1. Planteamiento del ejercicio:

Se nos pide simular el Problema de Parada propuesto por Alan Turing implementando patrones de comportamiento. Así que, en primer lugar, hace falta subdividir el ejercicio en problemas más pequeños. De este modo, encontramos dos problemas principales:

1. Entender y analizar el código.
2. Simular el problema de la parada.

Estos a su vez se subdividen en:

1.1. Reconocer el código.

1.2. Entender donde hay bucles y otras razones que puedan afectar a la salida del problema de la parada.

2.1. Saber si un programa parará o si, por el contrario, no termina nunca.

2.2. Simular un bucle infinito (para el Reverser) sin que realmente sea infinito ya que, en este ejercicio, el programa debe parar.

En este momento es cuando nos planteamos si algún patrón de diseño podría ayudarnos a resolver el ejercicio y, efectivamente, la Cadena de Responsabilidad (Chain of Responsability) nos ayuda tanto para el análisis del código como para las llamadas a las distintas máquinas. También encontramos un posible uso de un Método de Fábrica (Factory Method) para la creación del HaltChecker y el Reverser.

2. Resolución de los desafíos:

En primer lugar, vamos a analizar el código. Para eso necesitamos un método que lea el código y que nos lo devuelva como un String. Y es por eso que creamos una clase que gestione ficheros (“src/utils/IO/FileManager.java”) que tenga un método que lea los ficheros y que, gracias a un StringBuilder, genere un String con el código que luego analizaremos.

Tras esto, crearemos el analizador (“src/utils/Analyzer/CodeAnalyzer.java”) que tendrá una cadena de responsabilidad, tomando, en primer, lugar el código y verificando si tiene o no un bucle while. Hemos decidido que sea un bucle while por la simplicidad de los códigos de ejemplo que meteremos pero habría que comprobar si contiene cualquier tipo de bucle, ya que si no lo contiene, el programa siempre terminaría. Seguidamente, revisamos si el bucle (“src/utils/Analyzer/LoopAnalyzer.java”) para determinar si en algún momento termina o no. De la misma manera que antes, por simplicidad, solo comprobaremos si llega o no a cumplir la condición del bucle. Es aquí donde nos

damos cuenta que también nos hacen falta clases auxiliares que nos permitan el tratamiento de la salida de estos métodos.

Por último, creamos dos Máquinas llamadas HaltChecker (que comprueba si el programa introducido, junto con su input, se detiene o no) y Reverser (que en primer lugar duplica la entrada para metérsela como programa y como input a otra máquina; y, según el resultado, devuelve algo o ejecuta un bucle infinito).

3. Implementación y corrección de errores:

Para la representación del problema se ha implementado una GUI que ofrece los resultados del Reverser con los diferentes ejemplos como entrada y del HaltChecker con los diferentes ejemplos como entrada y unos input impuestos (en versiones futuras se le podría proponer al usuario elegir los input).

Para mejorar la legibilidad del código se han añadido comentarios, diagramas UML y JavaDoc.

También se ha creado un par de fábricas para las máquinas HaltChecker y Reverser por si en un futuro hay que hacer unas con las mismas características pero que funcionen de manera diferente; y se ha creado una interfaz MaquinaTuringHandler que obligue a estas clases abstractas a implementar las funciones setSiguienteMaquina() y call(), esta última, heredada de la clase interna de Java “Callable”.