Práctica 2: **Viaje galáctico(con ampliación)**

Programación II

Julio 2017

DLSI – Universidad de Alicante

Alicia Garrido Alenda

Normas generales

- Esta práctica contiene métodos nuevos que se tienen que implementar obligatoriamente ya que consisten en el 70 % de la nota de esta práctica en esta convocatoria.
- El plazo de entrega para esta práctica se abrirá el lunes 26 de junio a las 9:00 horas y se **cerrará el viernes 30 de junio a las 23:59 horas**. No se admitirán entregas fuera de plazo.
- Se debe entregar la práctica en un fichero comprimido de la siguiente manera:

  1. Abrir un terminal.
  2. Situarse en el directorio donde se encuentran los ficheros fuente (`.java`) de manera que al ejecutar `ls` se muestren los ficheros que hemos creado para la práctica y ningún directorio.
  3. Ejecutar:

     ```
     tar cfvz  practica2.tgz   *.java
     ```

Normas generales comunes a todas las prácticas

1. La práctica se debe entregar exclusivamente a través del servidor de prácticas del departamento de Lenguajes y Sistemas Informáticos, al que se puede acceder desde la página principal del departamento (`http://www.dlsi.ua.es`, enlace "Entrega de prácticas") o directamente en `http://pracdlsi.dlsi.ua.es`.
   - **Bajo ningún concepto** se admitirán entregas de prácticas por otros medios (correo electrónico, UACloud, etc.).
   - El usuario y contraseña para entregar prácticas es el mismo que se utiliza en UACloud.
   - La práctica se puede entregar varias veces, pero sólo se corregirá la última entrega.

2. El programa debe poder ser compilado sin errores con el compilador de Java existente en la distribución de Linux de los laboratorios de prácticas; si la práctica no se puede compilar su calificación será 0. Se recomienda compilar y probar la práctica con el autocorrector durante su implementación para detectar y corregir errores.

3. La práctica debe ser un trabajo original de la persona que entrega; **en caso de detectarse indicios de copia con una o más prácticas (o compartición de código) la calificación de la práctica será** `0`.

4. Se recomienda que los ficheros fuente estén adecuadamente documentados, con comentarios donde se considere necesario.

5. La primera corrección de la práctica se realizará de forma automática, por lo que es imprescindible respetar estrictamente los formatos de salida que se indican en este enunciado. Además, al principio de todos los ficheros fuente entregados se debe incluir un comentario con el `DNI` de la persona que entrega la práctica, con el siguiente formato:

   // DNI *tuDNI* Nombre

   donde *tuDNI* es el DNI del alumno correspondiente (sin letras), **tal y como aparece en UACloud**, y Nombre es el nombre completo; por ejemplo, el comentario podría ser:

   ```
   // DNI 2737189 CASIS RECOS, ANTONIA
   ```

6. El cálculo de la nota de la práctica y su influencia en la nota final de la asignatura se detallan en las transparencias de la presentación de la asignatura.

7. Para evitar errores de compilación debido a codificación de caracteres que **descuenta** `0.5` **de la nota de la práctica**, se debe seguir una de estas dos opciones:

   a) Utilizar EXCLUSIVAMENTE caracteres ASCII (el alfabeto inglés) en vuestros ficheros, **incluídos los comentarios**. Es decir, no poner acentos, ni ñ, ni ç, etcétera.

   b) Entrar en el menú de Eclipse `Edit > Set Encoding`, aparecerá una ventana en la que hay que seleccionar `UTF-8` como codificación para los caracteres.

8. El tiempo estimado por el corrector para ejecutar cada prueba debe ser suficiente para que finalice su ejecución correctamente, en otro caso se invoca un proceso que obliga a la finalización de la ejecución de esa prueba y se considera que dicha prueba no funciona correctamente.

Descripción del problema

Los navegantes han ampliado el diseño de sus naves, creando las naves *Dophraki*, que son *Romskip* pero que además de transportar mercancías pueden transportar navegantes, tener tripulación y realizar viajes. Además han ampliado la forma de embarcar mercancías, ya que en esta naves se puede embarcar directamente una mercancía.

Esta forma de embarque ha sido aprovechada por unos parásitos espaciales, los *Skrogem*, que se mimetizan como mercancías, de manera que son embarcados en la bodega de la nave de forma inadvertida. Una vez están en una nave, estos parásitos buscan un posible huésped entre los pasajeros que viajan en la nave, al que se adherieren de forma invisible para extraerle su "especia" hasta dejarlo seco, momento en el que se liberan y buscan un nuevo huésped.

Por este motivo la tripulación de la nave se entrega de forma solícita y congojosa a su labor de revisar la bodega de la nave para intentar eliminar estos molestos parásitos.

Por tanto un *Tripulante* de una de estas naves, es un *Navegante* que además ostenta un cargo y realiza acciones de mantenimiento en la nave.

Para esta práctica es necesario implementar las clases que se definen a continuación, a las que se pueden añadir variables de instancia y/o clase (siempre que sean privadas) y métodos cuando se considere necesario. También se pueden añadir métodos a las superclases cuando sea necesario.

Un **Navegante** se caracteriza por:

- *nombre*: nombre por el que es conocido en su sociedad (`String`).
- *especia*: energía de la que dispone (`double`).
- *tempo*: cantidad de tiempo trabajado (`double`).
- *cosecha*: especia recogida al trabajar (`double`).
- *pasaje*: número de asiento que ocupa cuando viaja en una nave (`int`).

Cuando se crea un **Navegante** se le pasan por parámetro un `String` y un real. El `String`[1] se corresponde con el nombre del navegante y el real se corresponde con la especia de la que dispone inicialmente, que debe ser mayor que `0`, en otro caso valdrá `7.5`. Inicialmente el tempo y la cosecha serán `0` y su pasaje `-1`, lo que significa que no es pasajero de ninguna nave.

Y las acciones que puede realizar son:

- *trabaja*: se le pasa por parámetro una mercancía y un real que indica el tiempo que le lleva realizar el trabajo. Si el tiempo es mayor que 0, se lleva a cabo el trabajo y, por tanto, se incrementa el *tempo* del navegante en esa cantidad. El trabajo consiste en comprobar si la mercancía es un parásito o no. Si se trata de un parásito es necesario desparasitarlo; para ello debe crear una mercancía con las mismas características pero sin que sea un parásito y devolverlo, pero previamente le extrae su especia, dejando la especia del parásito a `0`, y la agrega a su cosecha. Si no se trata de un parásito devuelve la misma mercancía que le habían pasado. Si no se lleva a cabo el trabajo, devuelve una referencia vacía.

[1] El `String` pasado por parámetro al constructor siempre será distinto de `null`.

⊙ *cobra*: el navegante percibe su sueldo en especia por el tiempo trabajado. Su tarifa por unidad de tiempo trabajada se calcula:

$$tarifa = \frac{especia}{cosecha}$$

De manera que su sueldo será:

$$sueldo = tarifa * tempo$$

De manera que incrementa su especia cobrando su sueldo, dejando su tempo a `0`. Además se devuelve el resto de la cosecha una vez descontado su sueldo, dejándola a `0` también.

⊙ *embarca*: se le pasa por parámetro la dophraki en la que quiere embarcar como pasajero. Si no tiene ya un pasaje válido, se le pasa un mensaje a la nave solicitando un asiento como pasajero invocando su método *embarque* pasándose a sí mismo como parámetro. Si la nave le devuelve un pasaje válido (mayor o igual a 0) significa que ha sido embarcado en esa nave, por tanto se guarda el pasaje y devuelve cierto. En cualquier otro caso devuelve falso.

⊙ *desembarca*: se le pasa por parámetro una dophraki a la que le pasa un mensaje de desembarque pasándole por parámetro su pasaje y a sí mismo, de manera que si la nave le devuelve cierto significa que ha desembarcado de la nave, inicializa su pasaje de nuevo a `-1` y devuelve cierto.

⊙ *paga*: se le pasa el importe que debe pagar. Si el importe es mayor que `0` y tiene suficiente especia se lo descuenta y devuelve la cantidad descontada. Si no tiene suficiente especia para pagar la cantidad indicada, devuelve `0.0`. Si el importe es menor o igual a `0` devuelve `-1.0`.

⊙ *getNombre*: devuelve el nombre del navegante.

⊙ *getEspecia*: devuelve la especia del navegante.

⊙ *getTempo*: devuelve el tempo del navegante.

⊙ *getCosecha*: devuelve la cosecha del navegante.

⊙ *getPasaje*: devuelve el pasaje del navegante.

El **Tripulante** es un *Navegante* que además se caracteriza por:

⁕ *conexion*: nave en la que ha sido enrolado (`Dophraki`).

⁕ *rango*: puesto que ocupa en el equipo de la nave (`String`).

Cuando se crea un **Tripulante** se le pasan por parámetro un `String`, un real y un `String`[2]. El primer `String` se corresponde con su nombre, el real con su especia y el segundo `String` con su rango. El rango puede ser `superior` o `subalterno`[3], en cualquier otro caso se le asigna el rango `subalterno`. Inicialmente no está enrolado en ninguna nave (`null`).

Y las acciones que puede realizar son las mismas que un *Navegante*, aunque algunas de forma diferente:

[2]Ambos `String` pasados por parámetro al constructor siempre serán distintos de `null`.

[3]Ignorando diferencias entre mayúsculas y minúsculas.

- ⊙ *cobra*: cobra como si fuera un navegante pero incrementando esta cantidad en un porcentaje en función de su rango:
  - superior: incrementa lo que cobra en un 35 %;
  - subalterno: incrementa lo que cobra en un 20 %;
- ⊙ *embarca*: se le pasa por parámetro la dophraki en la que se quiere enrolar como parte de la tripulación. Si no pertenece a ninguna tripulación y no viaja como pasajero en ninguna otra nave, le pasa un mensaje a la nave solicitando la admisión en su tripulación, para ello invoca su método *solicitudAdmision* pasándose a sí mismo como parámetro. Si la nave acepta la solicitud (le devuelve cierto), devuelve cierto. Si la nave no acepta su solicitud el tripulante intentará embarcar como pasajero en dicha nave, de manera que si lo consigue devolverá cierto. En cualquier otro caso devuelve falso.

Y además puede:

- ⊙ *trabaja*: se le pasa por parámetro el tiempo que invierte en eliminar un parásito. Si este tiempo es mayor que 0, recorre la bodega de la nave en la que ha sido enrolado buscando parásitos para eliminarlos, aunque como mucho puede eliminar 3 cada vez. Cuando encuentra uno, trabaja como cualquier otro navegante, pero además reemplaza en la bodega el parásito por una mercancía con las mismas características. Devuelve la cantidad de parásitos eliminados.
- ⊙ *trabaja*: se le pasa por parámetro el tiempo que invierte en eliminar un parásito y la cantidad máxima de parásitos a eliminar. Si ambos parámetros son mayores que 0, recorre la bodega de la nave en la que ha sido enrolado buscando parásitos para eliminarlos. Cuando encuentra uno, lo elimina de la bodega reemplazándolo en la bodega por una mercancía con las mismas características y almacena el parásito eliminado en el foso de la nave. Devuelve la cantidad de parásitos eliminados.
- ⊙ *desembarca*: el tripulante desea abandonar la tripulación de la nave, para ello antes de abandonar la nave tiene que obtener el finiquito, de manera que cobra y le envía a la nave un mensaje de dimisión invocando su método *solicitudDimision*, pasándole por parámetro lo que queda de su cosecha una vez descontado su sueldo y a sí mismo. Si la solicitud es admitida por la nave, devuelve cierto. En cualquier otro caso devuelve falso.
- ⊙ *enrolado*: devuelve cierto si el tripulante pertenece a la tripulación de alguna nave, y falso en caso contrario.
- ⊙ *setConexion*: se le pasa por parámetro la nueva dophraki para actualizar su conexión.
- ⊙ *getRango*: devuelve su rango.
- ⊙ getConexion**: devuelve su conexion.**

Las mercancías han adaptado su forma de ser almacenadas para adaptarse a las nuevas naves, de manera que ahora la **Mercancia** tiene un nuevo método *almacena* además del anterior. Este nuevo método sería:

- ⊙ *almacena*: se le pasa por parámetro la nave en la que va a ser almacenada. Si no está almacenada en ningún sitio, se actualiza su indicador de carga a cierto, se genera su etiqueta concatenando el nombre de la nave y su peso (la etiqueta se genera siempre, si tenía una anterior, es sustituída por la nueva) y devuelve cierto . En cualquier otro caso devuelve falso.

Un **Skrogem** es un parásito que se mimetiza como mercancía, por tanto es una *Mercancia*, que además se caracteriza por:

- *nave*: nave en la que se ha colado como polizón (Dophraki);
- *especia*: cantidad de especia sustraída a sus huéspedes (double);
- *huesped*: navegante parasitado (Navegante);
- *huespedes*: array con todos los navegantes parasitados por algún skrogem (static Navegante[]). Inicialmente tendrá un tamaño de 10;

Cuando se crea un **Skrogem** se le pasan por parámetro su peso y sus dimensiones, para mimetizarse como mercancía y la cantidad de especia que tiene en principio, que debe ser mayor que 0, en otro caso valdrá 5.5. Inicialmente no está en ninguna nave y no tiene huésped.

Un **Skrogem** puede realizar las mismas acciones que una *Mercancia* pero algunas de forma difererente:

- *almacena*: se le pasa por parámetro la nave en la que va a ser almacenado. Si no está almacenado en ningún sitio, se actualiza su indicador de carga a cierto, se genera su etiqueta concatenando el nombre de la nave y su peso (la etiqueta se genera siempre, si tenía una anterior, es sustituída por la nueva), actualiza su nave y devuelve cierto. En cualquier otro caso devuelve falso.
- *recogida*: si su indicador de carga está a cierto, se actualiza su valor a falso, actualiza el valor de su nave a referencia vacía (null) y el método devuelve cierto. En cualquier otro caso el método devuelve falso.

Y además puede realizar las siguientes acciones:

- *busca*: si no tiene huésped recorre el pasaje de la nave en la que se ha colado buscando un navegante al que usar como huésped. Como son muy agresivos, dos skrogem no pueden compartir un huésped, por tanto antes de adherirse comprueban si el navegante ya es huésped de otro skrogem. Para ello lo buscan en el array huespedes de los Skrogem, si lo encuentran pasan al siguiente navegante del pasaje y si no lo encuentran se adhieren a él, haciéndolo su huésped, agregándolo a los huéspedes de los skrogem y devuelve cierto. En cualquier otro caso devuelve falso. Si no hay posiciones libres para agregarlo a los huéspedes se debe redimensionar, aumentando su tamaño en 10 posiciones.
- *activa*: cuando un skrogem se activa extrae de su huésped cierta cantidad de especia. Si el huésped no dispone de especia (tiene 0) significa que el parásito lo ha dejado seco, por tanto se desliga de él eliminándolo de los huéspedes de los skrogem y actualizando su huésped a referencia vacía (null). Si el huésped dispone de una especia mayor que 4, el skrogem le extrae el 25 % de su especia, en otro caso le extrae a su huésped toda la especia que le queda (pero no se desliga), de manera que el huésped decrementa su especia mientras que el skrogem incrementa la suya propia. El método devuelve la cantidad de especia extraída del huésped.
- *eliminado*: el skrogem ha sido detectado y está siendo eliminado de la bodega de carga de la nave en la que estaba. Por tanto elimina a su huésped de los huéspedes de los skrogem, actualiza su nave y su huésped a referencia vacía (null) y devuelve su especia, dejándola a 0.
- *getEspecia*: devuelve la especia acumulada para su consulta.

- ⊙ *getHuesped*: devuelve su huésped.
- ⊙ *getNave*: devuelve su nave.
- ⊙ *getHuespedes*: devuelve los huéspedes de los *Skrogem*.
- ⊙ *recarga*: el skrogem calcula la media de la especia de todos los huéspedes de los skrogem e incrementa su especia con el 10 % de esta cantidad. Para calcular la media se debe aplicar la siguiente fórmula matemática:

$$\bar{E} = \frac{\sum_{i=1}^{n} e_i}{n}$$

donde e_i es la especia del huésped `i` y n el número de huéspedes. El método devuelve la cantidad con la que el skrogem incrementa su especia.

Una **Dophraki** es una *Romskip* que además se caracteriza por:

- ⁕ *pasaje*: asientos destinados a los navegantes que viajan en la nave (`Navegante[]`);
- ⁕ *tripulacion*: espacio reservado para los tripulantes de la nave (`Tripulante[]`);
- ⁕ *posicion*: coordenadas de la posición actual de la nave (`Coordenada`);
- ⁕ *especia*: indicador del nivel de especia, que es la fuente de energía necesaria para realizar viajes (`double`);
- ⁕ *mapa*: galaxia donde se sitúan estas naves (`static Galaxia`);
- ⁕ *foso*: espacio reservado para los parasitos eliminados de la bodega de la nave (`Skrogem[]`);

Cuando se crea una **Dophraki** se le pasan los mismos parámetros que a una *Romskip* y además, otros dos enteros y un real. El primer entero indica el tamaño que tendrá el pasaje, que como mínimo debe ser `3`, el segundo indica el tamaño de tripulacion, que como mínimo debe ser `1`, y por último el real indica la cantidad de especia inicial de la nave, que como mínimo debe ser `3.5`. La nave siempre está situada inicialmente en la coordenada `[0,0,0]`.

Las acciones que se pueden realizar con una **Dophraki** son las mismas que con una *Romskip*, pero algunas de ellas se llevan a cabo de forma diferente debido a los nuevos avances en tecnología, que permiten a estas naves transportar cualquier peso. Además como ahora transportan pasajeros, estos también se pueden teletransportar. Por tanto:

- ⊙ *embarque*: se le pasa por parámetro una cadena y una coordenada[4]. Se comprueba si la coordenada se corresponde con la coordenada en la que está situada la nave. Si no es así se devuelve la cadena:

```
1 2 3: emplazamiento incorrecto
```

donde `1 2 3` son los valores de la coordenada pasada por parámetro. Si la coordenada coincide, se debe recoger la primera mercancía que encuentre en el sector correspondiente a esa coordenada cuya etiqueta coincida con la cadena pasada por parámetro, ignorando diferencias entre mayúsculas y minúsculas. Si no encuentra ninguna mercancía con esa etiqueta devuelve la cadena:

[4]Estos parámetros siempre serán distintos de `null`.

```
barbie0.75-0.5: mercancia no encontrada
```

donde "`barbie0.75-0.5`" es la etiqueta pasada por parámetro.

Si encuentra una mercancía con esa etiqueta, tiene que recogerla del sector para embarcarla en la bodega de carga de la nave en la última posición libre que encuentre, pasando un mensaje de almacena a la mercancía, pasándose la propia nave como parámetro, para que actualice su etiqueta y su indicador de carga, y devolver la cadena:

```
Nostromo0.75: mercancia embarcada
```

donde "`Nostromo0.75`" es la etiqueta actualizada de la mercancía embarcada.

Si no dispone de posiciones libres, devuelve la cadena:

```
barbie0.75-0.5: bodega de carga completa
```

donde "`barbie0.75-0.5`" es la etiqueta de la mercancía no embarcada.

Si la nave no está situada en ninguna galaxia devuelve una referencia vacía (`null`).

⊙ ***`desembarque`***: se le pasa por parámetro una cadena. Busca entre sus mercancías el primer objeto cuya etiqueta coincida con la cadena pasada por parámetro, ignorando diferencias entre mayúsculas/minúsculas. Si no encuentra ninguno devolverá una referencia vacía. Si lo encuentra devolverá el objeto encontrado, dejando una posición vacía en la bodega de carga y sin compactar la bodega.

⊙ ***`teletransporte`***: se le pasa por parámetro la nave destino del teletransporte y un entero, que indica la posición a teletransportar. Si la nave destino es de tipo ***`Dophraki`*** se llevará a cabo la teletransportación del pasajero que ocupa la posición del pasaje indicada por el segundo parámetro, mientras que si es de tipo ***`Romskip`*** se teletransportará la mercancía que ocupa la posición de la bodega indicada por el segundo parámetro. Si la teletransportación se lleva a cabo con éxito el objeto teletransportado deja de estar en la nave, dejando una posición vacía en su lugar, y se devuelve cierto. Además en el caso de teletransportar un navegante se le pasa un mensaje para que actualice su pasaje, **y si lo que se teletransporta es un skrogem se le pasa un mensaje para actualizar su nave y su huésped a** `null`. En cualquier otro caso se devuelve falso.

⊙ ***`verifica`***: se le pasa por parámetro la mercancía que está siendo teletransportada. Debido a los nuevos avances, la mercancía será almacenada en la última posición libre de la bodega si dispone de posiciones libres, **y en el caso de que sea un skrogem se le pasa un mensaje para actualizar su nave**. Si no dispone de posiciones libres redimensiona la bodega ampliando su tamaño en 5 posiciones, prodece al almacenamiento como se ha comentado anteriormente y devuelve cierto.

Y además puede realizar las siguientes acciones:

⊙ ***`embarque`***: se le pasa por parámetro la mercancía que se desea almacenar en la bodega de carga de la nave. Antes de almacenarla se deben realizar unas comprobaciones:

1. la mercancía no puede estar almacenada ya en otro lugar. En ese caso se devuelve la cadena:

   ```
   ken1.2-2.1: mercancia ya almacenada
   ```

2. si queda espacio libre en la bodega para embarcar la nueva mercancía; si no hay espacio libre se devuelve la cadena:

   `ken1.2-2.1: bodega de carga completa`

donde `ken1.2-2.1` es un posible ejemplo de etiqueta de la mercancía que no se puede almacenar. Si se da más de un motivo por el que no se puede realizar el embarque, el orden de comprobación para la generación de la cadena es el orden en el que aparecen aquí.

Si se cumplen las condiciones necesarias se procede al embarque de la mercancía, de manera que se almacena en la última posición libre que encuentre de la bodega de carga de la nave, pasando un mensaje de almacena a la mercancía, pasándose la propia nave como parámetro, para que actualice su etiqueta y su indicador de carga, y devolver una cadena como la siguiente:

`Icarus1.2: mercancia embarcada`

donde "`Icarus1.2`" es la etiqueta actualizada de la mercancía embarcada.

⊙ ***`embarque`***: se le pasa por parámetro el navegante que desea viajar en la nave. Antes de asignarle una plaza se deben realizar las siguientes comprobaciones:

1. el navegante no puede tener ya un pasaje válido. En ese caso se devuelve `-1`.
2. si quedan plazas libres; si no hay plazas libres se devuelve `-2`.

Si se cumplen estas condiciones se busca la primera posición libre en el pasaje, que será la posible plaza del navegante. Éste debe pagar por su pasaje cuyo coste se calcula:

$$pasaje = (\frac{n}{i+1}) + pasajeros$$

donde n es el número de plazas para pasajeros de la nave, i la primera posición libre y *pasajeros* el número de pasajeros ya embarcados en la nave, teniendo en cuenta que si se trata de un ***`Tripulante`*** tiene un `30%` de descuento.

Si el navegante no puede pagar esta cantidad se devuelve `-3`.

Si se da más de un motivo por el que no se puede realizar el embarque, el orden de comprobación para devolver el valor correspondiente es el orden en el que aparece aquí.

Si se cumplen las condiciones necesarias se procede al embarque del navegante, de manera que se le asigna la plaza encontrada, se incrementa la especia de la nave con el pago del pasaje y se devuelve el número de la plaza asignada.

⊙ ***`solicitudAdmision`***: se le pasa por parámetro un tripulante. En la tripulación de una nave puede haber un máximo de 1 tripulante con el rango `superior` por cada 3 con el rango `subalterno`, de manera que:

1. el tripulante no está enrolado en la tripulación de otra nave ni viaja como pasajero en ninguna nave;
2. si dispone de sitio libre en su tripulación y no se excede la cantidad de miembros con el rango correspondiente;

le asigna la primera plaza libre en su tripulación, le pasa un mensaje para que actualice su conexión invocando su método ***`setConexion`*** y devuelve cierto. En cualquier otro caso, devuelve falso.

- ***solicitudDimision***: se le pasa por parámetro una cantidad de especia[5] y un tripulante. Si pertenece a la tripulación se suma la especia a la propia de la nave, se elimina al tripulante de la tripulación, le pasa un mensaje para que actualice su conexión invocando su método ***setConexion*** y devuelve cierto. En otro caso devuelve falso.

- ***desembarque***: se le pasa por parámetro un entero y un navegante. Se comprueba si dicho navegante ocupa la posición indicada por el entero, si es así lo quita de su pasaje y devuelve cierto. En cualquier otro caso devuelve falso.

- ***repostaje***: se recorre la tripulación pasando un mensaje a cada tripulante para que cobre, de manera que recolecta la especia recogida por los tripulantes para incrementar la especia de la nave.

- ***viaja***: calcula la posibilidad de realizar la travesía o no en función de la cantidad de especia que tiene la nave y la distancia a su destino, ya que se le pasa por parámetro la galaxia destino y las coordenadas del sector de esa galaxia al que se viaja[6]. La distancia se obtiene calculando la distancia euclídea entre ambas coordenadas:

$$d = \sqrt{\sum_{i=0}^{n-1} (x_i - y_i)^2}$$

  donde las x_i se corresponden con la coordenada en la que está situada la nave, y las y_i con la coordenada de destino. Si además la galaxia destino no es en la que se encuentra actualmente, es decir, se trata de un viaje intergaláctico, esta distancia se multiplica por el valor absoluto de la diferencia en el número de sectores, que no sean agujeros negros, que tiene una galaxia y otra[7]. Una vez obtenida la distancia, se divide dicha distancia por la cantidad de especia disponible en la nave, y si esta cantidad es menor o igual que la especia disponible, la travesía se realiza, se le resta a la especia la cantidad calculada, se actualiza la posición de la nave, la galaxia si es necesario y devuelve cierto. En otro caso la travesía no es viable y devuelve falso.

- ***verifica***: se le pasa por parámetro un navegante que está siendo teletransportado. Si no hay posiciones libres disponibles en el pasaje se devuelve `-1`. Si se dispone de posiciones libres, el navegante teletransportado ocupará la última posición libre, devolviendo dicha posición.

- ***getEspecia***: devuelve la especia de la nave.

- ***getPasaje***: devuelve el pasaje de la nave.

- ***getTripulacion***: devuelve la tripulacion de la nave.

- ***getPosicion***: devuelve la posición actual de la nave.

- ***setMapa***: obtiene el mapa de las Romskip y lo toma como propio.

- ***creaFoso***: se le pasa por parámetro un entero. Si dicho entero es mayor que `0` se crea el foso del tamaño indicado por el parámetro. En cualquier otro caso se crea un foso de tamaño `5`. El método devuelve el tamaño con el que se ha creado el foso.

[5] Esta cantidad siempre será mayor o igual a `0`.
[6] La galaxia destino nunca será `null` y no se puede viajar a un agujero negro.
[7] Si la nave no tiene mapa, se considera `0` para el número de sectores.

- ⊙ *crea*: se le pasa por parámetro una coordenada. Se consulta en la galaxia en la que se encuentra la nave si el sector correspondiente a esa coordenada es un agujero negro. Si se trata de un agujero negro, la nave crea un sector para esas coordenadas con un tamaño de almacen igual al tamaño de su foso, si éste existe, en otro caso el tamaño del almacen será 5; a continuación coloca el sector en la galaxia. El método devuelve cierto si coloca el sector y falso en cualquier otro caso.

- ⊙ *almacena*: se le pasa por parámetro un skrogem que debe almacenar en su foso en la primera posición libre que encuentre. Si no dispone de posiciones libres en el foso, la nave busca el sector más próximo al centro de la galaxia en la que se encuentra para transferir todos los parásitos que pueda al almacen de dicho sector, quitarlos de su foso y almacenar el skrogem pasado por parámetro. Las coordenadas de dicho sector se calculan dividiendo por 2 cada una de las dimensiones de la galaxia. Por ejemplo, en una galaxia de dimensiones 3x3x3, el sector central ocupa las coordenadas [1,1,1]. En cambio, en una galaxia de dimensiones 3x3x4, el sector más próximo al centro ocupa las coordenadas [1,1,2]. Si dicho sector es un agujero negro en la galaxia, la nave creará dicho sector. El método devuelve el número de skrogems que quedan finalmente en el foso de la nave.

Restricciones en la implementación

- ⊛ Todas las variables de instancia de las clases deben ser privadas (no accesibles desde cualquier otra clase).

- ⊛ Algunos métodos deben ser públicos y tener una *signatura* concreta:

  - En **Mercancia**
    - `public boolean almacena(Romskip r)`
  - En **Navegante**
    - `public Navegante(String n,double d)`
    - `public Mercancia trabaja(Mercancia m,double t)`
    - `public double cobra()`
    - `public boolean embarca(Dophraki n)`
    - `public boolean desembarca(Dophraki n)`
    - `public double paga(double d)`
    - `public String getNombre()`
    - `public double getEspecia()`
    - `public double getTempo()`
    - `public double getCosecha()`
    - `public int getPasaje()`
  - En **Tripulante**
    - `public Tripulante(String n,double p,String r)`
    - `public int trabaja(double t)`
    - `public int trabaja(double t,int c)`
    - `public double cobra()`
    - `public boolean embarca(Dophraki n)`

- `public boolean desembarca()`
- `public boolean enrolado()`
- `public void setConexion(Dophraki d)`
- `public String getRango()`
- `public Dophraki getConexion()`

- En **Skrogem**
  - `public Skrogem(double p,double[] d,double e)`
  - `public boolean almacena(Romskip r)`
  - `public boolean recogida()`
  - `public boolean busca()`
  - `public double activa()`
  - `public double eliminado()`
  - `public double getEspecia()`
  - `public Navegante getHuesped()`
  - `public Dophraki getNave()`
  - `public static Navegante[] getHuespedes()`
  - `public double recarga()`
- En **Dophraki**
  - `public Dophraki(String n,int i,double x,int j,int k,double e)`
  - `public String embarque(String s,Coordenada c)`
  - `public Mercancia desembarque(String s)`
  - `public boolean teletransporte(Romskip r,int i)`
  - `public boolean verifica(Mercancia m)`
  - `public String embarque(Mercancia m)`
  - `public int embarque(Navegante n)`
  - `public boolean solicitudAdmision(Tripulante n)`
  - `public boolean solicitudDimision(double d,Tripulante n)`
  - `public boolean desembarque(int i,Navegante n)`
  - `public void repostaje()`
  - `public boolean viaja(Galaxia g,Coordenada c)`
  - `public int verifica(Navegante n)`
  - `public double getEspecia()`
  - `public Navegante[] getPasaje()`
  - `public Tripulante[] getTripulacion()`
  - `public Coordenada getPosicion()`
  - `public static void setMapa()`
  - `public int creaFoso(int i)`
  - `public boolean crea(Coordenada c)`
  - `public int almacena(Skrogem s)`
  - `public Skrogem[] getFoso()`

⊛ Ninguno de los ficheros entregados en esta práctica debe contener un método `public static void main(String[] args)`.

⊛ Todas las variables de clase e instancia deben ser privadas. En otro caso se restará un `0.5` de la nota total de la práctica.

Probar la práctica

- En UACloud se publicará un corrector de la práctica con un conjunto mínimo de pruebas (se recomienda realizar pruebas más exhaustivas de la práctica).
- Podéis enviar vuestras pruebas (fichero con extensión `java`, con un método `main` y sin errores de compilación) a los profesores de la asignatura mediante tutorías de UACloud, para obtener la salida correcta a esa prueba. En ningún caso se modificará/corregirá el código de las pruebas. Los profesores contestarán a vuestra tutoría adjuntando la salida de vuestro `main`, si no da errores.
- El corrector viene en un archivo comprimido llamado `correctorP2.tgz`. Para descomprimirlo se debe copiar este archivo donde queramos realizar las pruebas de nuestra práctica y ejecutar:

  ```
  tar xfvz correctorP2.tgz
  ```

  De esta manera se extraerán de este archivo:

  - El fichero `corrige.sh`: *shell-script* que tenéis que ejecutar.
  - El directorio `practica2-prueba`: dentro de este directorio están los ficheros con extensión `.java`, programas en Java con un método `main` que realizan una serie de pruebas sobre la práctica, y los ficheros con el mismo nombre pero con extensión `.txt` con la salida correcta para la prueba correspondiente.
- Una vez que tenemos esto, debemos copiar nuestros ficheros fuente (sólo aquellos con extensión `.java`) al mismo directorio donde está el fichero `corrige.sh`.
- Sólo nos queda ejecutar el *shell-script*. Primero debemos darle permisos de ejecución si no los tiene. Para ello ejecutar:

  ```
  chmod +x corrige.sh
  corrige.sh
  ```
- Una vez se ejecuta `corrige.sh` los ficheros que se generarán son:

  - `errores.compilacion`: este fichero sólo se genera si el corrector emite por pantalla el mensaje "`Error de compilacion: 0`" y contiene los errores de compilación resultado de compilar los fuentes de una práctica particular. Para consultar el contenido de este fichero se puede abrir con cualquier editor de textos (gedit, kate, etc.).
  - Fichero con extensión `.tmp.err`: este fichero debe estar vacío por regla general. Sólo contendrá información si el corrector emite el mensaje "`Prueba p01: Error de ejecucion`", por ejemplo para la prueba `p01`, y contendrá los errores de ejecución producidos al ejecutar el fuente `p01` con los ficheros de una práctica particular.
  - Fichero con extensión `.tmp`: fichero de texto con la salida generada por pantalla al ejecutar el fuente correspondiente, por ejemplo `p01.tmp` contendrá la salida generada al ejecutar el fuente `p01` con los ficheros de una práctica particular.

  Si en la prueba no sale el mensaje "`Prueba p01: ok`", por ejemplo para la prueba `p01`, o algún otro de los comentados anteriormente, significa que hay diferencias en las salidas para esa prueba, por tanto se debe comprobar que diferencias puede haber entre los ficheros `p01.txt` y `p01.tmp`. Para ello ejecutar en línea de comando, dentro del directorio `practica1-prueba`, la orden: `diff -w p01.txt p01.tmp`