**Reporte del Sprint #5**

Las principales tareas de esta asignación son:

1. Agrega la función de grabar (record) un juego en un archivo de texto. Se requiere la historia de usuario y los criterios de aceptación tanto de grabación como de reproducción
2. Realización de un ejercicio de revisión de código.
3. Resumir las lecciones aprendidas del Sprint 0 al Sprint 5.

El siguiente es un diseño de GUI de muestra del producto final, donde "Replay" es opcional.

**El trabajo es de carácter individual.**

|  |  |  |
| --- | --- | --- |
| SOS Icon  Description automatically generated Simple game Icon  Description automatically generated General game Board size  8 | | |
| Blue player  Icon                                Description automatically generated Human  Icon  Description automatically generated S  Icon  Description automatically generated O  Icon                                Description automatically generated Computer | Chart, line chart  Description automatically generated | Red player  Icon  Description automatically generated Human  Icon  Description automatically generated S  Icon  Description automatically generated O  Icon  Description automatically generated Computer  Replay |
| ☒ Record game | Current turn: blue (or red) | New Game |

Figura 1. Sample GUI layout of the final product Diseño de GUI del producto final

**Puntos totales**

1. **Demostración (10 puntos)**

Envía un video de no más de 15 minutos, demostrando claramente que has implementado todas las funciones en la siguiente tabla. En el video, debes explicar lo que se está demostrando. **Presenta el diagrama de clases de tu código de producción y describe cómo la jerarquía de clases en su diseño trata con los requisitos del oponente de la computadora.**

|  |  |
| --- | --- |
|  | **Feature** |
| 1 | Se graba un juego simple completo de dos jugadores humanos. |
| 2 | Se graba un juego general completo de dos jugadores humanos |
| 3 | Se graba un juego simple completo de jugadores humano-computadora |
| 4 | Se graba un juego general completo de jugadores humano-computadora |
| 5 | Se graba un juego simple completo de jugadores computadora-computadora |
| 6 | Se graba un juego general completo de jugadores computadora-computadora |

Si has implementado la función de "replay" para obtener crédito adicional, debes incluir tu demostración en el video.

1. **Historias de usuario y criterios de aceptación para los requisitos para los requerimientos Record/Replay (1 punto)**

**Plantilla de historia de usuario**: Como <rol>, quiero <objetivo> [tal que <beneficio>]

Agrega o elimina filas si es necesario

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **Nombre de historia de usuario** | **Descripción de historia de usuario** | **Prioridad** | **Esfuerzo estimado (horas)** |
| 20 | Grabar Juego | Como jugador  Quiero tener la opción de grabar una partida de SOS Para poder ver qué movimientos se realizaron paso a paso en el tablero | ALTA | 2 |
| 21 | Reproducir Juego Grabado | Como jugador  Quiero reproducir una partida de SOS grabada  Para ver paso a paso cómo se jugó esa partida | MEDIA | 5 |

|  |  |  |  |
| --- | --- | --- | --- |
| **ID y nombre de la historia de usuario** | **AC**  **ID** | **Descripción del criterio de aceptación** | **Estado (completado, por hacer, en progreso)** |
| Historia 20 | 20.1 | AC 20.1 <Grabar Juego Simple Humano vs Humano>  Dado un juego simple de Humano vs Humano  Cuando esté marcada la opción de grabar y acabe el juego  Entonces se crea un archivo de texto que contiene las jugadas realizadas paso a paso. | COMPLETADO |
| 20.2 | AC 20.2 <Grabar Juego Simple Computador vs Computador>  Dado un juego simple de Computador vs Computador  Cuando esté marcada la opción de grabar y acabe el juego  Entonces se crea un archivo de texto que contiene las jugadas realizadas paso a paso. | COMPLETADO |
| 20.3 | AC 20.3 <Grabar Juego General Humano vs Humano>  Dado un juego general de Humano vs Humano  Cuando esté marcada la opción de grabar y acabe el juego  Entonces se crea un archivo de texto que contiene las jugadas realizadas paso a paso junto con el puntaje de cada jugador. | COMPLETADO |
| 20.4 | AC 20.4 <Grabar Juego General Computador vs Computador>  Dado un juego general de Computador vs Computador  Cuando esté marcada la opción de grabar y acabe el juego  Entonces se crea un archivo de texto que contiene las jugadas realizadas paso a paso junto con el puntaje de cada jugador. | COMPLETADO |
| Historia 21 | 21.1 | AC 21.1 <Reproducir Juego Simple Grabado>  Dado un archivo de texto que contenga un juego simple grabado  Cuando se reproduzca dicho archivo  Entonces se muestran los movimientos paso a paso en pantalla | POR HACER |
| 21.2 | AC 21.2 <Reproducir Juego General Grabado>  Dado un archivo de texto que contenga un juego simple grabado  Cuando se reproduzca dicho archivo  Entonces se muestran los movimientos paso a paso en pantalla | POR HACER |

**3. Revisión de código (4 puntos)**

Aplica la revisión del código fuente a una o dos de las clases más importantes (y a otras clases si el tiempo te permite) e informa de los resultados. Además de buscar errores, la revisión debe verificar: (1) si todo el proyecto ha seguido el estándar de codificación de manera consistente, (2) si el proyecto ha seguido los principios de diseño presentados en clase y (3) si hay olores de código que indican la necesidad de refactorización.

Las siguientes listas de verificación proporcionan pautas básicas. Puedes agregar nuevos elementos a cada una de las listas de verificación. Asegúrate de que tus respuestas sean el resultado del ejercicio de revisión del código. Si no hay hallazgos para una entrada, debes proporcionar una explicación. Por ejemplo, si tu respuesta a "¿Se violan las convenciones de nomenclatura?" es no, debes describir una convención de nomenclatura y presentar un ejemplo. No recibirás puntaje por si tus respuestas son simplemente sí o no sin información adicional.

Clases que han sido revisadas:

Fecha/hora de duración del ejercicio de revisión del código:

Revisaré las clases Tablero, Juego y Controller

|  |  |  |  |
| --- | --- | --- | --- |
| **Checklist** | **Items Checklist** | **Conclusiones** | |
| Estándares de codificación | Convenciones de nombres | Desde el inicio del proyecto se han seguido las convenciones, como podría ser escribir los métodos en Pascal Case. Como ejemplo: public void PrintGame(). También camel Casing, como ejemplo: public Juego(Tablero tablero). Sin embargo, se nos pasó las convenciones de no usar mayúsculas para constantes o variables de solo lectura, y la convención de no usar underlines antes de una variable sin asignar, sin embargo se hizo para evitar una mala admistración de variables: public override void MakeMove(int row, int column, Tablero.Cell \_ficha) por public override void MakeMove(int row, int column, Tablero.Cell ficha). Además, una pequeña omisión fue que el nombre del proyecto sea ReSOSgame en vez de ReSOSGame. | |
| Convención de ordenación de argumentos de método | C# no cuenta con esta convención per se, pero hemos ordenado de forma consistente los métodos. De esta forma el ejecutar los métodos ya sea dentro del código en las pruebas no se hizo confuso ni tedioso. | |
| Comentarios significativos y válidos. | A lo largo del código se logró comentar gran parte del código, los métodos, atributos y clases se han intentado hacer lo más autoexplicativos posibles. | |
| Estilo consistente de bloques de código | Los atributos y propiedades se han puesto en la parte de arriba de los bloques de código y luego los métodos. Hubo omisiones que ya se han corregido. El orden es: Atributos, Propiedades, Métodos. | |
| Indentación consistente | En aras de lograr un código ordenado y comprensible se ha mantenido una correcta identación. | |
| Typos | Hay muchos typos señalados por el IDE, sin embargo, estos se deben a que hay variables en español y los comentarios están en español también, por lo que después de revisar que no se tratan de violaciones de nomenclatura, se ha puesto como comentario. | |
| Principio de diseño | Clase o método no bien modularizado | En un principio la clase controller se encargaba no solo del juego, sino que también de su grabación. (Hace unos minutos). Pero ahora existe la clase GameRecorder que se encarga de la grabación del juego. | |
| Visibilidad adecuada de cada variable, método y clase. | Había variables y métodos cuya visibilidad no era la adecuada, muchas estaban en public cuando bien podrían estar en private o en el caso de muchas Propiedades, en protected para mantenerlo en casos de herencia y dentro de la misma clase. Con cada sprint se ha mantenido constante la omisión de buena encapsulación, sin embargo, en esta revisión de código he cambiado muchas de estas visibilidades, principalmente en las propiedades. | |
| Alguna clase con pobre abstracción | Se ha tenido muy en cuenta la abstracción para sacarle el mayor provecho al polimorfismo. Sin embargo, hay métodos o atributos que hacen que las clases estén estrechamente ligadas, como es el caso de Tablero y Juego, pues juego necesita de un tablero para poder trabajar. Quizá haya otro enfoque para trabajar esto. | |
| Diseño por contrato (pre/postcondiciones) | De forma implícita se ha implementado un enfoque de diseño por contrato. En este caso los if actúan como precondiciones que se aseguran de que las acciones que vienen debajo solo se realicen si se cumplen ciertas condiciones. Además en los métodos MakeMove y FinalGameState de la clase Juego, en cada una de sus clases hijas están de forma explícita las pre y post condiciones | |
| ¿Se viola el Principio Abierto-Cerrado? | En este caso no, tal como pide este principio, las clases cumplen con estar abiertas a adicionar/quitar funcionalidades, pero cerradas a la modificación de la clase per se. Esto se ha logrado gracias a un buen seguimiento del SRP y de la modularización de los métodos de clase. | |
| ¿Se viola el Principio de Responsabilidad Única [[1]](#footnote-1)? | En un principio sí, de forma muy evidente. La clase controller se encargaba de manejar el juego y además de grabar la partida, como ya mencioné antes, esta función se separó a la clase GameRecorder. | |
| Smells código | Números mágicos | En los tests se iniciaba muchos tableros con números que seleccionamos, se cambió por una constante PreferredSize en esos casos. | |
| Variable global /clase innecesaria | Las clases tienen propósitos bien definidos, además todas las variables están siendo utilizadas. | |
| Código duplicado | Se intentó, en la medida de lo posible, evitar la duplicación de código. Sin embargo, habían getters y propiedades que realizaban casi lo mismo, lo que hice fue añadirle más lógica a las Propiedades para descartar los getters. Por ejemplo se reemplazó GetCell(row,col) con una propiedad de indexación para poder usarlo de forma Tablero[row,col].  También la clase GameRecorder y Consola tenían gran similitud así que con ayuda de la clase TableroHelper se logró reducir esa duplicación de código. | |
| Métodos largos | Al centrarnos en que los métodos y clases estén modularizados, no hay presencia de métodos largos. | |
| Larga lista de parámetros | El método que pide más parámetros es MakeMove de la clase player, siendo 4 parámetros. Por lo que consideramos que los métodos en general no piden una larga lista de parámetros. | |
| Expresión demasiado compleja | No se evidencian expresiones muy complejas, están bien estructuradas, casi no hay acoplamiento y tienen buena cohesión. | |
| Switch o if-then-else que necesita ser reemplazado con polimorfismo | No, se tuvo desde el primer momento en mente el uso de polimorfismo para evitar ese tipo de situaciones. | |
| Nombre de método o variable cuya intención no está clara | El código se encuentra comentado en su mayoría, salvo algunos métodos, Propiedades o atributos self-explanatory (su función está en el nombre) que quizá se me habrá pasado. | |
| ¿Algún método similar en otras clases? | Si bien el MakeMove de la clase Player se parece a MakeMove de Juego, se ocupan de responsabilidades diferentes. Al punto que se podría decir que solo se parecen en el nombre. Me explico, mientras que en Juego se realiza un movimiento en el tablero independientemente del tipo de Jugador, en Player importa qué jugador y qué tipo de jugador está intentando hacer el movimiento. | |
| If anidados | A lo largo del código anidamos muchos if porque era la solución de ese momento, ahora se ha reemplazado para que, en vez de hacer comprobaciones largas, si no cumple ciertas condiciones el método hará un return. | |
| **Errores:** | **Fragmento de código con errores** | **¿Cuál es el error?** | **¿Por qué es un error?** |
| public bool HasOnePoint(int row, int col, Cell ficha)  {  if (ficha == Cell.S)  {  return (tablero.GetCell(row + 1, col + 1) == Cell.O && tablero.GetCell(row + 2, col + 2) == Cell.S) ||  (tablero.GetCell(row - 1, col - 1) == Cell.O && tablero.GetCell(row - 2, col - 2) == Cell.S) ||  (tablero.GetCell(row, col + 1) == Cell.O && tablero.GetCell(row, col + 2) == Cell.S) ||  (tablero.GetCell(row, col - 1) == Cell.O && tablero.GetCell(row, col - 2) == Cell.S) ||  (tablero.GetCell(row + 1, col) == Cell.O && tablero.GetCell(row + 2, col) == Cell.S) ||  (tablero.GetCell(row - 1, col) == Cell.O && tablero.GetCell(row - 2, col) == Cell.S) ;  }  else if (ficha == Cell.O)  {  return (tablero.GetCell(row - 1, col - 1) == Cell.S && tablero.GetCell(row + 1, col + 1) == Cell.S) ||  (tablero.GetCell(row + 1, col - 1) == Cell.S && tablero.GetCell(row - 1, col + 1) == Cell.S) ||  (tablero.GetCell(row - 1, col) == Cell.S && tablero.GetCell(row + 1, col) == Cell.S) ||  (tablero.GetCell(row, col - 1) == Cell.S && tablero.GetCell(row, col + 1) == Cell.S);  }  return false;  } | No tiene todas las formas de marcar SOS | No nos da la puntuación adecuada |
| public override void MakeMove(int row, int column, Cell \_ficha)  {  if (!validator.GameOver()) // Si es el estado de juego es JUGANDO  {  if (row >= 0 && row < tablero.Tamanio && column >= 0 && column < tablero.Tamanio  && tablero.GetCell(row, column) == 0)  {  tablero.ValidMove = true;  tablero.Ficha = \_ficha;  tablero.Grid[row, column] = \_ficha;  if (validator.HasOnePoint(row, column, tablero.Ficha) || validator.FullBoard()) // Cuando se ha hecho ganado 1 punto  {  FinalGameState();  }  if (tablero.EstadoDeJuego == GameState.JUGANDO)  {  tablero.Turno = (tablero.Turno == Jugador.JAZUL) ? Jugador.JROJO : Jugador.JAZUL;  }  }  else  {  tablero.ValidMove = false;  }  }  else  {  tablero.ValidMove = false;  }  } | if (validator.HasOnePoint(row, column, tablero.Ficha) || validator.FullBoard()) // Cuando se ha hecho ganado 1 punto  {  FinalGameState();  } | Pues cuando se obtiene un punto en la jugada que también rellena por completo el tablero da empate en vez de dar un ganador. |

**4. Resumen de todo el código (1 points)**

|  |  |  |
| --- | --- | --- |
| Nombre del archivo de código fuente | ¿Código de producción o prueba? | # lineas de código |
| Consola.cs | Producción | 21 |
| Controller.cs | Producción | 37 |
| GameRecorder.cs | Producción | 94 |
| Juego.cs | Producción | 199 |
| Player.cs | Producción | 85 |
| ScoreValidator.cs | Producción | 59 |
| Tablero.cs | Producción | 54 |
| Form1.cs | Producción | 311 |
| TableroHelper.cs | Producción | 38 |
| UnitTest1.cs | Prueba | 519 |
| Total de líneas de código | | 1418 |

**No recibirás puntaje por esta tarea a menos que envíes tu código fuente completo.**

5. Resume las lecciones aprendidas de todo el proyecto respondiendo las siguientes preguntas desde la perspectiva de los procesos de desarrollo, codificación, diseño, refactorización y prueba (**4 puntos**):

* ¿Qué ganaste personalmente con el proyecto?
* ¿Qué hace bien tu proyecto y qué podría hacer mejor tu proyecto?
* ¿Cómo podrías mejorar tu proceso de desarrollo si desarrollas un juego similar desde cero?

Requisito mínimo para (5): Una página completa a espacio simple, tamaño de fuente no mayor a 12 puntos.

**¿Qué ganaste personalmente con el proyecto?**

Principalmente el conocer mejor el estilo de trabajo de mi compañero de grupo, además de aprender a ceder en ciertas situaciones lo que implica ser más flexible: lo cual sirve mucho para dejar mi ansia de tener el control. Lo que lleva a aprender a confiar en mi partner, pues no puedo hacerlo todo solo, incluso si no fuese por eso muchas soluciones no se hubiesen dado en primer lugar. Sin embargo, lo anterior también me ayudó a ser más organizado y responsable, también a pedir ayuda.

Algo importante que destacar es que aprendí a usar más herramientas tales como el code coverage, git, markdown; y también a aprovechar mis conocimientos de programación, pues si bien conocía superficialmente C# y fue en ese lenguaje que aprendí a cómo hacer una GUI, aún tenía mucho camino por recorrer lo que me llevó a aprender sobre la marcha mucho más: cosas tales como las Properties, conocer más sobre el entorno de forms, etc. Además, el adaptar el código del proyecto TicTacToe en Java al nuestro me ayudó a comprender mejor ambos lenguajes y tenerle menos tirria a Java.

Sobre el TDD a tener una mejor comprensión de cómo desarrollar software, ya no desarrollo teniendo en mente que funcione perfecto a la primera, sino que pienso en ir mejorando progresivamente el código.

También he ido aplicando lo que he estado aprendiendo tanto en el curso como en el de diseño de algoritmos. Sobre lo primero he estado aplicando técnicas de refactorización lo que me ha ayudado a entender mi código, pues había veces en las que regresaba a él y no entendía nada si no fuese porque estaba comentado y las variables me explicaban qué hacen también.

También conocer y entender la importancia de tener un control de versiones, me ha salvado de varios fallos que sin un rollback no podría haber solucionado. Aunque también me enseñó a tener cuidado con los stash, push y pulls que hacía, y sí, aprendí por la mala (Recuerdos del Sprint 4).

Si bien en ing de software aprendí la importancia de capturar requerimientos, las historias de usuario y criterios de aceptación me ayudaron a entender cómo aplicarlas para crear código funcional y con propósito.

Que modularizar las clases y métodos hace más fácil la implementación de más funciones y actores, sin ello este Sprint 5 no hubiese sido tan sencillo.

Y puedo rescatar que, aunque medio obligué a Gabriel a que hagamos el proyecto en un lenguaje desconocido para él, supe guiarlo en sus primeros pasos y él mismo se dio cuenta de lo familiar que es y fue aprendiendo más por su cuenta. De hecho, le agradezco completamente el Sprint4.5 que fue donde logró conectar la lógica que hicimos a la interfaz gráfica lo que dejó que este sprint no sea tan trabajoso como lo tenía previsto desde donde dejamos el Sprint 4. Y también que me ayudó a tener la costumbre de comentar mi código porque en ese momento sé qué hace, pero días después puede que ya no.

**¿Qué hace bien tu proyecto y qué podría hacer mejor tu proyecto?**

Se puede jugar de forma funcional al SOS, tanto de forma simple como general, siendo los jugadores humanos o máquinas en cualquiera de sus 4 combinaciones.

En el juego general muestra puntajes, mientras que en el simple estos puntajes no se muestran pues solo se necesita un punto para ganar.

Además, está la opción de grabar el juego, independientemente del tipo y de los jugadores.

El proyecto evidencia buenas prácticas (o al menos la intención de ellas).

El código está modularizado y nos enfocamos en que siga el SRP.

Se hace uso del polimorfismo lo que ayuda a hacer un código más general y se apoya en el LSP para que funcione como debería.

Hay cosas que se pueden mejorar, como la implementación de un replay (estoy escribiendo esto el sábado y aún no está). También se podría implementar el dibujo de una línea para evidenciar un SOS (espero implementarlo antes del jueves, así que quizá lo implemente después de presentar).

Se podría implementar más pruebas, tales como las pruebas automatizadas de la GUI entre otras para lograr un 100% de code coverage.

Como último detalle, hacer más general el guardado de partidas, para que no haya conflictos al guardar en otro dispositivo que no sea el mío.

**¿Cómo podrías mejorar tu proceso de desarrollo si desarrollas un juego similar desde cero?**

Lo principal sería anotar en mi backlog los errores pendientes de resolución, para que no se acumulen o entorpezcan tanto el trabajo, ejemplo de esto fue que el error en saber si se hizo un SOS estuvo presente hasta el Sprint4 cuando empezó en el sprint3 y también un error que no cambiaba el estado de juego en un juego simple cuando la jugada ganadora también terminaba de rellenar el tablero desde el Sprint3 hasta este Sprint5.

Algo que también es muy importante es asegurarme de que capturé bien las historias de usuario para tener buenos criterios de aceptación y así seguir un TDD lo que facilita la arquitectura.

También podría darle más importancia a la GUI para que no se me escapen detalles como en este caso que desde el Sprint 2 no se puede hacer una línea que evidencia un SOS.

Sobre la arquitectura, el cambio de un diseño de cascada a uno más versátil como fue en este proyecto me ha ayudado a tener en mente que debo empezar el proyecto con una visión clara de lo que quiero lograr, para quién y para qué, de forma iterativa.

Junto a lo anterior, gracias a los principios SOLID y a la experiencia que adquirí, la modularidad de las clases podría empezar desde una fase más temprana.

Además, aprovechar más el uso de ReShaper para acelerar el refactor cuando se cambie de nombre, entre otras funcionalidades.

Como tengo una mejor comprensión del uso de Properties podría reducir la redundancia de código si uso C#.

Apuntar en orden los cambios pequeños que hago al momento de refactorizar por si se presenta un error ocasionado por uno de esos cambios para no tener que regresar mucho, digamos a un commit con mucha antigüedad o peor, a un Sprint anterior.

Apoyando lo anterior, un mejor uso del control de versiones gracias a la práctica que tengo evitará el tiempo perdido al tratar de arreglar una rama, repositorio o (intento de) commit.

Incluso podría aprovechar las pruebas por especificaciones para cubrir el mayor rango de cobertura posible.

Además, podría usar .NET.Xamarin que es multiplataforma para no limitar el uso de mi juego a Windows.

En cuanto a las pruebas y criterios de aceptación, no solo tomar pruebas esperando resultados correctos, sino también cuando se espera que el programa falle.

Para pruebas que dependen de valores aleatorios como es el caso de los movimientos por computadora, el uso de stubs y mocks haría una gran diferencia.

1. Revisa: [Violation solution for single responsibility principle](https://softwareengineering.stackexchange.com/questions/342051/violation-solution-for-single-responsibility-principle) [↑](#footnote-ref-1)