Trabajo Práctico Final Programación Orientada a Objetos

Integrantes

- Piñeiro, Eugenia - Legajo: 59449
- Torrusio, Lucía - Legajo: 59489
- Dell'Isola, Lucas - Legajo: 58025

Entrega: Final 2019, 1^{er} cuatrimestre

Informe - Candy Crush

Se eligió realizar las implementaciones de Golden Board y Cherry & Hazelnut

Funcionalidades Agregadas

En primer lugar, se creó la clase abstracta `Level` que realiza los métodos comunes a todos los niveles: llenar el tablero con los caramelos, revisar si un movimiento es válido y a su vez le agregamos un método para preguntar si el nivel tiene celdas doradas. Este último método fue agregado, no solo para la implementación del `Level2`, sino también en caso de que se tenga un nivel con celdas doradas y otros elementos y una vez ganado el nivel se pueda quitar el efecto dorado de esas celdas en específico.

Luego, se agregaron las clases `Level 2` y `Level 3` para la implementación de Golden Board y la de Cherry & Hazelnut respectivamente.

GOLDEN BOARD

Para esta implementación es necesario tener muy en claro la distinción entre "marcar como dorado" y "pintar de dorado". La primera hace referencia a, en el backend, setear el estado de una GoldenCell (clase que luego será explicada). En cambio, la segunda pone el efecto dorado a una celda en el frontend.

Se agregó la clase `Level2` que extiende de `Level`. Su objetivo es marcar las celdas como doradas

GoldenCell.java

Esta clase permite marcar a las celdas como doradas luego de hacer un intercambio.

GoldenGameListener.java

Se decidió hacer esta clase a fin de mejorar la eficiencia del `Level2`. Su objetivo es que aquellas celdas que ya están pintadas de dorado no vuelvan a pintarse.

CHERRY & HAZELNUT

Se agregó la clase `Level 3` que extiende de `Level`

Cherry.java - Hazelnut.java

Dentro del package `element` agregamos las clases `Cherry` y `Hazelnut` que extienden de la clase abstracta `UndestroyableElement`. Estos elementos se marcan como indestructibles.

Otras funcionalidades

Se agregó el Package Alert cuyo objetivo es mostrar en pantalla si se ganó o perdió el nivel. La clase `GameInfoListener` es la encargada de mostrar los paneles de Movimientos restantes, Score yGoldenCells pintadas o Cantidad de frutas restantes dependiendo de cada nivel.

Estructura del Proyecto

Para la estructura del proyecto se continuó con el patrón MVC establecido en el proyecto original, sumando a ello un par de clases que heredan `GameListener` para poder comunicar al modelo y el usuario.

Entre estas clases creadas se encuentra `BasicGameListener` cuya tarea es dibujar en pantalla al tablero con todos los elementos e íconos. También se implementó la clase `GoldenGameListener` que se encarga de detectar en que niveles se utilizan `GoldenCell` para poder pintar de dorado cuando una de esas celdas fue marcada como dorada en el modelo. A la hora de crear la clase se debatieron dos opciones: hacer al listener para que solo sirva en niveles donde todas las celdas son `GoldenCell`, que sería más simple, o intentar hacer una clase más general, que pueda operar en niveles donde hay solo una región con `GoldenCells` mientras que en otras partes hay celdas normales. Por último, se decidió hacer un Listener exclusivo para la información que se le muestra al jugador, como lo es el puntaje, los movimientos y el objetivo. Este listener esta manejando por tres clases: `GoalPanel`, `MovementsPanel` y `ScorePanel`.

Gracias a la implementación MVC, si en algún momento se quiere agregar una nueva funcionalidad como puede ser compartir la pantalla con otra computadora o simplemente agregar un nuevo efecto, solo es necesario crear el listener que implemente esa funcionalidad, sin modificar el código que ya esta escrito.

Por otro lado, dentro del modelo planteamos la siguiente estructura para los niveles. La clase `Grid` es la la grilla del nivel, indica las proporciones y la forma del nivel, además de darle funcionalidad, contar los movimientos, etc.

Luego, se cuenta con la clase abstracta `Level` que es la encargada de posicionar las celdas generadoras de caramelos y las paredes, por lo que aunque todos compartan la grilla de 9×9, `Level` indica donde están las paredes (`Wall`) y las celdas generadoras (`CandyGeneratorCell`) para hacer mas único al nivel manteniendo la funcionalidad que establece `Grid`.

Por ultimo tenemos a las clases `Level1`, `Level2` y `Level3` que son los niveles que el jugador va a jugar. Estas clases heredan de `Level` y también agregan funcionalidad como un objetivo a lograr. Si se desea mantener la forma de los niveles actuales, solo seria cuestión de hacer una clase que herede `Level` y luego agregarle una funcionalidad nueva.

Modificaciones al proyecto inicial

CandyGame.java

El cambio más grande fue en la inicialización de la clase. Ahora en vez de tomar como variable a un nivel, recibe una lista con los niveles que tiene que ejecutar, luego dentro de la clase hay un control interno sobre que nivel tiene que ejecutar:

```
private int levelIndex = 0;
private List<Class<?>> levels = new ArrayList<>();

public CandyGame(List<Class<?>> levels) {
    if(!levels.isEmpty()) {
        this.levels = levels;
        this.levelClass = levels.get(0);
    }
    else
        throw new IllegalArgumentException("At least one level must be passed");
}
```

Para el control de los niveles, y asegurarse que solo esta clase se encargue de manejarlos, se crearon los métodos `nextLevel()` y `hasNextLevel()` que se encargan de poner el próximo nivel:

```
public boolean hasNextLevel(){
    return levels.size() > levelIndex+1;
}

public void nextLevel() {
    if(hasNextLevel()) {
        grid.wasUpdated();
        this.levelClass = levels.get(++levelIndex);
        try {
            grid = (Grid) levelClass.newInstance();
        } catch (IllegalAccessException | InstantiationException e) {
            System.out.println("ERROR AL INICIAR");
        }
        state = grid.createState();
        grid.initialize();
        addGameListener(this);
    }
}
```

Por último, al agregar mayor funcionalidad a `GameState` se tuvieron que agregar algunos getters. Además se creó un método para poder actualizar los listeners:

```
public int getGoal(){return state.getGoal();}

public int getCurrentGoal(){return state.getCurrentGoal();}

public String getGoalDescription() {return state.getGoalDescription();}

public void updateListeners() {grid.wasUpdated();}

public int maxMovements() {return state.getMaxMoves();}

public int currMovements() {return state.getMoves();}
```

GameState.java

La clase `GameState` fue modificada para poder acomodar a los nuevos objetivos de los niveles. Como todos los niveles pueden terminar de dos formas distintas, si el jugador gana o si el jugador hace mas movimientos del máximo, por lo que ahora `gameOver()` se define en base a funciones que tienen que definir los estados de los niveles que implementen la clase. De forma similar, como hay varias formas de ganar según el nivel, se implementaron funciones para poder ver que tan lejos esta de ganar el jugador y cual es el objetivo.

```
public abstract int getMaxMoves();

public abstract int getGoal();

public abstract int getCurrentGoal();

public abstract String getGoalDescription();

public boolean gameOver(){
    return playerWon() || getMoves() > getMaxMoves()-1;
}
```

Grid.java

La clase abstracta `Grid` es la base del juego, por lo que se encarga solo de las cosas mas fundamentales como crear la grilla, cargarla de elementos y hacer que estos caigan cuando se hagan combinaciones validas. En esta clase se separó la acción de crear la grilla con la de llenarla de elementos y hacer que caigan, y se ubicaron en el método que inicializa al objeto. Esto permite que si un nivel requiere otro tipo de celda, pueden sobreescribir solo la función `fillCells()` para que se creen celdas que hereden de `Cell`.

```
public void initialize() {
    moveMaker = new MoveMaker(this);
    figureDetector = new FigureDetector(this);
    createGrid();
    fillCells();
    fallElements();
}

protected void createGrid(){
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            g[i][j] = new Cell(this);
            gMap.put(g[i][j], new Point(i,j));
        }
    }
}
```

Cell.java

El único cambio en esta clase es que solo se pueden combinar elementos si el elemento no es destruible.

```
public void clearContent() {
    if (content.isMovable() && content.isDestoyable()) {
        Direction[] explosionCascade = content.explode();
        grid.cellExplosion(content);
        this.content = new Nothing();
        if (explosionCascade != null) {
            expandExplosion(explosionCascade);
        }
        this.content = new Nothing();
    }
}
```

Element.java

Como se implementó la funcionalidad de las Cherries, se tuvo que agregar un método `isDestroyable()` que indique si el elemento puede ser destruido. Si un elemento no puede ser destruido, entonces tiene que sobreescribir a este método para indicarlo. También, se sobreescribió el método `equals()` para verificar si dos elementos son iguales en base a la clave que se les asigna.

```
public boolean isDestoyable() {return true;}

@Override
public boolean equals(Object obj){
    if(this == obj)
        return  true;
    if(!(obj instanceof Element))
        return  false;
    Element el = (Element) obj;
    return this.getFullKey().equals(el.getFullKey());
}
```

Nothing.java

Se sobreescribió el método `isDestroyable()` para indicar que no es posible destruir al vacío.

```
@Override
public boolean isDestroyable() {return false;}
```

Wall.java

Se sobreescribió el método `isDestroyable()` para indicar que no es posible destruir a las paredes.

```
@Override
public boolean isDestroyable() {return false;}
```

Level1.java

Probablemente la clase la más modificada, debido a la estructura del programa, se consideró conveniente crear una clase intermedia `Level` que tenga toda la funcionalidad general de un nivel con una configuración de celdas paredes (`Wall`) y celdas generadoras de caramelos (`CandyGeneratorCell`). Y como este nivel es muy básico no requería cambiar nada mas, salvo

crear la clase que implemente `GameState` para el nivel.

```
public class Level1 extends Level {

    private static int REQUIRED_SCORE = 1000;
    private static int MAX_MOVES = 20;

    @Override
    protected GameState newState() {
        return new Level1State(REQUIRED_SCORE, MAX_MOVES);
    }

    private class Level1State extends GameState {
        private long requiredScore;
        private int maxMoves;

        public Level1State(long requiredScore, int maxMoves) {
            this.requiredScore = requiredScore;
            this.maxMoves = maxMoves;
        }

        @Override
        public int getMaxMoves() {return maxMoves;}

        @Override
        public int getGoal() {return REQUIRED_SCORE;}

        @Override
        public int getCurrentGoal() {return (int)getScore();}

        @Override
        public String getGoalDescription() {return "Score";}

        public boolean playerWon() {return getScore() > requiredScore;}
    }
}
```

BombMove.java

Aquí se realizó una pequeña modificación en la función `removeElements()` para que las explosiones de las bombas no destruyan a las cherries ni las hazelnuts.

MoveMaker.java

En esta clase agregamos las interacciones entre las cherries y los demás caramelos, bombas, etc, dentro de la función `initMap()`.

CandyFrame.java

La clase `CandyFrame` es la encargada de manejar la interacción con el jugador. Se modificó la clase para poder implementar de forma modular los listeners (clases que heredan a `GameListener`), para lograrlo se creó el método `createGameListeners()` que inicializa los listeners del frontend como lo son `BasicGameListener`, `GoldGameListener` y `GameInfoListener` y luego estos se cargan al Model (`CandyGame`) con el método

`EnableGameListeners()`. De esta forma, se logró modularizar el código que estaba dentro del eventHandler.

```
private List<GameListener> listeners = new ArrayList<>();

public CandyFrame(CandyGame game) {
    this.game = game;
    getChildren().add(new AppMenu());
    images = new ImageManager();
    boardPanel = new BoardPanel(game.getSize(), game.getSize(), CELL_SIZE);
    getChildren().add(boardPanel);

    game.initGame();
    createGameListener();
    enableGameListener();

    addEventHandler(MouseEvent.MOUSE_CLICKED, event -> {
        if (lastPoint == null) {
            lastPoint = translateCoords(event.getX(), event.getY());
            System.out.println("Get first = " +  lastPoint);
        } else {
            Point2D newPoint = translateCoords(event.getX(), event.getY());
            if (newPoint != null) {
                System.out.println("Get second = " +  newPoint);
                game().tryMove((int)lastPoint.getX(), (int)lastPoint.getY(),
(int)newPoint.getX(), (int)newPoint.getY());

                if (game().isFinished()) {
                    if(game().playerWon())
                        new WonLevelAlert(game());
                    else
                        new LostLevelAlert(game());
                    enableGameListener();
                }
                lastPoint = null;
            }
        }
    });
}

private void createGameListener(){
    listeners.add(new BasicGameListener());
    listeners.add(new GoldenGameListener(game(),boardPanel));
    listeners.add(new GameInfoListener(this,game()));
}

protected void enableGameListener(){
    for(GameListener listener : listeners)
        game().addGameListener(listener);
    game().updateListeners();
}
```

Tambien se convirtió al listener anónimo que estaba definido en una clase interna llamada `BasicGameListener` para que sea mas claro.

```
private class BasicGameListener implements GameListener{
```

```
    @Override
    public void gridUpdated() {
        Timeline timeLine = new Timeline();
        Duration frameGap = Duration.millis(100);
        Duration frameTime = Duration.ZERO;
        for (int i = game().getSize() - 1; i >= 0; i--) {
            for (int j = game().getSize() - 1; j >= 0; j--) {
                int finalI = i;
                int finalJ = j;
                Cell cell = CandyFrame.this.game.get(i, j);
                Element element = cell.getContent();
                Image image = images.getImage(element);

                timeLine.getKeyFrames().add(new KeyFrame(frameTime, e ->
boardPanel.setImage(finalI, finalJ, null)));
                timeLine.getKeyFrames().add(new KeyFrame(frameTime, e ->
boardPanel.setImage(finalI, finalJ, image)));
            }
            frameTime = frameTime.add(frameGap);
        }
        timeLine.play();
    }
    @Override
    public void cellExplosion(Element e) {

    }
}
```

BoardPanel.Java

Se modificó `BoardPanel` para que pueda dibujar de dorado a la celda y tambien quitarle el efecto

```
public void setGoldenEffect(int row, int column) {
    Light.Distant spotLight = new Light.Distant();
    spotLight.setColor(Color.YELLOW);
    spotLight.setElevation(100);
    Lighting lighting = new Lighting(spotLight);
    cells[row][column].setEffect(lighting);
}

public void stopGoldenEffect(int row, int column){
    cells[row][column].setEffect(null);
}
```

ScorePanel.java

El único cambio en la clase `ScorePanel` fue para que tome `long` en vez de `String` y para que el texto indique que es el puntaje.

Problemas encontrados durante el desarrollo

Separación entre Frontend y Backend:

Aunque esto ya estaba implementado en el proyecto original, siguió siendo una complicación saber dónde debía estar cada clase. Sobretodo la distinción entre "marcar una celda como dorada" y "pintar de dorado una celda".

Estructura de los niveles:

Al principio del trabajo se discutió cuál era la mejor forma de organizar los niveles. Se debatió entre dos opciones: que los mismos extiendan de `Grid` o que haya una clase abstracta `Level` y luego cada nivel extienda de la misma. Se resolvió implementando la segunda opción.

Modularización:

Simultáneamente al problema de la organización de los “Levels” , se presentó otro problema: la repetición de código. Esto fue solucionado al mismo tiempo que se solucionó el caso anterior.

Diseño del GoldenBoard (Level 2) :

En un principio se pensó en crear una clase `GoldenGrid` la cual estaría formada solo por celdas de tipo golden, pero como sería utilizada únicamente por el nivel 2 se decidió que no era una buena implementación, ya que se separaba al nivel 2 del resto de los niveles. Esto fue solucionado cuando se optó por implementar la clase `GoldenGameListener`. Además no se podía hacer una buena implementación para que los cells se pinten de forma correcta. Luego de esto se buscó cómo optimizar la forma de pintar las celdas doradas con el objetivo de no volver a pintar un celda que ya tenía el efecto dorado.

Utilización de JavaFX:

No se contaba con un gran conocimiento sobre JavaFX, por lo tanto se dificultó interpretar algunas partes del código original.

Funcionamiento de la implementación provista por la cátedra:

Por último, cabe destacar que no resultó fácil interpretar todas las clases que ya estaban implementadas por la cátedra, pero una vez leído el código con mucho detenimiento se logró comenzar a trabajar.