Práctica 4: Simulador Físico

Fecha de entrega: 11 de Marzo de 2019 a las 9:00

Objetivo: Diseño Orientado a Objetos, Genéricos en Java y Colecciones.

1. Control de copias

Durante el curso se realizará control de copias de todas las prácticas, comparando las entregas de todos los grupos de TP. Se considera copia la reproducción total o parcial del código de otros alumnos o cualquier código extraído de Internet o de cualquier otra fuente, salvo aquellas autorizadas explícitamente por el profesor.

En caso de detección de copia se informará al Comité de Actuación ante Copias que citará al alumno infractor y si considera que es necesario sancionar al alumno propondrá una de las tres medidas siguientes:

- Calificación de cero en la convocatoria de TP a la que corresponde la práctica o examen.
- Calificación de cero en todas las convocatorias de TP del curso actual 2018/2019.
- Apertura de expediente académico ante la *Inspección de Servicios de la Universidad*.

2. Instrucciones Generales

Las siguientes instrucciones **son estrictas**, es decir, debes seguirlas obligatoriamente.

1. Descárgate del Campus Virtual la plantilla del proyecto que contiene el método main. Debes desarrollar la práctica usando esta plantilla.
2. Pon los nombres de los componentes del grupo en el fichero "`NAMES.txt`", cada miembro en una línea separada.
3. Debes seguir estrictamente la estructura de paquetes y clases sugerida por el profesor.

4. Cuando entregues la práctica, sube un fichero **zip** del proyecto, incluyendo todos los subdirectorios excepto el subdirectorio **bin**. **Otros formatos (por ejemplo `7zip`, `rar`, etc.) no están permitidos**.

3. Introducción al simulador físico

En esta práctica vamos a implementar un simulador para algunas *leyes de la física*, concretamente para las leyes del *movimiento* y de la *gravedad*. El simulador tendrá dos componentes principales:

- *Cuerpos*, que representan entidades físicas (por ejemplo planetas), que tienen una velocidad, aceleración, posición y masa. Estos cuerpos, cuando se les solicite, se pueden *mover*, modificando su posición de acuerdo a algunas leyes físicas.

- *Leyes de la gravedad*, que especifican cómo las fuerzas gravitacionales cambian las propiedades de un cuerpo (por ejemplo su aceleración).

Utilizaremos un diseño orientado a objetos para poder manejar distintas clases de cuerpos y de leyes de la gravedad. Además, utilizaremos genéricos para implementar factorías, tanto para los cuerpos como para las leyes de la gravedad.

Un paso de simulación consiste en aplicar las leyes de la gravedad para cambiar las propiedades de los cuerpos (e.g., la aceleración), y después solicitar a cada cuerpo que se *mueva*. En esta práctica:

- La entrada será: (a) un fichero que describe una lista de cuerpos en formato `JSON`; (b) las leyes de la gravedad que se van a usar y; (c) el número de pasos que el simulador debe ejecutar.

- La salida será una estructura `JSON` que describe el estado de los cuerpos al inicio y después de cada paso de la simulación.

En el directorio resources puedes encontrar algunos ejemplos de ficheros de entrada, con los correspondientes ficheros de salida. Debes asegurarte de que tu implementación genera la misma salida sobre estos ejemplos – al final de la Sección 4.3, explicaremos cómo comparar tu salida con la salida esperada. En la Sección 6 describimos un visor para poder ver de forma gráfica la simulación.

4. Material necesario para la práctica

4.1. Movimiento y gravedad

Recomendamos leer el siguiente material relacionado con el movimiento y la gravedad, aunque puedes implementar la práctica sin leerlo.

- `https://en.wikipedia.org/wiki/Equations_of_motion`

- `https://en.wikipedia.org/wiki/Newton%27s_law_of_universal_gravitation`

Operación	Descripción	En Java
Suma	$\vec{a}+\vec{b}$ se define como el nuevo vector $(a_1+b_1, \ldots, a_n+b_n)$	a.plus(b)
Resta	$\vec{a}-\vec{b}$ se define como el nuevo vector $(a_1-b_1, \ldots, a_n-b_n)$	a.minus(b)
Multiplicación escalar	$\vec{a} \cdot c$ (o $c \cdot \vec{a}$), donde c es un número real, se define como el nuevo vector $(c * a_1, \ldots, c * a_n)$	a.scale(c)
Longitud	la longitud (o magnitud) de $\vec{a}$, denotado como $\lvert\vec{a}\rvert$, se define como $\sqrt{a_1^2 + \cdots + a_n^2}$	a.magnitude()
Dirección	la dirección de $\vec{a}$ es un nuevo vector que va en la misma dirección que $\vec{a}$, pero su longitud es 1, i.e., se define como $\vec{a} \cdot \frac{1}{\lvert\vec{a}\rvert}$	a.direction()
Distancia	la distancia entre $\vec{a}$ y $\vec{b}$ se define como $\lvert\vec{a} - \vec{b}\rvert$	a.distanceTo(b)

Figura 1: Operaciones sobre vectores.

4.2. Vectores: Implementación

Un vector $\vec{a}$ es un punto $(a_1, \ldots, a_n)$ en un espacio Euclidiano n-dimensional, donde cada a_i es un número real (i.e., de tipo double). Podemos imaginar un vector como una línea que va desde el origen de coordenadas $(0, \ldots, 0)$ al punto $(a_1, \ldots, a_n)$.

En el paquete "simulator.misc" hay una clase Vector, que implementa un vector y ofrece operaciones para manipularlo. En la Figura 1) aparece la descripción de la clase Vector que vamos a utilizar.

4.3. Parseo y creación de datos JSON en Java

JavaScript Object Notation[1] (JSON) es un formato estándar de fichero que utiliza texto y que permite almacenar propiedades de los objetos utilizando pares de atributo-valor y arrays de tipos de datos. Utilizaremos JSON para la entrada y salida del simulador. Brevemente, una estructura JSON es un texto estructurado de la siguiente forma:

$$\texttt{\{ "key}_1\texttt{": value}_1\texttt{, ..., "key}_n\texttt{": value}_n\texttt{ \}}$$

donde $\texttt{key}_i$ es una secuencia de caracteres (que representa una clave) y $\texttt{value}_i$ puede ser un número, un *string*, otra estructura JSON, o un array $\texttt{[o}_1\texttt{,...,o}_k\texttt{]}$, donde $\texttt{o}_i$ puede ser un número, un *string*, una estructura JSON, o un array de estructuras JSON. Por ejemplo:

```
{
  "type" : "basic",
  "data" : {
             "id" : "planet",
             "pos"  : [0.0e00, 4.5e10],
             "vel"  : [1.0e04, 0.0e00],
             "mass" : 1.5e30
           }
}
```

[1] https://en.wikipedia.org/wiki/JSON

En el directorio lib hemos incluido una librería que permite parsear un fichero `JSON` y convertirlo en objetos Java. Esta librería ya está importada en el proyecto y se puede usar también para crear estructuras `JSON` y convertirlas a *strings*. Un ejemplo de uso de esta librería está disponible en el paquete "extra/json".

Para comparar la salida de tu implementación sobre los ejemplos suministrados, con la salida esperada, puedes usar la siguiente herramienta online: `http://www.jsondiff.com`. Además, el ejemplo en el paquete "extra/json" incluye otra forma de comparar estructuras `JSON`.

Ten en cuenta que dos estructuras `JSON` se consideran semánticamente iguales si tienen el mismo conjunto de pares atributo-valor. No es necesario que sean sintácticamente iguales.

5. Implementación del simulador físico

En esta sección describimos las diferentes clases (e interfaces) que debes implementar para desarrollar el simulador físico. Se recomienda fuertemente que sigas la estructura de clases y paquetes que aparece en el enunciado. Puedes encontrar diagramas UML en resources/uml.

5.1. Cuerpos

A continuación detallamos los diferentes tipos de cuerpos (*Body*) que debes implementar. Todas estas clases deben colocarse dentro del paquete "simulator.model" (no en subpaquetes).

Cuerpo básico

Un **cuerpo básico** se implementa a través de la clase Body, que representa una entidad física. Un objeto de tipo Body contiene un identificador *id* (String), una vector de velocidad $\vec{v}$, un vector de aceleración $\vec{a}$, un vector de posición $\vec{p}$ y una masa m (double). Además esta clase debe contener los siguientes métodos:

- public String getId(): devuelve el identificador del cuerpo.
- public Vector getVelocity(): devuelve una *copia* del vector de velocidad.
- public Vector getAcceleration(): devuelve una *copia* del vector de aceleración.
- public Vector getPosition(): devuelve una *copia* del vector de posición.
- double getMass(): devuelve la masa del cuerpo.
- void setVelocity(Vector v): hace una *copia* de v y se la asigna al vector de velocidad.
- void setAcceleration(Vector a): hace una *copia* de a y se la asigna al vector de aceleración.
- void setPosition(Vector p): hace una *copia* de p y se la asigna al vector de posición.
- void move(double t): mueve el cuerpo durante t segundos utilizando los atributos del mismo. Concretamente cambia la posición a $\vec{p}+\vec{v}\cdot t+\frac{1}{2}\cdot\vec{a}\cdot t^2$ y la velocidad a $\vec{v}+\vec{a}\cdot t$.

- public String toString(): devuelve un string con la información del cuerpo en formato JSON:

  { "id": *id*, "mass": m, "pos": $\vec{p}$, "vel": $\vec{v}$, "acc": $\vec{a}$ }

Observa que los métodos que cambian el estado del objeto son *package protected*. De esta forma se garantiza que ninguna clase fuera del modelo puede modificar el estado de los objetos correspondientes. La constructora de un cuerpo debe pasar los valores iniciales al objeto que crea.

Cuerpo que pierde masa

Representamos estos cuerpos a través de la clase MassLossingBody. Estos cuerpos se caracterizan porque pierden, con cierta frecuencia, masa cuando se mueven. La clase MassLossingBody extiende a Body, y tiene los siguientes atributos:

- lossFactor: un número (double) entre 0 y 1 que representa el factor de pérdida de masa.
- lossFrequency: un número positivo (double) que indica el intervalo de tiempo (en segundos) después del cual el objeto pierde masa.

Los valores para estos atributos deben obtenerse a través de su constructora. El método move se comporta como el de Body, pero además *después de moverse*, comprueba si han pasado lossFrequency segundos desde la última vez que se redujo la masa del objeto. En tal caso, se reduce la masa de nuevo en lossFactor, i.e, la nueva masa será $m * (1 - \text{lossFactor})$. Para implementar este proceso debes hacer lo siguiente: usa un contador c (inicializado a 0,0) para acumular el tiempo (i.e., el parámetro t de move) y cuando $c \geq$ lossFrequency aplica la reducción y pon de nuevo c a 0,0.

Otras clases de cuerpos

Si quieres, puedes inventarte nuevas clases de cuerpos con diferentes comportamientos.

5.2. Leyes de la gravedad

En esta sección describimos las diferentes clases de leyes de la gravedad que debes implementar. Todas las clases e interfaces deben colocarse en el paquete "simulator.model" (no en un subpaquete). Para modelar las leyes de la gravedad utilizaremos una interfaz GravityLaws, que tiene únicamente el siguiente método:

- public void apply(List<Body>bodies)

Este método, en las clases que implementan esta interfaz, debe aplicar las leyes de la gravedad particulares de la clase para cambiar las propiedades (e.g. el vector de aceleración) de los distintos cuerpos de la lista que va como parámetro.

Ley de Newton de la gravitación universal

Implementaremos esta ley en una clase NewtonUniversalGravitation, que cambiará la aceleración de los cuerpos de la siguiente forma: dos cuerpos B_i y B_j aplican una fuerza gravitacional uno sobre otro, i.e., se atraen mutuamente. Supongamos que $\vec{F}_{i,j}$ es la fuerza

aplicada por el cuerpo B_j sobre el cuerpo B_i (más tarde veremos como se calcula). La fuerza total aplicada sobre B_i se define como la suma de todas las fuerzas aplicadas sobre B_i por otros cuerpos, i.e., $\vec{F_i} = \sum_{i \neq j} \vec{F_{i,j}}$. Ahora, usando la segunda ley de Newton, i.e., $\vec{F} = m \cdot \vec{a}$, podemos concluir que la aplicación de $\vec{F_i}$ sobre B_i cambia su aceleración a $\vec{F_i} \cdot \frac{1}{m_i}$. Como un caso especial, si m_i es igual a 0,0, ponemos los vectores de aceleración y velocidad de B_i a $\vec{o} = (0, \ldots, 0)$ (sin necesidad de calcular $\vec{F_i}$).

Vamos a explicar ahora como calcular $\vec{F_{i,j}}$. Según la ley de la gravitación universal de Newton, los cuerpos B_i y B_j generan una fuerza, uno sobre otro, que es igual a:

$$f_{i,j} = G * \frac{m_i * m_j}{|\vec{p_j} - \vec{p_i}|^2}$$

donde G es la constante gravitacional, que es aproximadamente $6{,}67 * 10^{-11}$ (6,67E−11 usando la sintaxis de Java). Observa que $|\vec{p_j} - \vec{p_i}|$ es la distancia entre los vectores $\vec{p_i}$ y $\vec{p_j}$, i.e., la distancia entre los centros de B_i y B_j. Ahora, para calcular la dirección de esta fuerza, convertimos $f_{i,j}$ en $\vec{F_{i,j}}$ como sigue: Sea $\vec{d_{i,j}}$ la dirección de $\vec{p_j} - \vec{p_i}$, entonces $\vec{F_{i,j}} = \vec{d_{i,j}} \cdot f_{i,j}$.

Cayendo hacia el centro

Esta ley de gravedad se implementa en la clase FallingToCenterGravity. La ley simula un escenario en el cual todos los cuerpos caen hacia el "centro del universo", i.e. tienen una aceleración fija de $g = 9{,}81$ en dirección al origen $\vec{o} = (0, \ldots, 0)$. Técnicamente, para un cuerpo B_i, asumiendo que $\vec{d_i}$ es su dirección, su aceleración debería ponerse a: $-g \cdot \vec{d_i}$.

Sin gravedad

Esta ley de gravedad se implementa en la clase NoGravity. Simplemente no hace nada, i.e., su método apply está vacío. Esto significa que los cuerpos se mueven con una aceleración fija.

Otras leyes de la gravedad

Si quieres, puedes inventarte e implementar otras leyes de la gravedad.

5.3. Factorías

Una vez que hemos definido las diferentes clases de cuerpos y leyes de la gravedad, utilizaremos factorías para separar su creación desde el simulador. Necesitamos dos factorías: una para los cuerpos y otra para las leyes de la gravedad. Las factorías las implementaremos usando genéricos, ya que tienen una parte en común. A continuación detallamos cómo implementar estas factorías paso a paso. Todas las clases e interfaces deben colocarse en el paquete "simulator.factories" (no en subpaquetes).

La interfaz "Factory"

Modelamos esta factoría a través de la interfaz genérica Factory<T>, con los siguientes métodos:

Cuerpo básico

```
{
  "type": "basic",
  "data": {
    "id": "b1",
    "pos": [0.0e00,  0.0e00],
    "vel": [0.05e04, 0.0e00],
    "mass": 5.97e24
  }
}
```

Cuerpo que pierde masa

```
{
  "type": "mlb",
  "data": {
    "id": "b1",
    "pos": [-3.5e10, 0.0e00],
    "vel": [0.0e00, 1.4e03],
    "mass": 3.0e28,
    "freq": 1e3,
    "factor": 1e-3
  }
}
```

Ley de Newton de la gravitación universal

```
{
  "type": "nlug",
  "data": {}
}
```

Cayendo hacia el centro

```
{
  "type": "ftcg",
  "data": {}
}
```

Sin gravedad

```
{
  "type": "ng",
  "data": {}
}
```

Figura 2: Formato JSON para cuerpos y leyes de la gravedad.

- public T createInstance(JSONObject info): recibe una structura JSON que describe el objeto a crear (ver la sintaxis más abajo), y devuelve una instancia de la clase correspondiente – una instancia de un subtipo de T. En caso de que info sea incorrecto, entonces lanza una excepción del tipo IllegalArgumentException.
- public List<JSONObject>getInfo(): devuelve una lista de objetos JSON, que son "plantillas" para structuras JSON válidas. Los objetos de esta lista se pueden pasar como parámetro al método createInstance. Esto es muy útil para saber cuáles son los valores válidos para una factoría concreta, sin saber mucho sobre la factoría en si misma. Por ejemplo, utilizaremos este método cuando mostremos al usuario los posibles valores de las leyes de la gravedad.

La estructura JSON que se pasa como parámetro al método createInstance incluye dos claves: *type*, que describe el tipo de objeto que se va a crear y; *data*, que es una estructura JSON que incluye toda la información necesaria para crear el objeto. En la Figura 2 se incluye una tabla con las estructuras JSON que utilizaremos para crear instancias de cuerpos y leyes de la gravedad.

Los elementos de la lista que devuelve getInfo son estructuras JSON de las listadas en la Figura 2, con algunos valores por defecto para las claves de la sección *data* (o en lugar de valores, podemos usar *strings* que describan las clases correspondientes). Además cada elemento de la lista incluye una clave *desc*, que contiene un *string* para describir la plantilla. Por ejemplo, para la ley de Newton de la gravitación universal, podemos usar:

```
"desc": "Newton's law of universal gravitation"
```

Constructores basados en factorías

Las factorías concretas que vamos a desarrollar están basadas en el uso de constructores. Un constructor es un objeto que es capaz de crear una instancia de un tipo específico, i.e., puede manejar una structura JSON con un valor concreto para la clave *type*. Un constructor se modela usando la clase genérica Builder<T>, con los siguientes métodos:

- public T createInstance(JSONObject info): si la información suministrada por info es correcta, entonces crea un objeto de tipo T (i.e., una instancia de una subclase de T). En otro caso devuelve null para indicar que este constructor es incapaz de reconocer ese formato. En caso de que reconozca el campo *type* pero haya un error en alguno de los valores suministrados por la sección *data*, el método lanza una excepcion IllegalArgumentException.
- public JSONObject getBuilderInfo(): devuelve un objeto JSON que sirve de plantilla para el correspondiente constructor, i.e., un valor válido para el parámetro de createInstance (ver getInfo() de Factory<T>).

Usa esta clase para definir los siguientes constructores:

- BasicBodyBuilder que extiende a Builder<Body>, para crear objetos de la clase Body – pone la aceleración a $(0, \ldots, 0)$.
- MassLosingBodyBuilder que extiende a Builder<Body>, para crear objetos de la clase MassLosingBody – pone la aceleración a $(0, \ldots, 0)$.
- NewtonUniversalGravitationBuilder que extiende a Builder<GravityLaws>, para crear objetos de la clase NewtonUniversalGravitation
- FallingToCenterGravityBuilder que extiende a Builder<GravityLaws>, para crear objetos de la clase FallingToCenterGravity
- NoGravityBuilder que extiende a Builder<GravityLaws>, para crear objetos de la clase NoGravity

Una vez que los constructores están preparados, implementamos una factoría genérica BuilderBasedFactory<T>, que implementa a Factory<T>. La constructora de esta clase recibe como parámetro una lista de constructores:

public BuilderBasedFactory(List<Builder<T>> builders)

El método createInstance de la factoría ejecuta los constructores uno a uno hasta que encuentre el constructor capaz de crear el objeto correspondiente — debe lanzar una excepcion IllegalArgumentException en caso de fallo. El método getInfo() devuelve en una lista las estructuras JSON devueltas por getBuilderInfo().

El siguiente ejemplo muestra como se puede crear una factoría de cuerpos usando las clases que hemos desarrollado:

```
ArrayList<Builder<Body>> bodyBuilders = new ArrayList<>();
bodyBuilders.add(new BasicBodyBuilder());
bodyBuilders.add(new MassLosingBodyBuilder());
Factory<Body> bodyFactory = new BuilderBasedFactory<Body>(bodyBuilders);
```

5.4. La clase simulador

El simulador lo implementamos en la clase PhysicsSimulator, dentro del paquete "simulator.model" (no en un subpaquetes). Su constructora tiene los siguientes parámetros, para inicializar los campos correspondientes:

- *Tiempo real por paso*: un número de tipo double que representa el tiempo (en segundos) que corresponde a un paso de simulación — se pasará al método move de los cuerpos. Debe lanzar una excepción IllegalArgumentException en caso de que el valor no sea válido.

- *Leyes de la gravedad:* un objeto del tipo GravityLaws, que representa las leyes de la gravedad que el simulador aplicará a los cuerpos. Si el valor es null, debe lanzar una excepción del tipo IllegalArgumentException.

La clase debe mantener además una lista de cuerpos de tipo List<Body>y el tiempo actual, que inicialmente será 0,0. Esta clase ofrece los siguientes métodos:

- public void advance(): aplica un paso de simulación, i.e., primero llama al método apply de las leyes de la gravedad, después llama a move(dt) para cada cuerpo, donde dt es el *tiempo real por paso*, y finalmente incrementa el tiempo actual en dt segundos.
- public void addBody(Body b): añade el cuerpo b al simulador. El método debe comprobar que no existe ningún otro cuerpo en el simulador con el mismo identificador. Si existiera, el método debe lanzar una excepción del tipo IllegalArgumentException.
- public String toString(): devuelve un string que representa un estado del simulador, utilizando el siguiente formato JSON:

  $$\texttt{\{ "time": } T\texttt{, "bodies": [}json_1\texttt{, }json_2\texttt{, ...] \}}$$

  donde T es el tiempo actual y $json_i$ es el *string* devuelto por el método toString del i-ésimo cuerpo en la lista de cuerpos.

5.5. El controlador

El controlador se implementa en la clase Controller, dentro del paquete "simulator.control" (no en un subpaquete). Es el encargado de (1) leer los cuerpos desde un InputStream dado y añadirlos al simulador; (2) ejecutar el simulador un número determinado de pasos y mostrar los diferentes estados de cada paso en un OutputStream dado. La clase recibe en su constructora un objeto del tipo PhysicsSimulator, que se usará para ejecutar las diferentes operaciones y un objeto del tipo Factory<Body>, para construir los cuerpos que se leen del fichero. La clase Controller ofrece los siguientes métodos:

- public void loadBodies(InputStream in): asumimos que in contiene una estructura JSON de la forma:

  $$\texttt{\{ "bodies": [}bb_1\texttt{,...,}bb_n\texttt{] \}}$$

  donde bb_i es una estructura JSON que define un cuerpo de acuerdo a la sintaxis de la Figura 2. Este método primero transforma la entrada JSON en un objeto JSONObject, utilizando:

  ```
  JSONObject jsonInupt = new JSONObject(new JSONTokener(in));
  ```

  y luego extrae cada bb_i de jsonInupt, crea el correspondiente cuerpo b usando la *factoría de cuerpos*, y lo añade al simulador llamando al método addBody.
- public void run(int n, OutputStream out): ejecuta el simulador n pasos y muestra los diferentes estados en out, utilizando el siguiente formato JSON:

  $$\texttt{\{ "states": [}s_0\texttt{,}s_1\texttt{,...,}s_n\texttt{] \}}$$

  donde s_0 es el estado del simulador antes de ejecutar ningún paso, y cada s_i con $i \geq 1$, es el estado del simulador inmediatamente después de ejecutar el i-ésimo paso de simulación. Observa que el estado s_i se obtiene llamando al método toString() del simulador.

5.6. La clase Main

En el paquete "simulator.launcher" puedes encontrar una versión incompleta de la clase Main. Esta clase procesa los argumentos de la línea de comandos e inicia la simulación. La clase también parsea algunos argumentos de la línea de comandos utilizando la librería common-cli (incluida en el directorio lib y ya importada en el proyecto). Tendrás que extender la clase Main para parsear todos los posibles argumentos.

Ejecutar Main con el argumento -h (or --help) debe mostrar por consola lo siguiente:

```
usage: simulator.launcher.Main [-dt <arg>] [-gl <arg>] [-h] [-i <arg>] [-m
       <arg>] [-o <arg>] [-s <arg>]
 -dt,--delta-time <arg>     A double representing actual time, in seconds,
                            per simulation step. Default value: 2500.0.
 -gl,--gravity-laws <arg>   Gravity laws to be used in the simulator.
                            Possible values: 'nlug' (Newton's law of
                            universal gravitation), 'ftcg' (Falling to
                            center gravity), 'ng' (No gravity). Default
                            value: 'nlug'.
 -h,--help                  Print this message.
 -i,--input <arg>           Bodies JSON input file.
 -o,--output <arg>          Output file, where output is written. Default
                            value: the standard output.
 -s,--steps <arg>           An integer representing the number of
                            simulation steps. Default value: 150.
```

Este texto de ayuda describe como ejecutar el simulador. Por ejemplo:

```
-i resources/examples/ex4.4body.txt
-o resources/examples/ex4.4body.out -s 100 -gl nlug
```

ejecutaría el simulador 100 pasos, usando las leyes de la gravedad nlug (ley de Newton de la gravitación universal). La entrada la lee del fichero resources/examples/ex4.4body.txt y la salida la escribe en resources/examples/ex4.4body.out. En la cabecera de la clase Main puedes encontrar mas ejemplos de uso de la línea de comandos. Modifica la clase Main para que tenga la siguiente funcionalidad:

- Añade la opción -o (o --output) en la línea de comandos para que el usuario pueda escribir el nombre del fichero en el cual escribir la salida. En caso de que no se especifique el fichero de salida, se utilizará la salida por consola System.out.
- Añade la opción -s (o --steps) en la línea de comandos para que el usuario pueda especificar el númeo de pasos de simulación. Sino se especifica nada, el valor por defecto será de 150.
- Completa la implementación del método init() para crear e inicializar las factorías (campos _bodyFactory and _gravityLawsFactory) – ver el final de la Sección 5.3.
- Completa la implementación del método startBatchMode() de forma que cree una instancia del simulador y del controlador; establezca las leyes de gravedad del simulador de acuerdo con lo especificado a través de la opcion -gl; cree los ficheros correspondientes de entrada y salida teniendo en cuenta las opciones -i y -o, añada los cuerpos al simulador (invocando al método loadBodies del controlador); e inicie la simulación llamando al método run del controlador.

6. Visualización de la salida

Como habrás observado, la salida de la práctica es una estructura `JSON` que describe los diferentes estados de la simulación, y que no es fácil de leer. En la Práctica 5 desarrollaremos una interfaz gráfica que permitirá visualizar los estados con animación. Pero hasta entonces, puedes usar resources/viewer/viewer.html para visualizar la salida de tu programa. Es un fichero HTML que utiliza JavaScript para ejecutar la visualización. Abreló con un navegador, como por ejemplo Firefox, Safari, Internet Explorer, Chrome, o el navegador de Eclipse.