Práctica 1

Cambio Dinámico

Nombre: Pablo Ramos Muras
DNI: 76584372F
Email: pramos280@estudiante.uned.es

Indice

1. Introducción y Cuestiones.

1.1 Introduccion

Para esta primera práctica de Programación Dinámica, he decidido centrarme en la reusabilidad del código, implementando un patrón de diseño MVC. En principio, esto me permite reutilizar gran parte del código para futuras prácticas, como por ejemplo, la vista, que se encarga de solicitar y mostrar información al usuario.

El algoritmo implementa una interfaz “Lógica”, pensando en la expansibilidad, ya que mi idea original en esta práctica era implementar también el Algoritmo Voraz, y usar un patrón “AbstractFactory” para crear las diferentes implementaciones y poder compararlas directamente en la ejecución. Por razones de tiempo esto no se encuentra terminado.

Estas apreciaciones pueden ser vistas en los comentarios, incluidos sobre el código, donde pongo procedencia de snippets concretos, dudas, o flujo general del programa.

1.2 Cuestiones.

1.2.1 Indica y razona sobre el coste temporal y espacial del algoritmo.

- Ya que en la práctica se pedía tanto la cantidad de monedas como el valor de cada una de ellas, he implementado también el Algoritmo Voraz que devuelve el tipo de las monedas a partir de la tabla generada previamente. Este algoritmo se explica en la pag. 157 del texto base.

Sabiendo esto, tenemos que el coste temporal es la suma del coste de los dos algoritmos, el principal y el voraz en función a C (Cantidad a cambiar) y N (Numero de monedas en el sistema).

El coste temporal será: $\Theta(CN) + \Theta(C) = \Theta(2CN)$

- El coste espacial del algoritmo depende de los datos que se guarden. Tenemos, como primer dato, la Tabla de la solución, con un tamaño de C*N*size_of(int). También tenemos el vector de monedas, N*size_int y por ultimo un integer de cantidad.

Así que inicialmente el algoritmo ocupa, para un tamaño de int de 32 bits.

$$(32 + C*N^2*32) \text{ bits}$$

Durante el tiempo de ejecución se añaden dos variables temporales más, i y j.

Al finalizar, volvemos al tamaño inicial.
En el caso de mi implementación, el tamaño en bits no es exacto, ya que implemento una clase para monedas, en vista de una mayor expansibilidad. También uso un ArrayList para las monedas del sistema monetario. Esto estaba pensado para implementar la solución de no usar un segundo algoritmo para la identificación de las monedas. Aunque al final lo descarte, ya que el coste espacial era mayor.

1.2.2 Explica qué otros esquemas pueden resolver el problema y razona sobre su idoneidad.

- Tal y como explica el texto base, este problema también puede ser solucionado a través de un algoritmo voraz que vaya probando con la moneda más grande. Esto, a pesar de ser más eficiente, puede dar soluciones erroneas, como en el caso de:

  3
  1 6 10
  24

  Donde la solución de un algoritmo voraz serían 6 monedas (10 10 1 1 1 1) mientras que el esquema de programación dinámica devuelve 4 (6 6 6 6).

- A mayores, como comenté anteriormente, usamos un algoritmo voraz para identificar las monedas, existiendo la opción de memorizarlo en la tabla.

2. Ejemplos de ejecución.

Junto a la práctica, envió 3 archivos con diferentes pruebas que he ejecutado para probar la práctica. Los archivos incluyen todo el trace, que se puede habilitar con -t.

Ejemplo 1.

Entrada	Salida
3 1 6 10 12	2 6 6

Ejemplo 2.

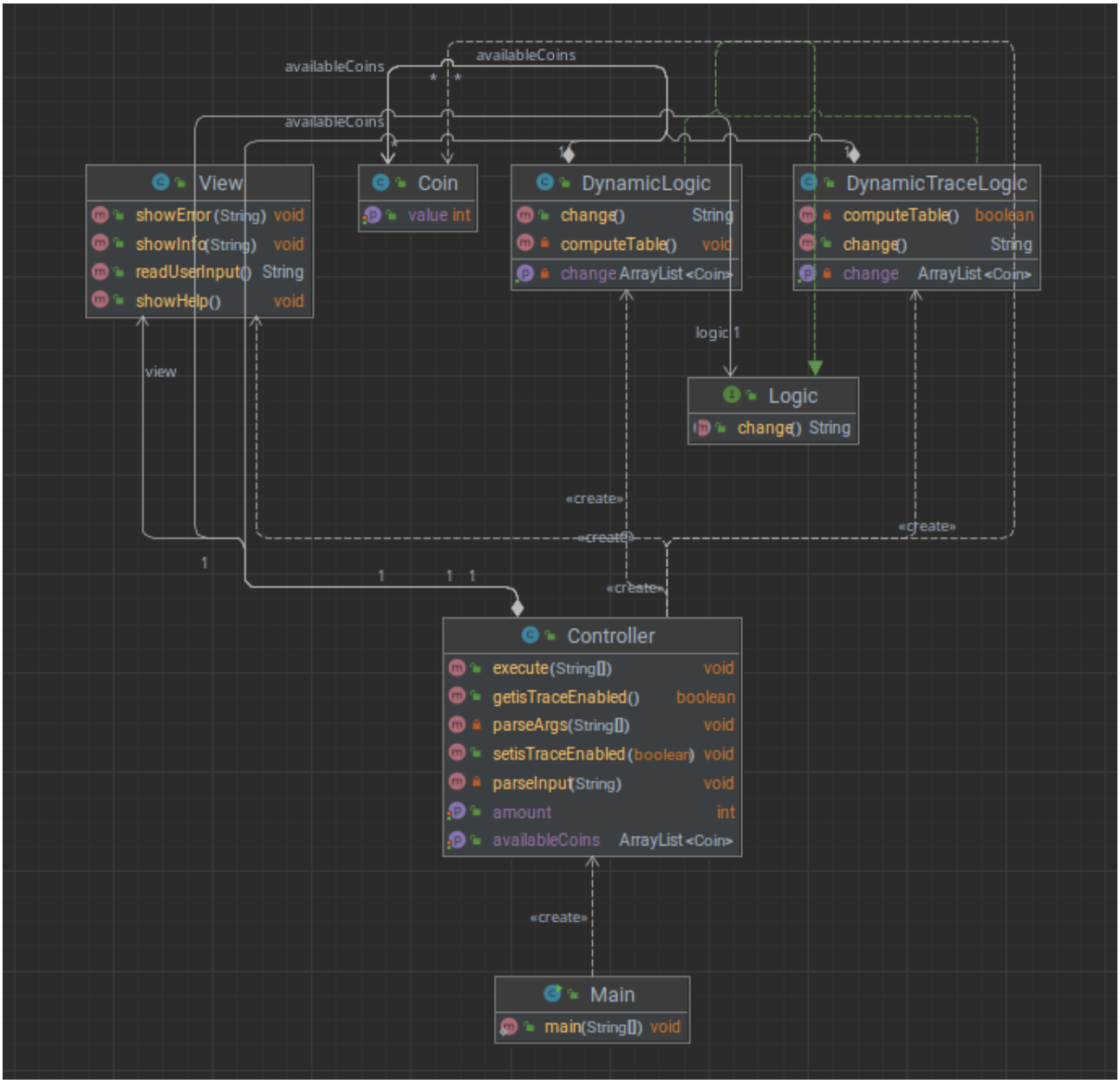
Entrada	Salida
3 1 6 10 24	4 6 6 6 6

Ejemplo 3.

Entrada	Salida
8 1 2 5 10 20 50 100 200 24	3 20 2 2

3. Listado de Código.

```
src/main/java/
└── org
    └── murapa
        └── cambio
            ├── controller
            │   └── Controller.java
            ├── Main.java
            ├── model
            │   ├── Coin.java
            │   ├── DynamicLogic.java
            │   ├── DynamicTraceLogic.java
            │   └── Logic.java
            └── view
                └── View.java
```



Este es el árbol de directorio de mi práctica y su diagrama UML. Como comentaba anteriormente, se separa en 3 paquetes:

- Controller: Maneja la view y el model para resolver el problema.-
- Model: Describe los datos que vamos a manejar, en este caso la Moneda y la Logic (Con sus 2 implementaciones diferentes)
- View: Maneja la entrada y salida de información con el usuario.El motivo por el que he elegido esta estructura, es tener una base para futuras prácticas no tener que empezar de 0. Pudiendo reutilizar la gestión de argumentos y procesado de entrada / salida.

Todo este código esta disponible en un repositorio de GitHub.

REPOSITORIO