Corso di Programmazione Concorrente e Distribuita

Relazione Assignment #2

Simulazione del movimento di N corpi in un piano bidimensionale
Libreria asincrona per analisi di codice sorgente Java basata su Event Loop
Libreria per analisi di codice sorgente Java basata su programmazione reattiva

Alessandro Magnani, Simone Montanari, Andrea Matteucci

Parte 1

Analisi del problema

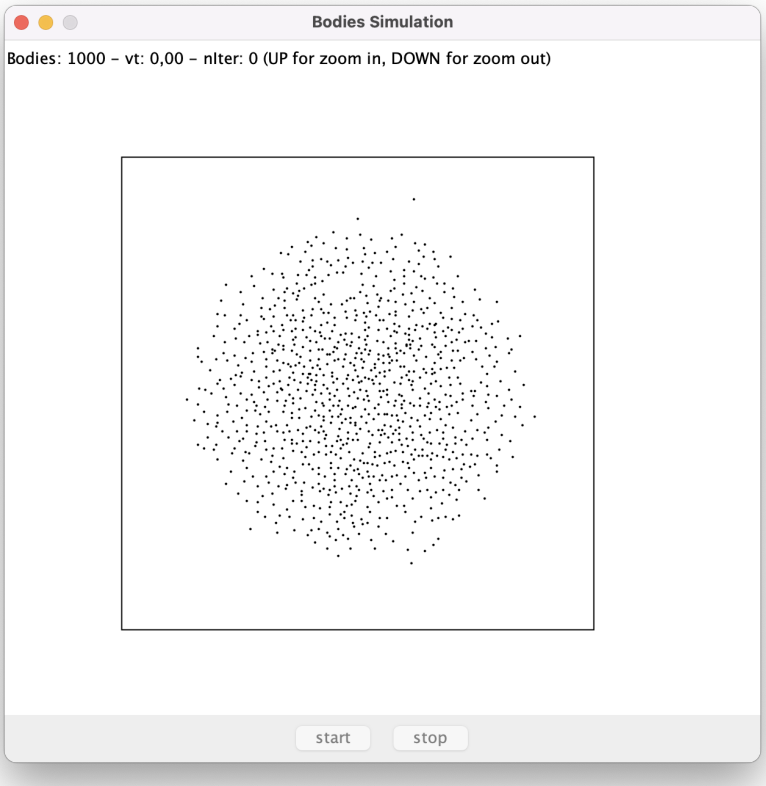
Il problema riguarda la gestione concorrente di un numero N di palline che si muovono in uno spazio bidimensionale, soggette a due tipi di forze:

- Forza repulsiva, esercitata da un corpo sugli altri;
- Forza di attrito, contraria al moto del singolo corpo, ed opposta alla sua velocità.

Nella versione precedente relativa all'Assignment #1, l'algoritmo per poter gestire il movimento delle palline in modo concorrente faceva uso di Thread (calcolo delle forze, aggiornamento delle posizioni e controllo di eventuali collisioni con i bordi).

Dunque, il principale problema da affrontare è quello di adattare la versione concorrente basata sui Thread ad una, sempre concorrente, ma basata sui Task.

È di seguito riportato uno screenshot del sistema in esecuzione.



Architettura proposta e strategia risolutiva

All'interno del Main relativo alla parte 1 è possibile avviare due versioni diverse del programma proposto:

- Una composta dalla GUI ed i relativi bottoni di avvio e terminazione
- Una senza GUI, per la sola misurazione delle performance, ovvero del tempo di esecuzione

Versione senza GUI

All'interno di questa versione si è scelto di realizzare un thread, rappresentato dalla classe *MasterPerformance*, con l'obiettivo di generare e gestire le varie computazioni sulle palline, generando pool di Task di dimensioni pari al numero di Core a disposizione della macchina sulla quale viene avviata l'esecuzione.

All'interno del ciclo principale (con *nSteps* iterazioni), similmente a quanto fatto con il primo Assignment, si è deciso di maneggiare le palline a gruppi utilizzando due tipologie di Task (uno per la prima operazione, la *ComputeTotalForceAndUpdateVelocity*, e uno per la seconda e terza, ovvero *updatePosition* e *checkBoundaryCollision*).

Nello specifico, la pool di Task creata agisce su gruppi di 50 palline per volta, rendendo così l'implementazione più efficiente, dal momento che ogni task esegue piccole operazioni su poche palline per volta. Tuttavia, si è notato che gestirne un numero minore andava a peggiorare le prestazioni.

Vi è la certezza che non si verifichino corse critiche tra i task che modificano la lista di palline, in quanto a seguito di ogni operazione su un gruppo di 50 palline, è possibile asseverare un blocco dell'esecuzione attraverso il metodo *get()* della *Future,* fino a quando queste non sono state computate tutte.

Versione con GUI

Nella costruzione del codice si è seguito il pattern **MVC**, in modo da avere una struttura che ci permettesse un collegamento efficiente tra le diverse parti: Model, View e Controller.

Inoltre si è deciso di utilizzare un **Monitor** per riuscire a interrompere l'esecuzione della simulazione.

La struttura del programma è analoga alla versione senza GUI, i cambiamenti sono dovuti ovviamente al bisogno di stampare a video la simulazione e sono i seguenti:

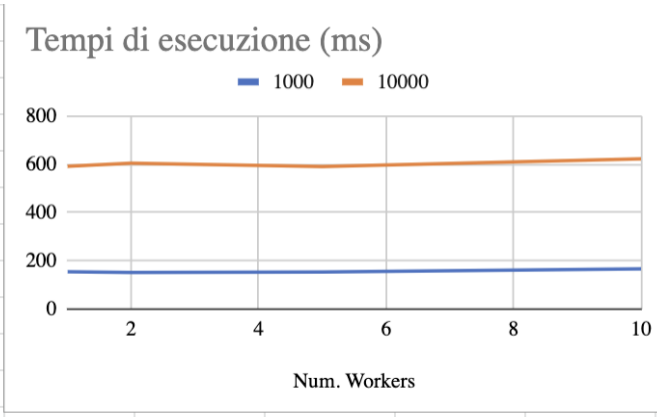
- Il Thread Master (rappresentato dalla classe *MasterGUI*) riceve come parametro anche la variabile (*stopFlag*) che rappresenta il monitor;
- Il corpo del ciclo principale del Master è racchiuso da un controllo (*stopFlag.isStopped()*) per poter interrompere l'esecuzione in modo reattivo e, inoltre, viene eseguita la *display()*;
- I due Task ricevono come parametro la *stopFlag* e all'interno delle loro computazioni sono stati inseriti dei controlli su essa per lo stesso scopo.

Performance

100 corpi

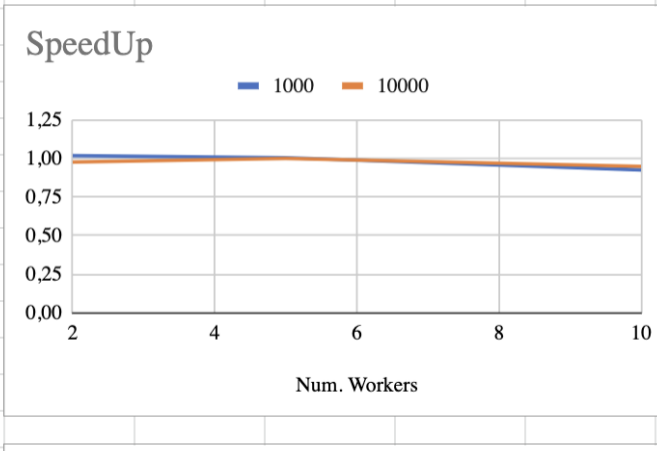
100 corpi
Tempi di esecuzione (ms)

	Iterazioni	
Num. Core	1000	10000
1	153	592
2	150	605
5	152	591
10	165	623



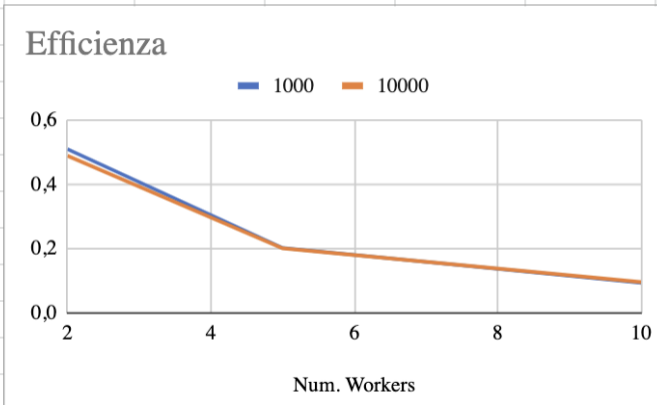
100 corpi
SpeedUp

	Iterazioni	
Num. Core	1000	10000
2	1,02	0,9785123
5	1,0065789	1,0016920
10	0,9272727	0,9502407



100 corpi
Efficienza

	Iterazioni	
Num. Core	1000	10000
2	0,51	0,4892561
5	0,2013157	0,2003384
10	0,0927272	0,0950240

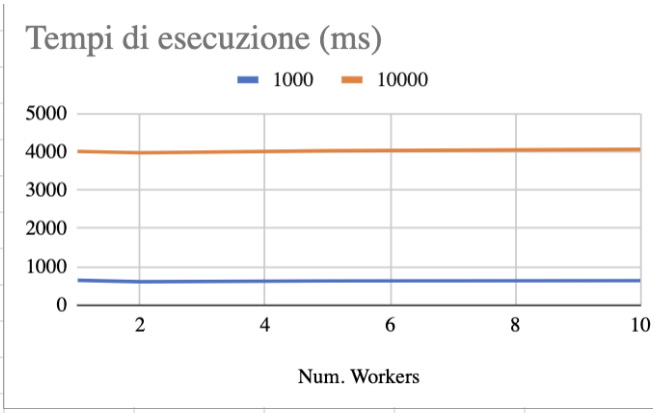


Abbiamo eseguito i test riguardanti le performance con gli stessi valori dell'Assignment #1, al fine di avere un migliore confronto tra le due realizzazioni. Prendendo inizialmente in esame la situazione con 100 corpi, notiamo come in questo caso i tempi e lo SpeedUp si mantengano costanti.
L'efficienza invece diminuisce aumentando il numero di core.

1000 corpi

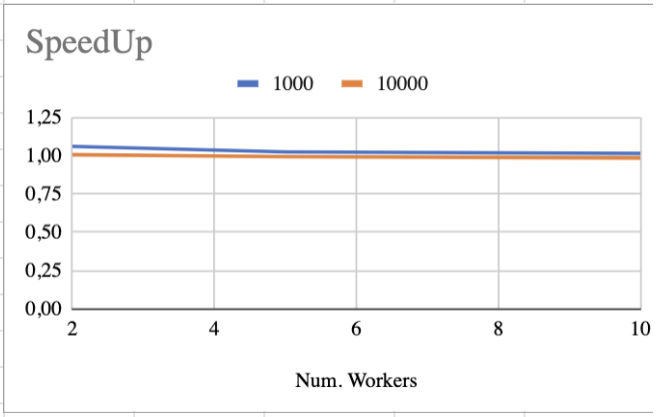
1000 corpi
Tempi di esecuzione (ms)

	Iterazioni	
Num. Core	1000	10000
1	641	4019
2	603	3983
5	624	4036
10	630	4073



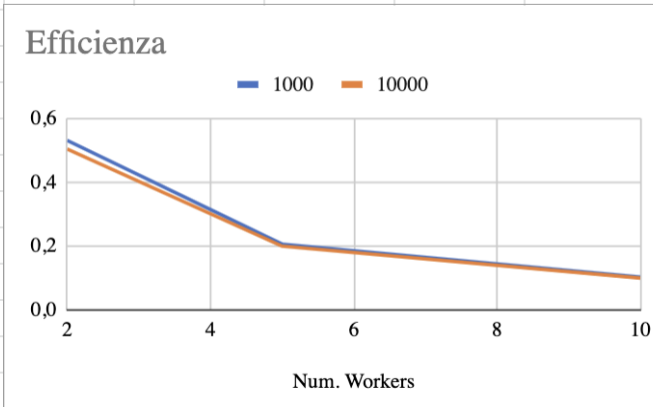
1000 corpi
SpeedUp

	Iterazioni	
Num. Core	1000	10000
2	1,0630182	1,0090384
5	1,0272435	0,9957879
10	1,0174603	0,9867419



1000 corpi
Efficienza

	Iterazioni	
Num. Core	1000	10000
2	0,5315091	0,5045192
5	0,2054487	0,1991575
10	0,1017460	0,0986741

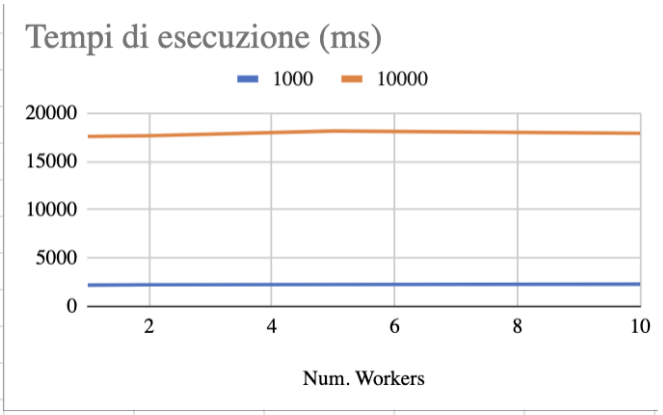


Anche in questo caso notiamo come rimangano costanti i valori di tempo e SpeedUp, con l'efficienza che diminuisce via via che aumentano i core.

5000 corpi

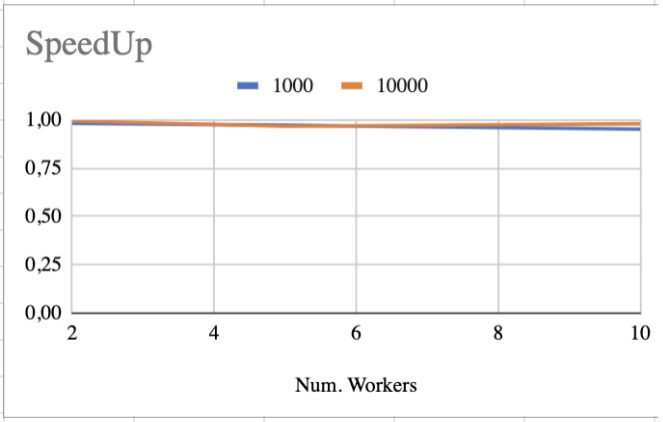
5000 corpi
Tempi di esecuzione (ms)

	Iterazioni	
Num. Core	1000	10000
1	2161	17604
2	2198	17680
5	2218	18181
10	2267	17937



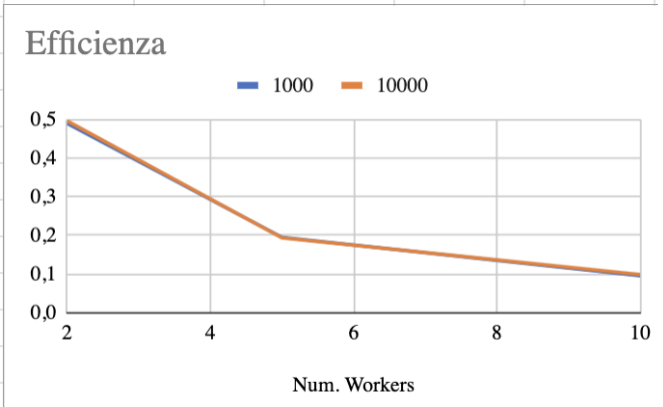
5000 corpi
SpeedUp

	Speedup	
Num. Core	1000	10000
2	0,9831665	0,9957013
5	0,9743011	0,9682635
10	0,9532421	0,9814350



5000 corpi
Efficienza

	Efficienza	
Num. Core	1000	10000
2	0,4915832	0,4978506
5	0,1948602	0,1936527
10	0,0953242	0,0981435



In conclusione, possiamo dire che, in modo analogo a come avviene nell'Assignment #1, l'efficienza diminuisce con l'aumento dei core. Tuttavia, se nel primo Assignment aveva l'efficienza migliore con l'utilizzo di due core, in questo caso siamo riusciti ad ottenere un'efficienza massima di 0,49.

Comportamento del sistema

È di seguito riportato il comportamento del sistema con GUI:

- All'avvio del programma viene istanziato il componente grafico (classe *SimulationView*), composto dai vari pulsanti;

- Una volta premuto il pulsante START, viene inviata una notifica al controller (metodo *notifyStarted*) che provvede a creare un Thread (rappresentato dalla classe *MasterGUI*) per poi iniziare l'esecuzione avviandolo (metodo *start()*);

- Nel corpo del Thread Master vengono inizialmente generate le palline (*nBalls* specificato come parametro).
  Successivamente è presente un ciclo (con *nSteps* iterazioni) rappresentante il funzionamento della simulazione;

- All'interno del ciclo principale vengono eseguiti altri due cicli (in base all'ordine delle operazioni da eseguire sulle palline):
  - il primo contenente la submit dei task che devono svolgere la prima operazione (*computeTotalForce*)
  - il secondo contenente la submit dei task che devono svolgere la seconda e la terza operazione (*updatePosition* e *checkBoundary*)

- Si è scelto di eseguire le computazioni su gruppi di 50 palline alla volta per rendere maggiormente efficiente l'algoritmo.
  All'interno dei cicli sono stati inseriti controlli sul Monitor per interrompere l'esecuzione.

- Alla fine di ogni iterazione del ciclo principale viene visualizzato lo spostamento aggiornato delle palline nella GUI (metodo *display()*);

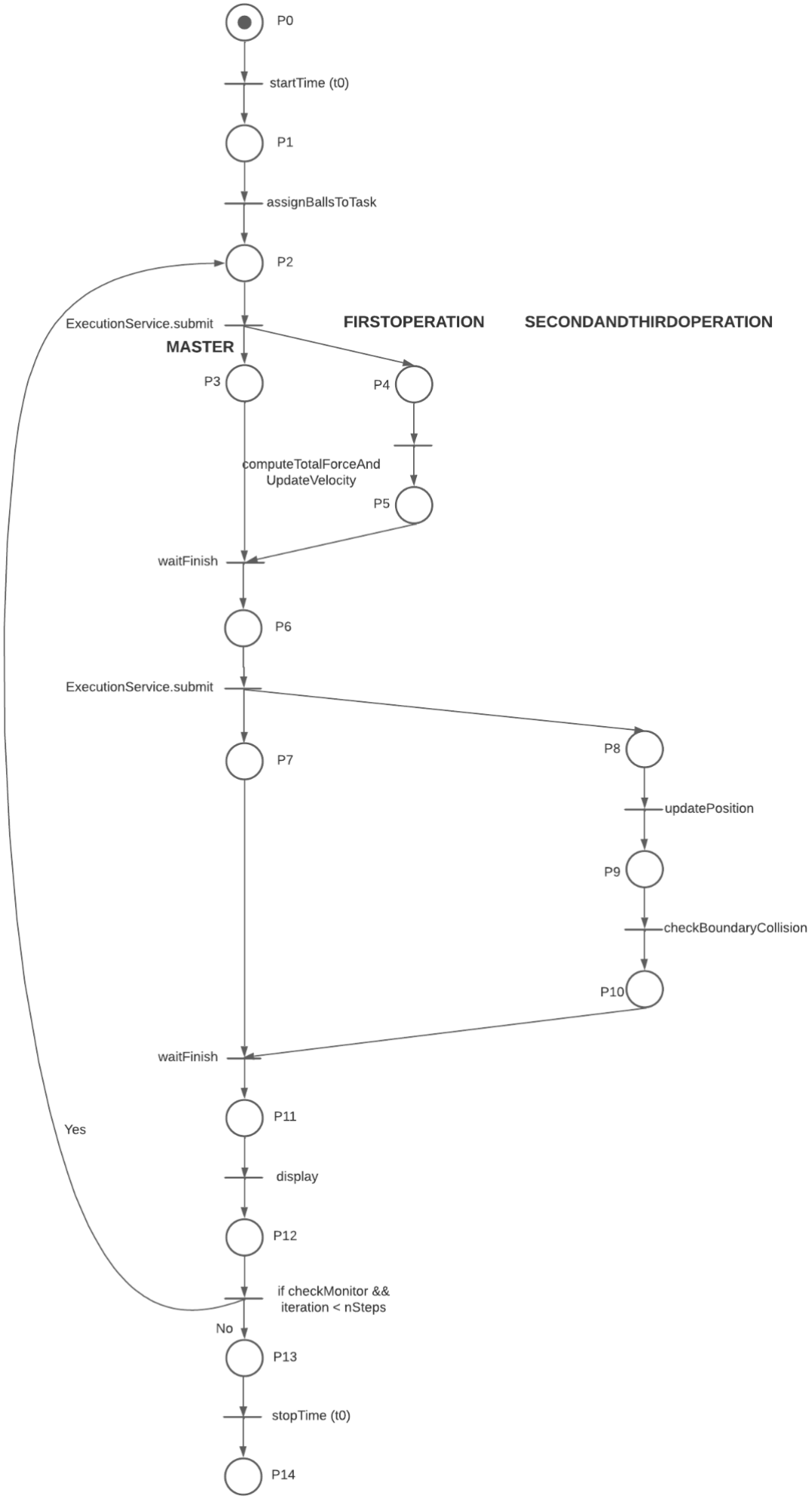
- Durante l'esecuzione, inoltre, è possibile premere il pulsante STOP che avverte il controller di interrompere l'esecuzione (metodo *notify Stopped()*), provvedendo a cambiare lo stato del Monitor (metodo *stop()*);

- Sia in caso di terminazione, sia in caso di interruzione dell'esecuzione, viene stampato in console il tempo impiegato;

La struttura del programma è simile alla versione con GUI, i cambiamenti sono dovuti all'assenza della parte grafica e sono i seguenti:

- Non viene più istanziato il componente grafico ma viene direttamente creato il Thread Master nel Main (classe *MasterPerformance*);
- Non viene utilizzato il Monitor in quanto non occorre la funzionalità di interruzione;
- Nel ciclo principale non occorre visualizzare le palline nella GUI dunque non è presente la *display()*.

Rete di Petri



Parte 2

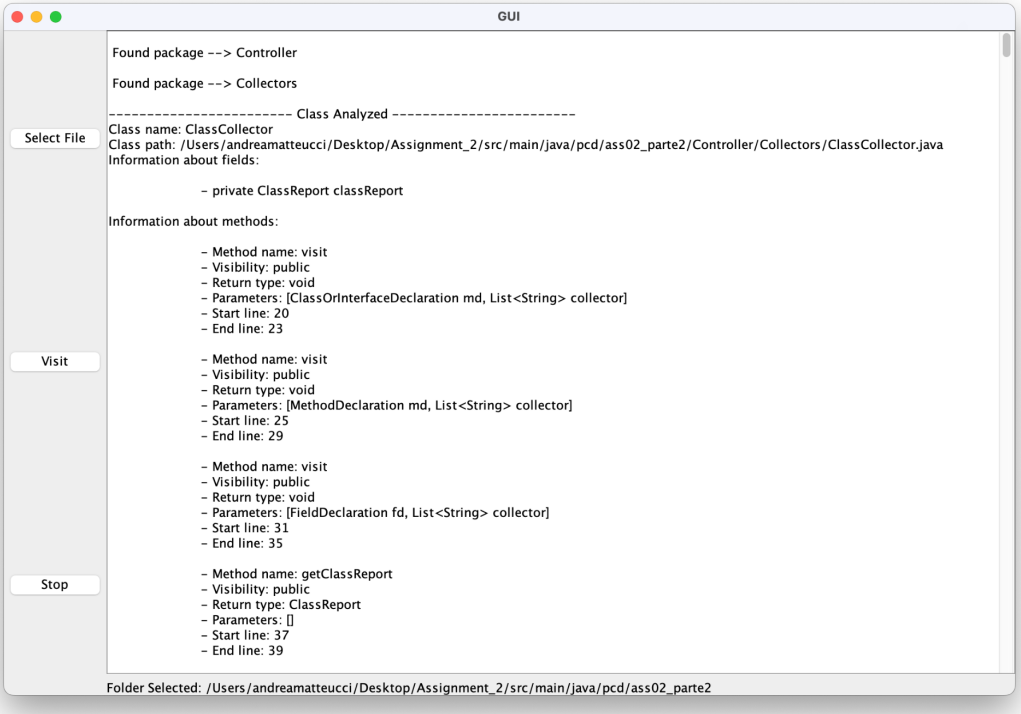
Analisi del problema

L'obiettivo del programma è la realizzazione di una libreria open-source che fornisca metodi asincroni per eseguire il parsing (utilizzando la libreria JavaParser di java) e per l'analisi del codice sorgente Java, con la successiva creazione dell'AST e la navigazione attraverso il pattern Visitor.

La libreria deve fornire i seguenti metodi:

- getInterfaceReport, che consente di ottenere il report di una interfaccia;
- getClassReport, che consente di ottenere il report di una classe;
- getPackageReport, che consente di ottenere il report di un package;
- getProjectReport, che consente di ottenere il report di un progetto;
- analyzeProject, che permette di effettuare un'analisi incrementale, generando eventi sul topic.

È di seguito riportato uno screenshot del sistema in esecuzione, con un esempio di analisi:



Architettura proposta e strategia risolutiva

Punto A

Per quanto riguarda il punto A è stata creata una libreria (rappresentata dalla classe *Library*) che mette a disposizione i metodi sopra citati.

Per rendere asincrono il funzionamento sono state utilizzate delle *Future* per tutte le computazioni all'interno dei vari *executeBlocking()*, così facendo dal Main è possibile recuperare i risultati (detti reports, rappresentati dalle classe *ClassReport, InterfaceReport, PackageReport* e *ProjectReport*) sfruttando il metodo *onComplete()* delle *Future*.

Per quanto riguarda il metodo *analyzeProject()* si è deciso di implementare la versione che sfrutta l'Event Bus di *Vertx*.
Il funzionamento ottenuto è sempre asincrono (grazie alla *executeBlocking()*), ma con la differenza che è stato sfruttato il metodo *deployVerticle()* a due argomenti di Vertx, per poter creare una struttura in cui due agenti (rappresentati dalle classi *AgentSender* e *AgentReceiver*) si potessero scambiare messaggi sul canale (*topic*) specificato come parametro.

Punto B

Il programma è stato costruito seguendo l'approccio del pattern MVC (Model-View-Controller), in modo da riuscire a creare una versione il più efficiente e leggibile possibile.

La differenza sostanziale con il punto A è l'annessione di un **Monitor** atto all'interruzione dell'analisi e direttamente connesso con il pulsante Stop presente nella GUI.

Comportamento del sistema

È di seguito riportato il comportamento del sistema (punto A):

- Viene creata un'istanza di *Vertx*;

- Viene istanziata la libreria richiesta dall'esercizio (*Library*) a cui viene passata l'istanza di *vertx*;

- A questo punto dal main è possibile richiamare i metodi messi a disposizione dalla libreria (*getClassReport()*, *getInterfaceReport()*, *getPackageReport()*, *getProjectReport()*) passando il path del file da analizzare, per poi recuperare il report corrispondente, sfruttando la *onComplete()* delle *Future* (ad esclusione di *analyzeProject()*).

Funzionamento *analyzeProject()* (punto A):

- Dal main viene richiamato il metodo void *analyzeProject()* passando il path del progetto da analizzare;

- Nel corpo del metodo viene eseguita la *deployVerticle(),* in maniera asincrona (essendo all'interno di *executeBlocking()*), passando le due classi che rappresentano gli agenti (*AgentSender* e *AgentReceiver*);

- Funzionamento *AgentSender*
  Sfruttando la classe *File* di Java vengono analizzati tutti file e le directory all'interno del progetto e, ogni volta che si imbatte in un elemento, quest'ultimo viene analizzato, sfruttando le visite (similmente a prima), wrappato in una stringa (sfruttando i metodi della classe *Wrapper*) e pubblicato sul canale (attraverso il metodo *publish()*);

- Funzionamento *AgentReceiver*
  L'agente è in ascolto sul canale e tramite il metodo *consumer()* ogni qualvolta legga un messaggio sul canale, lo stampa a console.

Comportamento del sistema con GUI (punto B):

- Come da prassi con MVC nel main vengono create le istanze di *View* e *ControllerView* passando la view al controller;

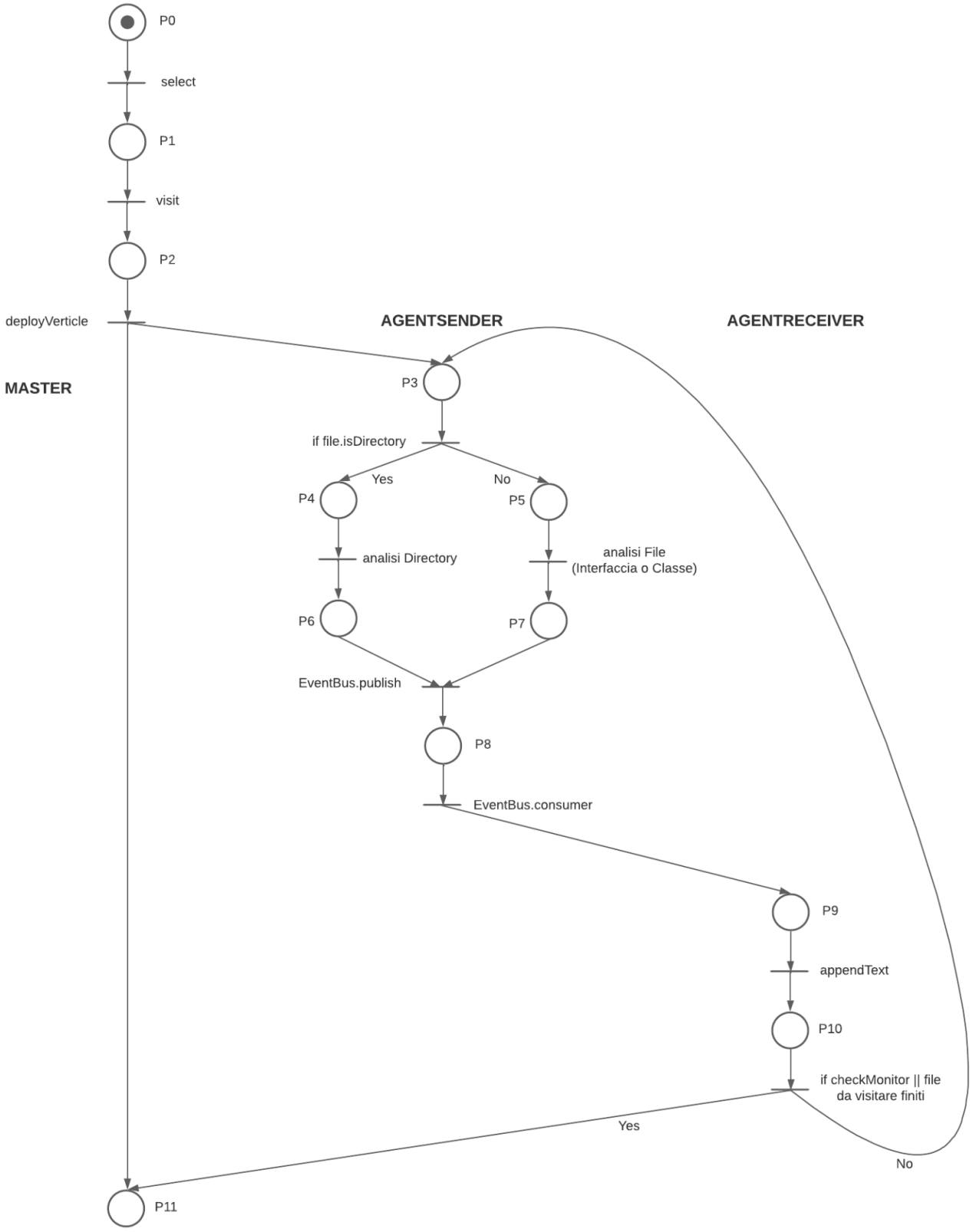
- A questo punto il sistema è in attesa che venga selezionato un progetto;

- Premendo sul pulsante “Select file” si potrà scegliere un progetto su cui poi far partire l’analisi premendo il bottone “Visit”;

- Alla pressione del pulsante di visita, il controller viene notificato (attraverso il metodo *notifyStarted()*). Quest’ultimo crea le istanze di *Vertx* e *Library* e richiama il metodo *analyzeProjectWithGUI()* che, similmente al metodo *analyzeProject()*, tramite la *deployVerticle()* asincrona, istanzia i due agenti (*AgentSender* e *AgentReceiver*).

  Vengono utilizzati costruttori diversi di modo che ogni qualvolta vengano analizzati e pubblicati elementi nel canale, l’agente ricevente possa stampare nella GUI (e non più nella console come in precedenza).

Rete di petri



Parte 3

Analisi del problema

La parte 3 del progetto ha come obiettivo la realizzazione di una soluzione al problema descritto nella parte 2, attraverso, però, un approccio basato su estensioni RxJava. ReactiveX infatti è una API che consente di gestire la programmazione asincrona attraverso degli stream observable.

In particolare, l'obiettivo richiesto era quello di:

- ripensare e ricostruire la libreria della parte 2 con un approccio basato su programmazione reattiva;
- costruire una GUI, che andasse ad utilizzare la nuova libreria fornita.

Si è, quindi, deciso di mantenere la componente front-end della GUI tale e quale a quella della parte 2, modificando esclusivamente la logica back-end, in modo che sfruttasse le estensioni Rx.

Architettura proposta e strategia risolutiva (con focalizzazione sulla concorrenza)

L'obiettivo è la realizzazione di una libreria che svolga gli stessi compiti di quella precedente ma con un'implementazione basata su programmazione reattiva. É stato, quindi, adottato un approccio basato sulla libreria ReactiveX, come consigliatoci a lezione.

Per quanto riguarda i metodi del punto A (ad esclusione di *analyzeProject()*) si è scelto di utilizzare la classe *Flowable* (per l'esattezza *Flowable.fromCallable()*) in quanto vi era la necessità di restituire in output solamente un elemento per visita, come per esempio un ClassReport o un InterfaceReport.
Mentre, per quanto riguarda il metodo *AnalyzeProject()* e la parte B, è stato utilizzato *Observable* (nello specifico la factory *Observable.create()* che ci ha permesso di gestire le chiamate a cascata, in modo simile a degli *Stream*) in quanto occorre implementare un meccanismo con il quale stampare in modo incrementale, a differenza di come fatto in precedenza dove era sufficiente restituire in uscita un solo elemento per ogni visita.

Comportamento del sistema

È di seguito riportato il comportamento del sistema (punto A):

- Inizialmente viene creata un'istanza di *Vertx*;

- A questo punto dal main è possibile richiamare i metodi messi a disposizione dalla libreria passando il path dell'oggetto da analizzare, per poi recuperare il report tramite il metodo *subscribe()* dei *Flowable* (ad esclusione di *analyzeProject()*);

Funzionamento *analyzeProject()* (punto A):

- Dal main viene richiamato il metodo void *analyzeProject*(), passando il path del progetto da analizzare;

- Nel corpo del metodo viene generata un'istanza della classe *ReportStreamBuilder* (passando un'istanza di *File*) e i relativi metodi *createStringStream()* e *subscribe()*;

- Il metodo *createStringStream()* crea un oggetto *Observable* grazie al quale viene visitato il progetto (similmente a quanto fatto nel punto 2). Ogni elemento incontrato viene analizzato, wrappato e emesso nel flusso (metodo *onNext()*);

È di seguito riportato il comportamento del sistema (punto B):

- Nel Main vengono istanziati *View* e *Controller*;

- A questo punto il sistema è in attesa che venga selezionato un progetto (tramite il pulsante *Select File*) e che venga fatta partire l'analisi (tramite il pulsante *Visit*);

- Una volta fatta partire l'analisi, il *Controller* viene notificato della pressione del pulsante (metodo *notifyStarted()*) e come diretta conseguenza, genera un Thread Master (classe *Master*) e lo esegue attraverso il metodo *start()*;

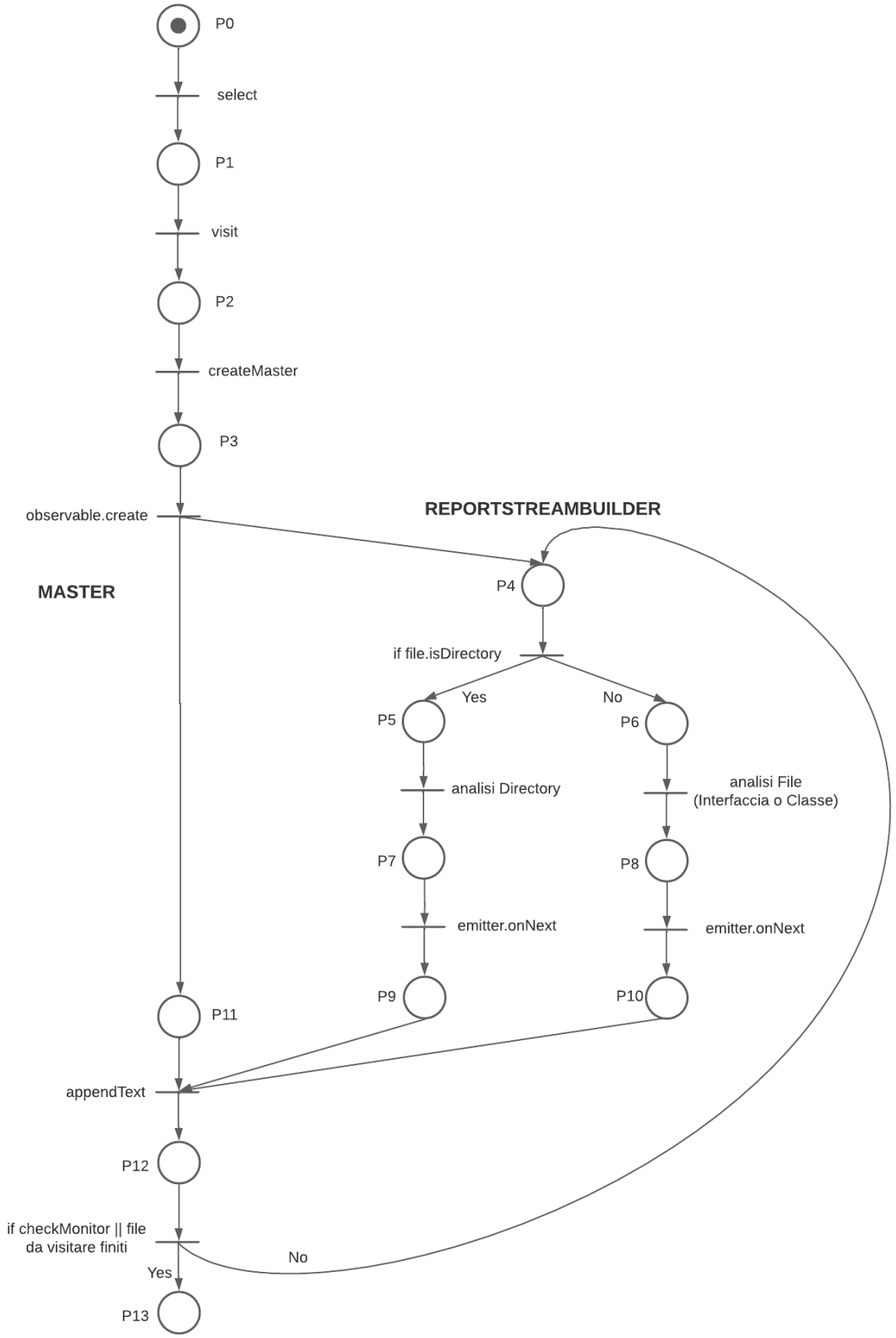
- Nel corpo del Thread Master viene istanziato un oggetto *ReportStreamBuilder* (passando il Monitor per permettere l'interruzione reattiva) e viene sfruttata la

factory chiamando il metodo *createStringStream()*;

- A questo punto l'oggetto *ReportStreamBuilder* inizia a emettere i report wrappati nel flusso;

- Nel Thread Master, successivamente alla creazione dello Stream Builder, viene eseguita la *subscribe()* dell'*Observable* e per ogni elemento incontrato esso viene inserito nella GUI (metodo *appendText()*) riuscendo ad ottenere la stampa incrementale richiesta.

Rete di Petri



Conclusioni

Nella realizzazione di questo Assignment, abbiamo trovato maggiore difficoltà nella costruzione della seconda parte. Per questo motivo, siamo arrivati a migliorare via via l'implementazione, cercando però di mantenere un approccio MVC che permettesse di avere sempre una migliore visione d'insieme della struttura del programma.

In conclusione, siamo dell'idea che la realizzazione di questo secondo Assignment ci abbia aiutato a comprendere meglio i concetti teorici spiegati durante le lezioni.