Práctica 2: Programas en la TPMV

Fecha de entrega: 7 de Enero de 2014, 16:00

Objetivo: Herencia y polimorfismo: uso e implementación

1. Nueva funcionalidad

La funcionalidad principal de esta nueva práctica consiste en añadir a la TPMV el concepto de *programa*. En efecto, hasta ahora la TPMV consistía únicamente de una memoria y una pila de operandos; las instrucciones que ejecutaba eran las que el usuario iba introduciendo paulatinamente por teclado.

En esta nueva práctica vamos a extender la implementación de la TPMV para que maneje una lista de instrucciones a ejecutar (que llamaremos *programa-mv*) y tenga un *contador de programa* que marque cuál es la siguiente instrucción. De esta forma el proceso de uso de la aplicación consistirá en dos partes claramente diferenciadas:

- En la primera parte, el usuario introducirá (por orden) todas las instrucciones que componen el programa. Cuando haya terminado de escribir todas esas instrucciones, tecleará "`END`" (que no es una instrucción válida).
- En la segunda parte, el usuario podrá ejecutar el programa utilizando diferentes comandos que permitirán su ejecución completa, paso a paso o por lotes. Esta segunda fase (y la aplicación completa) terminará cuando el programa-mv termine, o el usuario teclee "`QUIT`".

Además, y siguiendo con la evolución de nuestra TPMV, añadiremos soporte para nuevas instrucciones. En concreto, instrucciones de comparación y de manejo de valores booleanos e instrucciones de salto que modifican el *contador de programa* para permitir la ejecución de bucles. Finalmente, la aplicación se debe lanzar desde la clase `tp.pr2.mv.Main`.

2. Primera parte: registro del programa

Nada más arrancar la aplicación, la TPMV estará vacía (es decir, la cpu estará sin contenido en la pila de operandos, memoria y programa). La aplicación solicitará entonces

al usuario que introduzca, una a una, todas las instrucciones del programa-mv a ejecutar. Debido a la existencia de instrucciones de salto, la última instrucción del mismo *no* tiene por qué ser `HALT` por lo que el usuario utilizará la palabra `END` para indicar que ya no quedan más instrucciones que añadir.

Esta primera parte terminará mostrando el programa-mv completo cargado en la máquina, donde además cada instrucción irá acompañada de un número que indica su orden dentro del programa-mv. A continuación aparece un ejemplo de ejecución en el que se introducen tres instrucciones (se incluye también la segunda fase de la ejecución en la que únicamente se solicita terminar):

```
Introduce el programa fuente
> push 1
> push 2
> ADD
> end
El programa introducido es:
0: PUSH 1
1: PUSH 2
2: ADD
> quit
```

3. Segunda parte: ejecución del programa

En la segunda fase la aplicación mostrará, igual que antes, un *prompt* (>) en donde el usuario podrá ejecutar distintos comandos de ejecución del programa. Los comandos disponibles serán los siguientes:

- `STEP`: hace que la TPMV ejecute una instrucción (la siguiente del programa-mv de acuerdo al contador de programa).
- `STEP <num instrucciones>`: similar a la anterior, permite indicar cuántas instrucciones se quieren ejecutar.
- `RUN`: ejecuta el programa-mv completo hasta su terminación.
- `QUIT`: sale de la aplicación.

En los tres comandos que involucran ejecución de instrucciones hay que tener en cuenta que:

- Tras la ejecución de *cada instrucción* máquina, se mostrará el estado de la máquina virtual de forma similar a lo que se hacía en la práctica anterior. Eso significa que, por ejemplo, el `STEP 4` mostrará cuatro veces el estado de la máquina, lo que permitirá ver su evolución instrucción a instrucción.
- Si la ejecución de la(s) instrucción(es) hace que el programa-mv llegue a su final (ya sea porque el contador de programa apunta a una instrucción fuera de las disponibles, o por la ejecución de `HALT`), la aplicación terminará (después de mostrar el estado tras la ejecución de la última instrucción).
- Si al ejecutar una instrucción ésta falla, se dejarán de ejecutar el resto de instrucciones solicitadas y se volverá a mostrar el prompt (esto afecta a `STEP n` y a `RUN`).

La sección 6 muestra algún ejemplo de ejecución.

4. Nuevas instrucciones

Como ya hemos dicho, en esta práctica se añadirán numerosas instrucciones a nuestra TPMV. En concreto, las instrucciones nuevas son las siguientes:

- Operación unaria de negación, `NEG`: cambia el signo al valor almacenado en la cima de la pila. Aunque en realidad esta instrucción no aporta mayor expresividad a la TPMV (equivale a la secuencia `PUSH 0`, `FLIP`, `SUB`), sí facilita la comprensión del código y hace que los programas que la utilicen sean más rápidos y cortos.

- Operaciones de comparación, `LT`, `GT`, `EQ`, `LE`: todas cogen la subcima y la cima de la pila y comparan sus valores enteros. Si la comparación es satisfactoria, apilan un 1. Si no lo es, apilan un 0. `LT` ("*less-than*") equivale a la comparación $subcima < cima$; `GT` ("*greater-than*") a $subcima > cima$; *EQ* ("*equals to*") $subcima == cima$ y por último `LE` ("*less-or-equal*") a $subcima \leq cima$.

- Operaciones booleanas, `AND`, `OR` y `NOT`: equivalen a las operaciones `&&`, `||` y `!` de lenguajes como C++. Cogen de la pila los dos operandos (o el operando, en el caso de la `NOT`) y lo interpretan como un valor binario (donde 0 indica *falso* y cualquier otra cosa es *cierto*). Realizan la operación booleana y apilan el resultado (0=falso, 1=cierto).

- Salto incondicional, `JUMP n`: provoca un cambio en el contador de programa en base al parámetro de la instrucción. Tras el salto, la siguiente instrucción que se ejecutará será la que ocupe la posición `n` en el programa (ten en cuenta que la primera instrucción del programa es la *cero* no la *uno*).

- Saltos condicionales, `BT n` y `BF n`: son instrucciones donde el salto únicamente se produce bajo una condición concreta relacionada con el valor de la cima de la pila. El primero, `BT` ("*branch-if-true*") salta (a la instrucción indicada en el parámetro) si el valor de la cima de la pila es cierto (cualquier cosa que no sea cero), mientras que `BF` ("*branch-if-false*") salta si el valor es falso. Como siempre, las dos *eliminan* el valor de la cima de la pila.

5. Implementación

Para implementar esta nueva versión de la TPMV necesitarás al menos las siguientes clases:

- Memory y OperandStack son similares a las de la práctica 1.

- ProgramMV: Clase que representa un programa. Por lo tanto contendrá la secuencia completa de instrucciones de las que consta el programa.

- CPU: Además de la memoria y la pila de operandos, la CPU tendrá un contador de programa que indica la siguiente instrucción a ejecutar y el programa actual cargado. El método de la práctica 1 que permitía ejecutar una instrucción pasada como parámetro desaparece y aparecen (al menos) los dos siguientes:

  - public void loadProgram(ProgramMV): que permite cargar un programa.

    - public boolean step(): que ejecuta la siguiente instrucción, devolviendo false si hay algún error. Se encargará también de gestionar el contador de programa, para que la siguiente invocación a step() ejecute la instrucción de la TPMV o la instrucción a la que se saltó.

- Instruction: la clase de la práctica anterior sufrirá una refactorización para eliminar las distinciones de casos que en la práctica anterior aparecían en el execute(Instruction). Se convierte en una clase que dispone del método `abstract public boolean execute(...)` –los parámetros del método se discutirán en clase. A partir de esta clase debe implementarse la jerarquía de instrucciones de la nueva versión de la TPMV. Por ejemplo podemos considerar los siguientes bloques de instrucciones comunes:

    - Operaciones aritméticas: `ADD`, `SUB`, `MULT` y `DIV`.
    - Operaciones booleanas: `AND`, `OR` y `NOT`.
    - Instrucciones de salto. Contienen tanto a las de salto condicional como incondicional, es decir `BT n`, `BF n`, `JUMP n`.
    - Instrucciones de comparacion: `LT`, `GT`, `EQ`, `LE`.
    - Resto de instrucciones todas ellas secuenciales.

  Piensa con cuidado como realizar la jerarquía de clases de forma que sea lo mayor reutilizable posible. En clase se darán más indicaciones sobre como realizar dicha jerarquía.

- InstructionParser: Similar a la de la práctica 1 añadiendo el nuevo conjunto de instrucciones. Ahora en vez de devolver objetos de la clase Instruction configurados para representar cada instrucción concreta, devolverá objetos de las clases derivadas.

Haz una distribución de las clases en paquetes de forma que cada paquete contenga un conjunto de clases que mantengan alguna relación.

6. Ejemplos de ejecución

El siguiente ejemplo muestra el uso de las instrucciones **`STEP`** y **`STEP n`**. Observa que el `STEP 5` termina prematuramente porque la TPMV alcanza el final del programa, lo que hace que la aplicación termine.

```
Introduce el programa fuente
> push 2
> push 3
> add
> halt
> end
El programa introducido es:
0: PUSH 2
1: PUSH 3
2: ADD
3: HALT
> Step
Comienza la ejecución de PUSH 2
El estado de la maquina tras ejecutar la instrucción es:
Memoria: <vacia>
```

```
Pila de operandos: 2
> Step 5
Comienza la ejecución de PUSH 3
El estado de la maquina tras ejecutar la instrucción es:
Memoria: <vacia>
Pila de operandos: 2 3
Comienza la ejecución de ADD
El estado de la maquina tras ejecutar la instrucción es:
Memoria: <vacia>
Pila de operandos: 5
Comienza la ejecución de HALT
El estado de la maquina tras ejecutar la instrucción es:
Memoria: <vacia>
Pila de operandos: 5
```

La siguiente ejecución demuestra qué sucede cuando se produce un error en alguna instrucción de la TPMV; en este caso hay un fallo lógico en el programa, en el que se han intercambiado la segunda y la tercera instrucción. Al ejecutar el ADD mal colocado la ejecución se detiene con un pequeño mensaje y se le muestra el prompt al usuario. Si intenta proseguir la ejecución, el error volverá a aparecer, pues la TPMV volverá a intentar ejecutar la misma instrucción incorrecta.

```
Introduce el programa fuente
> Push 65
> Add
> Push 2
> out
> end
El programa introducido es:
0: PUSH 65
1: ADD
2: PUSH 2
3: OUT
> step 15
Comienza la ejecución de PUSH 65
El estado de la maquina tras ejecutar la instrucción es:
Memoria: <vacia>
Pila de operandos: 65
Comienza la ejecución de ADD
Error en la ejecución de la instruccion
> step
Comienza la ejecución de ADD
Error en la ejecución de la instruccion
> quit
```

En el siguiente ejemplo se puede ver que, igual que en la práctica anterior, el usuario puede confundirse al introducir tanto las instrucciones como los comandos.

```
Introduce el programa fuente
> apila 65
Error: Instrucción incorrecta
> push 65
> out
> end
El programa introducido es:
0: PUSH 65
```

```
1: OUT
> ejecuta 3
No te entiendo.
> step 3
Comienza la ejecución de PUSH 65
El estado de la maquina tras ejecutar la instrucción es:
Memoria: <vacia>
Pila de operandos: 65
Comienza la ejecución de OUT
A
El estado de la maquina tras ejecutar la instrucción es:
Memoria: <vacia>
Pila de operandos: <vacia>
```

Por último, a continuación se introduce un programa-mv que calcula el factorial de 4 y lo deja en la posición 1 de memoria. Como puedes observar, el programa tiene una comparación y un par de saltos para implementar la versión iterativa del factorial. Incluímos la ejecución de unas pocas instrucciones. No mostramos el proceso completo pues es demasiado largo para incluirlo en el enunciado, pero puedes probar el programa en tu propia práctica para comprobar que, efectivamente, calcula el factorial de 4.

```
Introduce el programa fuente
> push 0
> store 0
> push 1
> store 1
> load 0
> push 3
> gt
> bt 19
> load 1
> load 0
> mult
> load 1
> add
> store 1
> load 0
> push 1
> add
> store 0
> jump 4
> halt
> end
El programa introducido es:
0: PUSH 0
1: STORE 0
2: PUSH 1
3: STORE 1
4: LOAD 0
5: PUSH 3
6: GT
7: BT 19
8: LOAD 1
9: LOAD 0
10: MULT
11: LOAD 1
```

```
12: ADD
13: STORE 1
14: LOAD 0
15: PUSH 1
16: ADD
17: STORE 0
18: JUMP 4
19: HALT
> step
Comienza la ejecución de PUSH 0
El estado de la maquina tras ejecutar la instrucción es:
Memoria: <vacia>
Pila de operandos: 0
> step 2
Comienza la ejecución de STORE 0
El estado de la maquina tras ejecutar la instrucción es:
Memoria: [0]:0
Pila de operandos: <vacia>
Comienza la ejecución de PUSH 1
El estado de la maquina tras ejecutar la instrucción es:
Memoria: [0]:0
Pila de operandos: 1
> step
Comienza la ejecución de STORE 1
El estado de la maquina tras ejecutar la instrucción es:
Memoria: [0]:0 [1]:1
Pila de operandos: <vacia>
> quit
```

Observa que el factorial de 4 está almacenado en la posición de memoria 1.

7. Sugerencias adicionales de implementación

Para la implementación de la segunda parte de la aplicación (la que permite ejecutar el programa-mv paso a paso), puedes construir una jerarquía de clases similar a la de Instruction-InstructionParser:

- CommandInterpreter: clase abstracta que representa los distintos comandos que podemos ejecutar sobre la máquina virtual. Entre sus atributos tendrá a la CPU cpu y dispondrá de un método abstracto public abstract boolean executeCommand() que implementarán sus distintas subclases de acuerdo con el comando que se desee ejecutar. Esta clase ofrece el metodo public static void configureCommandInterpreter para inicializar el valor del atributo cpu una vez que la CPU dispone del programa a ejecutar.

  De esta clase heredarán las subclases Run, Step, Steps y Quit que implementan, respectivamente, los comandos con el mismo nombre. Las clases Steps y Run heredan de la clase Step.

- CommandParser: Clase que se encarga de generar los comandos. Contiene el método estático public static CommandInterpreter parseCommand(String line), que procesa el string de entrada y o bien devuelve un objeto de la clase CommandInterpreter o null en caso de que la línea de entrada no se corresponda con ningún comando.

8. Parte Opcional

Opcionalmente puedes mejorar la aplicación en dos aspectos: dar más funcionalidad a la segunda parte de la aplicación y/o añadir otras tres instrucciones nuevas a la TPMV.

8.1. Depuración del programa

Como hemos visto, en la segunda fase de ejecución de nuestra aplicación, si la TPMV alcanza una instrucción que no puede ejecutarse, se muestra un error y la aplicación se para esperando que el usuario tome una decisión.

En realidad, el usuario de nuestra aplicación puede hacer más bien poco: tras analizar cómo ha ido evolucionando el estado de la TPMV podrá determinar qué ha sido lo que ha provocado ese error, pero poco más.

En esta parte opcional le permitiremos al usuario que "arregle" el estado de la TPMV que ha provocado el fallo y prosiga la ejecución. Para eso, además de las ya vistas `STEP [<n>]` y `RUN`, también podrá usar:

- `PUSH n`: añadirá un valor a la cima de la pila.
- `POP`: eliminará la cima de la pila (si existe).
- `WRITE pos value`: escribirá en la posición de memoria indicada el valor dado.

Como ejemplo, se muestra la forma en la que se puede arreglar (sobre la marcha) la ejecución del programa mostrado anteriormente donde había dos instrucciones intercambiadas:

```
Introduce el programa fuente
> Push 65
> Add
> Push 2
> out
> end
El programa introducido es:
0: PUSH 65
1: ADD
2: PUSH 2
3: OUT
> run
Comienza la ejecución de PUSH 65
El estado de la maquina tras ejecutar la instrucción es:
Memoria: <vacia>
Pila de operandos: 65
Comienza la ejecución de ADD
Error en la ejecución de la instruccion
> push 2
> step
Comienza la ejecución de ADD
El estado de la maquina tras ejecutar la instrucción es:
Memoria: <vacia>
Pila de operandos: 67
> step
Comienza la ejecución de PUSH 2
El estado de la maquina tras ejecutar la instrucción es:
```

```
Memoria: <vacia>
Pila de operandos: 67 2
> pop
> step
Comienza la ejecución de OUT
C
El estado de la maquina tras ejecutar la instrucción es:
Memoria: <vacia>
Pila de operandos: <vacia>
```

8.2. Instrucciones relativas de salto

Las instrucciones de salto anteriores, `JUMP`, `BT`, `BF` tienen un parámetro que indica la *dirección absoluta* del salto. Estas instrucciones dificultan mucho la *relocalización* del código y dificultan la labor de los compiladores. Es por esto que las máquinas suelen estar también provistas de instrucciones de salto cuyas direcciones son relativas al contador de programa.

Añade las versiones de saltos relativos de las tres instrucciones, `RJUMP`, `RBT` y `RBF` de forma que, por ejemplo, `RJUMP 2` incremente el contador de programa en dos unidades (en vez de saltar a la instrucción 2 del programa-mv). Eso significa que la instrucción siguiente a `RJUMP` *no* se ejecutará. Fijate que:

- `RJUMP 0` provocaría que la máquina entre en un bucle infinito (el contador de programa no se modifica).
- `RJUMP 1` es una instrucción sin efecto, pues hace que se ejecute la siguiente instrucción a *RJUMP* (algunas arquitecturas *hardware* utilizan una instrucción equivalente a esta para simular la operación `NOP` disponible en muchos lenguajes ensambladores).
- Estos saltos pueden tener parámetros negativos para saltar hacia atrás en el programa.

A continuación se muestra una ejecución en la que se introduce un programa-mv que comprueba si el número apilado en la primera instrucción es primo (escribe `P`) o no (escribe `N`). El programa (cuya ejecución no se muestra entera) utiliza todas las instrucciones de salto relativo añadidas.

```
Introduce el programa fuente
> push 17
> dup
> store 0
> push 2
> dup
> store 1
> div
> store 2
> load 0
> dup
> load 1
> div
> load 1
> mult
> eq
> not
```

```
> load 1
> load 2
> le
> and
> rbf 6
> load 1
> push 1
> add
> store 1
> rjump -17
> load 1
> load 2
> le
> rbt 4
> push 80
> out
> rjump 5
> push 78
> out
> end
El programa introducido es:
0: PUSH 17
1: DUP
2: STORE 0
3: PUSH 2
4: DUP
5: STORE 1
6: DIV
7: STORE 2
8: LOAD 0
9: DUP
10: LOAD 1
11: DIV
12: LOAD 1
13: MULT
14: EQ
15: NOT
16: LOAD 1
17: LOAD 2
18: LE
19: AND
20: RBF 6
21: LOAD 1
22: PUSH 1
23: ADD
24: STORE 1
25: RJUMP -17
26: LOAD 1
27: LOAD 2
28: LE
29: RBT 4
30: PUSH 80
31: OUT
32: RJUMP 5
33: PUSH 78
34: OUT
```

```
> step 2
Comienza la ejecución de PUSH 17
El estado de la maquina tras ejecutar la instrucción es:
Memoria: <vacia>
Pila de operandos: 17
Comienza la ejecución de DUP
El estado de la maquina tras ejecutar la instrucción es:
Memoria: <vacia>
Pila de operandos: 17 17
> quit
```

9. Entrega de la práctica

La práctica debe entregarse utilizando el mecanismo de entregas del campus virtual, no más tarde de la fecha y hora indicada en la cabecera de la práctica.

El fichero debe tener al menos el siguiente contenido[1]:

- Directorio `src` con el código de todas las clases de la práctica.
- Fichero `alumnos.txt` donde se indicará el nombre de los componentes del grupo.

[1] Puedes incluir también opcionalmente los ficheros de información del proyecto de Eclipse

Apéndice

1. Detalles de Implementación. TP grupo B

- **ProgramMV** (Package tp.pr2.mv.cpu): Clase que representa un programa. Por lo tanto contendrá la secuencia completa de instrucciones de las que consta el programa. Recuerda que la primera instrucción del programa es la cero, no la uno. Sirve para almacenar un programa de la máquina virtual. Tiene un método para añadir una nueva instrucción al final del programa public void addInstruction(Instruction).

- **ExecutionManager**: Clase que gestiona la ejecución del programa. Contiene un contador (`currentPc`) de la instrucción a ejecutar, un contador de la siguiente instrucción a ejecutar (`nextPc`) y una variable booleana (`halt`) que indique si la ejecución ha terminado. El contador `nextPc` sirve para registrar la siguiente instrucción en caso de que se haya ejecutado una instrucción de salto. Contiene (entre otros) un método `public void incrementPc()` que actualiza los contadores cada vez que se ejecuta con éxito una instrucción del programa.

- **CPU**: (Package tp.pr2.mv.cpu) Además de la memoria y la pila de operandos, la CPU tendrá el programa actual cargado. Para el control de ejecución, la CPU tendrá un gestor de ejecución. El método de la práctica 1 que permitía ejecutar una instrucción pasada como parámetro desaparece y aparecen (al menos) los dos siguientes:

  - public void loadProgram(ProgramMV): que permite cargar un programa.
  - public boolean step(): que ejecuta la siguiente instrucción, devolviendo false si hay algún error. La siguiente instrucción a ejecutar es la indicada por el gestor de ejecución de la CPU.La siguiente invocación a step() debe ejecutar la instrucción de la TPMV o la instrucción a la que se saltó.

  La clase CPU también dispone de un método public boolean isHalted() que permite saber si la máquina está parada o no (devuelve true si la máquina está parada). La máquina está parada si ejecutó la instrucción HALT o el contador de programa alcanzó una instrucción que no existe.

- Instruction:(Package tp.pr2.mv.ins) La clase de la práctica anterior sufrirá una refactorización para eliminar las distinciones de casos que en la práctica anterior aparecían en el método execute(Instruction). Se convierte en una clase que dispone del método abstracto abstract public boolean execute(Memory, OperandStack, ExecutionManager).

A partir de esta clase debe implementarse la jerarquía de instrucciones de la nueva versión de la TPMV. Por ejemplo podemos considerar los siguientes bloques de instrucciones:

- Operaciones aritméticas: `ADD`, `SUB`, `MULT` y `DIV`.
- Operaciones booleanas: `AND`, `OR` y `NOT`.
- Instrucciones de salto. Contienen tanto a las de salto condicional como incondicional, es decir `BT n`, `BF n`, `JUMP n`.
- Instrucciones de comparación: `LT`, `GT`, `EQ`, `LE`.
- Etc.

Ten en cuenta que podemos tener varios niveles de instrucciones en la jerarquía.

Cada tipo de instrucción se corresponde con una clase diferente, por lo que tendremos que crear las clases derivadas (subclases) Add, Sub, Jump, Store, etc.

Piensa con cuidado cómo realizar la jerarquía de clases de forma que sea lo más reutilizable posible. Los bloques anteriores sugieren la creación de la clase abstracta Arithmetic como subclase de Instruction, y la creación de las clases Add, Sub, etc. como subclases de Arithmetic. El método abstracto abstract public boolean execute(Memory, OperandStack, ExecutionManager) ha de implementarse en algún punto de la jerarquía, de forma que sea lo más reutilizable posible. Este método puede redefinirse en caso de ser necesario.

Presta atención a la hora de definir la visibilidad de los miembros de las clases de la jerarquía.

- InstructionParser: (Package tp.pr2.mv.ins) Similar a la de la práctica 1 añadiendo el nuevo conjunto de instrucciones. Ahora el método estático parse, en vez de devolver objetos de la clase Instruction configurados para representar cada instrucción concreta, devolverá objetos de las clases derivadas.

- CommandInterpreter: (Package tp.pr2.mv.command) Clase abstracta que representa los distintos comandos que podemos ejecutar sobre la máquina virtual. Entre sus atributos tendrá a la CPU cpu y una variable booleana para determinar si la ejecución de la aplicación ha terminado (si se ha ejecutado quit). Dispondrá de un método abstracto public abstract boolean executeCommand() que implementarán sus distintas subclases de acuerdo con el comando que se desee ejecutar. Esta clase ofrece el método public static void configureCommandInterpreter para inicializar el valor del atributo cpu una vez que la CPU dispone del programa a ejecutar. También dispondrá de un método public CommandInterpreter parseComm (String cadena) que procesa el string de entrada y o bien devuelve un objeto de la clase CommandInterpreter o null en caso de que la línea de entrada no se corresponda con ningún comando. Por último, el método estático public static boolean isQuit() devuelve true si el usuario ha ejecutado el comando QUIT.

  De esta clase heredarán las subclases Run, Step, Steps y Quit que implementan, respectivamente, los comandos con el mismo nombre. Las clases Steps y Run heredan de la clase Step.

- CommandParser: (Package tp.pr2.mv.command) Clase que se encarga de generar los comandos. Contiene un atributo estático commands que es un array de objetos de la clase CommandInterpreter:

```
private static CommandInterpreter[] commands = {
     new step(), new steps(),  ...
};
```

Contiene el método estático public static CommandInterpreter parseCommand(String line), que procesa el string de entrada y o bien devuelve un objeto de la clase CommandInterpreter o null en caso de que la línea de entrada no se corresponda con ningún comando. Este método recorre mediante un bucle cada uno de los elementos del atributo commands y lo parsea.