Arca, Gonzalo - 60303
Revich, Igal Leonel - 60390
Rodriguez, Joaquin Manuel - 60258

Trabajo Práctico Especial
Programación Orientada a Objetos
Diciembre 2019
INFORME

Funcionalidades Agregadas

- **Level**

  Level es una clase abstracta que extiende de *Grid*, la cual modela el comportamiento en común de todos los niveles del juego. Dicha clase se encarga de llenar la grilla de caramelos, dado que dicho proceso es igual para todos los niveles. Posee un método que retorna una celda generadora de caramelos, la cual varía dependiendo del nivel. De esta clase extienden *Level1* y *TimeLevel.*

  ➢ **Level1**

  Level1 define las constantes con la cantidad máxima de movimientos y el score necesario para terminar el juego, así como los *getters* de los mismos. Incluye dentro una inner class *Level1State* que hereda el comportamiento de GameState y que manipula sus valores para determinar el estado de la partida.

  ➢ **TimeLevel**

  TimeLevel es una clase abstracta, dentro de la cual se define una variable de instancia *quota,* en donde se almacena el máximo establecido de caramelos especiales, y posee distintos métodos que le permiten manejar la cantidad de dichos caramelos especiales. Esta clase a su vez contiene una inner class *SpecialLevelGameState* que extiende de *GameState*, la cual almacena datos del juego como la cantidad de especiales a eliminar, eliminados y aparecidos. *TimeLevel* luego adquiere el rol de comunicar el estado dentro del GameState al resto del código.

  ❑ **Level2** ***(Time Bomb - Caramelos con contador)*****:**

  Level2 es un nivel del juego que se agregó al código como nueva funcionalidad, el cual consiste en la adición de un nuevo tipo de caramelos con movimientos en ellos, los cuales van

decrementando a medida que el jugador va haciendo movimientos válidos. Si alguno de ellos llega a 0, el jugador pierde. Para evitar que ocurra esto, se deben eliminar dichos caramelos antes de que sus contadores lleguen a 0. Si se logran eliminar todos los caramelos "*Time Bomb*", cuya cantidad es fija y visualizable desde la interfaz del usuario, el jugador gana la partida. Incluye dentro la clase *Level2State* que extiende de *SpecialLevelGameState*, ésta se encargará del tratamiento de los caramelos especiales y de las variables vinculadas al estado del juego, las cuales se almacenan en enteros y colecciones como mapas.

- **Level3 *(Límite de tiempo)*:**

  Level3 consiste en eliminar caramelos especiales antes de que transcurra un tiempo determinado. Dichos caramelos especiales poseen la funcionalidad de otorgar tiempo extra al usuario si se eliminan. Se establece un máximo de caramelos especiales y el usuario contará con un tiempo de juego inicial fijado previamente. Define la clase *Level3State* que extiende de *SpecialLevelGameState*, en donde se especifica cómo tratar los caramelos del nivel en caso de que se deban manipular por otras secciones del código durante el transcurso de la partida. Asimismo, maneja el estado interno de la misma.

- **L1ScorePanel**

  L1ScorePanel extiende de ScorePanel y agrega un *Label* para ir mostrando los movimientos restantes que le quedan al usuario, que se calculan mediante el método privado *getRemainingMoves*, y se actualiza sobreescribiendo el método *updateData*, llamando al método *updateRemainingMoves* luego de haber realizado una llamada a *super*. El puntaje en este nivel se muestra junto al puntaje esperado para ganar.

- **SpecialScorePanel**

  SpecialScorePanel es una clase abstracta creada con el propósito de modelar el comportamiento compartido entre los *ScorePanel* correspondientes a los niveles 2 y 3. Esta agrega un *Label* para mostrarle al usuario cuántos caramelos especiales le quedan por eliminar, cuya cantidad se irá actualizando durante el juego. De esta clase extienden *L2ScorePanel* y *L3ScorePanel*.

  - **L2ScorePanel**

L2ScorePanel incorpora otro *Label* para mostrar la mínima cantidad de movimientos que el jugador puede hacer antes de perder el nivel, que se irá actualizando a lo largo del juego. Si no hay caramelos especiales en pantalla, el texto del *Label* de movimientos restantes pasa al *String* vacío.

➢ **L3ScorePanel**

*L3ScorePane*l adiciona un *Label* para mostrar el tiempo de juego restante. Dicho tiempo se actualiza a cada segundo, y se controla si el usuario ganó o perdió. Dicho tiempo se muestra en el formato *mm:ss*.

- **TimeCandy**

TimeCandy (extiende de *Candy*) es una clase abstracta, con el objetivo de reunir el comportamiento del *TimeBombCandy* y *TimeBonusCandy*. Esta recibe en su constructor, además de su color, un entero que indicará el valor de su variable *timer*, a la cual se le definió un *getter* para permitir su acceso desde las clases de *Level*. Se sobredefinió el método *isSpecial* (ver en *Cambios en la implementación de la Cátedra, Back-end* para que retorne `true`. También se sumó un método abstracto *getLabel* debido a que las clases hijas tendrán un texto que deberá ser puesto encima de la imagen del caramelo.

➢ **TimeBombCandy**

TimeBombCandy es utilizada en *Level2* y la misma cuenta con una constante que indica el valor inicial del contador de movimientos del caramelo, y con una variable de clase que se utiliza en el método *setId,* llamado en el contructor para definir el *id* al nuevo caramelo de esta clase e incrementarlo para el próximo, para poder guardarlos en el *TreeMap* de acuerdo a su momento de inserción. De esta manera, los primeros serán los que menos movimientos restantes tendrán, considerando que en el *TreeMap* sólo aparecen los *TimeBombCandy* activos. El método *getLabel* retornara el valor de *timer* de la instancia como *String*. Cuenta también con los métodos *decTimer*, que decrementa el valor del contador del caramelo (es llamado por *Level2* cada vez que se realice un movimiento válido).

➢ **TimeBonusCandy**

TimeBonusCandy cuenta con una constante que indica la cantidad de tiempo a sumar al contador y la sobreescritura del método *getLabel,* para que retorne un *String* con el formato ”+ *number*”.

- **TimeCandyGeneratorCell**

  TimeCandyGeneratorCell extiende de CandyGeneratorCell, y se encarga de generar caramelos especiales, tanto para Level 2 como para Level 3, a partir de una constante entera *N*, la cual determina la frecuencia con la cual se generarán caramelos especiales. La relación está dada de forma tal que, mientras mayor sea la constante *N,* menor será la frecuencia de generación de caramelos especiales. De esta clase extienden *L2CandyGeneratorCell* y *L3CandyGeneratorCell.*

  - ➢ **L2CandyGeneratorCell**

    L2CandyGeneratorCell se encarga de generar los *TimeBombCandy*.

  - ➢ **L3CandyGeneratorCell**

    L3CandyGeneratorCell se encarga de generar los *TimeBonusCandy*.

- **TimeCell**

  TimeCell extiende de *Cell* y redefine el método *clearContent*, de tal forma que si el caramelo a sacar es especial, se ejecute también el *removeSpecial* del mismo.

- **TimerManager**

  TimerManager se utiliza para manejar el comportamiento del *timer* involucrado en *Level3*. Es necesaria, pues hay veces que se debe finalizar el *timer* desde fuera de donde se instanció el mismo, para evitar que se siga corriendo estando fuera de *Level3*.

- **Seleccionador de nivel agregado a la barra de menú**

  Dicha funcionalidad consiste en un nuevo *Menu* agregado al *MenuBar* ya existente. Este posee 3 *MenuItems* (*Level1, Level2, Level3),* y al apretar alguno de estos, se inicia un juego nuevo con dicho nivel.

- **Pantalla de inicio con imagen de Candy Crush.**

  Esta implementación consiste en que al arrancar la aplicación no se inicia el *Level1* por default, sino que se muestra una imagen del Candy Crush como inicio, y luego el usuario, mediante el seleccionador de niveles, puede elegir en qué nivel comenzar a jugar.

Cambios en la implementación de la cátedra

Front-end:

Se modificó ***GameApp*** de tal forma que no inicialice el nivel 1 del código fuente, si no que muestre una escena de "inicio", donde el jugador tendrá que seleccionar en la barra de menú cuál será el nivel a ejecutar. Se agregó un método (*startLevel*) en la clase *GameApp* para generalizar el inicio del nivel que corresponda, el cual se llama desde *AppMenu* para realizar el cambio a un nuevo nivel. Luego, en dicha clase se agrego el método *sizeToScene,* para que el tamaño de la ventana se adapte a la escena, y evite la formación de márgenes.

En la clase ***AppMenu*** se modificó la funcionalidad del botón *"Salir"* proveniente del menú *"Archivo"* y del botón de cerrar de la ventana, de forma que en lugar de realizar `Platform.exit()`, realice `System.exit(0)`. Esto asegura de que en caso de cerrar el juego una vez empezada una partida del nivel 3 (sin haberla ganado o perdido) se cancele el Timer creado desde el mismo nivel, evitando que el proceso de Java se quede colgado. También se agregó un botón que permite la selección de los niveles disponibles.

Dentro de ***CandyFrame*** se cambió que al *scorePanel* a utilizar se le setea el game actual, para que pueda acceder a los datos del mismo (así obtiene datos propios de cada nivel). Luego en la implementación de *gridUpdated* dentro de la misma clase se modificó la última línea, donde se utiliza un método nuevo proveniente de ***BoardPanel*** (sobrecarga de *setImage*) que recibe, además de lo que ya recibe *setImage*, un *element*. Esto resulta útil para poder saber si al caramelo que se le asigna una imagen le corresponderá o no la etiqueta del *"TimeBomb"* o del *"TimeBonus"*. Volviendo a ***CandyFrame***, en el *addEventHandler* del click del mouse se agregó un `if` externo al código fuente, que entra si el juego todavía no finalizó. De esta forma, una vez finalizado el juego, el usuario no puede realizar más movimientos. También se modificó el llamado que se realiza al *updateScore* del *scorePanel* por el *updateData*, pues en ciertos niveles no era suficiente con solo actualizar el valor del *score*, ya que tienen otras variables requieren de actualización. Adicionalmente, se modificó la disposición del mensaje de "juego terminado", donde ahora tiene el orden inverso; primero el estado de la partida, y luego el puntaje, pues resulta más cómodo para *L1ScorePanel*, ya que ahora se muestra el score mínimo luego del actual.

En cuanto a la clase ***ScorePanel***, se decidió hacerla sea abstracta, con el fin de crear clases hijas que compartan comportamiento (mostrar el estado del juego), pero difieren en detalles por pertenecer a niveles distintos. Esta clase cuenta con un *Label*, donde se muestra el *score* y una instancia de *CandyGame*, cuya función es relacionar el *ScorePanel* con el *back-end* del juego actual. Dicha instancia se setea mediante el método *setGame*, que guarda la referencia de la instancia de *CandyGame* recibida en *CandyFrame*. Por otro lado, se creó un método en dicha clase llamado *updateData*, el cual se llama desde *CandyFrame* luego de que el usuario realice un movimiento válido. Este se encarga de que cada *ScorePanel* que extienda de dicha clase abstracta actualice sus datos, y de esta forma cada uno de ellos sabrá qué datos debe actualizar sin la necesidad de preguntar por el nivel actual (polimorfismo). De esta forma, de la clase *ScorePanel* extienden *L1ScorePanel* y *SpecialScorePanel*. Cabe

mencionar que cualquier instancia de *Timer* que haya en la partida será cancelada aquí en el momento al instanciar la clase, puesto que es un fragmento de código que se corre cada vez que se cambia de nivel, y que además los *Timers* luego se instancian en la clase hija *L3ScorePanel*.

Back-end:

Se agregó en ***Element*** el método booleano *isSpecial*, que tiene como valor predeterminado `false`, y será sobreescrito para determinar lo contrario en los caramelos pertenecientes a los niveles 2 y 3.

Se modificó en la clase ***Grid*** el *initialize*, verificando que la constante *SIZE* tenga un valor válido para poder jugarse (mayor o igual a 3, pues la figura más chica requiere 3 caramelos). Luego, en lugar de asignar a cada posición de la matriz una nueva *Cell*, se llama a un nuevo método *assignCell*, que recibirá cada posición para asignarle una nueva instancia de *Cell*. Se decidió hacer esto para que luego *assignCell* pueda ser sobreescrito por las clases hijas y así puedan asignarles otro tipo de celda. Es necesario remarcar que este último llama al método *setGridCell* (nuevo) con la posición y una nueva celda, el cual hubo que crear para poder setear distintos tipos de celdas en la matriz desde clases hijas.

Al método *tryMove* se lo cambió de tal forma que si el movimiento es válido, luego de realizar el *removeElements* se llame a un nuevo método en el cual se ejecutarán acciones. Este será protected para que las clases hijas puedan redefinirlo y agregarle comandos en caso de que quieran realizar algo más si el movimiento es válido.

En cuanto al método ***fillCells,*** éste se movió para la clase *Level*, dado que este proceso se lleva a cabo de la misma manera para todos los niveles, con la diferencia del tipo de *CandyGeneratorCell* que se utilizará. Por éste último detalle, ahora la variable de instancia *candyGenCell* de *Level* se inicializa desde el método *getCandyGenerator*, que, dependiendo del nivel, retornara un generador distinto a través de sobreescritura. Empleando estos cambios, cada clase hija puede reutilizar *fillCells* a pesar de sus distintas características.

Dentro de ***Level1State***, se cambió el método *playerWon* para que el usuario gane si su puntaje es mayor o igual al puntaje requerido para ganar la partida (antes era mayor estricto, lo cual no es demasiado intuitivo).

BombWrappedMove ahora extiende de la clase *BombMove*, pues primero realiza lo mismo que su *super,* y luego elimina los alrededores de los caramelos (estos antes se dividía en dos ciclos y ahora se integró en el mismo).

Se modificó ***TwoWrappedMove*** para que sea más claro, pues en los dos casos de la condición del código base se realizaba la misma acción.

Se agregó en la clase ***Cell*** el método *getGrid*, pues se necesitaba el acceso a la misma desde la clase hija *TimeCell* para poder utilizar el *removeSpecial* de *TimeLevel,* (que extiende de Grid). Dicho acceso era también necesario dentro de las clases de *TimeCandyGeneratorCell* y *L2CandyGeneratorCell*.

Problemas encontrados durante el desarrollo

Durante el desarrollo del Trabajo Práctico, se encontraron dificultades en los siguientes puntos:

1. **Cancelación del *Timer* de *L3ScorePanel*:**

   Dado que al arrancar el *Level3* se seteaba un *timer* desde *L3ScorePanel*, al terminar la partida, cambiar de nivel, o cerrar el juego en su totalidad, éste debía finalizar su funcionamiento, mediante las funciones *cancel* y *purge*. En caso de terminarse el juego, esto se realizaba dentro de la misma instancia de *timer*, ya que cada segundo controlaba si el usuario ya perdió o ganó, y de suceder alguna de esas cosas, el proceso de *timer* se finalizaba. En caso de que se cerrara el juego, fue necesario cambiar el `Platform.exit` por `System.exit(0)`. La diferencia entre ambos comandos es que el `System.exit(0)` cierra todos los procesos corriendose en el momento en que se llama, incluido el *timer* instanciado en *L3ScorePanel*. De no ser por esto, la aplicación de Java se cuelga al intentar cerrarla mientras el *timer* se ejecuta.

   Por último, el caso en donde se encontró mayor dificultad fue el de cambiar de nivel antes de llegar a ganar o perder la partida ya empezada de *Level3*. Para dicho escenario, se optó por crear la clase *TimerManager* mencionada anteriormente, cuya función es guardar la referencia del *Timer* creado en *L3ScorePanel*, y, cuando se cree una nueva partida de cualquier nivel, en el constructor de la clase abstracta *ScorePane*l se cierre dicho *Timer*. Resulta coherente realizar esta última tarea allí, dado que el *ScorePanel* siempre se instancia antes de crear el *game* en sí, y al implementarlo desde la clase abstracta, todos los niveles terminan ejecutando la tarea, y se evita la repetición innecesaria de código.

2. **Método de cambio de Nivel:**

   Se encontraron dificultades en hallar un método para que, al apretar algún nivel en el menú, se inicie un nuevo juego con dicho nivel, dado que cada nivel debe tener un *ScorePanel* diferente, y de alguna manera se debe saber qué nivel se iba a inicializar. Se decidió crear el método *startLevel* en *GameApp,* el cual recibe la clase del nivel que se quiere iniciar, una nueva instancia del *ScorePanel* correspondiente a dicho nivel, y la instancia del *AppMenu* desde donde se llama al método. De esta manera, al *CandyFrame* se le pasa una clase de nivel y un ScorePanel, donde cada uno sabe que hacer dependiendo del nivel y no hay necesidad de preguntar por el nivel que se quiere arrancar, y por ende se evita repetición de código.

3. **Colección para guardar los *TimeBombCandy* activos:**

   Teniendo en cuenta que dichos caramelos deben decrementar su contador en el momento que se realiza un movimiento válido, el grupo se vio obligado a almacenar

las instancias de los *TimeBombCandy* activos, para poder así decrementar todos sus contadores en simultáneo en el momento que se deba. Asimismo, era necesario además poder acceder el mínimo contador de dichas instancias para el *ScorePanel* del nivel.

Para lograr esto, se optó en primer instancia por guardarlos en un *ArrayList*. Esto generó un inconveniente, ya que, llegado el momento de tener que remover los caramelos de la lista en el instante que se eliminan de la grilla se llama el método `remove(Object obj)`, el cual utiliza el equals de la clase del objeto para poder encontrarlo, y luego elimina la primer instancia encontrada. El problema era que, el equals de *TimeBombCandy*, al ser el mismo que el de *Candy* por herencia, eliminaba el primero de la lista que tuviera el mismo color que el que se deseaba realmente eliminar. Sobreescribir el equals tampoco funcionó, ya que en a la hora de detectar figuras, esta tarea se ejecutaba incorrectamente considerando que el nuevo equals de *TimeBombCandy* causaba que los caramelos con mismo color pero de distinto tipo (uno *TimeBomb* y otro cualquiera) no formaran figuras por ser "distintos".

Considerando todo lo ocurrido, se concluyó que la mejor opción es utilizar como colección un *TreeMap<Integer, TimeBombCandy>* , donde la clave es el *id* del *TimeBombCandy*, del cual se deduce el orden del tiempo de creación de dicho caramelo. Por ende, al estar ordenados mediante este id, la primer entrada del mapa es el caramelo especial activo más antiguo, y por ende este tiene la mínima cantidad de movimientos que le quedan por hacer al usuario antes de perder.