Tecnología de Programación (TdP)

Segundo Cuatrimestre 2024

Planilla de Corrección Proyecto 1
Comisión 14

Nota final: A +	**Aprobado**

Análisis preliminar del proyecto: *descarga y compilación del código fuente; uso de la aplicación en ejecución.*	
¿Todos los alumnos mostraron actividad en el uso del repositorio y las defensas?	**SI** - Se observan commits de consideración de todos los alumnos.
¿El código se compila?	**SI** - El código se compila correctamente, y ante la ejecución no se visualizan errores por consola.
¿El juego muestra tiempo, puntaje y vidas en todo momento? ¿El juego implementa dos modos de juego diferentes? ¿El juego opera el ranking de forma consistente?	**SI** - Se muestra puntaje, vidas y tiempo restante todo el tiempo. - El juego implementa dos modos de juego completamente diferentes. El modo de juego alternativo está ambientado en Doom. - El juego opera el ranking de forma consistente.
¿El juego se comporta como se espera (estado inicial, condición de finalización, transición de como mínimo 3 niveles)?	**SI** - El juego inicia correctamente, mostrando las imágenes de los elementos del juego. - Se cumplen las condiciones de finalización (victoria, derrota y cuando se acaba el tiempo). - El juego cuenta con transición de tres niveles con claro nivel de dificultad ascendente.
¿El juego se comporta como se espera ante la interacción con los power-ups (cambio de estado, cambio de sprites)?	**SI** - Al tomar el PowerUp de estrella, el personaje principal cambia de tamaño. - El resto de PowerUps funciona según lo esperado.
¿El juego se comporta como se espera ante la interacción con los enemigos (condición para muerte, estados intermedios de enemigos tales como tortuga)?	**SI** - Cuando el personaje principal aplasta a un Koopa, el saltito que hace luego del primer golpe no derrota al caparazón.
¿El juego se comporta como se espera respecto del lanzamiento de bolas de fuego?	**SI** - Muy bien logrado el lanzamiento de bolas de fuego como en el juego original.
Virtudes de la GUI y usabilidad	**SI** - Los personajes tienen animaciones.

	- El modo de juego alternativo está muy bien trabajado. No solo cuenta con imágenes alternativas, sino que también incluye sonidos diferentes para los personajes e interacciones. - Los enemigos aplastados tienen animación de muerte. - El movimiento del personaje principal y los enemigos es fluido
Debilidades de la GUI y usabilidad	**SI** -En ocasiones, si un bloque de pregunta tiene dos bloques sólidos adyacentes, se lo debe chocar más centrado para que interactúe. - Al finalizar la partida no se indica motivo de finalización (victoria, derrota, se acabó el tiempo). - En ocasiones se ven parpadear levemente a los Labels cuando se avanza rápido. - Para navegar por los botones del menú principal se deben utilizar las teclas “w”, “s” y “enter”. Esto puede ser no muy intuitivo para alguien que no esté familiarizado con videojuegos.
Errores y/o características que deberían modificarse	**NO** -Sin observaciones.

(sigue debajo)

Análisis preliminar del proyecto: *diagramas reducidos, extendidos y de secuencia.*	
¿Se encuentran accesibles los *diagramas reducidos* y *extendidos* de clases de cada una de las defensas?	**SI**
¿Se encuentran accesibles los *diagramas de secuencia* solicitados para la defensa correspondiente?	**SI**
¿Se encuentran accesibles los *diagramas reducido y completo de clases* de la entrega final?	**SI** - Ambos diagramas están muy completos en cuanto al nivel de detalle. El diagrama reducido muestra los nombres y cardinalidades de las relaciones. El diagrama extendido incluso está organizado en los paquetes del código. - Para reducir la cantidad de flechas en el diagrama reducido se podrían haber omitido modelar los componentes de la librería java.swing tales como JButton. Las asociaciones bidireccionales pueden modelarse como una recta sin flechas, con los nombres y cardinalidades de las relaciones en los extremos de las mismas. - Además de los estados de Mario esperados, se modela MarioRecuperación para el momento inmediato luego de que Mario recibe un golpe, lo cual es una acertada decisión de diseño. - Se modelan los distintos sonidos para los modos de juego utilizando el patrón factory, similar a como se lo utiliza para los Sprites, lo cual es una acertada decisión de diseño. - En DetectorDireccionColisión, en lugar de tener cuatro métodos, uno por cada dirección en la que se puede chocar, se podría tener un solo método que retorne el tipo de un enumerado. - Como la característica de emitir sonidos es transversal a muchas clases, es apropiado en este caso utilizar el patrón Singleton.

(sigue debajo)

Análisis del código fuente: *inspección del código fuente entregado.*	
¿El código fuente se encuentra adecuadamente organizado en paquetes?	**SI** - Sin observaciones.
¿El código exhibe de forma consistente y adecuada las prácticas propuestas por Clean Code?	**SI** - Se utilizan las prácticas de clean code consistentemente a lo largo del código. Los nombres de las variables y métodos son significativos y se subdividen en métodos privados los métodos complejos, mostrando una buena subdivisión de los problemas a resolver.
¿Existe una adecuada división de responsabilidades entre las clases que implementan cuestiones de lógica/gráfica?	**NO.** - En el método ControladorVistas.mostrarPantallaDeJuego, en lugar de utilizar getters para acceder a Partida y activar movimientos desde la capa vista (lo cual no es su responsabilidad), que Juego ofrezca un método, iniciar() por ejemplo, y que en ese método se activen los movimientos. Así se preserva mejor el encapsulamiento. - Que ElementoDeJuego tenga un atributo para mantener la posición gráfica atenta contra la separación entre las capas vista y lógica, las clases que pertenezcan a ésta última capa no deberían saber cómo son representadas gráficamente. El observer asociado es la clase correspondiente para almacenar este atributo ya que pertenece a la capa de la vista. - No debería ser necesario BucleVentana para actualizar los Labels, se debe utilizar el patrón observer. La interfaz Observer debería ofrecer un método adicional eliminar(). Las implementaciones de Observer tendrían una asociación a la pantalla que lo contiene. Entonces, cuando un ElementoDeJuego deba ser eliminado, inmediatamente avisa a su Observer y éste último se encarga de comunicarle a la pantalla que desaparezca. Con esto se evita el atributo booleano removido. Se puede usar observer también para el HUD al mostrar los datos del jugador.
¿Se implementa de forma consistente la detección de colisiones?	**SI** - La detección de colisiones se realiza según lo esperado utilizando dos hilos: uno para el jugador y otro para el resto de elementos. La detección de colisiones se realiza luego de que se concreten los movimientos y se revisan con las entidades dinámicas y estáticas.
¿Se implementa de forma consistente la resolución de colisiones?	**SI** - Se aplica correctamente el patrón visitor con una pequeña modificación en los nombres de los métodos visit(). Para reducir la cantidad de visitors, intentar utilizar herencia para aquellos métodos comunes. Por ejemplo, VisitorBuzzyBeetle y VisitorLakitu realizan lo mismo al recibir a Mario. La mayoría del resto de enemigos realiza las mismas acciones por lo que se podría tener un VisitorEnemigo. Si alguna colisión específica es distinta, se extiende el visitor y se sobreescribe el método correspondiente.

¿Se implementa de forma consistente los diferentes estados de Mario?	**SI** - En los estados, en lugar de retornar un Sprite con una imagen según lo que esté haciendo Mario, se debe extender la clase Sprite para que pueda tener más de una imagen y proveer métodos para cambiar su imagen actual de entre las que tiene almacenadas. La fábrica devolvería este Sprite.
Virtudes	**SI** - La consistente aplicación de clean code hace que el código sea fácilmente entendible.
Debilidades	**SI** - Se debe utilizar el .gitignore para que los archivos que no sean código fuente sean ignorados (.project, /bin, .settings, .classpath, etc). Es decir que no sean trackeados por Git y que en consecuencia no se suban a GitHub. - Varias clases ofrecen métodos getters a sus atributos internos y son utilizados para luego utilizar sus operaciones en otras clases que no tienen la responsabilidad de ejecutar ese tipo de métodos. Por ejemplo, en MasterMind.actualizarEnemigos(), se obtiene el observer del enemigo y se llama a su método actualizar. ¿Por qué MasterMind tiene que saber que Enemigo internamente tiene un observer si su responsabilidad es mover enemigos? Es mejor delegar en el método mover() de Enemigo la actualización de su representación gráfica. MasterMind solo se encarga de moverlo. De esta forma se preserva mejor el encapsulamiento.
Errores y/o características que deberían modificarse	**SI** - Aplicar el patrón observer para eliminar elementos gráficos. - Refactorizar ElementoDeJuego para que no tenga información sobre la vista.