Sommario

1 Analisi

1.1 Requisiti

Il software progettato realizza la simulazione di un sistema solare in 2D con il quale l'utente ha la possibilità di interagire, creando e modificando la simulazione a proprio piacimento. E' infatti possibile l'aggiunta di nuovi corpi celesti, i quali interagiranno dinamicamente con i corpi preesistenti ed è presente un sistema per la gestione degli impatti tra essi.
L'applicazione si pone come obiettivo di dare supporto alla comprensione del comportamento di un sistema solare nel tempo.

Requisiti concordati

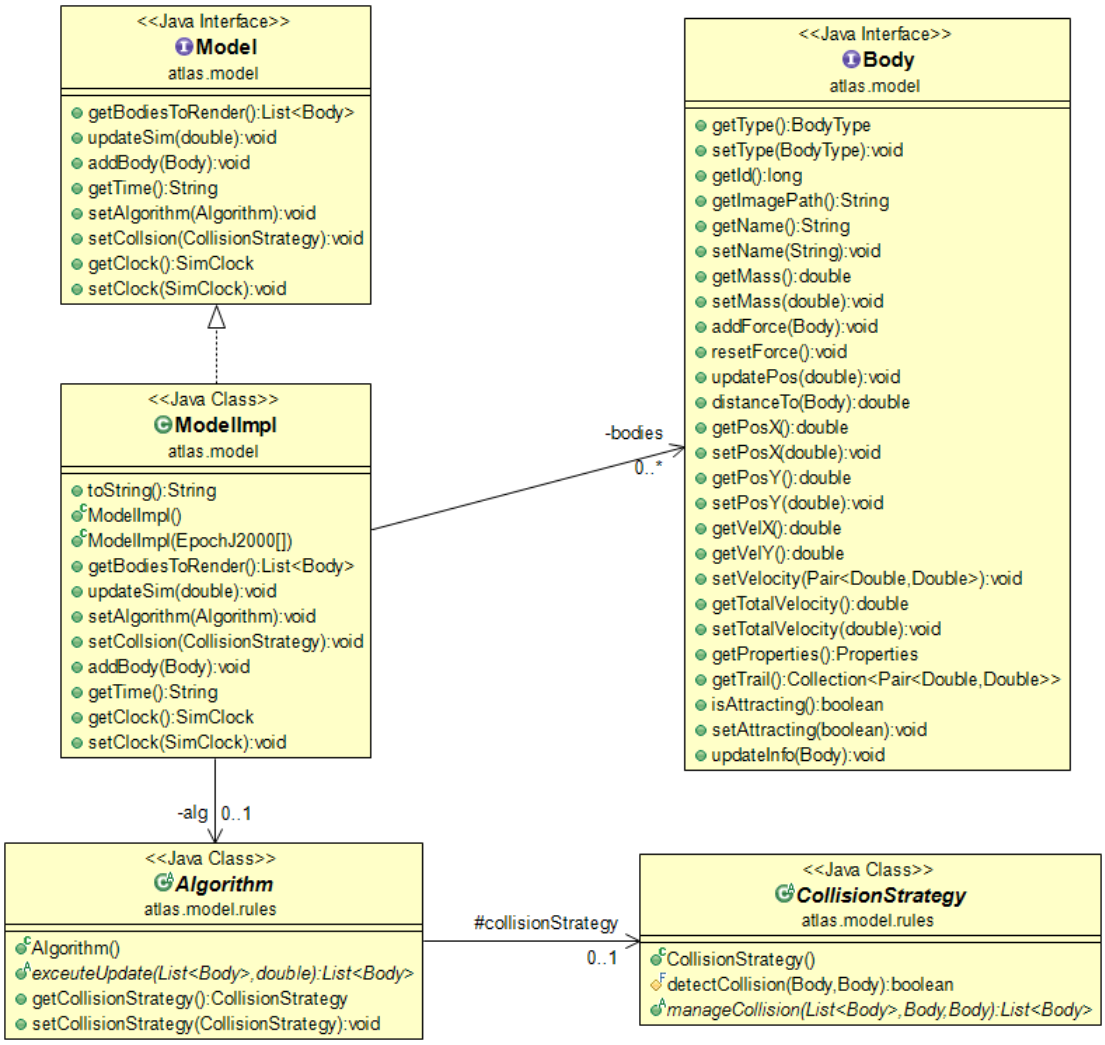
-Creazione di un nuovo sistema solare-Inserimento di nuovi corpi all'interno di un sistema preesistente
-Calcolo delle orbite dei pianeti
-Modifica delle proprietà dei corpi
-Variare la velocità della simulazione, modificando anche l'unità di misura del tempo
-Salvare/caricare una simulazione o un singolo corpo
-Zoom in dettaglio su un corpo, mostrando informazioni aggiuntive
-Gestione dell'impatto tra corpi e la generazioni di frammenti

1.2 Analisi del modello del dominio

ATLAS dovrà una effettuare una “N-body simulation”, ovvero gestire un arbitrario numero di corpi dotati di massa che si attraggono gli uni con gli altri, calcolandone le coordinate cartesiane (x, y) in relazione alle forze esercitate calcolate tramite la legge di gravitazione universale.

La difficoltà principale sarà quella di riuscire mantenere un livello di precisione il più possibile elevato, la quale dipende strettamente dal numero di corpi coinvolti nel calcolo delle forze e dal time stamp utilizzato nell’aggiornamento, ovvero di quanto (tempo) aggiornare. In particolare la precisione aumenta all’aumentare dei corpi considerati e al diminuire del time stamp (deve essere positivo), questo inevitabilmente significa che le performance decadono all’aumentare della precisione a parità di tempo aggiornato. L’obiettivo è quello garantire un’esperienza d’uso fluida, cioè, dal punto di vista delle performance, riuscire ad ottenere almeno 20-30 FPS con decine di corpi su schermo. Inoltre si dovranno gestire le collisioni fra i corpi in modo realistico, questo richiederà ricerche che non potranno essere

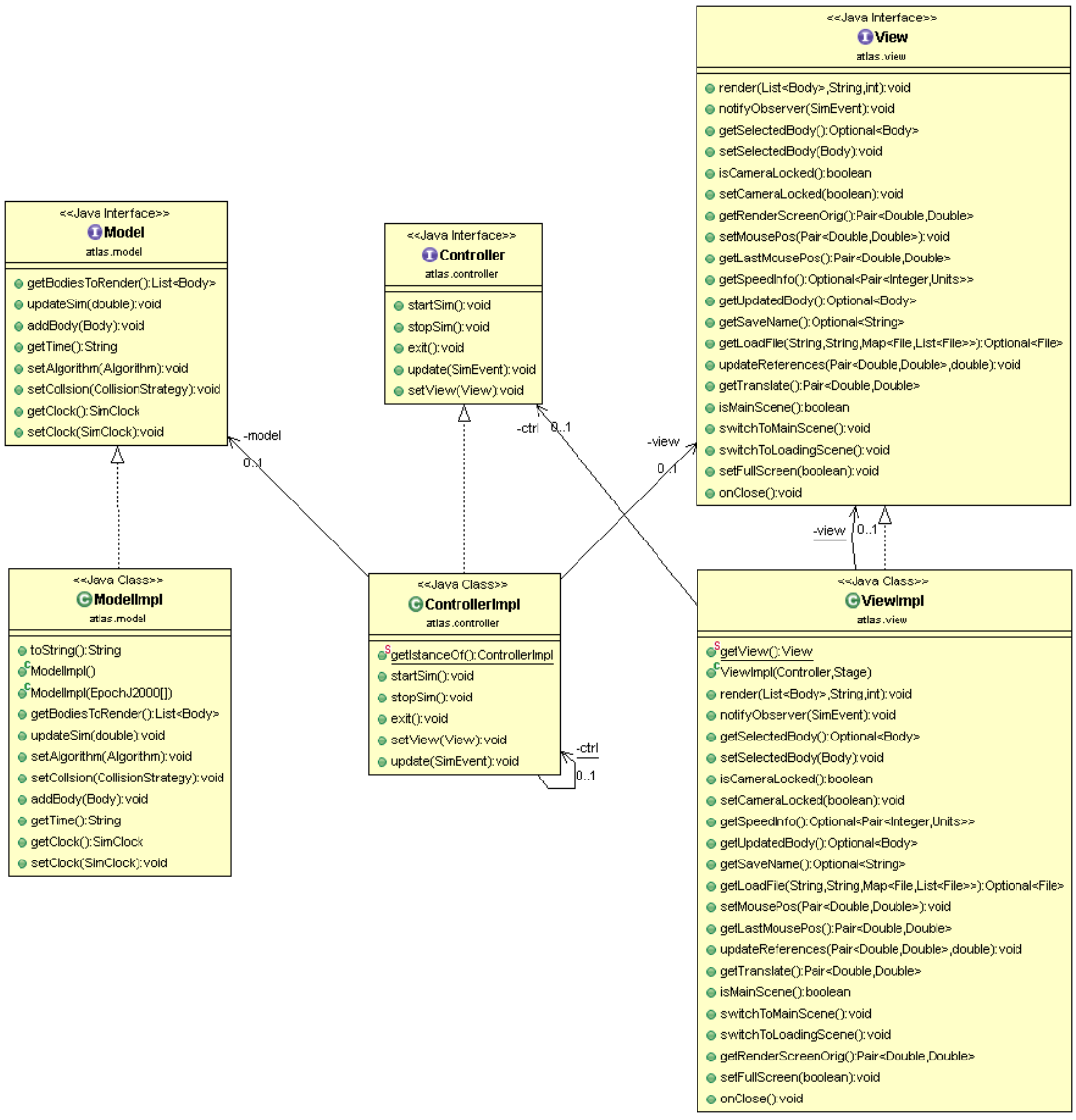
effettuate all'interno del monte ore previsto, perciò si cercherà di approssimare nel migliore dei modi un comportamento adeguato e sarà oggetto di futuri lavori per miglioramenti.



2 Design

2.1 Architettura

Come pattern architetturale abbiamo utilizzato il Pattern MVC, per mantenere separati i compiti svolti dal Modello View e Controller. (Vedi figura UML)
L'applicazione viene avviata tramite la classe ApplicationLauncher, la quale si occupa di creare una View ed un Controller, il quale a sua volta crea il Model. Utilizzando questa strategia la View resta completamente indipendente, avendo il solo compito di mostrare l'output; il Controller accetta un oggetto che implementi l'interfaccia View. Per la notifica degli input dell'utente abbiamo utilizzato il Pattern Observer, tutto ciò rende riutilizzabile il codice rendendo possibile un possibile cambiamento totale della View senza condizionare gli altri componenti software.



2.2.1 Design dettagliato model

Ma Xiang Xiang

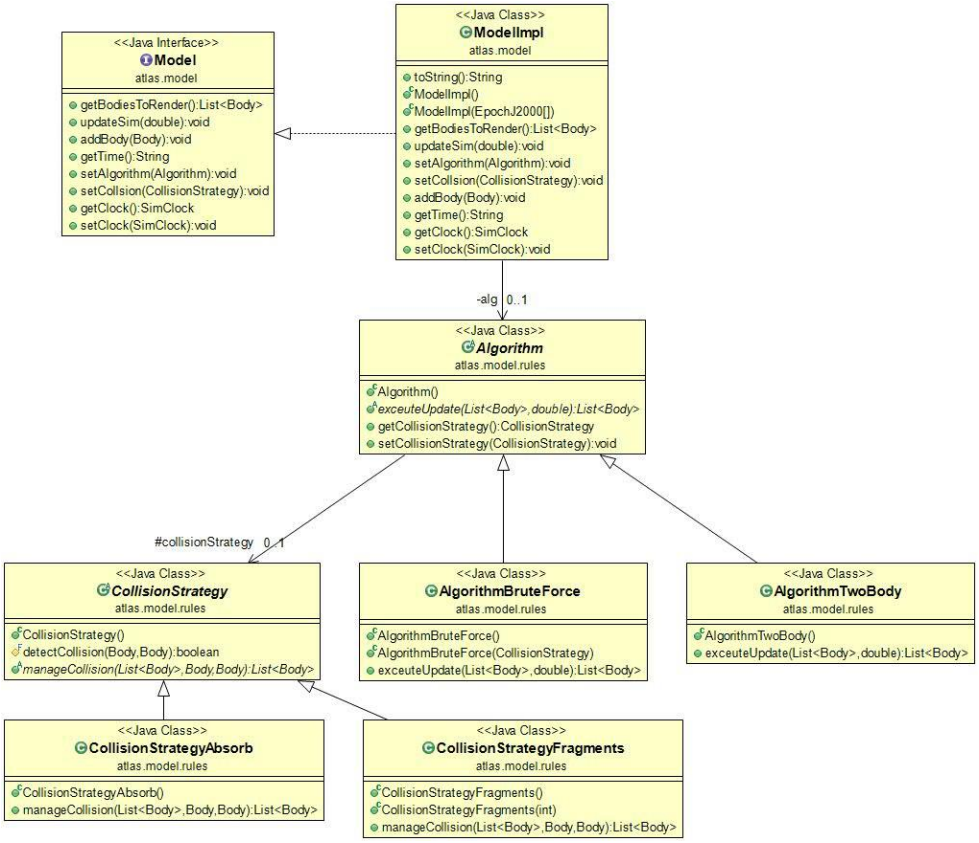
MODEL

Come già detto nell'analisi dei requisiti, il model deve essenzialmente occuparsi di eseguire la N-body simulation e di restituire i corpi con le modifiche al controller. Tenendo conto dei problemi esposti precedentemente sono stati implementati due algoritmi (con la possibilità di aggiungerne altri) per la gestione dell'aggiornamento:

- Brute force = per ogni corpo si sommano le forze di tutti gli altri corpi.
  - Vantaggi:
    - Semplice da implementare.
    - Possibilità di individuare le collisioni tra i corpi.
    - La precisione dipende solamente dal time stamp.
  - Svantaggi:
    - Ha una complessità computazionale uguale a $O(n^2)$.
    - Numero limitato di corpi in concorrenza (circa 100).

- Two-body = si semplifica n-body, sommando solo la forza dei corpi più influenti.
  - Vantaggi:
    - Estremamente più veloce di Brute force, complessità = $\Omega(n)$.
    - Possibilità di gestire centinaia/migliaia di corpi contemporaneamente.
  - Svantaggi:
    - No collisioni.
    - Preciso solamente in alcuni casi (es sistema solare).

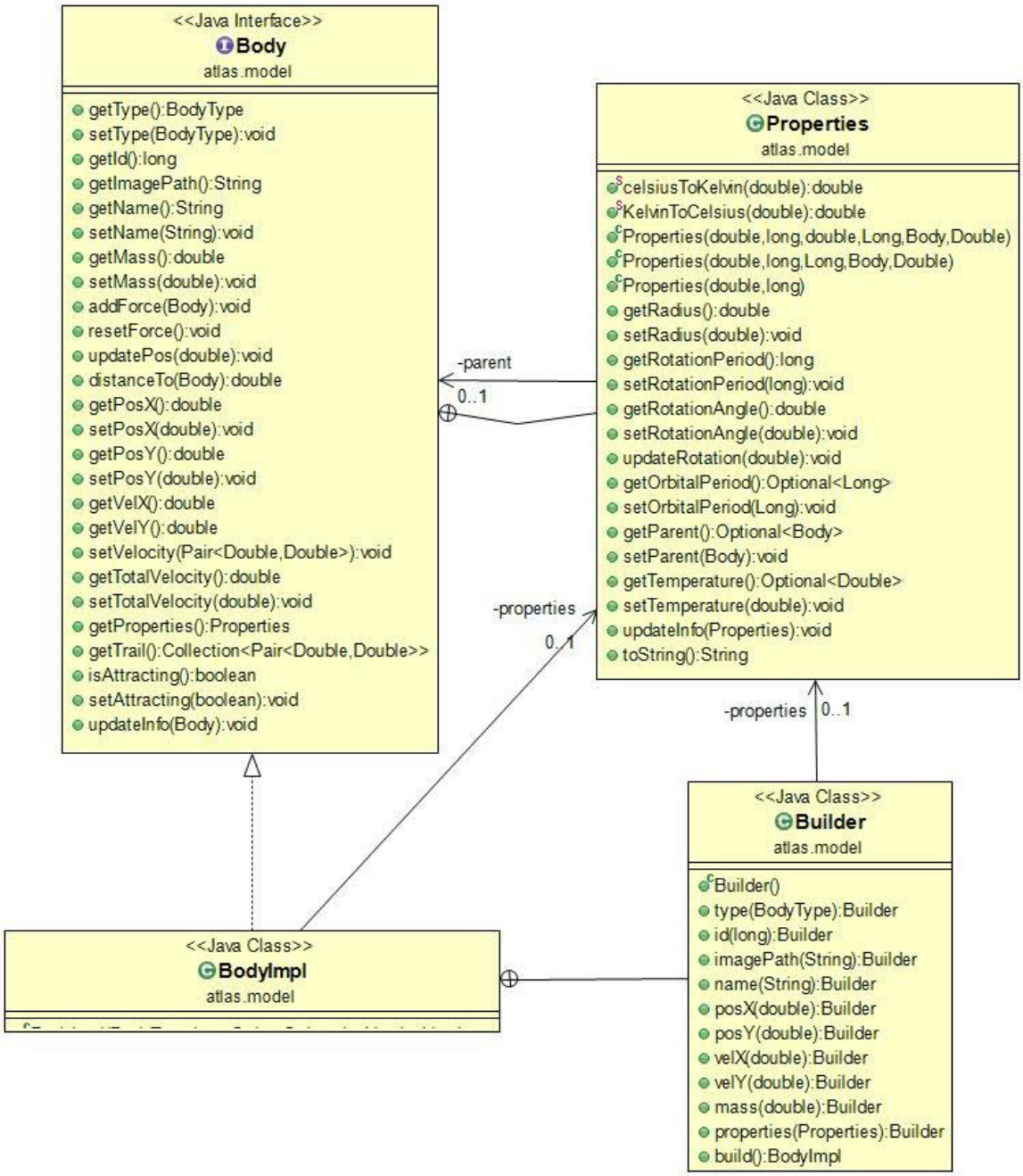
Questi due algoritmi sono implementati nelle rispettive classi, che estendono una classe astratta (Algorithm). All'interno di quest'ultima è presente un oggetto CollisionStrategy che rappresenta l'algoritmo di collisione da eseguire nel caso di collisioni. Ho scelto di usare un pattern strategy sia per i algoritmi che per le collisioni, in modo da renderli intercambiabili e soprattutto espandibile per future aggiunte senza modificare il codice del model.



All'interno del model le classi Algorithm e CollisionStrategy sono entrambe estese da più classi, per questo motivo è stato ritenuto opportuno inserirle in un subpackage .model.rules, rendendo più navigabile il package atlas.model.

Durante lo sviluppo mi sono accorto del elevato numero di campi necessari all'implementazione della classe Body, che creava i seguenti problemi: bassa leggibilità del codice, ridondanza eccessiva di getters, setters e costruttori. Per arginare a questi problemi ho adottato due provvedimenti:

- Separazione dei campi secondari (quelli meno usati e non essenziali per il funzionamento) da quelli primari tramite la creazione di una classe annidata di nome Properties nell'interfaccia Body perché aumenta l'incapsulamento in quanto è usato solamente da una classe (BodyImpl).
- Pattern builder in BodyImpl per facilitare la creazione di nuovi corpi.

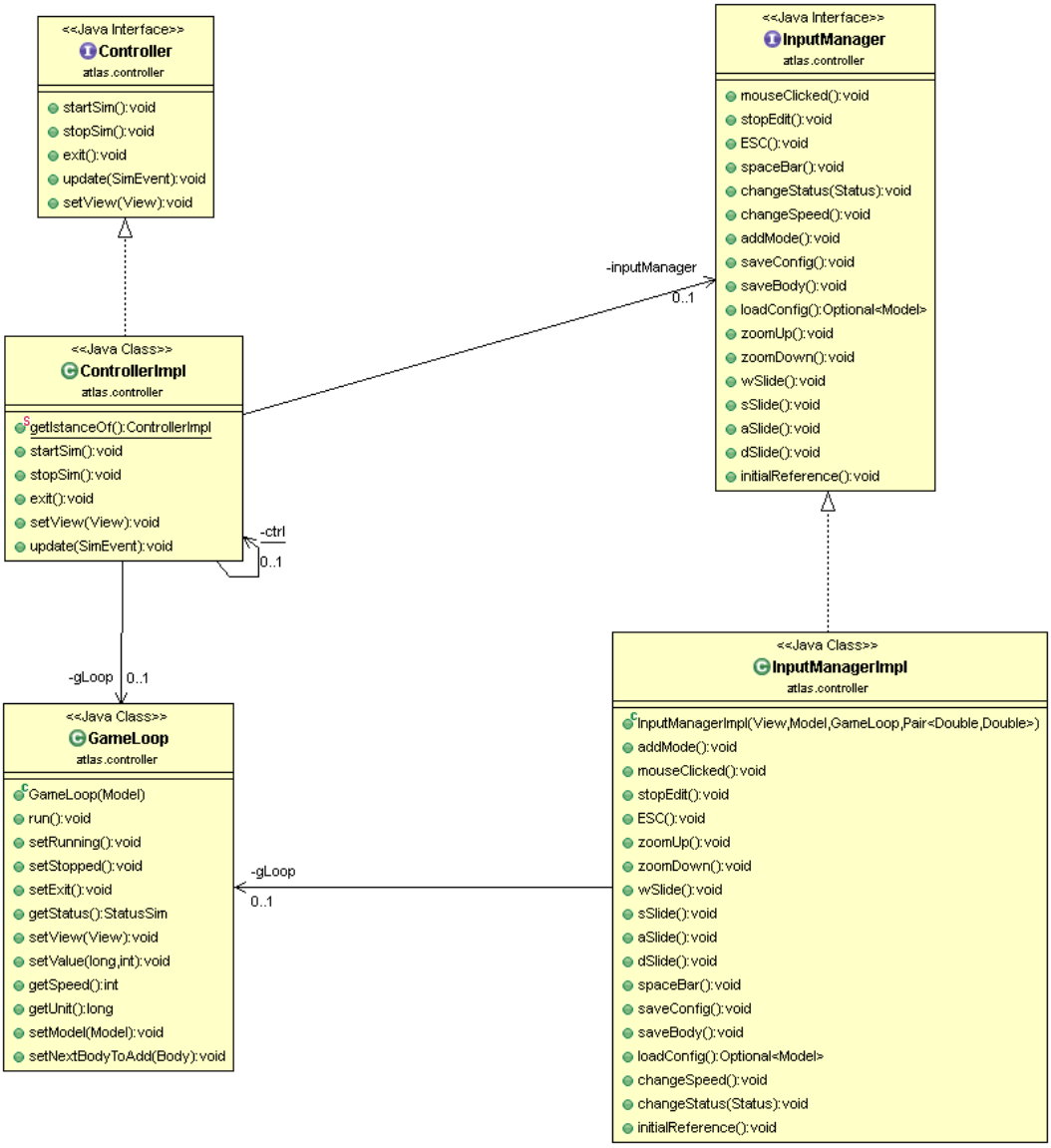

<<Java Interface>>
Body
atlas.model
getType():BodyType
setType(BodyType):void
getId():long
getImagePath():String
getName():String
setName(String):void
getMass():double
setMass(double):void
addForce(Body):void
resetForce():void
updatePos(double):void
distanceTo(Body):double
getPosX():double
setPosX(double):void
getPosY():double
setPosY(double):void
getVelX():double
getVelY():double
setVelocity(Pair<Double,Double>):void
getTotalVelocity():double
setTotalVelocity(double):void
getProperties():Properties
getTrail():Collection<Pair<Double,Double>>
isAttracting():boolean
setAttracting(boolean):void
updateInfo(Body):void
<<Java Class>>
Properties
atlas.model
celsiusToKelvin(double):double
KelvinToCelsius(double):double
Properties(double,long,double,Long,Body,Double)
Properties(double,long,Long,Body,Double)
Properties(double,long)
getRadius():double
setRadius(double):void
getRotationPeriod():long
setRotationPeriod(long):void
getRotationAngle():double
setRotationAngle(double):void
updateRotation(double):void
getOrbitalPeriod():Optional<Long>
setOrbitalPeriod(Long):void
getParent():Optional<Body>
setParent(Body):void
getTemperature():Optional<Double>
setTemperature(double):void
updateInfo(Properties):void
toString():String
-parent
0..1
-properties
0..1
-properties
0..1
<<Java Class>>
BodyImpl
atlas.model
<<Java Class>>
Builder
atlas.model
Builder()
type(BodyType):Builder
id(long):Builder
imagePath(String):Builder
name(String):Builder
posX(double):Builder
posY(double):Builder
velX(double):Builder
velY(double):Builder
mass(double):Builder
properties(Properties):Builder
build():BodyImpl

2.2.2 Design dettagliato Controller

Andrea Bondanini

CONTROLLER

Il Controller server da coordinatore e viene interposto tra la View e il Model. Siccome deve gestire un numero elevato di Input distinti, esso è stato progettato suddividendo tutte le funzioni in due classi: ControllerImpl (che implementa l'interfaccia Controller) e InputManagerImpl (che estende l'interfaccia InputManager). In ControllerImpl sono presenti tutti i metodi principali per la gestione dello stato della simulazione, InputManagerImpl invece contiene i metodi relativi sia agli input dell'utente che alla gestione del salvataggio/caricamento dei files. Il Controller viene creato tramite *getIstanceOf()* (Pattern Singleton) e a sua volta crea il Model e il GameLoop. Il GameLoop è un Thread che si occupa di chiamare ciclicamente il metodo per il calcolo delle posizioni di tutti i corpi (Body) presente nel Model e della renderizzazione grafica. Tutto ciò può richiedere un tempo variabile di calcolo (soprattutto se sono presenti tanti corpi), perciò è possibile che il gameloop "skippi" un certo numero di frame se viene impiegato troppo tempo per eseguire l'update della simulazione, oppure il ciclo viene stoppato per mantenere il frame rate costante. Per mantenere indipendenti View e Controller abbiamo utilizzato il Pattern Observer; il metodo update accetta un Evento dalla View tramite la *notify* e si occupa di chiamare il metodo tramite InputManager.



2.2.3 Design dettagliato view

Ma Xiang Xiang + Andrea Bondanini

VIEW

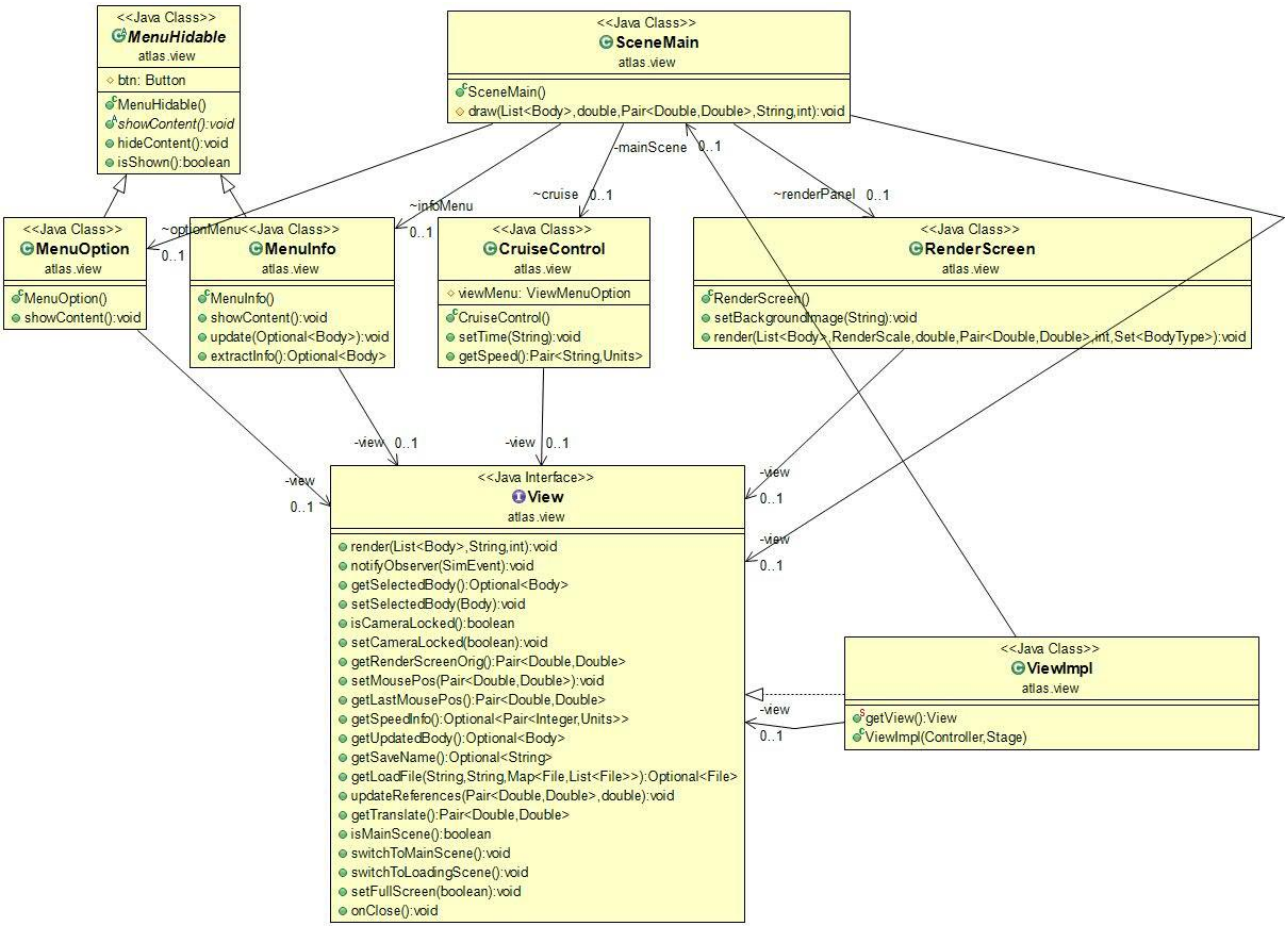
Per la realizzazione della view abbiamo scelto di usare la libreria JavaFX in quanto più si addiceva alle nostre esigenze, ovvero una GUI dall'aspetto moderno facilmente personalizzabile.

L'aspetto dell'interfaccia utente è ispirata a quella di "Universe Sandbox 2", un gioco disponibile su steam.

La GUI è composta da una schermata (SceneLoading) di caricamento, usata all'avvio in attesa che la schermata principale (SceneMain) sia pronta. Quest'ultima quindi,

costituisce l’intera interfaccia usabile. Al suo interno ci sono tutti i comandi disponibili all’utente sotto forma di pulsanti e menu.

La View offre delle funzioni e comunica con il controller tramite la *notify,* per non creare dipendenze dirette tra le parti. (Pattern Observer) Ciò rende possibile una possibile sostituzione futura o ampliamento della View senza creare complicazioni.



Ma Xiang Xiang

Mi sono occupato della realizzazione dei pannelli, del loro aspetto e integrazione all'interno di un'unica scena. Ho iniziato dal panello più importante, ovvero RenderScreen (dove avviene il rendering). Essa è organizzata in layers con 3 livelli gerarchici, dove il quello più alto sovrascrive quelli inferiori. In particolare il livello più basso è un canvas dove vengono disegnate le scie, metre i livelli superiori ospitano le immagini e i labels con i nomi. Inoltre ho realizzato un sistema di caching per le immagini e un algoritmo second chance per rimuovere quelle non più usate.

Il pannello principale è un BorderPane, dove al centro è posto RenderScreen, nella parte inferiore il CruiseControl con i comandi principali e infine ai lati due menu apribili tramite l'azione di un pulsante.

I pannelli sono stati sviluppati tenendo presente della possibilità di ridimensionare la finestra come se fosse una finestra normale, perciò gli elementi al suo interno si devono adattare alle nuove dimensioni. Per ottenere questo risultato ho usato una feature di JavaFX, ovvero il binding dell'altezza e della larghezza.

Bondanini Andrea

Mi sono occupato del collegamento tra tutti i componenti attivabili della view da parte dell'utente e la gestione delle periferiche (mouse e tastiera). Siccome il nostro software prevede molte tipologie di input ho sviluppato una Enumerazione SimEvent per la gestione logica delle interazioni. Ho deciso di utilizzare degli stati in cui collocare la simulazione in determinati periodi (Es: fasi di adding, edit, dragging ecc ecc) perché alcuni input (come il click del mouse) svolgono azioni differenti a seconda della situazione. Mi sono occupato anche delle modifiche estetiche tramite CSS.

3 Sviluppo

3.1 Testing automatizzato

Sono stati realizzati test automatizzati in Junit che riguardano il dominio del model, ovvero test per la creazione di corpi, collisioni e correttezza dei calcoli delle forze.

Data la poca prevedibilità nel tempo dei sistemi come l'N-body, abbiamo testato manualmente la stabilità delle orbite.

3.2 Metodologia di lavoro

Lo sviluppo è stato complicato a causa dell'abbandono del progetto da parte dell'incaricato alla View. Inizialmente il piano di lavoro era così suddiviso:

-Taddei Nicola (View) Ritirato

-Bondanini Andrea (Controller)

-Ma Xiang Xiang (Model)

A causa di questo inconveniente, è risultato necessario ridisegnare tutta la View a progetto in corso. Per quanto riguarda le componenti del Controller e Model il lavoro è stato abbastanza fluido e senza intoppi; tra componenti del gruppo ci siamo accordati sui metodi da implementare nelle interfacce di ogni parte software. La View è stata svolta parallelamente da Ma e Bondanini; lavorando su un DVCS sugli stessi file si sono create spesso "nuove teste" e conflitti. Tuttavia siamo riusciti a lavorare su singolo Branch. Per rendere il codice chiaro e navigabile, abbiamo utilizzato una suddivisione di esso in package. Il gruppo ha utilizzato il DVCS Mercurial ed è stato utilizzato un repository BitBucket.

3.3 Note di Sviluppo

Nel package *atlas.Utils* è presente la classe *Pair* realizzata da *Mirko Viroli* e *Danilo Pianini*. Per quanto riguarda il CSS sono stati consultati alcuni tutorial online.

Per gli algoritmi di N-body si è fatto riferimento al seguente link. Invece per i dati abbiamo utilizzato il database di JPL Horizons della nasa.

4 Commenti Finali

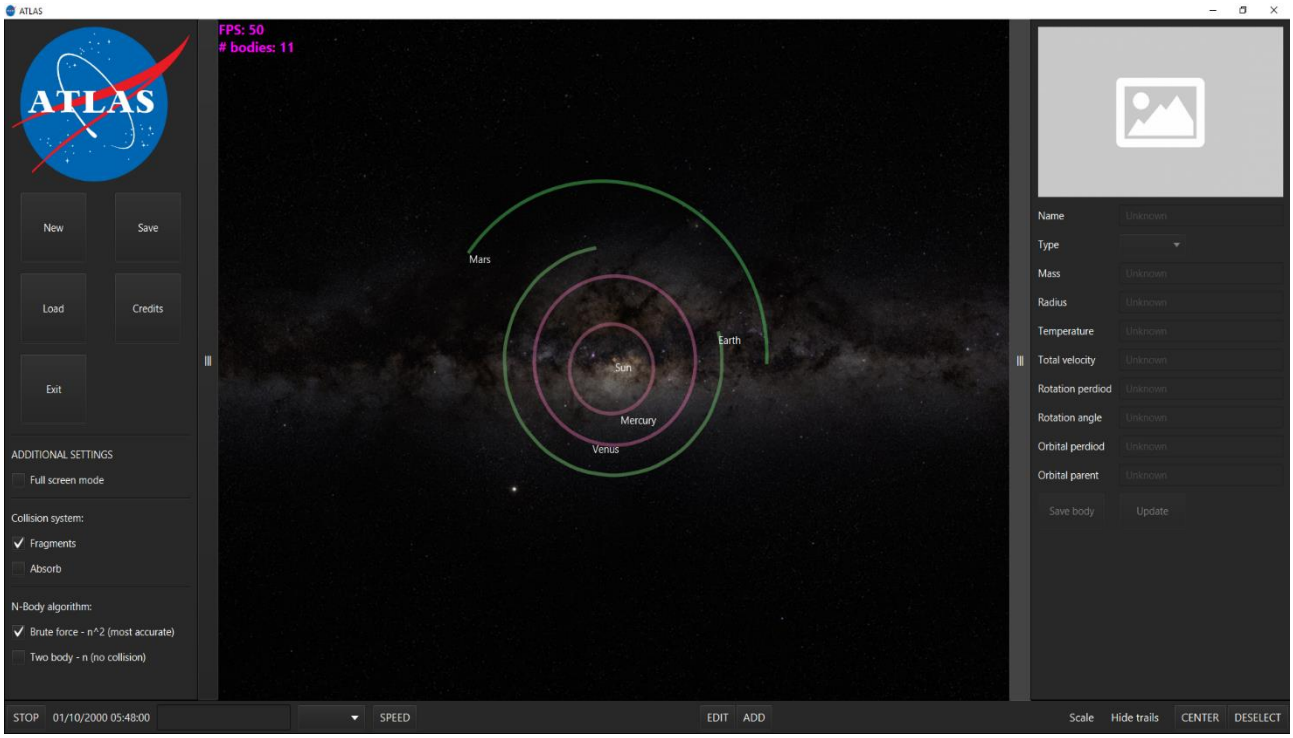
4.1 Autovalutazione

Il gruppo si ritiene abbastanza soddisfatto del lavoro svolto, nonostante alcune features opzionali non sono state implementate per mancanza di tempo (raggiunto il limite delle ore). Il software è abbastanza flessibile ed estendibile, tenendo ben separate tutte le parti software. L'aggiunta di funzionalità o la sostituzione della view saranno implementabili efficacemente senza problemi complessi. Abbiamo utilizzato una suddivisione in package abbastanza chiara, che ha facilitato il lavoro a livello corale.

-**Bondanini Andrea**: Sono complessivamente soddisfatto del mio lavoro. Il codice abbastanza flessibile non dovrebbe creare problemi per eventuali aggiunte in futuro. Ho capito quanto sia importante "disegnare" lo scheletro del software di iniziare a sviluppare. Molto importante è anche la qualità del codice con relativa documentazione, per facilitarne la comprensione ai colleghi permettendo una linea di sviluppo più efficiente. Probabilmente avrei potuto sviluppare un codice più "pulito" evitando qualche ripetizione.

-**Ma Xiang Xiang**: Penso di aver svolto un buon lavoro soprattutto nel model, nel quale ho cercato fin dall'inizio di eseguire un design pulito e adatto a future estensioni. Per view sono molto contento dell'aspetto ottenuto grazie alla combinazione di JavaFX e CSS. Inoltre sviluppando questo progetto, oltre alle competenze tecniche acquisite, mi sono reso conto dell'importanza cruciale che ha la comunicazione nella riuscita del progetto.

4.2 Guida Interfaccia



All’avvio della applicazione la simulazione parte dall’anno 2000.

Quella riportata sopra è l’interfaccia principale della simulazione. Partendo dal pannello in basso troviamo una lista di pulsanti per il controllo:

-Stop: stoppa/riprende la simulazione.

-Speed: nel campo testuale posso inserire un numero da 0 a 1000, mentre nella -combo-box affianco posso selezionare una unità di misura. Premendo speed applico la nuova velocità all’applicazione.

-Edit: premendo edit e poi una Lable di un corpo, esso seguirà il cursore fino al successivo click, dove il corpo prenderà la nuova posizione.

-Add: si apre un pannello dove è possibile selezionare nuovi corpi da aggiungere alla simulazione, caricandoli da file.

-Scale: è una combo-box che seleziona la dimensione delle immagini dei corpi.

-Hide Trails: è una combo-box dove è possibile nascondere le traiettorie di alcuni/tutti corpi.

-Center: tasto che ritorna in posizione centra la simulazione nella posizione iniziale

-Deselect: deseleziona il corpo

A destra è presente un pannello dove sono indicate le caratteristiche del corpo selezionato. Tramite i pulsanti *Save Body* e *Update* è possibile rispettivamente salvare su file un corpo o modificare alcune caratteristiche.

Nel pannello di sinistra è possibile selezionare la modalità full-screen, il sistema di collisioni e il tipo di algoritmo (modifica la precisione dello spawning).

-New crea una nuova simulazione vuota.

-Save salva una configurazione

-Load carica una configurazione precedentemente salvata

-Exit termina l'applicazione