Pablo Edu García

PROCESO DE IMPLEMENTACIÓN

“HALTING PROBLEM” EN JAVA

LUNES 29 ABRIL - JUEVES 2 MAYO

Una vez habiendo leído el enunciado de la práctica, lo que he entendido es que debo realizar una simulación interactiva que permita obtener las observaciones de Alan Turing con respecto al “Problema de la parada”. Para hacer esto he considerado que habría que hacer los siguientes procedimientos:

1. Crear una clase donde se desarrolle un programa de incremento infinito, donde haya un bucle while que, dado un número se ejecute hasta que se obtenga un valor menor a dicho número. Este hecho es imposible que se produzca por lo que en condiciones normales el bucle nunca se interrumpiría.

2. Crear una clase donde se desarrolle un programa de decremento finito, donde haya un bucle while que, dado un número entero positivo se ejecute hasta que se obtenga un valor igual a cero, momento en el que se interrumpiría el bucle.

3. Crear una interfaz *haltChecker*, que sea implementada por una clase y que permita añadir *HaltChecker* como parámetro en cada una de las clases anteriores.

4. Una clase Reverser, que implemente *haltChecker* y que según el resultado obtenido por esta función, se ejecute de manera contraria, es decir si dice que el programa va a detenerse, ejecute un bucle infinito y si dice que no va a detenerse, se detenga.

Para poder realizar una implementación adecuada del problema he pensado que el patrón de diseño más adecuado podría ser **Strategy**.

Este patrón nos permite definir una familia de algoritmos, colocar cada uno de ellos en una clase separada y hacer sus objetos intercambiables. En este caso podría ser útil ya que la función haltChecker deberá ser implementada de distintas formas según el programa que se quiera probar.

VIERNES 3 MAYO

Después de llevar varios días tratando de implementar el patrón Strategy, he empezado a dudar de si es el más adecuado para este proyecto. Por ello, he decidido centrarme en primer lugar en patrones Creacionales que son más sencillos de implementar en un principio. He optado por utilizar el patrón de diseño creacional **Abstract Factory** para la creación de instancias de los distintos programas. Tendríamos una interfaz Programa, que sería implementada por las distintas fábricas concretas que se encargarían de la creación de los programas Interrumpible Ininterrumpible y Reverser.

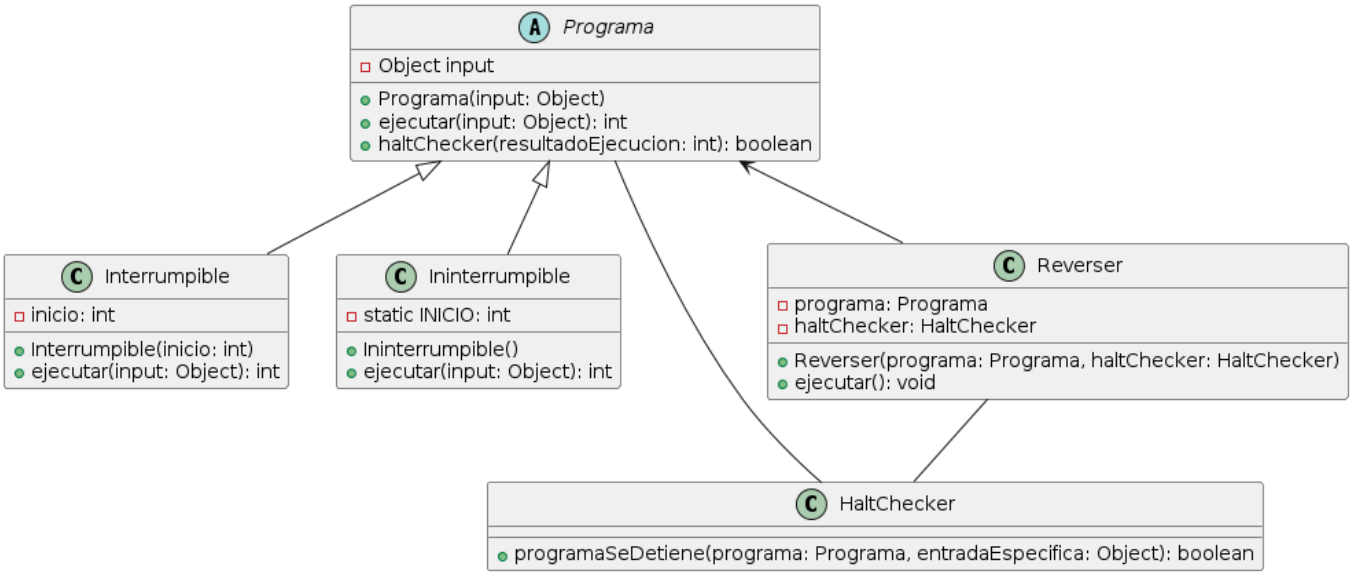
SÁBADO 4 - LUNES 6 MAYO
He decidido hacer una serie de cambios en la implementación del programa.

1. En primer lugar, en vez de crear haltChecker como un programa individual que sigue el contrato de una interfaz podría implementarse simplemente como una clase, prescindiendo de la interfaz ya que no va a comprobar de distinta forma si un programa se detiene o no. Si la ejecución del programa devuelve 0 se va a parar y si no no (al menos con nuestra implementación particular). Esta clase tendría una relación de asociación con los programas que la usen

2. Otro cambio que se me ha ocurrido es que, en vez de aplicar un Abstract Factory, aplicar un patrón **Factory Method** que puede ser más óptimo ya que reduciríamos el número de clases necesarias simplificando el programa general.

Para llevar a cabo esta nueva implementación se crearía una clase abstracta *Programa* que extienda a cada uno de los programas individuales, y que cuente con la función *haltChecker* (de forma que pueda ser utilizada por los distintos programas), una función estática llamada *crearPrograma* (al ser estática podrá ser accedida desde cualquier parte del código sin la necesidad de crear instancias) y una función abstracta de tipo entero llamada *ejecutar* (que se implemente según los requerimientos de cada programa particular y que devuelva un número que indicará a *haltChecker* si se ha interrumpido o no el programa).

El diagrama de clases quedaría así:



MARTES 7 - VIERNES 10 MAYO
Una vez hecho esto ya podemos centrarnos en cumplir la finalidad del programa: Determinar si un bucle dentro de un programa concreto es o no infinito. Para hacer esto lo que haremos es analizar los 2 programas y buscar elementos concretos que nos verifiquen que esto se cumple. Esto lo haremos mediante el patrón de diseño de comportamiento **Chain of Responsability**, en el cual definiremos clases concretas (Handlers) que verifiquen que se cumplan sentencias muy concretas, uno que busque palabras reservadas como while o for, que nos indican la existencia de un bucle, elementos de comparación como '<' o '==' para comprobar la sentencia del bucle, etc.
En resumen lo que haremos es un analizador de sentencias sintácticas de nuestros programas por lo que el patrón Strategy quedaría descartado en este caso.

Obviamente analizar un código como tal, que es lo primero que se me ocurrió es muy complejo así que en su lugar crearemos 2 archivos txt, los convertiremos en cadena de texto (String) que le iremos pasando a cada Handler para que busque la sentencia concreta que le indiquemos. Para definir qué es lo que debe buscar cada manejador haremos uso de sentencias “regex” que nos permitirán verificar muchos casos en los que pueda aparecer la sentencia buscada.

Todo esto lo manejaremos desde el main, donde crearemos las instancias de cada Handler y estableceremos el orden de llamada de cada uno de ellos.

Es prácticamente imposible determinar todos los casos en los que un programa puede pararse o no, incluso acotando las posibilidades a nuestro caso particular en el que tenemos un programa que va contando hacia arriba de forma indefinida y otro que cuenta hacia abajo hasta un límite es relativamente complicado. Por ello la solución a la que he llegado es que la función haltChecker verifique el estado de una serie de flags que he añadido en los ciertos handlers, en mi caso si se verifica en alguno de los códigos la existencia de la palabra reservada while, un incremento de uno a uno y un comparador mayor que se activarán estas flags como true, los valores de estas variables serán pasados a haltChecker, que en caso de encontrar los 3 valores como true, determinará que el bucle es infinito.

```java
public class HaltChecker {
    private WhileLoopHandler whileLoopHandler;
    private IncrementoDirectoHandler incrementoDirectoHandler;
    private ComparadorMayorHandler comparadorMayorHandler;

    public HaltChecker(WhileLoopHandler whileLoopHandler, IncrementoDirectoHandler incrementoDirectoHandler,
            ComparadorMayorHandler comparadorMayorHandler) {
        this.whileLoopHandler = whileLoopHandler;
        this.incrementoDirectoHandler = incrementoDirectoHandler;
        this.comparadorMayorHandler = comparadorMayorHandler;
    }

    public boolean haltCheck() {
        boolean flag1 = WhileLoopHandler.isPosibleInfinito();
        boolean flag2 = IncrementoDirectoHandler.isPosibleInfinito();
        boolean flag3 = ComparadorMayorHandler.isPosibleInfinito();
        if (flag1 && flag2 && flag3) {
            return true;
        } else {
            return false;
        }

    }
```

Otro detalle importante es que el primer handler que se ejecuta en la cadena de responsabilidad es el llamado SyntaxAnalysisHandler. En esta clase se transforma el fichero txt en cadena de texto, así que además de formar parte del patrón de diseño Chain of Responsability, podría considerarse como una implementación muy simplificada del patrón de diseño estructural **Adapter**.