
Universidad Nacional de Córdoba


Facultad de Ciencias Exactas Físicas y Naturales

# Los piratas

# Ingeniería de Software

---

# PacMan

## Diseño, Implementación y Pruebas

### Arquitectura del Sistema

---

Autores:

- Arnolfo Emanuel
- Landaeta Ezequiel
- Montero Felipe
- Olocco Laureano

Versión del documento: 1.4

# Historial de versiones

| Versión | Fecha | Cambios |
|---|---|---|
| 1.1 | 2/06/2022 | Arquitectura y diseño del sistema |
| 1.2 | 9/06/2022 | Patrones de diseño |
| 1.3 | 11/06/2022 | Pruebas unitarias y pruebas de integración |
| 1.4 | 12/06/2022 | Actualización de matriz de trazabilidad |

# Arquitectura del sistema

El sistema fue estructurado como una máquina de estados finitos (FSM, por sus siglas en inglés). La misma cuenta con dos estados principales (Menu y Game) en los cuales se desarrolla toda la funcionalidad del sistema. Estos estados se conectan mediante transiciones que se invocan llamando al método standBy(). En la Figura N° 1 se muestra un diagrama de estados que representa dicha arquitectura del sistema.

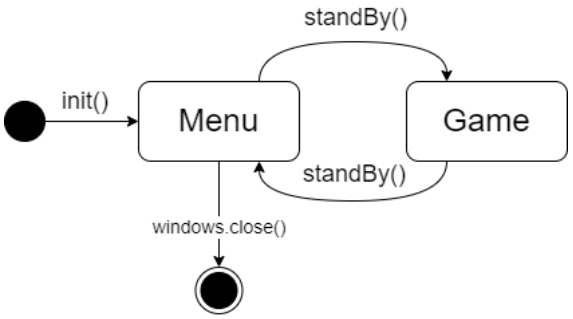


**Figura N° 1. Diagrama de estados que representa la arquitectura del sistema.**

El usuario interacciona con ambos estados, a través de entradas por teclado. Una representación de alto nivel que diagrama la interacción entre el usuario y los componentes del sistema se muestra en el siguiente diagrama de casos de uso (Figura N° 2).

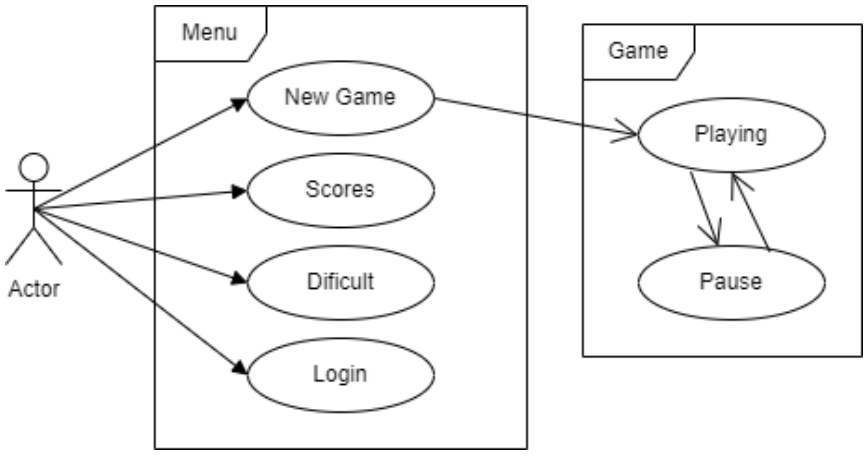


**Figura N° 2. Diagrama de estados que representa la arquitectura del sistema.**

El diseño de arquitectura como máquina de estados se realizó teniendo en cuenta la modularización y desempeño del sistema. Cada estado tiene una funcionalidad definida y es independiente del otro estado. Los estados no se comunican directamente entre sí, sino que interaccionan con la máquina de estados (GameController) a un nivel superior. Esto da origen a un bajo acoplamiento entre los estados. El sistema, por otra parte, tiene una secuencia de ejecución fija y continua, no sujeta a condiciones. Solo cambia el estado hacia el que se refiere. Este principio intenta cumplir con una rápida ejecución e inicio del sistema,

debido a que el inicio y cambio entre vistas del sistema (despliegue de la ventana en el estado Menu y Game) está sujeto simplemente al estado al que hace referencia el controlador. El funcionamiento concreto de la máquina de estados se explica con más detalle en la sección de patrones de diseño.

El sistema está diseñado como un juego de un jugador (single player) para computadoras de escritorio o notebooks, como única pieza de hardware. No utiliza conexión a internet, no interacciona con bases de datos u otras computadoras. El sistema crea dos archivos de texto, scores.txt y subscribers.txt utilizados para almacenar los 5 máximos puntajes y la lista de usuarios a notificar, respectivamente. Esto queda representado en el siguiente diagrama de despliegue (Figura N° 3).

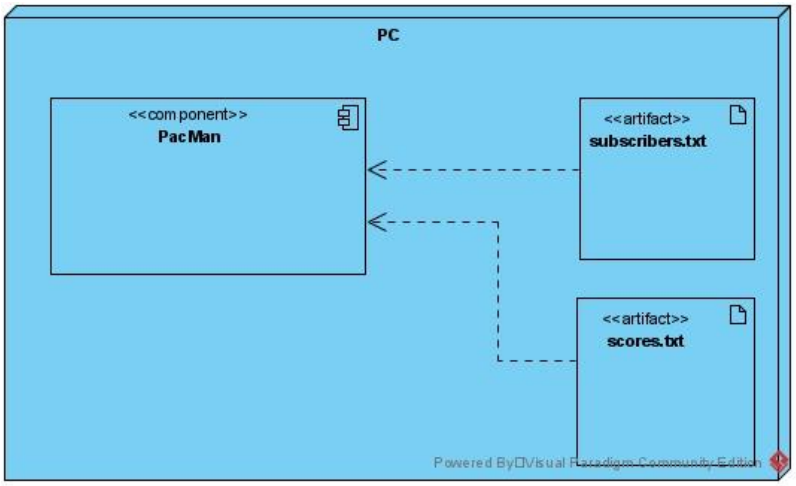


**Figura N° 3. Diagrama de despliegue del sistema.**

Los principales componentes del sistema y las interfaces de interacción de las mismas se muestran en la Figura N° 4.

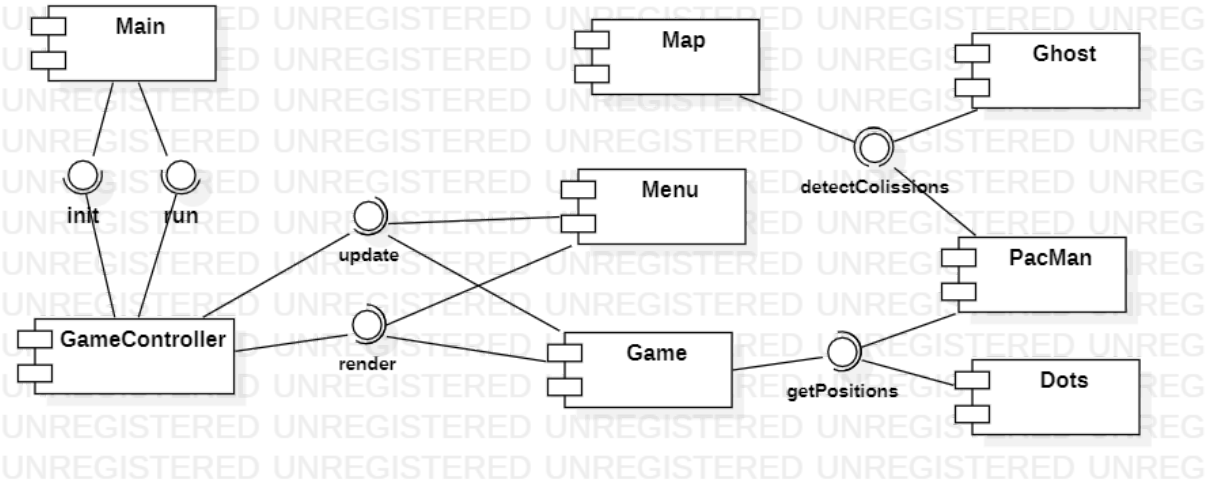


**Figura N° 4. Diagrama de componentes del sistema.**

# Diseño del sistema

La representación del diseño del sistema se realizó mediante diagramas de clase, de secuencias y de paquetes. Debido al tamaño de los diagramas de clase y secuencias, estos se presentan como imágenes a continuación.

- Sequence Diagram.jpg
- Sequence Diagram Menu.jpg
- Class Diagram.jpeg

Con respecto a los diagramas de secuencias, se presenta uno que modela la interacción entre el usuario y el menú de inicio y otro que modela una ejecución del juego. Además se presenta un diagrama de paquetes, que muestra la organización de directorios del proyecto. El mismo se presenta en la Figura N° 5.

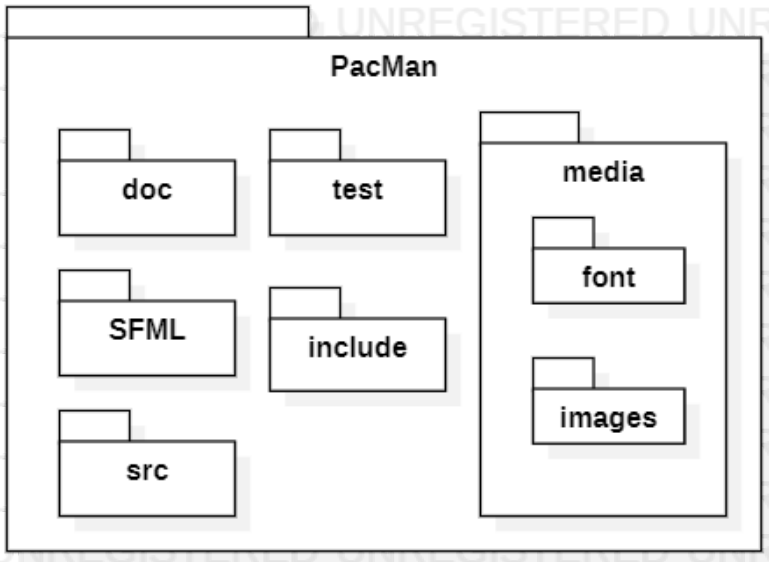


**Figura N° 5. Diagrama de paquetes. Muestra la organización de directorios del proyecto.**

## Patrones de Diseño

En el proyecto se llevó a cabo la implementación de cuatro patrones de diseño: State, para estructurar el sistema como máquina de estados; Singleton, para asegurar la creación de una única instancia de los objetos principales del sistema; Strategy, para modularizar los diferentes comportamientos de la clase Ghost y Observer, para notificar a los usuarios logueados sobre su estado de puntuación y facilitar una vista alternativa al rango de puntajes. A continuación se da una descripción detallada sobre la implementación de cada patrón.

## State

Como se observa en la Figura N° 1, el sistema fue desarrollado como una máquina de estados finitos. Esto se logró mediante la implementación de cuatro clases: GameController, State, Game y Menu. Un diagrama de clases de estas se muestra en la Figura N° 6.

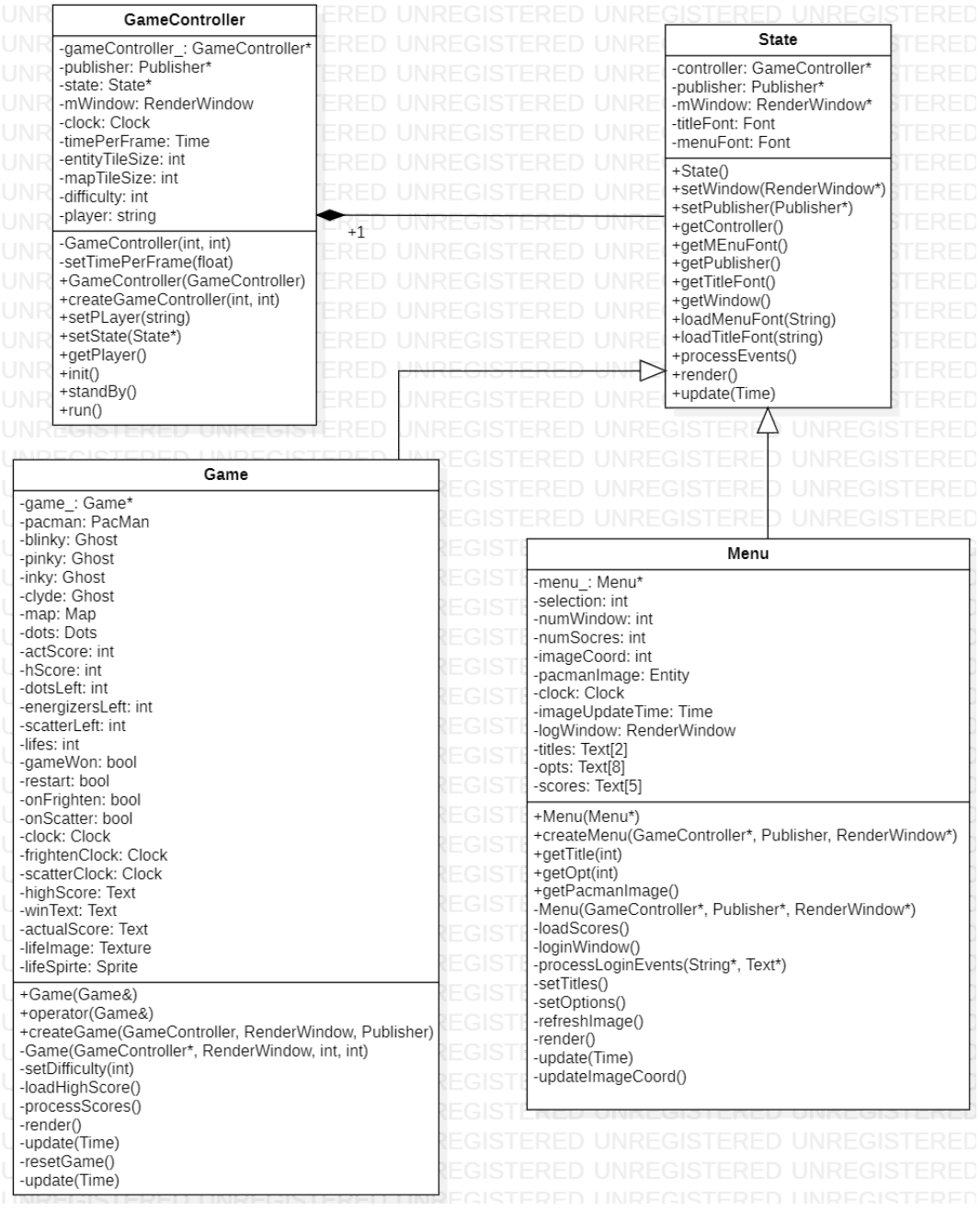


**Figura N° 6. Diagrama de clases del patrón State.**

GameController representa la máquina de estados en sí misma. Es el motor que ejecuta los métodos principales de cada estado y mantiene una selección del estado actual en ejecución. También almacena variables que sirven de comunicación entre los estados y configuración de los mismos. El estado actual lo almacena en un puntero (state) a un objeto

de tipo State. Sus métodos principales son run() y standBy(). El método run() es un bucle que ejecuta constantemente la mecánica del estado al que apunta la variable state. Una vez iniciado este método, el mismo se ejecuta sin parar hasta cerrar la ventana, es decir hasta finalizar el programa. El método standBy() cambia el objeto hacia el que apunta la variable state, cambiando así el estado actual. Una vez ejecutado el método standBy(), los métodos ejecutados dentro de run() serán los del otro estado. En las Figuras N° 7 y N° 8 se muestra la implementación de los métodos run() y standBy(), respectivamente.

```
void GameController::run()
{
    sf::Time timeSinceLastUpdate = sf::Time::Zero;
    while (mWindow.isOpen())
    {
        state->processEvents();
        timeSinceLastUpdate += clock.restart();
        while (timeSinceLastUpdate > timePerFrame)
        {
            timeSinceLastUpdate -= timePerFrame;
            state->processEvents();
            state->update(timePerFrame);
        }
        state->render();
    }
}
```

**Figura N° 7. Implementación del método run().**

```
void GameController::standBy()
{
    if (typeid(Menu).hash_code() == typeid(*state).hash_code())
    {
        setTimePerFrame(1.f);
        state = Game::createGame(this, publisher, &mWindow, entityTileSize, mapTileSize, difficulty);
    }
    else if (typeid(Game).hash_code() == typeid(*state).hash_code())
    {
        setTimePerFrame(5.f);
        state = Menu::createMenu(this, publisher, &mWindow);
    }
}
```

**Figura N° 8. Implementación del método standBy().**

La clase State sirve de clase base, de la cual Game y Menu son clases derivadas. Contiene las definiciones de los métodos y variables que son comunes entre ambas clases. Los métodos principales de estas clases son update() y render(). En el método update() se actualizan las variables, objetos (imágenes de los mismos, posiciones, etc.) del estado actual y se reciben las entradas por teclado. En el método render() se renderizan estos cambios en la ventana.

Es importante remarcar que el GameController (el motor) ejecuta constantemente los métodos update() y render(), a lo largo de toda la ejecución del programa. Es dentro de los estados Game y Menu que se invoca al método standBy(), para determinar efectivamente a qué estado pertenecen los métodos update() y render() en ejecución.

## Strategy

Cada fantasma (instancias de la clase Ghost) tiene tres formas diferentes de actuar: Chase, Scatter y Frightened. En modo Chase, los fantasmas actúan de acuerdo a su personalidad: Blinky, Pinky e Inky persiguen a PacMan y Clyde se mueve de forma aleatoria por el mapa. En modo Scatter cada fantasma intenta ir hacia una esquina del mapa. En modo Frightened los fantasmas se mueven de forma aleatoria, se tornan azules y pueden ser comidos por PacMan. Los fantasmas entran en modo Scatter 4 veces por partida, cada 20 segundos, por 4 segundos. Solo si PacMan “come” un “energizer” (una de las 4 bolitas grandes repartidas cerca de cada esquina del mapa), los fantasmas entran en modo Frightened, por 4 segundos.

La implementación de cada comportamiento se logró mediante la implementación de cuatro clases: Strategy, Chase, Scatter y Frightened. Un diagrama de clases de las mismas se presenta en la Figura N° 9.

La clase Strategy sirve de clase base para las otras tres. Todas las clases comparten los mismos métodos: act(), actBlinky(), actPinky(), actInky() y actClyde().

Cada fantasma posee un puntero a un objeto de tipo Strategy. El objeto hacia donde apunta dicho puntero, determina el modo de actuar del fantasma. El actuar de cada fantasma se lleva a cabo mediante la invocación de los métodos getStrategy()->act(). En las Figuras N° 10 y N° 11 se muestra la implementación de los métodos setStrategy() y getStrategy(), respectivamente. En la Figura N° 12 se muestran las líneas 348 a 351 del archivo Game.cpp, donde se hace actuar a cada fantasma.

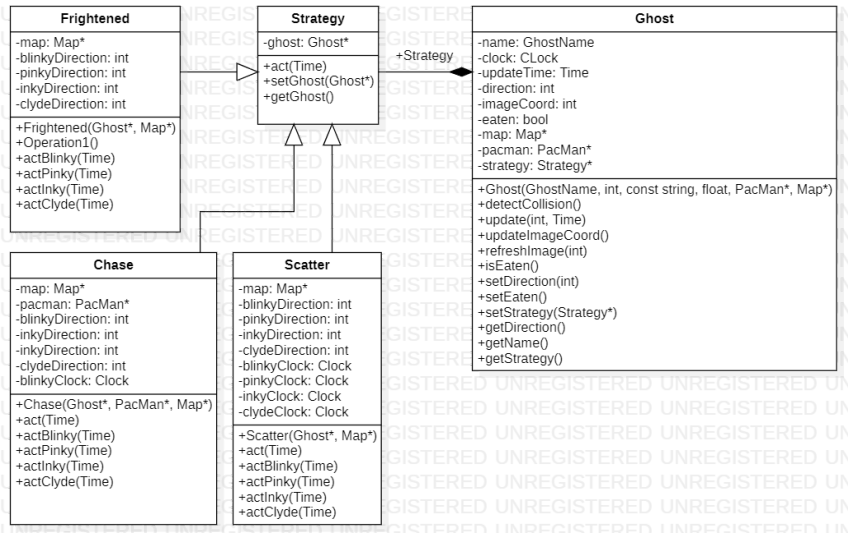


**Figura N° 9. Diagrama de clases del patrón Strategy.**

```
void Ghost::setStrategy(Strategy* s)
{
    delete(strategy);
    strategy = s;
}
```

**Figura N° 10. Implementación del método setStrategy().**

```
Strategy* Ghost::getStrategy()
{
    return strategy;
}
```

**Figura N° 11. Implementación del método getStrategy().**

```
else
{
    blinky.getStrategy()->act(deltaTime);
    pinky.getStrategy()->act(deltaTime);
    clyde.getStrategy()->act(deltaTime);
    inky.getStrategy()->act(deltaTime);
}
```

**Figura N° 12. Invocación del método  getStrategy()->act() durante la ejecución del juego.**

## Singleton

Como se mencionó anteriormente, el patrón de diseño Singleton permite crear una sola instancia de la clase que implementa dicho patrón. Se hizo uso de este patrón en las clases GameController, Game, Menu y Publisher. La implementación se llevó a cabo definiendo como privado el constructor de cada clase Singleton. Se crearon un puntero de auto referencia y un método de creación público que invoca al constructor privado, en caso de no existir instancia de la clase. El método de creación setea el puntero al objeto creado y si se vuelve a invocar el método de creación, no se invoca al constructor, simplemente se devuelve el puntero de auto referencia. En la Figura N° 13 muestra, como ejemplo, los métodos y atributos de la clase GameController. Se puede observar el puntero un objeto de la misma clase y el método createGameController(). En la Figura N° 14 se muestra la implementación del método createPublisher() de la clase Publisher.

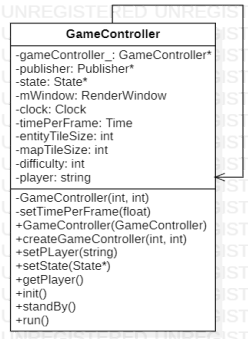


**Figura N° 13. Clase GameController implementada como Singleton.**

```
Publisher* Publisher::publisher_{ nullptr };

Publisher::Publisher()
{
    if (!menuFont.loadFromFile(filename: "../media/fonts/namco.ttf"))
        throw std::runtime_error("Failed to load menu font");
}

Publisher* Publisher::createPublisher()
{
    if (publisher_ == nullptr)
    {
        publisher_ = new Publisher();
    }
    return publisher_;
}
```

**Figura N° 14. Implementación del método createPublisher().**

En la Figura N° 14 se aprecia como la manera de crear un objeto Publisher es a través del método createPublisher() el cual crea una instancia sólo si no existe. En el caso de que ya exista una instancia de Publisher, y se llame al método createPublisher(), éste nos devolverá la instancia ya creada.

## Observer

Observer es un patrón de diseño que define una dependencia del tipo uno a muchos entre objetos, de manera que cuando uno de los objetos cambia su estado, notifica este cambio a todos los dependientes. En el proyecto fue implementado dicho patrón para enviar una notificación al usuario, cuando éste queda fuera del top 5 de puntajes. La implementación se realizó creando la clase Publisher. La instancia de esta clase crea un archivo de registro (subscribers.txt) donde guarda el nombre de los usuarios que quedan fuera de la lista de mejores puntuaciones. Cuando un usuario se “loguea” con algún nombre que esté en la lista de subscribers, esté es notificado sobre el hecho de que no está más en la lista de mejores puntuaciones, y se le muestra el listado actual. En la Figura N° 15 se muestra el diagrama de clases para la clase Publisher. En la Figura N° 16 se muestra la notificación enviada a un usuario que se loguea luego de ser removido de la lista de mejores puntuaciones y en la Figura N° 17 se muestra la lista de puntuaciones a la que se accede desde el menu principal.

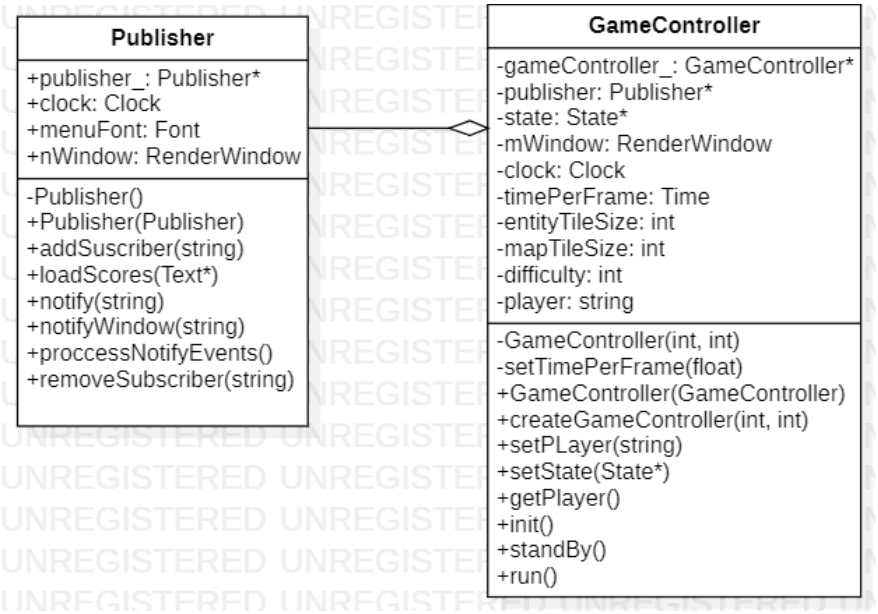


**Figura N° 15. Diagrama de clase de la clase Publisher.**

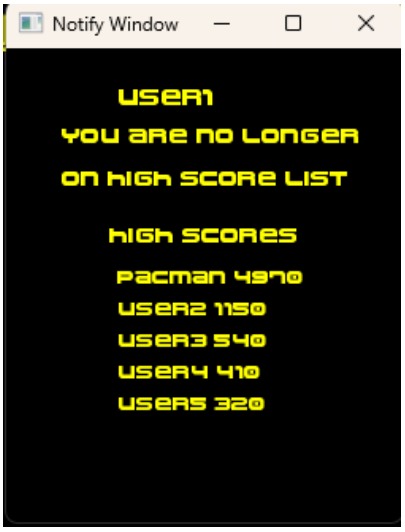


**Figura N° 16. Ventana de notificación generada por la clase Publisher.**


**Figura N° 17. Ventana de notificación generada por la clase Publisher.**

## Pruebas unitarias

Los tests unitarios se encargan de comprobar que funcionen los métodos de las distintas clases que forman parte del programa. Además de correrlos localmente, los mismos fueron automatizados con la herramienta de Integración Continua “CircleCI”. Esta herramienta fue integrada con la herramienta de control de versiones (GitHub), de forma tal que cada vez que se suben cambios al repositorio, CircleCI verifica que los cambios compilen y los test unitarios pasen las pruebas.

## Pruebas de integración

Las pruebas de sistema comprueban la integración del sistema. Para esto se prueban los requerimientos más importantes del software. Las pruebas de integración T01 y T02 se pueden automatizar con la herramienta CircleCi. para esto se inicia el juego y se verifica la correcta ejecución. Si esto no es así, el test devuelve una interrupción y notificaa una falla en la integración del sistema.

**TI01**
**Descripción**: Iniciación de un nuevo juego

**Función a testear:** Se probara que se pueda iniciar el juego , una partida y que se cargue la ventana con el mapa, el PacMan, los fantasmas y los Dots.

**Ejecución del test (pasos):**

1. Iniciar el juego
2. Iniciar una nueva partida
3. Cargar el mapa , con todos sus elementos correspondientes (pacman , fantasmas , pelotitas).

**Resultado esperado:** Correcta iniciación del juego y la partida con todos sus elementos.

**TI02**

**Descripción:** Se probará el movimiento y las colisiones de los personajes dentro del mapa principal del juego

**Función a testear:** integración de los personajes y el mapa.

**Ejecución del test (pasos):**

1. Iniciar el juego
2. Iniciar una nueva partida
3. Mover PacMan

4. Verificar cambio de posición en el PacMan
5. Verificar cambio de posición en los fantasmas.

**Resultado esperado:** Si el juego se inicia correctamente, el jugador podrá mover el PacMan y se deberá observar el movimiento de los fantasmas.

**TI03**

**Descripción:** Se probará si se pierde una vida si el fantasma toca al PacMan

**Función a testear:** integración del pacman con los fantasmas.

**Ejecución del test (pasos):**
1. Iniciar el juego
2. Iniciar una nueva partida

**Resultado esperado:** Que el fantasma al comer al PacMan se pierda una vida y luego de esto se empiece en la posición inicial con el mismo estado del mapa antes de ser comido

## Pruebas unitarias

**UT1**
**Nombre:** GhostDirectionTest
**Descripción:** Se prueba que cualquier fantasma se pueda mover en todas las direcciones posibles

**UT2**
**Nombre:** GhostEatenTest
**Descripción:** Se prueba que se pueda comer un fantasma

**UT3**
**Nombre:** GhostImageTest
**Descripción:** Se prueba que se visualicen los fantasmas

**UT4**
**Nombre:** GhostMoveTest
**Descripción:** Se prueba el movimiento de los fantasmas en todas las direcciones posibles.

**UT5**
**Nombre:** GhostWallTest
**Descripción:** Se prueba que cualquier fantasma no traspase una pared
**UT6**
**Nombre:** MapImageTest

**Descripción:** Se prueba que el mapa se visualice

**UT7**
**Nombre:** MenuImageTest
**Descripción:** Se prueba que visualice el menú

**UT8**
**Nombre:** PacManDirectionTest
**Descripción:**

**UT9**
**Nombre:** PacManGhostCollisionTest
**Descripción:** Se prueba que haya contacto entre el PacMan y los distintos fantasmas

**UT10**
**Nombre:** PacManImageTest
**Descripción:** Se prueba que el PacMan se visualice en la pantalla

**UT11**
**Nombre:** PacManMoveTest
**Descripción:** Se prueba que el PacMan se mueva en todas las direcciones posibles.

**UT12**
**Nombre:** PacManSpriteTest
**Descripción:** Se prueba el metodo setSprite

**UT13**
**Nombre:** PacManWallTest
**Descripción:** Se prueba que el personaje no traspase una pared

**UT14**
**Nombre:** ScoretTest
**Descripción:** Se prueba que se impriman los puntajes de los jugadores

**UT15**
**Nombre:** SetDificultyTest
**Descripción:** Se probará el cambio de dificultad dentro del menú de configuraciones del juego.

**UT16**
**Nombre:** StrategyTest
**Descripción:** Se prueba el cambio de estrategia del fantasma

| | CU 1 | CU 2 | CU 3 | CU 4 | CU 5 | CU 6 | CU 7 | CU 8 | CU 9 | CP1 | CP2 | CP3 | CP4 | CP5 | CP6 | CP7 | CP8 | CP9 | CP 10 | CP 11 | CP 12 | CP 13 | CP 14 | CP 15 | CP 16 | CP 17 | CP 18 | CP 19 | CP 20 | CP 21 | CP 22 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Req 1 | UT 7 | | | | | | | | | UT 7 | | | | | | UT 7 | UT 7 | | | | | | | | | | | | | | |
| Req 2 | | | | | | | | | | TI 2 | TI 2 | | TI 2 | | | | | | | TI 2 | | | | | | | | | | | |
| Req 3 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Req 4 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Req 5 | | | | | | UT 14 | | | | | | | | | | UT 14 | | | | | | | | | | | | UT 14 | | | |
| Req 6 | | | | | | | | | | UT 11 | UT 11 | | | | | | | | | | | | | | | | | | | | |
| Req 7 | | | | | | UT 14 | | | | | | | | | | | | | | | | | | | | | UT 14 | | | | |
| Req 8 | | | | | | | | | | | | | | | | | | | | UT 2 | UT 9 | | | | | | | | | | |
| Req 9 | | | | | | | | | | | UT 11 | | | | | | | | | | | UT 4 | | | | | | | UT 11 | | |

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Req 10 | | | | | | | | | | | | | UT 13 | | | | | | | | | | UT 5 | | | | | | | | |
| Req 11 | | | | | | | | | | | | | | | | | | | | UT 9 | | | | | | | | | | |
| Req 12 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Req 13 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Req 14 | | | | | | | | | | | | | | | | | | | | UT 6 | | | | | | | | | | |
| Req 15 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Req NF 1 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Req NF 2 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Req NF 3 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Req NF 4 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

## Pruebas adicionales

Además de las pruebas antes mencionadas, todo el código generado se analizó con la herramienta de análisis de código estático cppCheck. Se verificó que todo el código no presentase errores anunciados por dicha herramienta. En la Figura N° 18 se presenta un caso de uso de la herramienta.

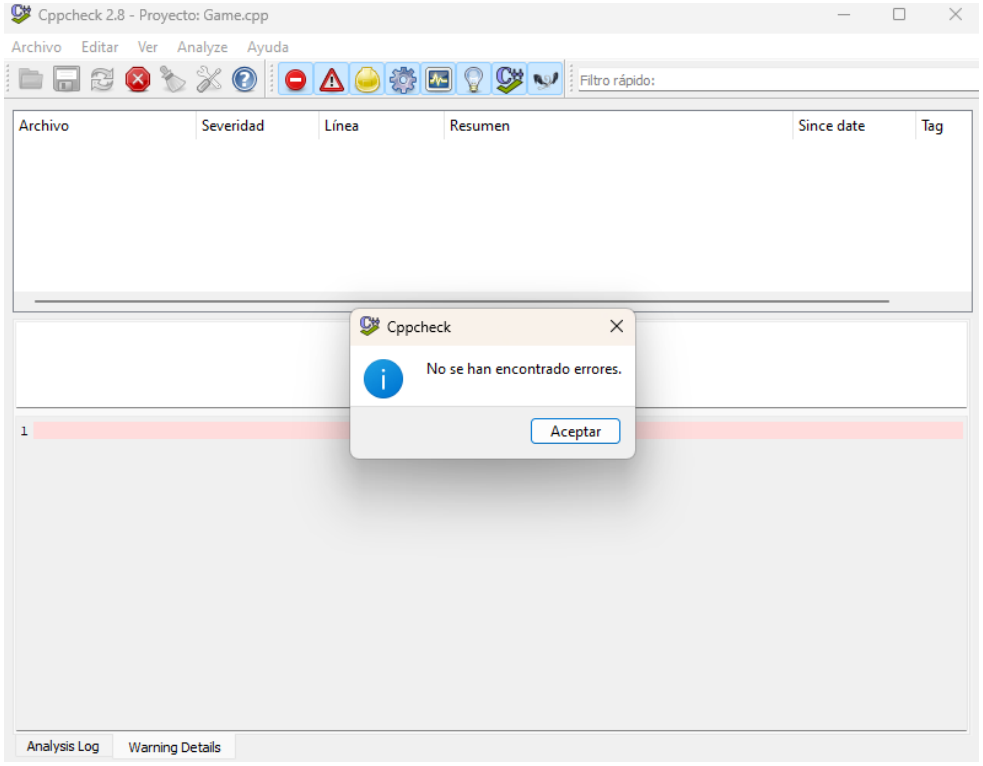


**Figura N° 18. Caso de uso de cppCheck.**