Programación Orientada a Objetos

Trabajo Práctico Especial

CandyType

Autores:

Cristian Ontivero - 51102

Daniel Lobo - 51171

Agustín Prado - 51239

Resumen

La aplicación *CandyType* es una versión académica y muy reducida del juego *Candy Crush*. El juego consiste en una grilla orientada verticalmente, donde caen y se apilan caramelos de diferentes colores.

8 de junio de 2013

Índice

1. Introducción

La mecánica del juego consiste en elegir un par de caramelos adyacentes para intercambiar sus posiciones. Este intercambio es válido solo si al realizarlo se forma alguna figura válida con caramelos de un mismo color.

Las figuras son líneas horizontales o verticales de 3, 4 o 5 caramelos, o bien una T (de tres caramelos en línea y dos perpendiculares en el medio) en cualquier orientación, o bien una L en cualquier orientación. Si el intercambio es válido, porque debido al mismo se forman una o dos figuras, los caramelos que las componen explotan y desaparecen. Y en su lugar caen los caramelos que estaban por encima de estos (esto es un proceso recursivo).

Ciertas figuras al explotar dejan en su lugar caramelos especiales, que cuando explotan generan explosiones en cascada de distinto tipo. Cada caramelo que explota le otorga puntaje al usuario.

2. Enumeración y breve descripción de las funcionalidades agregadas

2.1. JellyCell

La *JellyCell* es una celda la cual tiene un borde gris y sobre la cual hay que efectuar algún tipo de movimiento sobre ella para que se elimine el borde gris. La existencia de este tipo de celda aporta una condición de ganado o perdido adicional al juego. En el juego original hay una versión similar a la implementada.

2.2. Food (Cherry y Hazelnut)

Si bien la aplicación que creamos tiene un *Cherry* y una *Hazelnut*, está implementado de manera tal que se puedan agregar más tipos de comida (*Food*). La función que cumple el *Cherry* y la *Hazelnut* es agregar una condición de ganado adicional, otorgándonos la posibilidad de crear más niveles. Además, queda estéticamente bien la combinación de caramelos con frutas.

2.3. Blank

Este elemento es como si fuera un tipo de celda que no puede ser ocupada y que hace saltear elementos siempre en dirección vertical. Genera una dinámica un poco diferente a la de otros niveles que no poseen este elemento y le cambia un poco la estética al juego.

2.4. Niveles

Decidimos agregar tres niveles además del básico provisto por la cátedra con el fin de poder mostrar claramente las funcionalidades agregadas al código original y probar, al mismo tiempo, la integración de las distintas modificaciones que fuimos agregando al desarrollar el trabajo. Además, poder tener más niveles hace que no sea tan monótono el estilo del juego ni la apariencia. Creemos que en futuras versiones se pueden hacer muchas mejoras en el plano estético y en el plano funcional.

3. Explicación de todas las modificaciones realizadas al proyecto original

En principio, diagramamos el *Grid* de otra manera para que cada nivel pueda modificarlo a su manera y así poder agregar distintos elementos sobre él.

Fueron agregadas funcionalidades al generador así como también la manera de generar distintos tipos de *GameState* para permitir al desarrollo ser más genérico y así, en futuros cambios del proyecto, poder cambiar más fácilmente el código de ganar y perder. Inicialmente, durante el desarrollo, la clase *GameState* era abstracta y el diseño estaba pensado como para que los niveles posean una clase inner que extendiera de *GameState* sobreescribiendo sus metodos abstractos con las condiciones de ganado. Luego de un tiempo, nos dimos cuenta que esto generaba mucha repetición innecesario de código, por lo tanto lo cambiamos. La clase *GameState* pasó a ser instanciable (dejó de ser abstracta) y agregamos una variable de instancia del tipo *Condition*. *Condition* es una interface con la que modelamos las condiciones para terminar el juego (*gameOver()*) y para ganar el juego (*playerWon()*). Gracias a esto, con definirlas una vez y que implementen esa interfaz es posible lograr que cada nivel instancie las que necesite. A su vez, existe la clase *MixedCondition* que también implementa *Condition* y guarda en una lista condiciones, de esa forma es posible componer varias condiciones en un mismo nivel.

El único inconveniente que este modelo presentó es que *GameState* contiene la información de los movimientos y puntaje del juego y la condición básica de ganar o perder en base a movimientos y puntaje (representada por *BasicCondition*) necesita acceso a eso para poder trabajar. Por lo tanto, la única opción es, primero instanciar un *MixedCondition*, pasárselo como parámetro al constructor de *GameState* para instanciar un *GameState* y luego agregar la condición de *BasicCondition* en *MixedCondition* pasandole el *GameState* instanciado antes por el constructor a *BasicCondition*. Es por esto que, aun si el mapa tiene una sola condición (como en el caso del nivel 1), si esta es *BasicCondition* es necesario usar *MixedCondition*, de lo contrario habría que agregar setters o getters que no deberían estar.

A algunos de elementos contenidos dentro del tablero se les fue adicionada la propiedad de ser explotables, como por ejemplo, los caramelos. Por ejemplo, se definieron diferentes propiedades como *isExplodable()* en los elementos para que sepan si deben desaparecer o no con los wrapped y stripped candies y *isSkippable()* en Cell para permitir saltearlas en los fallUpperContent (en el caso de Blank). De todas maneras, existen elementos, tales como la comida, que no poseen dichas propiedades.

Los mapas que utilizan la *FoodCondition* deben constar con una variable de tipo *Map¡FoodType, FoodTypeState¿*, en donde las keys son los nombres de las diferentes comidas definidas por el enum *FoodType*, y los values son una clase en la cual se almacena la información relevante a cada cómida, es decir, el total de la misma, cuantas queda por tirar al grid, y cuantas llegaron a destino.

El backend y el frontend estan en muchas partes interlazados siendo una de las cosas mas obvias tener el DELAY_MS en *Cell* (clase del back) cuando es algo exclusivamente del frontend. Hay otros métodos también en los que ocurren hechos similares. En el momento en que agregame el append de la imagén png de la *JellyCell* dejamos un instanceof. Más allá de que generalmente se deben evitar, la forma correcta sería desligar el frontend del backend, sacando los métodos que son claramente del frontend de las clases del backend, y ubicándolos en clases espejos por asi decirlo. Por ejemplo, podría ser clases del frontend que representen a sus análogos del backend como un*FrontCell* que contenga un *Cell* del backend y los métodos del fronted que maneja actualmente *Cell*. Se podría además, agregar un método como *isAppendable()* para saber si debe hacer un append o no de una imágen y de esta manera sería posible deshacerse del problema de preguntar por clases específicas y cada objeto sabe contestar de acuerdo a quién es. Es importante destacar que un cambio de esta magnitud es bastante mayor a lo que apuntaba en principio el trabajo práctico inicilamente, por lo que se tomó la decisión de mantener el instanceof.

4. Problemas encontrados durante el desarrollo

El hecho de utilizar un código aportado por alguien que no era ninguno de nosotros tres, complicó inicialmente el desarrollo ya que tuvimos que detenernos a analizar todo el código y entender realmente cómo funcionaban cada una de las partes en lugar de pensar qué queríamos hacer e implementarlo. Además, el código estaba sin comentarios por lo que el esfuerzo era línea por línea en algunos casos.

El funcionamiento del grid tuvo que ser cambiado para adaptarse a las necesidades que fuimos teniendo con respecto a las celdas y a los distintos tipos de elementos.

No hubo grandes complicaciones en términos generales, pero la mayoría de los obstáculos al momento de desarrollar pasaban por realmente entender el código y no sobreescribir o inventar métodos y funcionalidades que ya existían en el código original. Intentamos relacionar y utilizar el código y diseño existentes lo más posible. Gran parte del desafío radicaba en este punto, el de pensar el diseño de la aplicación.