



Universidad
Nacional
de Córdoba



Facultad de
Ciencias Exactas
Físicas y Naturales

Trabajo Práctico Final



Super Mario Bros



Ingeniería de Software

Nombre de Grupo:

“A de Ayme”

Integrantes

Maria Wanda Molina

43676123

Augusto Gabriel Cabrera

42259653

Fecha

22/06/2023



Índice

- 1. Introducción.**
- 2. Plan de Gestión de las Configuraciones.**
- 3. Documentos de Requerimientos.**
- 4. Diagrama UML.**
- 5. Patrón de Arquitectura.**
- 6. Patrones de Diseño.**
- 7. Integración Continua.**
- 8. Políticas.**
- 9. Bibliografía y Software.**





Introducción

El presente informe describe el trabajo final de la materia Ingeniería en Software, el cual tiene como objetivo principal la implementación de un proyecto de software a realizar como cierre de la materia en cuestión.

El proyecto seleccionado por el grupo es una versión modificada del famoso videojuego de 1985 "Super Mario Bros", este mismo pretende conservar la jugabilidad clásica del videojuego pero con referencias de Argentina.





Plan de Gestión de las Configuraciones

El Plan de Gestión de la Configuración describe la manera en que la información sobre los elementos del proyecto, así como cuáles elementos, serán registrados y actualizados de modo que el producto, servicio o resultado del proyecto se mantenga consistente y/u operativo.

Las configuraciones implementadas se enumeran a continuación:

1. Dirección y forma de acceso a la herramienta de control de versiones.

La herramienta de control de versiones a utilizar para el proyecto en cuestión será **GitHub**.

Link: <https://github.com/AugustoCabrera/superMarioBros/tree/main>

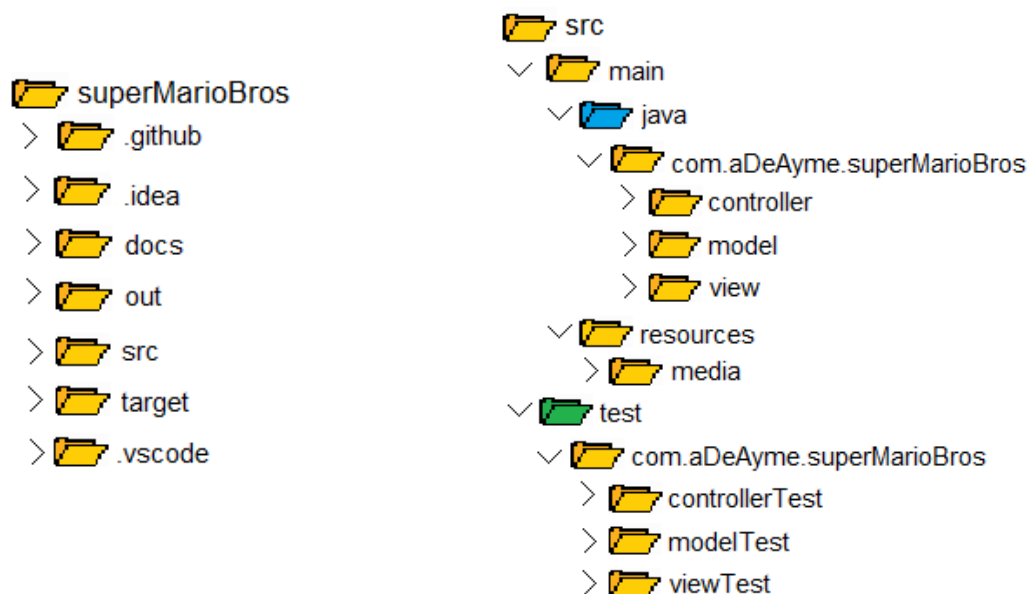
2. Dirección y forma de acceso a la herramienta de integración continua.

Con **GitHub Actions**, ya que puede crear workflow de integración continua (CI) directamente en el repositorio de GitHub anteriormente mencionado.

Link: <https://github.com/AugustoCabrera/superMarioBros/tree/main>

3. Esquema de directorios y propósito de cada uno.

El esquema de directorio que se utilizó para el desarrollo del proyecto en cuestión es el ilustrado a continuación:



El mismo sigue con los lineamientos recomendados por Maven para integrarlo al proyecto.





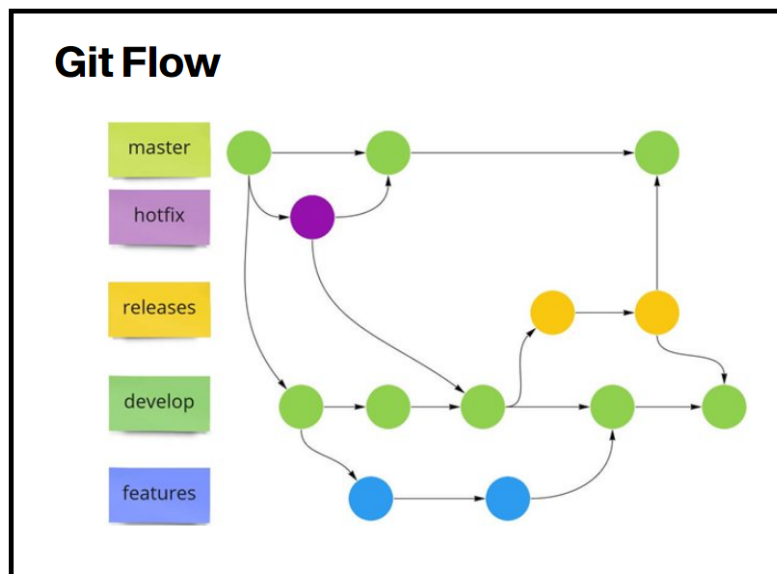
- Dentro de la carpeta *superMarioBros* tenemos:
 - Carpeta **.github**: contiene el workflow que corre mediante Github Actions (*pipeline.yml*)
 - Carpeta **.idea**: contiene archivos utilizados por IntelliJ para almacenar información relacionada con el entorno de desarrollo integrado y la configuración del proyecto.
 - Carpeta **docs**: contiene el informe presente en formato “.pdf”.
 - Carpeta **out**: contiene los archivos compilados y generados durante el proceso de compilación y construcción del proyecto.
 - Carpeta **src**: contiene el código y archivos de multimedia del juego.
 - Carpeta **target**: contiene los resultados de la construcción del proyecto, incluyendo los archivos compilados, los archivos generados y otros recursos relacionados con la compilación y empaquetado del proyecto.
 - Carpeta **.vscode**: almacenar configuraciones específicas de Visual Studio Code
 - Archivo **README**: presenta una breve descripción del juego con gráficas de como se ve y con qué lenguaje está desarrollado.
 - Archivo **pom.xml**: archivo de configuración utilizado en proyectos gestionados por Apache Maven
 - Archivos **1G.Super-Mario-Bros.iml** y **superMarioBros.iml**: contienen información relacionada con la configuración y estructura del módulo en IntelliJ IDEA
- Dentro de la carpeta *src* tenemos:
 - Carpeta **main.java.com.aDeAyme.superMarioBros.controller**: contiene los archivos de código sobre la administración de mapas, sonidos, estatus y botones de acción.
 - Carpeta **main.java.com.aDeAyme.superMarioBros.model**: contiene los archivos de código que especifica el accionar del personaje interactivo y de los enemigos, obstáculos y recompensas que se encuentra.
 - Carpeta **main.java.com.aDeAyme.superMarioBros.view**: contiene los archivos de código que definen las animaciones que rigen sobre las gráficas del juego.
 - Carpeta **main.resources.media**: contiene audios, fuente de letras e imágenes de los mapas.
 - Carpeta **test**: contiene los unit test de todo el proyecto, separado en carpetas controller, model y view. Además contiene un archivo “TestSuite.java”

4. Normas de etiquetado y de nombramiento de los archivos.





Las nombres de etiquetado utilizados en el desarrollo del proyecto, está basado en el plan de esquemas de ramas de “Git Flow”, el cual se describe a continuación:



Se basa en **dos ramas principales** con una vida infinita. Para cada tarea que se le asigna a un desarrollador se crea una rama feature en la cual se llevará a cabo la tarea, esta presenta la siguiente nominación “**Feature_**funcionalidad****” (con **funcionalidad** se refiere a una breve palabra que describa que cambios contiene ese branch, ej: FeatureOldCI). Una vez que ha finalizado, realizará un pull request (validación) contra develop para que validen el código.

Pasamos a detallar las dos ramas principales que se utilizan:

Master: Contiene el código de producción. Todo el código de desarrollo, a través del uso de releases, se mergea (fusiona) en esta rama en algún momento.

Develop: Contiene código de pre-producción. Cuando un desarrollador finaliza su feature, lo mergea contra esta rama.

Durante el ciclo de desarrollo, se usan varios tipos de ramas para dar soporte:

Feature: Por cada tarea que se realiza, se crea una nueva rama para trabajar en ella. Esta rama parte de develop.

Hotfix: Parte de master. Rama encargada de corregir una incidencia crítica en producción.

Releases: Parte de develop. Rama encargada de generar valor al producto o proyecto. Contiene el código que se desplegará, y una vez que se han probado las features integradas en la release, se "mergeará" a la rama master.





El proyecto utilizará el siguiente nombramiento de archivos:

- Para carpetas y archivos **“.wav”**: Camel Case
- Para archivos **“.java”**: Pascal Case
- Para archivos **“.png”**: Kebab Cases
- Para archivos Map **“.png”**: tienen el formato “Map X” donde X es el número de mapa.

5. Plan del esquema de ramas a usar.

El proyecto utilizará el esquema **Git Flow** para el desarrollo (antes mencionado).

6. Políticas de fusión de archivos y de etiquetados de acuerdo al progreso de calidad de los entregables.

Con GitHub se puede crear una regla de protección de rama a fin de aplicar determinados flujos de trabajo para una o varias ramas, como exigir una revisión de aprobación o pasar comprobaciones de estado para todas las solicitudes de incorporación de cambios combinadas en la rama protegida, el equipo de Scrum es cuestión **NO** utilizara política de fusión de archivos.

7. Forma de entrega de los releases, instrucciones mínimas de instalación y formato de entrega.

El proyecto se entrega en formato **“superMarioBros-AdeAyme.jar”** dentro de una carpeta **“.zip”** el cual este último se obtiene a través del artefacto otorgado en el pipeline (Deploy).

8. Plan de ceremonia de Scrum y roles de los miembros del equipo.

Para el desarrollo de este proyecto, el equipo de Scrum se desempeñará a lo largo de Sprints de **2 Semanas**, las cuales comenzarán los días Viernes de las semanas.

Cada ceremonia tiene también un tiempo limitado.

A continuación, se describen las etapas de trabajo del Equipo de Scrum:





- **Sprint Planning**

Tiene lugar al comienzo del Sprint. Esta ceremonia está diseñada para asegurar que cada miembro del equipo esté preparado y que en cada Sprint se hagan las cosas correctamente. Es una hora por cada semana de Sprint.

El equipo de Scrum realiza la Sprint Planning los días Viernes de 17:00 hs a 19:00 hs GMT -3

- **Daily Scrum**

Al menos debe hacerse una vez por día, preferiblemente por la mañana. No es una reunión «pesada», pues no supera los 15 minutos. El equipo se reúne y se comunica el progreso individual –siempre con base en la meta del Sprint–. Eso sí, aunque el tono de la ceremonia debe ser ligero y divertido, la junta también es informativa.

- **Sprint Review**

Es un momento dedicado a mostrar el trabajo completado durante el Sprint a las partes interesadas. De esta forma, las partes interesadas pueden ver cómo van las cosas y dedicarse a inspeccionar o adaptar el producto.

Se trata de estar centrados en el valor comercial que se está entregando. La duración de esta ceremonia es de entre 30 minutos y una hora por semana de Sprint.

El equipo de Scrum realiza la Sprint Review los días Jueves de 17:00 hs a 19:00 hs GMT-3

- **Sprint Retrospective**

Esta ceremonia consiste en obtener una retroalimentación rápida con el propósito de mejorar la cultura y desarrollo del producto.

Se realiza al final para que el equipo pueda “mirar hacia atrás” en su trabajo e identificar elementos que podrían mejorarse. La reunión no es mayor a una duración de 2 horas por cada Sprint de dos semanas.

Nuestro equipo de Scrum realiza la Sprint Review los días Jueves de 21:00 hs a 23:00 hs GMT-3.

Roles de los miembros del equipo.

Product Owner (PO): Molina, Maria Wanda - D.N.I.: 43.676.123

Scrum Master (SM): Cabrera, Augusto Gabriel - D.N.I.: 42.259.653

Equipo de Desarrollo (DS): Cabrera, Augusto Gabriel - D.N.I.: 42.259.653
Molina, Maria Wanda - D.N.I.: 43.676.123





Documento de Requerimientos

El documento de requerimientos de software, es el lugar donde se da descripción a las características y requisitos de un software, producto, programa o conjunto de programas. Los requisitos se expresan en lenguaje natural, sin consideraciones ni términos técnicos. La especificación de requisitos de software es el resultado del levantamiento de información con el usuario o cliente del producto. Son un método para una comunicación más concisa y clara entre los encargados de desarrollar el software y el área de negocio o clientes que usarán el producto.

A continuación se enuncian los Requerimientos Funcionales implementados a través de “*Historias de Usuarios*”:

Requerimientos Funcionales:



El programa “Super Mario Bros” se basa en un videojuego el cual se describe a continuación:






Los jugadores deben ser capaces de controlar a Mario mientras atraviesa el Reino Champiñón para rescatar a la Princesa Toadstool en su castillo. Mario debe atravesar etapas de desplazamiento lateral mientras evita peligros como enemigos y pozos con la ayuda de potenciadores como la Super Mushroom y la Fire Flower.

Historia de Usuario	
Número: 1	Usuario: Jugador
Nombre de Historia: Como jugador, quiero poder mover a Mario hacia la derecha o izquierda usando las flechas “←” y “→”, respectivamente, de manera que pueda avanzar por el nivel.	
Prioridad en negocio: ★★ ★	Riesgo de desarrollo: 🍄 🍄 🍄
Puntos estimados: 100 (POKER PLANNING)	Iteración asignada: 2do Sprint (Sprint que se realiza)
Programador responsable: Cabrera, Augusto Gabriel	
Descripción: El videojuego permitirá al jugador mover a Mario dentro de la dirección horizontal, pudiendo elegir el sentido del mismo mediante el uso de las teclas de dirección.	
Criterio de aceptación: <ul style="list-style-type: none">- Mario se mueve hacia la derecha o izquierda al presionar las flechas correspondientes.- La velocidad de Mario aumenta de manera uniforme mientras se mantiene presionada la flecha correspondiente.- Mario se detiene cuando no se presiona ninguna flecha.	










Historia de Usuario	
Número: 2	Usuario: Jugador
Nombre de Historia: Como jugador, quiero poder hacer saltar a Mario con la flecha “↑” para poder superar obstáculos y llegar a lugares más altos.	
Prioridad en negocio: 	Riesgo de desarrollo: 
Puntos estimados: 40	Iteración asignada: 2do Sprint
Programador responsable: Molina, Maria Wanda	
Descripción: El videojuego permitirá al jugador mover a Mario dentro de la dirección vertical, pudiendo sólo elevarse hasta una coordenada específica y luego descender a su posición inicial mediante el uso de las teclas de dirección.	
Criterio de aceptación: <ul style="list-style-type: none">- Mario salta al presionar la flecha “↑”.- La altura y distancia del salto son consistentes.- Mario puede saltar y romper bricks.- Mario puede matar monstruos saltando sobre ellos.	




Historia de Usuario	
Número: 3	Usuario: Jugador
Nombre de Historia: Como jugador, quiero poder recoger “coins” corriendo hacia ellas o saltando debajo de “coin boxes” para aumentar mi puntaje y tener una vida extra cuando recolecta una “1-up Mushroom”.	
Prioridad en negocio: 	Riesgo de desarrollo: 
Puntos estimados: 20	Iteración asignada: 3er Sprint
Programador responsable: Molina, Maria Wanda	
Descripción: El videojuego permitirá al jugador tomar y llevar cuenta de las “coins” que va encontrando en su camino, accediendo a “coin boxes”. También llevará cuenta del puntaje por cada recompensa adquirida y tendrá la posibilidad de aumentar en una unidad sus vidas al recolectar un “1-up Mushroom”.	
Criterio de aceptación: <ul style="list-style-type: none">- Mario puede recoger “coins”  saltando debajo de “coin boxes” .- Al recolectar una “1-up Mushroom” , el jugador obtiene una vida extra.	





- El puntaje del jugador aumenta en cincuenta al recoger "coins".



Historia de Usuario	
Número: 4	Usuario: Jugador
Nombre de Historia: Como jugador, quiero encontrar "Power Ups", como "Super Mushrooms" y "Fire Flowers", para hacer que Mario sea más poderoso.	
Prioridad en negocio: 	Riesgo de desarrollo: 
Puntos estimados: 8	Iteración asignada: 3er Sprint
Programador responsable: Cabrera, Augusto Gabriel	
Descripción: El videojuego permitirá al jugador tomar ciertos objetos del mapa denominados "Power Ups" para mejorar los poderes de Mario durante la partida. Según el objeto que adquiera al atravesarlo, tendrá distintos poderes.	
Criterio de aceptación: <ul style="list-style-type: none">- Mario encuentra "Power Ups" en ciertas partes del nivel.- Al encontrar un "Super Mushroom",  el tamaño de Mario aumenta y puede romper bricks adicionales.- Al encontrar un "Fire Flower",  Mario obtiene el poder de lanzar bolas de fuego  a los enemigos con la tecla "SPACE".- Los "Power Ups" tienen efectos visuales y de sonido distintivos .	


Historia de Usuario	
Número: 5	Usuario: Jugador
Nombre de Historia: Como jugador, quiero poder pausar  el juego en cualquier momento para tomar un descanso o atender otros asuntos.	
Prioridad en negocio: 	Riesgo de desarrollo: 
Puntos estimados: 13	Iteración asignada: 4to Sprint
Programador responsable: Cabrera, Augusto Gabriel	
Descripción: El videojuego permitirá al jugador pausar la partida y reanudarla, dando libertad de poder continuar cuando lo desee.	
Criterio de aceptación:	





- El juego puede ser pausado en cualquier momento presionando el botón "SCAPE".
- La pausa se muestra en la pantalla y se detiene cualquier acción del juego.
- El juego puede ser reanudado desde la pantalla de pausa.




Historia de Usuario	
Número: 6	Usuario: Jugador
Nombre de Historia: Como jugador, quiero elegir el mapa al cual voy a jugar.	
Prioridad en negocio: 	Riesgo de desarrollo: 
Puntos estimados: 5	Iteración asignada: 4to Sprint
Programador responsable: Molina, Maria Wanda	
Descripción: El videojuego permitirá al jugador elegir el mapa el cual va a jugar.	
Criterio de aceptación: <ul style="list-style-type: none">- El juego tiene 2 mapas.- El progreso del jugador se guarda al perder la partida.	



Historia de Usuario	
Número: 7	Usuario: Jugador
Nombre de Historia: Como jugador, quiero tener 3 vidas al comienzo del juego para tener una oportunidad de completar el nivel.	
Prioridad en negocio: 	Riesgo de desarrollo: 
Puntos estimados: 3	Iteración asignada: 5to Sprint
Programador responsable: Cabrera, Augusto Gabriel	
Descripción: El videojuego permitirá al jugador hacer uso y contabilizar los intentos para pasar de nivel mediante vidas, perdiendo una cuando muera y ganando una cuando agarre un "1-up Mushroom".	
Criterio de aceptación: <ul style="list-style-type: none">- El jugador comienza con 3 vidas  al comenzar el juego.- El número de vidas se muestra en la pantalla de juego.- Cuando el jugador pierde una vida, el número de vidas disminuye en uno.	





- El jugador al recoger un "1-up Mushroom"  incrementa en uno la cantidad de vidas .

Historia de Usuario	
Número: 8	Usuario: Jugador
Nombre de Historia: Como jugador, quiero que el juego termine cuando se acaben las vidas de Mario, para poder ver mi puntuación final.	
Prioridad en negocio: 	Riesgo de desarrollo: 
Puntos estimados: 3	Iteración asignada: 5to Sprint
Programador responsable: Cabrera, Augusto Gabriel	
Descripción: El videojuego finaliza cuando la cantidad de vidas de Mario es cero  , a lo que se muestra en pantalla un mensaje con la puntuación <u>100</u> ganada por el usuario.	
Criterio de aceptación: <ul style="list-style-type: none">-Cuando la vida de Mario llegue a cero, el juego deberá mostrar una pantalla de "Game Over".-La pantalla de "Game Over" debe mostrar la puntuación final del jugador en el juego.-La pantalla de "Game Over" debe permitir al jugador volver al menú con la tecla "SCAPE".	

Historia de Usuario	
Número: 9	Usuario: Jugador
Nombre de Historia: Como jugador, quiero que la dificultad del juego sea cada vez más elevada conforme progreso en el mismo mapa.	
Prioridad en negocio: 	Riesgo de desarrollo: 
Puntos estimados: 3	Iteración asignada: 5to Sprint
Programador responsable: Cabrera, Augusto Gabriel	
Descripción: El videojuego incrementa su dificultad a medida que el usuario obtiene puntos.	
Criterio de aceptación: <ul style="list-style-type: none">- El videojuego incrementará en 1 la velocidad de los enemigos , cuando la cantidad de puntos de Mario sea mayor a cincuenta.	





Requerimientos NO Funcionales

El videojuego debe poder ejecutarse con Java 8 instalado en el ordenador. Como primera instancia, el juego podría funcionar solo en el sistema operativo Windows. Además, el videojuego debe admitir más de 60 FPS de velocidad. Las características base que debe tener el dispositivo en el cual se ejecuta el videojuego son:

- RAM: 4 GB
- Procesador: Intel Celeron 1,50GHz
- Tipo de sistema: 64 bits

A continuación, se presentan los requerimientos en formato SRS según IEEE:

1. Introducción:

El presente documento describe los requerimientos del juego de plataformas Mario. El objetivo del juego es proporcionar una experiencia divertida y emocionante para el usuario, permitiendo que Mario se mueva de manera fluida en el mapa y realice una serie de acciones para avanzar en el juego.

2. Descripción general:

El juego de plataformas Mario deberá permitir al usuario controlar a Mario usando las flechas "←" y "→" para moverlo hacia la izquierda y hacia la derecha, respectivamente, y la flecha "↑" para hacerlo saltar. Mario deberá ser capaz de saltar y romper "bricks", matar monstruos y agarrar "coins" y "power ups" que se encuentren en el mapa.

3. Requisitos específicos:

- El juego deberá permitir que Mario tome una velocidad constante luego de acelerarlo de manera uniforme.
- Mario deberá ser capaz de saltar y romper "bricks".
- Mario deberá ser capaz de matar monstruos que se le presenten en el mapa.
- Mario deberá ser capaz de agarrar "coins" corriendo hacia ellas o saltando debajo de "coin boxes".
- Mario deberá encontrarse con "power ups" que lo harán más poderoso, "super mushrooms" que harán que crezca su tamaño, "fire flowers" que le darán el poder de lanzar bolas de fuego y "1-up mushrooms" que le darán una vida extra.
- El usuario deberá ser capaz de mover el mapa hacia la izquierda cuando Mario se mueva hacia la izquierda.
- El usuario deberá tener 3 vidas al comienzo del juego.
- El usuario deberá tener una vida más cuando recolecta una "1-up mushroom".
- El usuario deberá ser capaz de comenzar un nivel nuevo cuando termine otro.
- El usuario deberá ser capaz de pausar el juego en cualquier momento.





4. Requisitos de rendimiento:

1. Velocidad de fotogramas (framerate): El juego debe ser capaz de mantener una velocidad de fotogramas constante y suave, preferiblemente de al menos 60 fps para una experiencia de juego fluida.
2. Tiempo de carga: El tiempo de carga del juego debe ser mínimo para que los jugadores no tengan que esperar demasiado tiempo antes de comenzar a jugar. A partir de una notebook con 8Gb de RAM y un procesador Intel i3, no debe exceder los 3 segundos de demora.
3. Java instalado: El videojuego debe poder ejecutarse con Java 8 (o superiores) instalado en la computadora.
4. Escalabilidad: Este juego se adapta a versiones de sistema operativo Windows a partir del Windows Vista/7.

5. Requisitos de diseño:

- La interfaz de usuario deberá ser de 1268 x 708 píxeles como mínimo.

6. Requisitos de interfaces externas:

- El juego requiere tener un teclado y un monitor integrados a la computadora como mínimo para su ejecución y jugabilidad. La adición de un mouse y parlantes mejora considerablemente la jugabilidad.

7. Requisitos de seguridad:

- El juego no requerirá medidas de seguridad especiales y otras.

8. Requisitos de pruebas:

- El juego deberá ser probado exhaustivamente antes de su lanzamiento para asegurar su funcionamiento correcto.
- Se deberán realizar pruebas de unidad para garantizar la calidad del juego.

9. Requisitos de mantenimiento:

- El juego deberá ser fácil de mantener y actualizar en caso de errores o mejoras futuras.



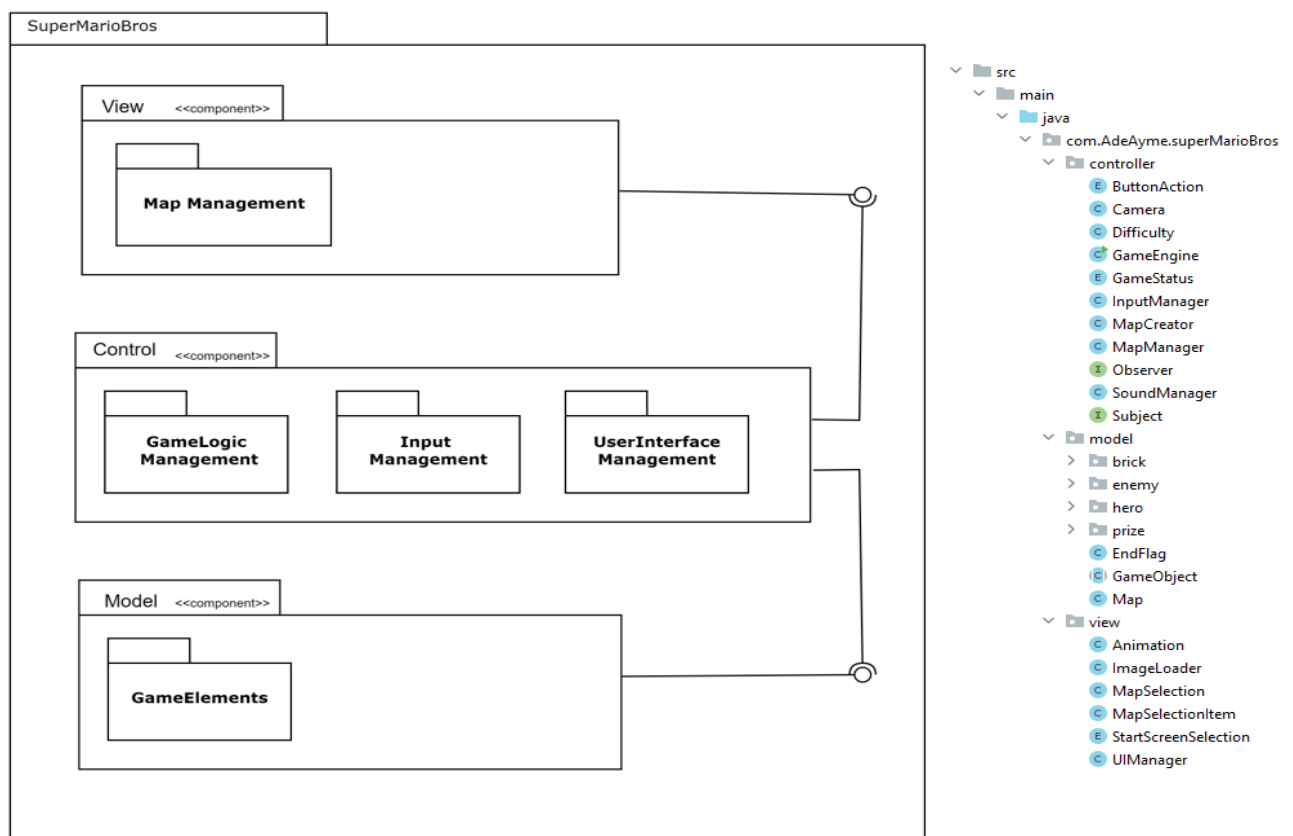


Diagrama UML

Una imagen vale más que mil palabras. Es por eso que se creó la generación de diagramas con el Lenguaje Unificado de Modelado (UML): ***“para forjar un lenguaje visual común en el complejo mundo del desarrollo de software”***.

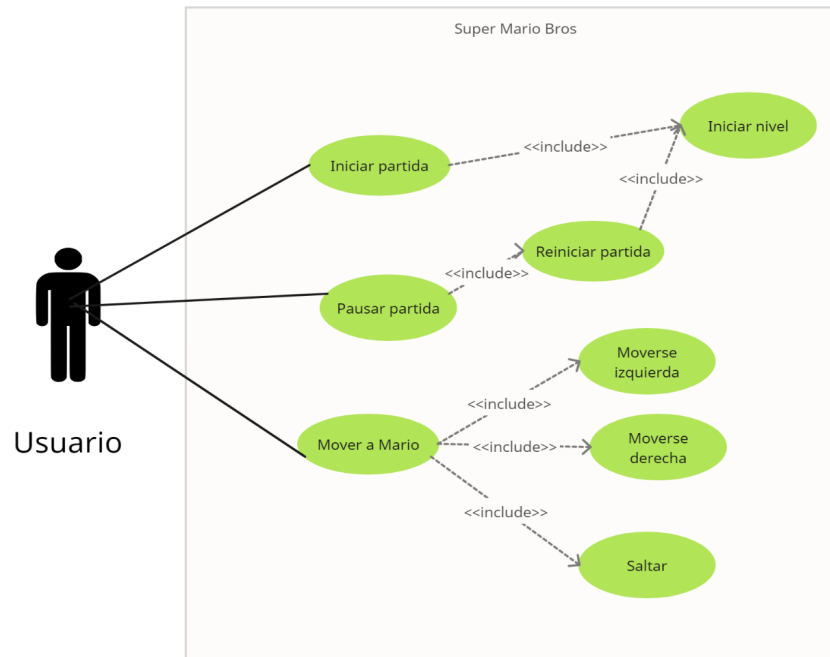
A continuación se muestran diferentes diagramas UML que describen distintas características del proyecto en su última etapa:

- **Diagrama de Clases**
Presente en el Drive que se encuentra junto a este informe.
- **Diagrama de Secuencias**
Presente en el Drive que se encuentra junto a este informe.
- **Diagrama de Componentes**

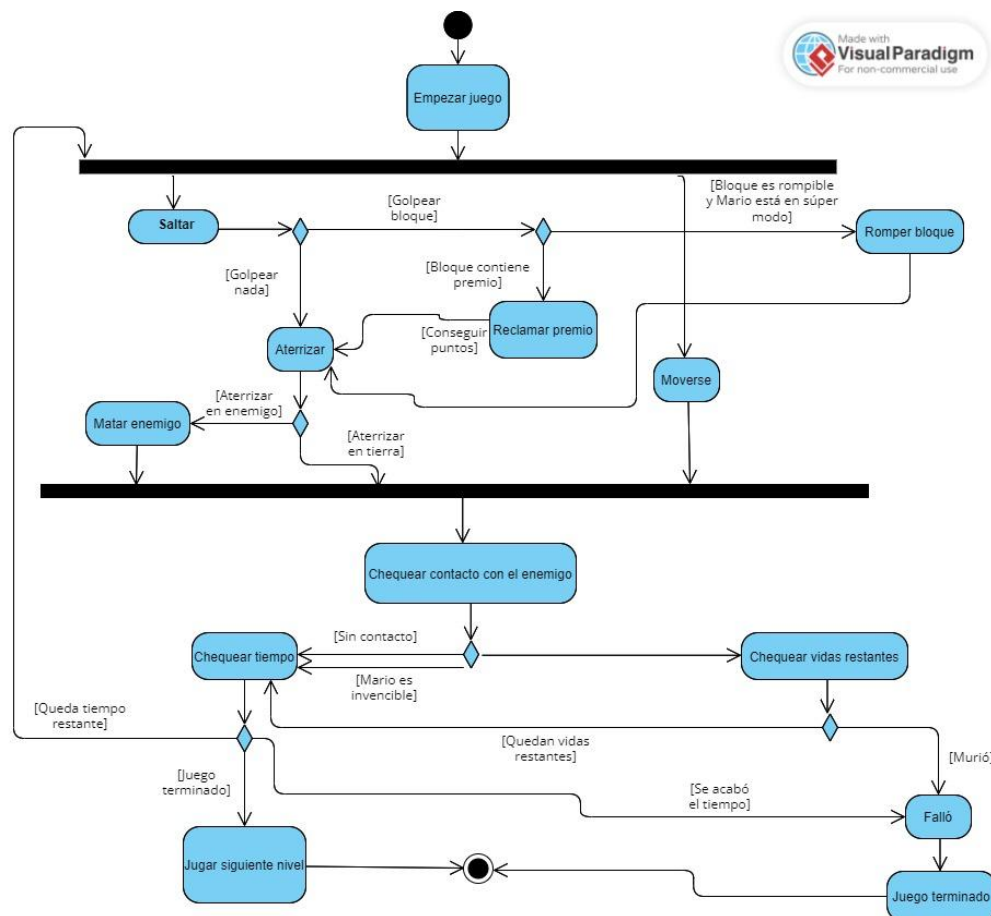




- **Diagrama de Casos de uso**



- **Diagrama de Actividades**





Patrón de Arquitectura

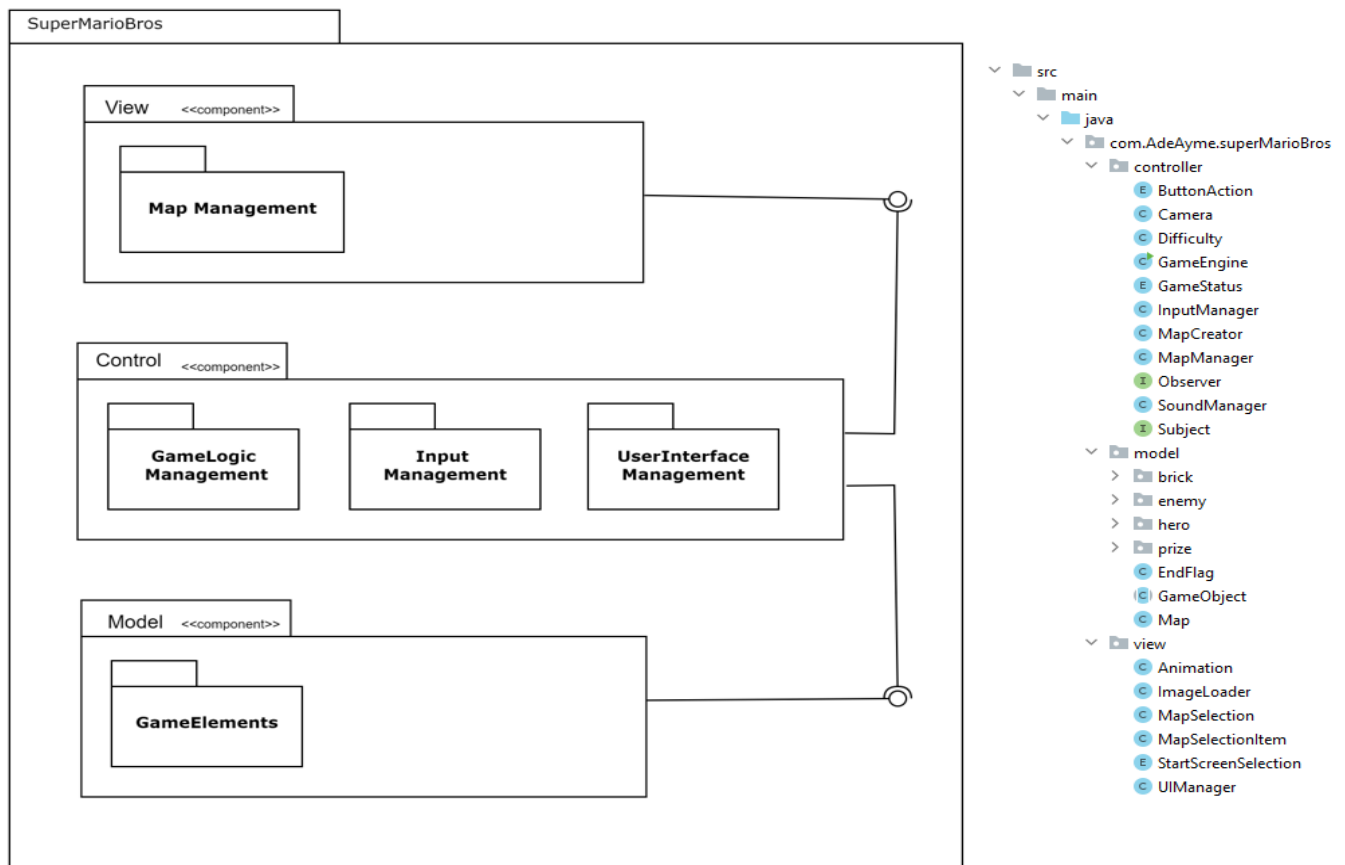
Esta temática es crítica a la hora de desarrollar el proyecto, ya que esta misma define como serán las conexiones entre los requerimientos, las funcionalidades y cómo se comporta el sistema y cuales son las capacidades del mismo, es un enfoque en cómo se organizan los componentes, donde están ubicada cada una de las capas y cada uno de los módulos de software.

Antes de diseñar el sistema, se debe elegir el estilo arquitectónico más adecuado. Para el proyecto presentado, la arquitectura **MVC** (Modelo-Vista -Controlador) es la más adecuada. Este estilo de arquitectura también viene con sus propias ventajas:

1. En esta arquitectura el sistema sufre una descomposición del sistema, esto permite un proceso de desarrollo más rápido y paralelo.
2. El uso de MVC permite realizar modificaciones en las partes del subsistema sin afectar las otras partes. Esto facilita futuras actualizaciones del sistema.
3. Este estilo arquitectónico desacopla el acceso a datos y la presentación de datos.

De acuerdo con el estilo arquitectónico MVC, todo el sistema se ha descompuesto en tres componentes principales. El paquete del controlador contiene todas las clases que se ocupan con control. Del mismo modo, tenemos un paquete modelo que representa objetos en el juego. Estas clases contienen estados, ubicaciones, tipos, habilidades, etc. El paquete de vista contiene clases que están relacionadas con el límite de visualización de usuario.

El siguiente diagrama de componentes especifica los componentes que ayudan a hacer esas funcionalidades, representando la forma en la que estos se organizan con el código implementado.



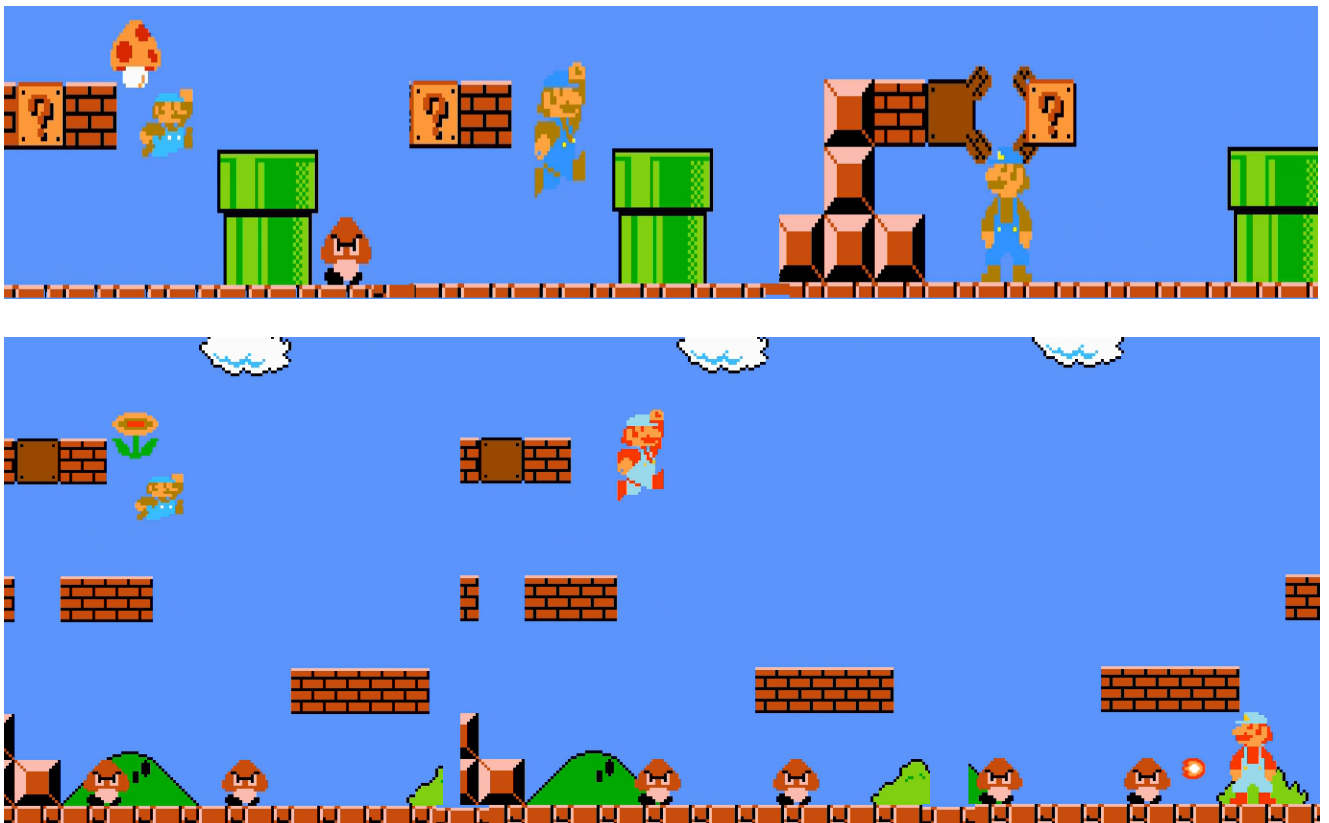


Patrones de Diseño

Está pensado en la reutilización, en la fácil compresión, por eso un sinónimo claro de esto es “Soluciones comunes a Problemas Comunes”. Cada patrón es como un plano que se puede personalizar para resolver un problema de diseño particular del código en cuestión. En este trabajo se implementaron dos patrones de diseño de propósito Conductual, los cuales se muestran a continuación:

Strategy

Este patrón es totalmente aplicable a los programas del estilo del proyecto en cuestión, ya que este mismo define una familia de algoritmos, encapsula cada uno, y los hace intercambiables. El patrón de Strategy sugiere que tome una clase que haga algo específico de muchas maneras diferentes y extraiga todos estos algoritmos en clases separadas llamadas estrategias, para este proyecto , el momento en el cual Mario agarra el “Super Mushroom” o la “FireFlower”, obtiene nuevo tamaño y capacidades en Runtime(romper bloques ordinarios y lanzar bolas de fuego), por ende, mario cambió su comportamiento en ejecución.



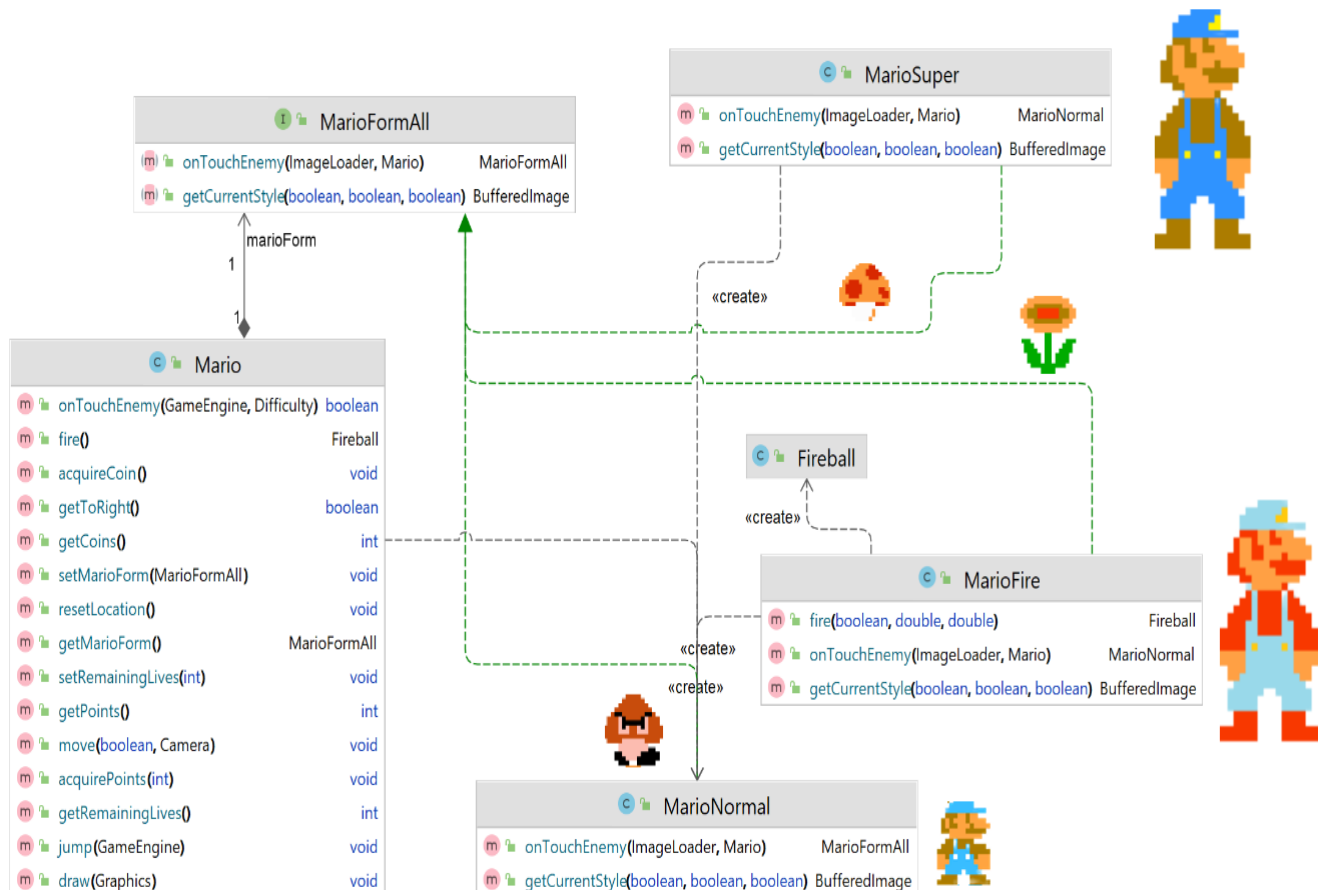
Implementación del patrón Strategy: Se utiliza la interfaz “MarioFormAll” para encapsular los comportamientos (Formas) de Mario, esta interfaz es implementada por tres clases que representan los posibles estados de mario (“MarioNormal” , “MarioSuper” y “MarioFire”),





por la acción de implementación, cada una de estas tres clases deberá redefinir los métodos señalizados en esa interfaz, los cuales son:

- **onTouchEnemy(ImageLoader,Mario):**
 - MarioNormal: No hace nada. Ya que la administración de vidas se encarga la clase “Mario”.
 - MarioSuper: vuelve al comportamiento de “MarioNormal”.
 - MarioFire: vuelve al comportamiento de “MarioNormal”.
- **getCurrentStyle(boolean, boolean, boolean):**
 - MarioNormal: Administra las ilustraciones del Mario en **estado Normal** de movimiento hacia la derecha y la izquierda, y también las ilustraciones de salto y caída.
 - MarioSuper: Administra las ilustraciones del Mario en **estado Super** de movimiento hacia la derecha y la izquierda, y también las ilustraciones de salto y caída.
 - MarioFire:Administra las ilustraciones del Mario en **estado Fire** de movimiento hacia la derecha y la izquierda, y también las ilustraciones de salto y caída.



Métodos Exclusivos de la clase implementadora:



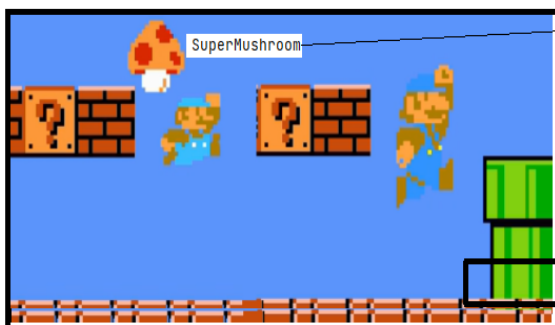


- **Fire(boolean,double,double): SOLO PARA COMPORTAMIENTO DE MARIOFIRE**, crea una bola de fuego que elimina al enemigo que toque.
- En la clase **OrdinaryBrick** se define el método **reveal(GameEngine)** el cual define que Mario en este estado puede romper ladrillos comunes (**SOLO PARA COMPORTAMIENTO DE MARIOSUPER**).

¿Cómo se alterna entre los estados y comportamientos de Mario?

A continuación, se muestra gráficamente y en código el acto de setear el comportamiento para el caso:

- **mario.setMarioForm(MarioSuper):**



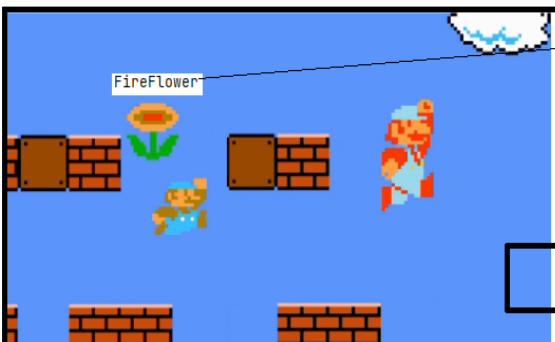
```
public class SuperMushroom extends BoostItem implements MagicObject{
    @Override
    public void setChangeMarioForm(Mario mario) {
        ImageLoader imageLoader = new ImageLoader();

        BufferedImage[] leftFrames = imageLoader.getLeftFrames(Mario.SUPER);
        BufferedImage[] rightFrames = imageLoader.getRightFrames(Mario.SUPER);

        Animation animation = new Animation(leftFrames, rightFrames);

        MarioFormAll newForm = new MarioSuper(animation, mario);
        mario.setMarioForm(newForm);
    }
}
```

- **mario.setMarioForm(MarioFire):**



```
public class FireFlower extends BoostItem implements MagicObject{
    @Override
    public void setChangeMarioForm(Mario mario) {
        ImageLoader imageLoader = new ImageLoader();

        BufferedImage[] leftFrames = imageLoader.getLeftFrames(Mario.FIRE);
        BufferedImage[] rightFrames = imageLoader.getRightFrames(Mario.FIRE);

        Animation animation = new Animation(leftFrames, rightFrames);

        MarioFormAll newForm = (MarioFormAll) new MarioFire(animation, mario);
        mario.setMarioForm(newForm);
    }
}
```

Observer

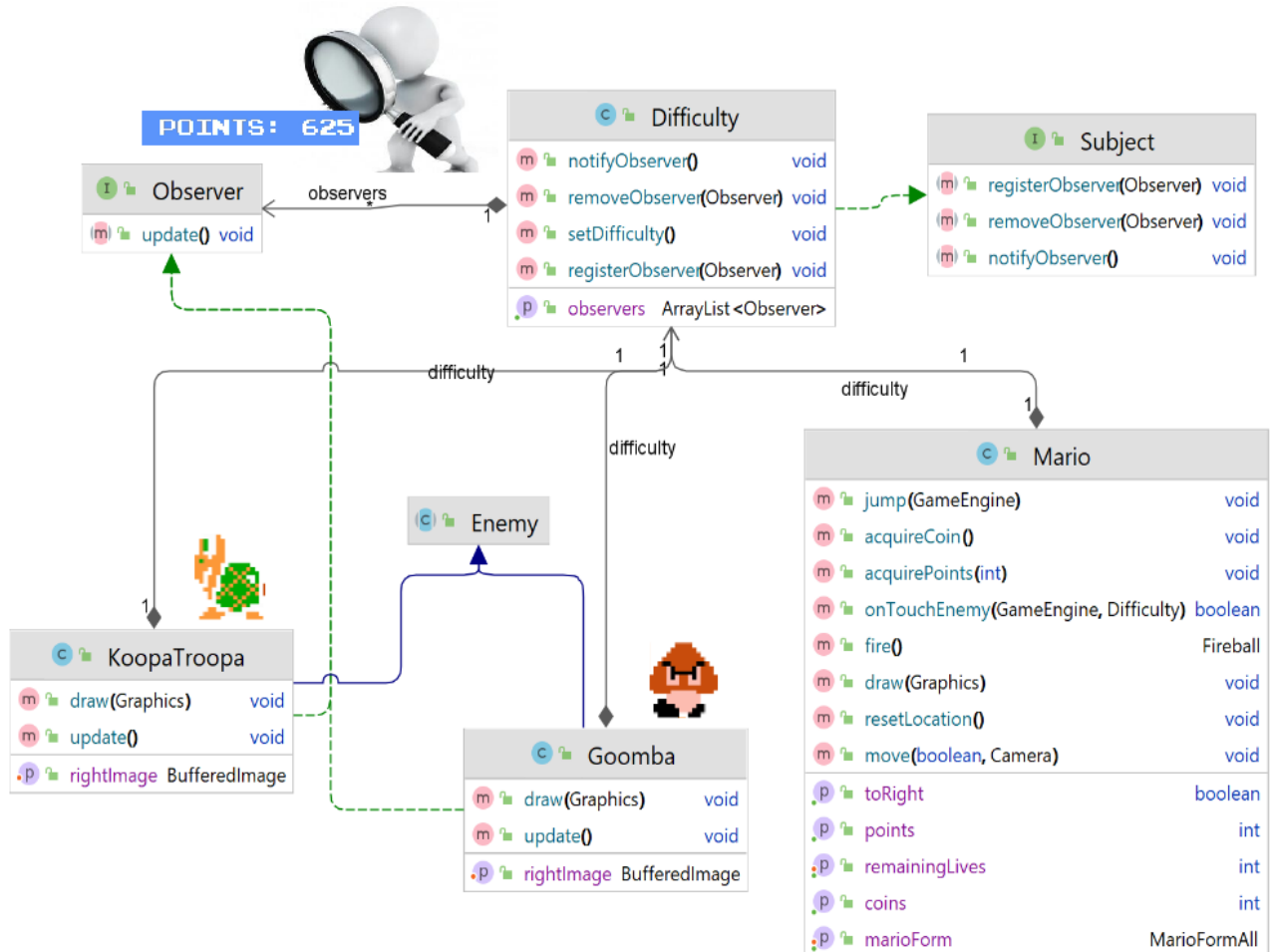
La idea básica es la de un sujeto que emite notificaciones y tenemos observadores que reciben esas notificaciones, permite definir un mecanismo de suscripción para notificar a varios objetos sobre cualquier evento que le suceda al objeto que están observando. Sirve para notificar a los observadores cuando los valores del sujeto cambian. Para este proyecto, se pensó integrar este patrón a través de una filosofía bastante frecuente en los videojuegos actuales: “El progreso es severamente penalizado”, se desarrolló una nueva clase, la clase “Difficulty” esta funciona como el “sujeto notificador” y en el momento que se crea el mapa TODOS los enemigos (Goomba y KoopaTroopa) se unen a la colección de observadores, para que cuando se supere el valor de puntos 50, puntaje que va a ser observado constantemente por el sujeto, éste notifique a TODOS los observadores para que aumenten





su velocidad de movimiento, para así de esta forma sea mucho más difícil para el jugador, poder esquivarlos o eliminarlos.

A continuación, se observa el diagrama de clases que resume lo anterior explicado:



¿Cómo se incrementa la dificultad en el juego utilizando el patrón Observer?

Una vez que mario supera los cincuenta puntos, cada vez que se realiza una aquire en los puntos, se realiza la siguiente secuencia:

Se llama al método **setDifficulty()** del objeto **Difficulty**, que proporciona el mapa. Este método llama a **notifyObserver()**, encargado de iterara en la lista **Observers** (donde se encuentran los observadores del patrón) y llama al método **update()** de cada objeto. Este método **update()** toma la velocidad horizontal de los objetos observadores y la incrementa en uno.

(Nota: La clase Difficulty posee una única instancia en todo el desarrollo del videojuego, para versiones posteriores, se podría incorporar algún patrón de propósito Creacional Singleton)





Integración Continua

La integración continua (**CI**) es una práctica de desarrollo de software mediante la cual los desarrolladores combinan los cambios en el código en un repositorio central de forma periódica, tras lo cual se ejecutan versiones y pruebas automáticas (**JUnit**). La integración continua se refiere en su mayoría a la fase de creación o integración del proceso de publicación de software y conlleva un componente de automatización. Para el desarrollo de este trabajo las prácticas CI son implementadas a través de “**GitHub Actions**”, las dependencias “**JUnit**” (Implementación de Test Unitarios) y “**JaCoCo**” (cobertura de código). Se enuncian los siguientes apartados:

1. *JUnit*

Una prueba unitaria o test unitario es una forma efectiva de comprobar el correcto funcionamiento de las unidades individuales más pequeñas del programa. Por ejemplo en diseño estructurado o en diseño funcional una función o un procedimiento, en diseño orientado a objetos una clase. En este trabajo, los JUnit fueron desarrollados a través de la dependencia “**JUnit**”, a continuación se observan los test unitarios implementados para la correcta comprobación de que se cumplen los **Requerimientos Funcionales**.

```
package com.aDeAyme.superMarioBros.modelTest;

import ...
3 usages
public class MarioTest {

    @Test
    public void onTouchEnemyTest(){
        GameEngine engine=new GameEngine( msgj: "MarioTest");
        Difficulty difficulty= new Difficulty();
        Mario mario= new Mario( x: 0, y: 0, difficulty);
        //test para un marioNormal
        mario.onTouchEnemy(engine, difficulty);
        assertEquals(mario.getRemainingLives(), actual: 2, delta: 0.0);
        //test para un marioFire
        FireFlower flower = new FireFlower( x: 0, y: 0, style: null);
        flower.onTouch(mario, engine);
        mario.onTouchEnemy(engine, difficulty);
        assertTrue(mario.getMarioForm() instanceof MarioNormal);
        //test para un marioSuper
        SuperMushroom mushroom = new SuperMushroom( x: 0, y: 0, style: null);
        mushroom.onTouch(mario, engine);
        mario.onTouchEnemy(engine, difficulty);
        assertTrue(mario.getMarioForm() instanceof MarioNormal);
    }
}
```

Como ejemplo agregamos el siguiente Test del metodo **onTouchEnemy()** el cual prueba como al llamarlo, convierte la forma **MarioSuper** o **MarioFire** en **MarioNormal** y, si se encontraba con la forma **MarioNormal**, pierde una vida.





```
package com.aDeAyme.superMarioBros.modelTest;
import ...

1 usage
public class KoopaTroopaTest {
    no usages
    @Test
    public void updateTest(){ //Prueba del update del Observer
        KoopaTroopa koopa_troopa = new KoopaTroopa( x: 0, y: 0, style: null);
        koopa_troopa.update();
        double speed= koopa_troopa.getVelX();
        assertEquals( expected: 4, speed, delta: 0.0);
    }
}
```

Este otro test prueba el método *update()* de la clase *KoopaTroopa*, el cual es llamado por el objeto *Difficulty* del mapa. Este test asegura que la velocidad del enemigo aumenta luego de llamar al *update()*.

```
1 usage new *
public class BricksTest {
    no usages new *
    @Test
    public void revealTest(){
        GameEngine engine = new GameEngine( msg: "BrickTest");
        //reveal de OrdinaryBrick
        OrdinaryBrick ordinary=new OrdinaryBrick( x: 0, y: 0, style: null); //Creo brick ordinario
        Object prizeOrdinary = ordinary.reveal(engine); // como es un bloque ordinario no tiene sorpresa
        assertNull(prizeOrdinary); //verifico si me da alguna sorpresa (para Mario Normal)
        SuperMushroom mushroom =new SuperMushroom( x: 0, y: 0, style: null);
        mushroom.onTouch(engine.getMapManager().getMario(), engine);
        assertNull(ordinary.reveal(engine)); //para Mario Super
        assertTrue(ordinary.isBreakable());
        assertEquals( expected: -27, ordinary.getX(), delta: 0.0);
        assertEquals( expected: -27, ordinary.getY(), delta: 0.0);

        //reveal de SurpriseBrick
        SurpriseBrick surprise=new SurpriseBrick( x: 0, y: 0, style: null, (Prize) new Coin(x: 0, y: 0, style: null, point: 3));
        Prize prizeS= surprise.reveal(engine); // Obtengo la sorpresa en el bloque sorpresa
        assertNotSame(prizeS, actual: null); // verifico que no sea null
        assertEquals(prizeS.getPoint(), actual: 3); // verifico que valga lo que definido

        //reveal de GroundBrick
        GroundBrick ground=new GroundBrick( x: 0, y: 0, style: null); //creo un brick piso
        assertNull(ground.reveal(engine)); //verifico que no revele nada
    }
}
```

Este nuevo test prueba el metodo ***reveal()*** que tiene cada tipo de Brick (**OrdinaryBrick**, **SurpriseBrick** y **GroundBrick**).





2. JaCoCo

Este mecanismo proporciona la cobertura de código, que no es más que una métrica del testing de software, que se encarga de determinar la cantidad de líneas de código que se validan en las pruebas unitarias.

A continuación se explica la información proporcionada por el mismo:

my-app												
my-app												
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.aDeAyme.superMarioBros.controller	<div><div></div></div>	34%	<div><div></div></div>	13%	228	271	491	674	83	112	1	9
com.aDeAyme.superMarioBros.view	<div><div></div></div>	32%	<div><div></div></div>	34%	55	90	173	266	24	46	1	6
com.aDeAyme.superMarioBros.model	<div><div></div></div>	37%	<div><div></div></div>	26%	51	90	98	186	32	64	0	3
com.aDeAyme.superMarioBros.model.hero	<div><div></div></div>	42%	<div><div></div></div>	12%	35	52	67	127	11	27	1	5
com.aDeAyme.superMarioBros.model.brick	<div><div></div></div>	50%	<div><div></div></div>	0%	12	20	27	59	9	17	0	5
com.aDeAyme.superMarioBros.model.enemy	<div><div></div></div>	40%	<div><div></div></div>	12%	7	13	18	36	3	9	0	3
com.aDeAyme.superMarioBros.model.prize	<div><div></div></div>	80%	<div><div></div></div>	40%	8	26	12	74	5	21	0	5
Total	3,714 of 5,950	37%	434 of 532	18%	396	562	886	1,422	167	296	3	36

Esta imagen muestra el porcentaje de test realizados que cubren el código de cada carpeta especificada. Dentro de cada carpeta encontramos el porcentaje de cobertura de cada clase:

com.aDeAyme.superMarioBros.view

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
UIManager	<div><div></div></div>	0%	<div><div></div></div>	0%	23	23	110	110	16	16	1	1
MapSelection	<div><div></div></div>	36%	<div><div></div></div>	31%	15	23	28	51	2	7	0	1
StartScreenSelection	<div><div></div></div>	46%	<div><div></div></div>	0%	11	13	16	21	3	5	0	1
Animation	<div><div></div></div>	33%	<div><div></div></div>	0%	5	8	11	19	2	5	0	1
ImageLoader	<div><div></div></div>	93%	<div><div></div></div>	100%	1	17	8	51	1	7	0	1
MapSelectionItem	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	6	0	14	0	6	0	1
Total	853 of 1,270	32%	58 of 88	34%	55	90	173	266	24	46	1	6

Si seleccionamos uno de los archivos, podemos ver los métodos que faltan por cubrir y cuáles ya están cubiertos:

MapSelection

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
draw(Graphics)	<div><div></div></div>	0%	<div><div></div></div>	0%	3	3	17	17	1	1
selectMap(Point)	<div><div></div></div>	0%	<div><div></div></div>	0%	8	8	8	8	1	1
changeSelectedMap(int, boolean)	<div><div></div></div>	70%	<div><div></div></div>	66%	2	4	2	7	0	1
createItems(ArrayList)	<div><div></div></div>	95%	<div><div></div></div>	75%	1	3	1	8	0	1
MapSelection()	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	5	0	1
selectMap(int)	<div><div></div></div>	100%	<div><div></div></div>	75%	1	3	0	3	0	1
getMaps()	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	3	0	1
Total	180 of 283	36%	22 of 32	31%	15	23	28	51	2	7





```
38. private void getMaps(){ //Devuelve los mapas
39. //TODO: read from file
40. maps.add("Map 1.png");
41. maps.add("Map 2.png");
42. }
43.
44. private MapSelectionItem[] createItems(ArrayList<String> maps){ //Crea los items de los mapas
45. if(maps == null)
46. return null;
47.
48. int defaultGridSize = 100;
49. MapSelectionItem[] items = new MapSelectionItem[maps.size()];
50. for (int i = 0; i < items.length; i++) {
51. Point location = new Point(0, (i+1)*defaultGridSize+200);
52. items[i] = new MapSelectionItem(maps.get(i), location);
53. }
54.
55. return items;
56. }
57.
58. public String selectMap(Point mouseLocation) { //Elige un mapa de la lista de items mediante el mouse
59. for(MapSelectionItem item : MapSelectionItems){
60. Dimension dimension = item.getDimension();
61. Point location = item.getLocation();
62. boolean inX = location.x <= mouseLocation.x && location.x + dimension.width >= mouseLocation.x;
63. boolean inY = location.y >= mouseLocation.y && location.y - dimension.height <= mouseLocation.y;
64. if(inX && inY){
65. return item.getName();
66. }
67. }
68. return null;
69. }
```

Mediante el uso de la acción **cicirello/jacoco-badge-generator@v2** configurada en el pipeline, tenemos el siguiente badge en la sección de Summary de Github Actions:

test summary

JaCoCo Test Coverage Summary

- Coverage: 35.713%
- Branches: 16.541%
- Generated by: jacoco-badge-generator

Job summary generated at run-time

3. GitHub Actions

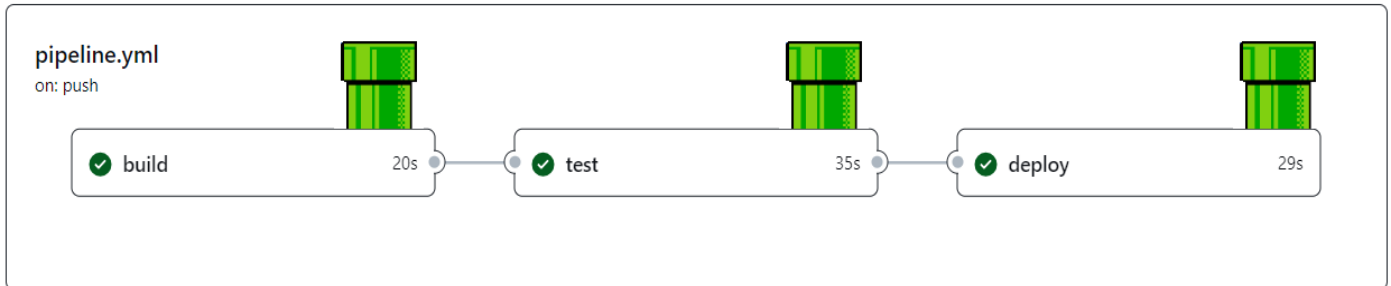
Es una plataforma de integración continua (CI) y entrega continua (CD) que le permite automatizar su canalización de compilación, prueba e implementación. Puede crear flujos de trabajo que construyen y prueban cada solicitud de extracción en su repositorio, o implementar solicitudes de extracción combinadas en producción.



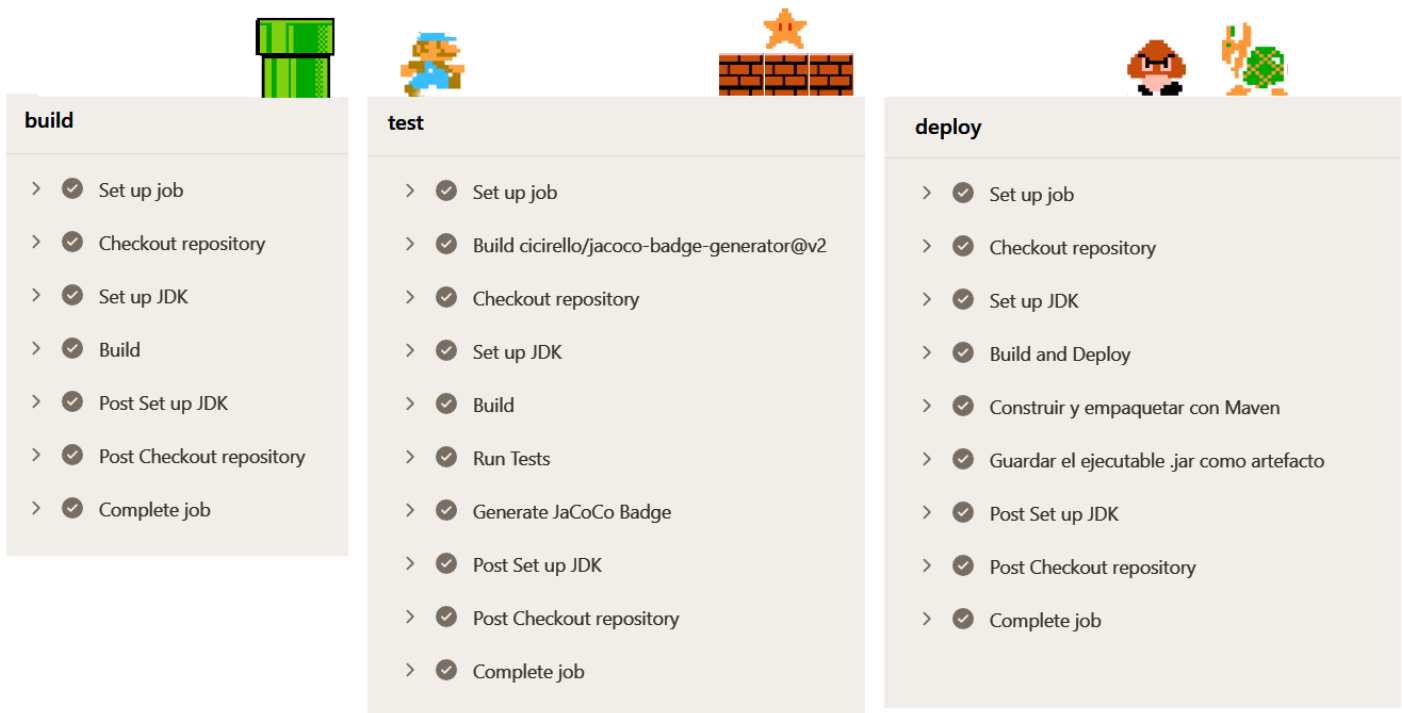


A continuación se muestra la utilización de la plataforma en el repositorio GitHub del presente informe:

Se puede observar el siguiente **Pipeline**:



A continuación se especifican las tres etapas:



- **build:**
 - **Set up job:** Prepara la máquina virtual.
 - **Checkout repository:** Obtiene una copia del código fuente del repositorio, en el entorno de ejecución del workflow.
 - **Set up JDK:** Configurar el JDK de java.
 - **Build:** Compila el proyecto.
 - **Post Set up JDK:** Se continúa configurando el JDK





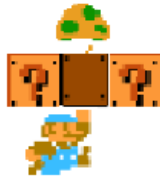


- **Post Checkout repository:** Realiza acciones específicas relacionadas con la configuración del entorno y la manipulación del código fuente del repositorio
- **Complete job:** Indica que esta sección del Pipeline fue finalizada.
- **test:**
 - **Set up job:** Prepara la máquina virtual.
 - **Build cicirello/jacoco-badge-generator@v2:** Ejecución de una acción específica llamada "cicirello/jacoco-badge-generator"
 - **Checkout repository:** Obtiene una copia del código fuente del repositorio, en el entorno de ejecución del workflow.
 - **Set up JDK:** Configurar el JDK de java.
 - **Build:** Compila el proyecto.
 - **Run tests:** Corre TODOS los Unitest.
 - **Generate JaCoCo Badge:** Realiza la acción de generar insignias (badges) para informes de cobertura de código generados por Jacoco.
 - **Post Checkout repository:** Realiza acciones específicas relacionadas con la configuración del entorno y la manipulación del código fuente del repositorio.
 - **Complete job:** Indica que esta sección del Pipeline fue finalizada.
- **deploy:**
 - **Set up job:** Prepara la máquina virtual.
 - **Checkout repository:** Obtiene una copia del código fuente del repositorio, en el entorno de ejecución del workflow.
 - **Set up JDK:** Configurar el JDK de java.
 - **Build and Deploy:** Se realiza una compilación completa del proyecto, incluida la ejecución de pruebas.
 - **Construir y empaquetar con Maven:** Empaqueta el proyecto compilado en un formato específico (.jar).
 - **Guardar el ejecutable .jar como artefacto:** Ejecuta la acción actions/upload-artifact@v2
 - **Post Set up JDK:** Se continúa configurando el JDK.
 - **Post Checkout repository:** Realiza acciones específicas relacionadas con la configuración del entorno y la manipulación del código fuente del repositorio.
 - **Complete job:** Indica que esta sección del Pipeline fue finalizada.

Estas configuraciones del Pipeline , se realizan en el archivo “**pipeline.yml**”.





A continuación, se observa la ventana de Github Actions en la cual se puede obtener el ejecutable:

Artifacts Produced during runtime			
Name		Size	
	superMarioBros-AdeAyme	11.9 MB	





Políticas

Las políticas en los equipos de Scrum son acuerdos y reglas establecidas para guiar el trabajo colaborativo y asegurar un funcionamiento eficiente y efectivo del equipo. A continuación, para el desarrollo de este trabajo, el equipo de Scrum “A de Ayme” planteó las siguientes políticas:

1. Al momento de hacer un push a la rama Develop, se debe obligatoriamente, antes de realizar esta acción, coordinar una videollamada e informar acerca de si los cambios son aprobados por TODOS los miembros del equipo, caso negativo, el opositor deberá justificar la causa de su disgusto.
2. Siguiendo con la política anterior, el push OBLIGATORIAMENTE se debe realizar dentro de la videollamada, revisando la actualizando en Github.
3. Solo se podrá realizar un push en la rama Develop o Main, si se cuenta con un Coverage del 40% o superior, con respecto a los archivos “Models”.
4. En el momento de realizar un commit a cualquier rama, se debe seguir el siguiente formato:
 - a. **“ ACCIÓN REALIZADA FECHA HORA ”**
 - i. *ej: En caso de que se haya agregado una clase llamada “Enemigo” el commit sería el siguiente*
commit: “NuevoEnemigoAgregado 15/04/2023 16:15”





Conclusión

Con el trabajo realizado, se pudo adoptar y conocer nuevas “buenas prácticas” de programación, y conocer aún más sobre el desarrollo de software, tanto a baja escala como a gran escala. También fue de ayuda para conocer las distintas herramientas utilizadas en el ámbito laboral, tales como Github, Github Actions , Java, Maven , entre otros.

Además, se implementó la forma de trabajo acorde a las metodologías Ágiles, El Scrum, el cual demostró ser una forma totalmente óptima y organizada a la hora de planificar y mejorar tanto en el desarrollo del código, como en el diseño del mismo, y también a organizar hábitos de trabajo mucho más ordenadas cronológicamente.





Bibliografía y Software.

Bibliografía:

- SOMMERVILLE - INGENIERÍA DE SOFTWARE, en su novena edición
- Información sobre los Patrones: <https://refactoring.guru/es/design-patterns>
- Apuntes de las clases del año corriente.

Software:

- IDE: IntelliJ - 2022.3.3
- Gestor de Dependencias: Apache Maven - Version 3.8.6
 - Dependencias
 - jacoco-maven-plugin-Version 0.8.3
 - Junit - Version 4.13.1
- Plataforma de integración continua (CI) y entrega continua (CD): Github Actions
- Herramienta de Control de Versiones: Github
(<https://github.com/AugustoCabrera/superMarioBros>)
- Herramienta para hacer graficos: <https://www.lucidchart.com/>

