ad un particolare Player o ad un particolare Stock, il nome di questo è indicato a lettere maiuscole.

I Player generano output nei seguenti casi: registrazione, acquisto, vendita;
Il Market genera output nei seguenti casi: registrazione player, registrazione stock, acquisto, vendita, produzione, deperimento;
Gli Stock generano output nei seguenti casi: registrazione, acquisto, vendita, deperimento, produzione;

MonitorX si presta a osservare l'andamento del sistema mentre esso è in esecuzione, o a ripercorrere l'evoluzione di una particolare esecuzione una volta che questa è terminata. In quest'ultimo caso è sufficiente killare Players e StockMng lasciando in vita Market; il sistema è fermo ma la finestra non scompare ed è possibile scorrere gli output fino in cima.

Le istruzioni che invocano MonitorX

Nota: La chiusura del monitor non determina l'interruzione di alcuna parte del sistema.

Struttura filesystem

config/

constants.iol	→ definizioni costanti utilizzate
stocks/	→ file di configurazione di ciascuno Stock
Grano.xml	
Oro.xml	
Petrolio.xml	

embeddedService/

Stock.ol	→ rappresenta singola istanza Stock (embeddata in StocksMng)

interfaces/ → definizioni strutture e interfacce

commonInterface.iol	→ interfaccia comune a tutti i servizi
playerInterface.iol	→ descrive le operazioni che i Player possono invocare sul Market (e le relative strutture)
stockInterface.iol	→ operazioni e strutture tra StocksMng, Stocks e Market

main/

Market.ol	→ gestione Market
player1.ol	→ player tipo “Random”
player2.ol	→ 2 player paralleli con diverse politiche
player3.ol	→ player tipo “Speculativo”
StocksMng.ol	→ Dispatcher / gateway tra le singole istanze di Stocks embeddate ed il Market

Out/ → parte Java per la gestione di MonitorX

jolie.jar
MonitorX$2.class MonitorX.class MonitorX.java
MonitorX$1.class MonitorX$AscoltatoreResize.class MonitorX.jar

Demo

Dopo avere verificato la presenza dei file .xml all'interno della directory *config/stocks* (Stock su sui su vuole attivare la contrattazione) si procedere con l'avvio delle varie componenti.

Avvio indistinto di StocksMng.ol e Market.ol.
L'avvio di Market.ol attiva la dashboard di contrattazione MonitorX su cui cominceranno ad essere visualizzate le operazioni di registrazione, deperimento e produzione dei vari Stock forniti.

Solo dopo avere avviato Market e StocksMng è possibile avviare i Player.

Avvio dei player: player1.ol, player2.ol, player3.ol
L'esecuzione dei Player dà il via alle contrattazioni che verranno mostrate sempre sulla dashboard di contrattazione.

Per terminare le contrattazioni occorre interrompere (ctrl-c) l'esecuzione di Market.ol questo produrrà a catena una eccezione intercettata da *StocksMnsg* e *Players*, i Player visualizzeranno il loro patrimonio accumulato sino a quel momento.

Osservazioni

La scelta di non implementare alcun tipo di priorità sull'accesso alle strutture dati del Market, garantendo un assoluto non determinismo nelle operazioni di scrittura / lettura, viene enfatizzato dal comportamento dello *Speculative Player* ed è riassunto di seguito.

Supponiamo che il Player abbia avviato un numero elevato di operazioni di acquisto o vendita. Esse si posizionano nella coda dei semafori del Player e dello Stock all'interno di Market, in attesa di acquisire i lock.
In maniera non deterministica gli Stock procedono con le operazioni di deperimento e produzione ed aggiornano correttamente le informazioni di disponibilità al loro interno, ma l'invocazione delle operazioni *addStock@Market e destroyStock@Market* (che agiscono sul prezzo dello Stock) potrebbero non essere in grado di acquisire il lock a causa dell'accodamento delle operazioni di *sell* e *buy* innescate dal Player.

Per ovviare al problema si poteva definire una priorità alle operazioni *addStock e destroyStock* e quindi passare da una soluzione *non deterministica* ad una implementazione *deterministica.*

Abbiamo notato un'incongruenza nella determinazione del prezzo degli Stock durante la fase di *production* e, in particolare, nel seguente caso: qualora venga prodotta una quantità pari alla quantità disponibile, al Market sarà comunicato un incremento della quantità del 100% che equivarrà ad un decremento del prezzo del 100%. Il prezzo dello Stock verrà impostato a 10, ovvero il minimo prezzo consentito.
In particolare quando la disponibilità è bassa, è verosimile presupporre che il prezzo sia elevato, ma al verificarsi della casistica prima descritta, il prezzo cala drasticamente al valore di default.

Conclusioni

Durante lo sviluppo il team ha avuto modo di misurarsi con le problematiche legate all'esecuzione di un sistema software che sfrutta la concorrenza, ha studiato, compreso e applicato le soluzioni che a questi problemi offre il linguaggio Jolie.
Oltre ai numerosi incontri e discussioni, la comunicazione è avvenuta via mail, la condivisione del

codice attraverso un repository Git.
Il costante confronto tra i membri del gruppo è stato fondamentale per individuare le imperfezioni più sottili nonché per costruire soluzioni migliori, che spesso emergevano dalla fusione di più soluzioni proposte dai singoli.