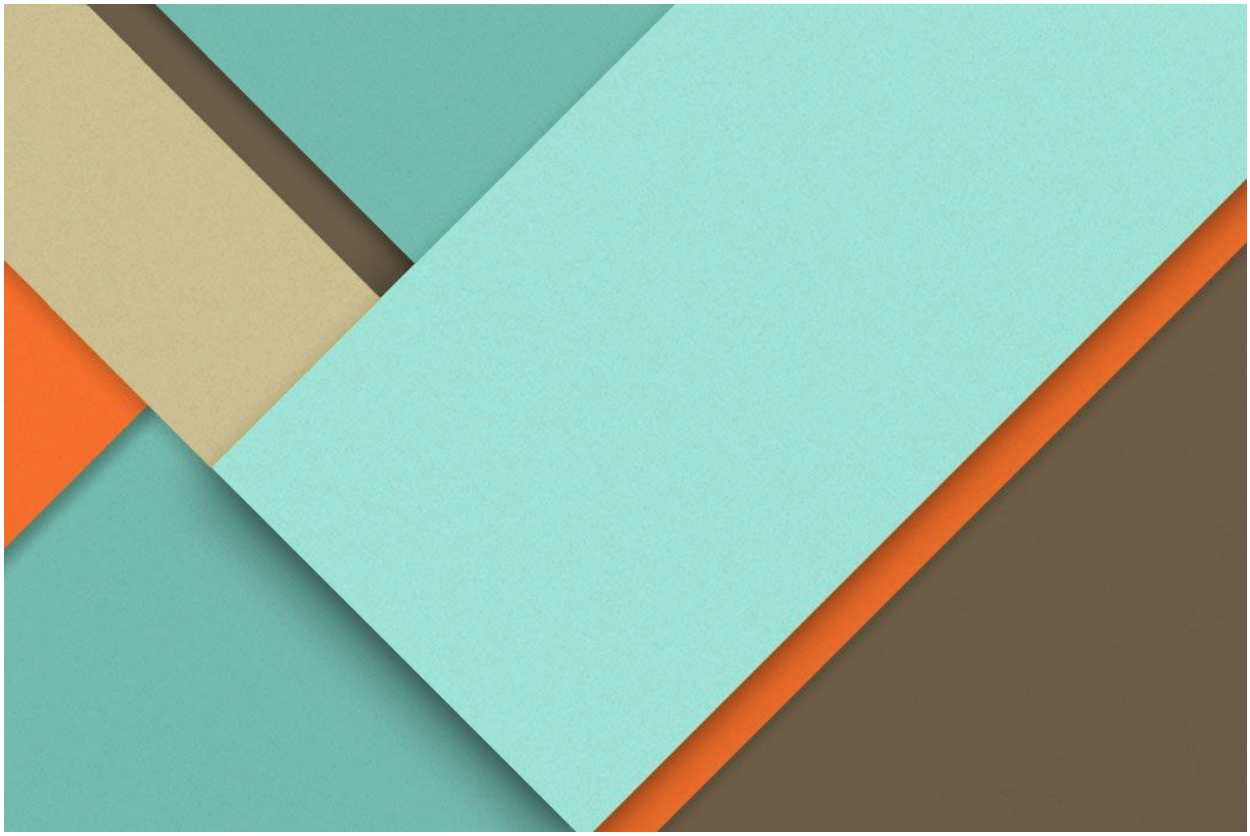


Juego de Ahorcado

Diseño del Sistema

V1.0.1

23.06.2020

Grupo Random

Sebastián Klincovitzky

Cristian Pereyra

Ignacio Ruano

Santiago Moutón

Historial de Cambios

Versión	Fecha	Resumen	Autor
1.0.0	19-06-2020	Creación del archivo	Ignacio Ruano, Cristian Pereyra, Santiago Mouton, Klincovitzky Sebastian
1.0.1	23-06-2020	Corrección de diagramas	Ignacio Ruano, Cristian Pereyra, Santiago Mouton, Klincovitzky Sebastian

Índice

Diagrama de Paquetes

Muestra cómo se encuentra organizado el código y las dependencias separándolo en subsistemas. En este caso, muestra cómo existe una relación de dependencia entre las pruebas y el código en sí.

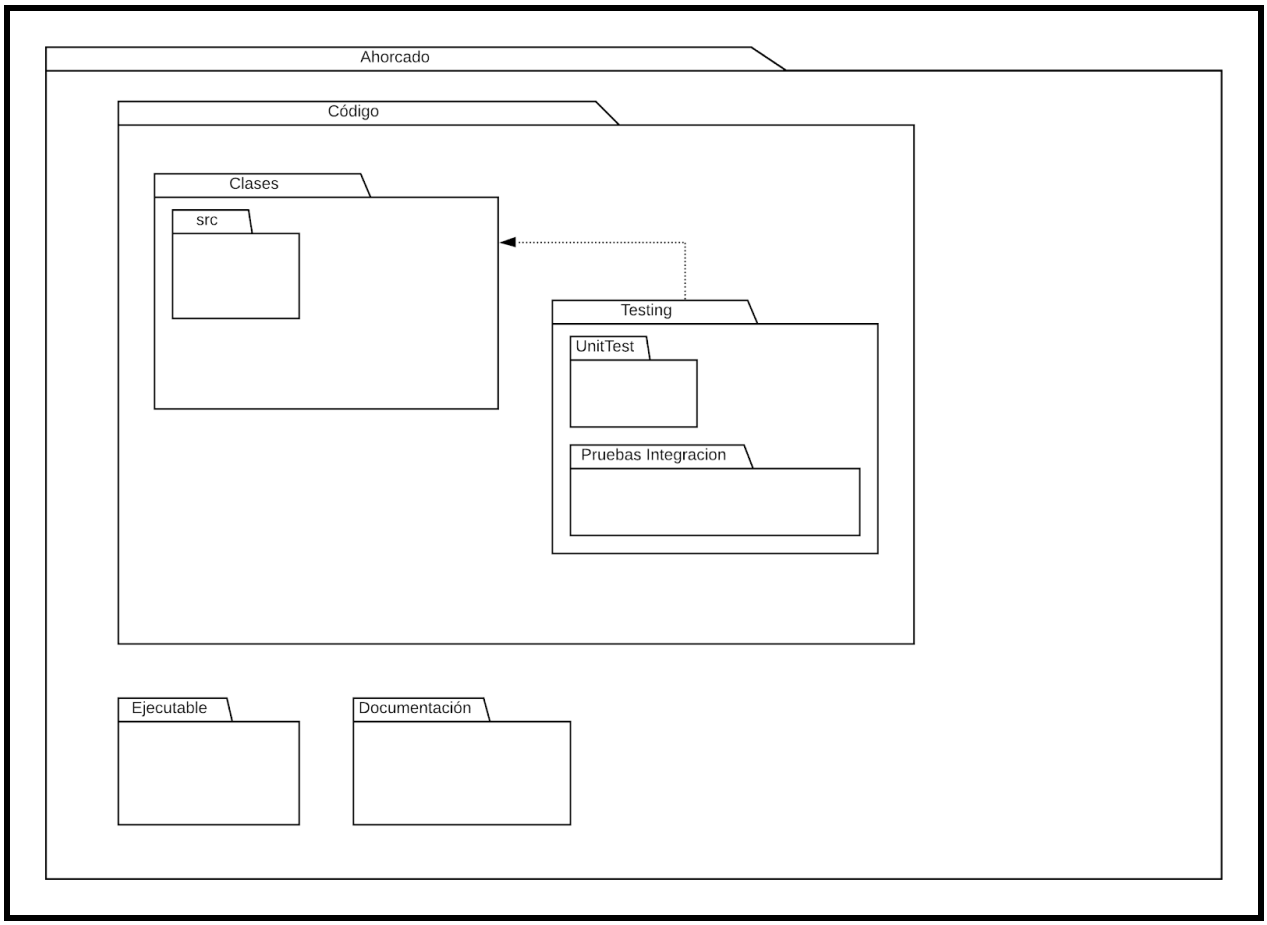


Diagrama de Clases

Describe clases de objetos y sus relaciones.

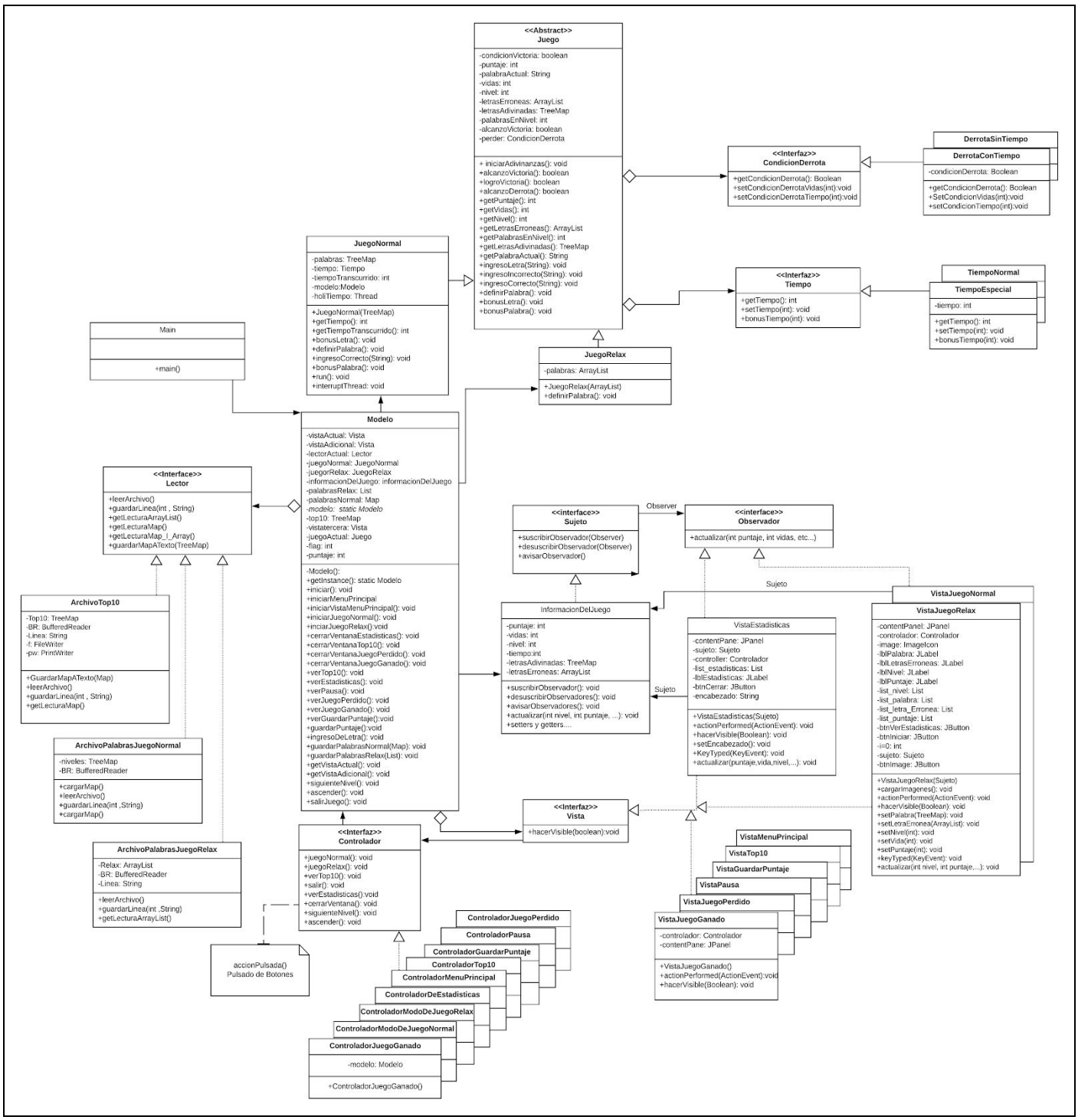
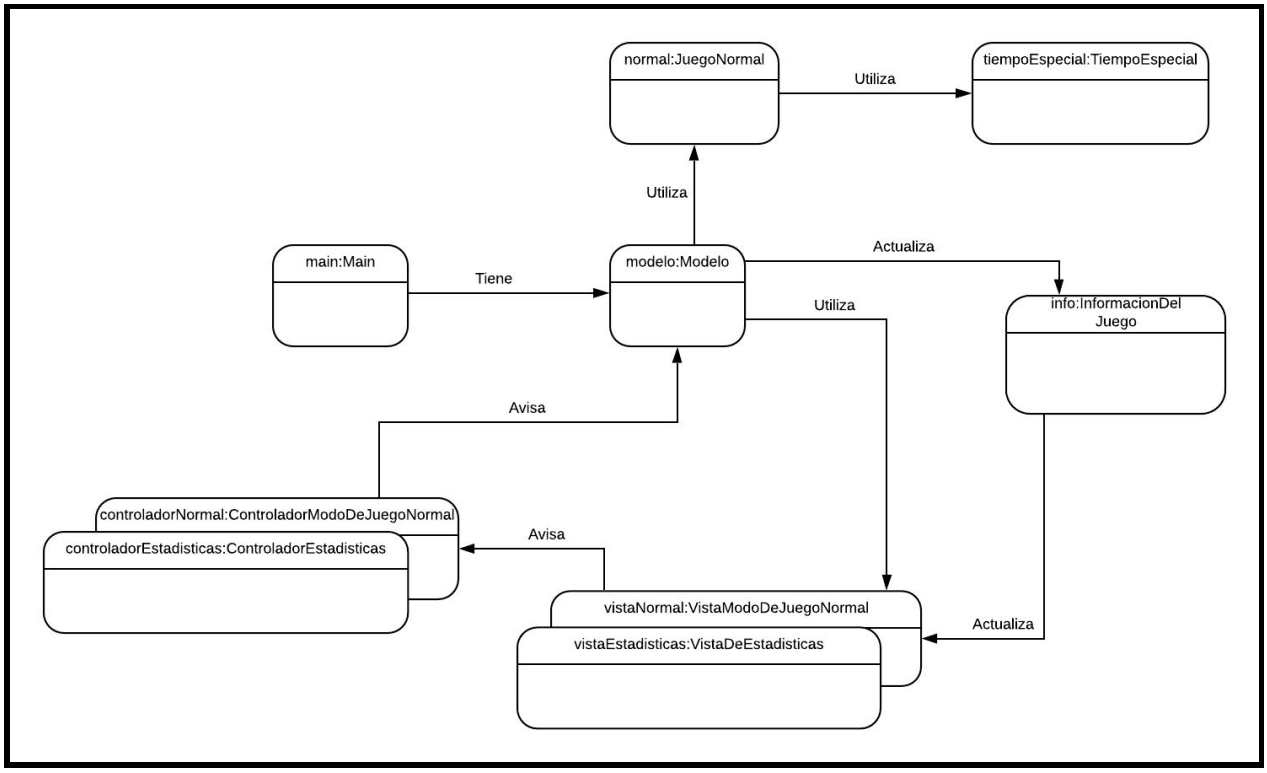


Diagrama de Objetos

El diagrama de objetos representa al programa en un momento del tiempo, es decir, muestra las clases y las instancias del diagrama de clases, y las interacciones entre estas para realizar una actividad determinada. Este nos permite observar la perspectiva cronológica de las interacciones. En este caso, presentamos un momento en que el programa se encuentra tanto con la vista del juego normal, como con la de estadisticas, ademas de estar haciendo uso del tiempo especial.

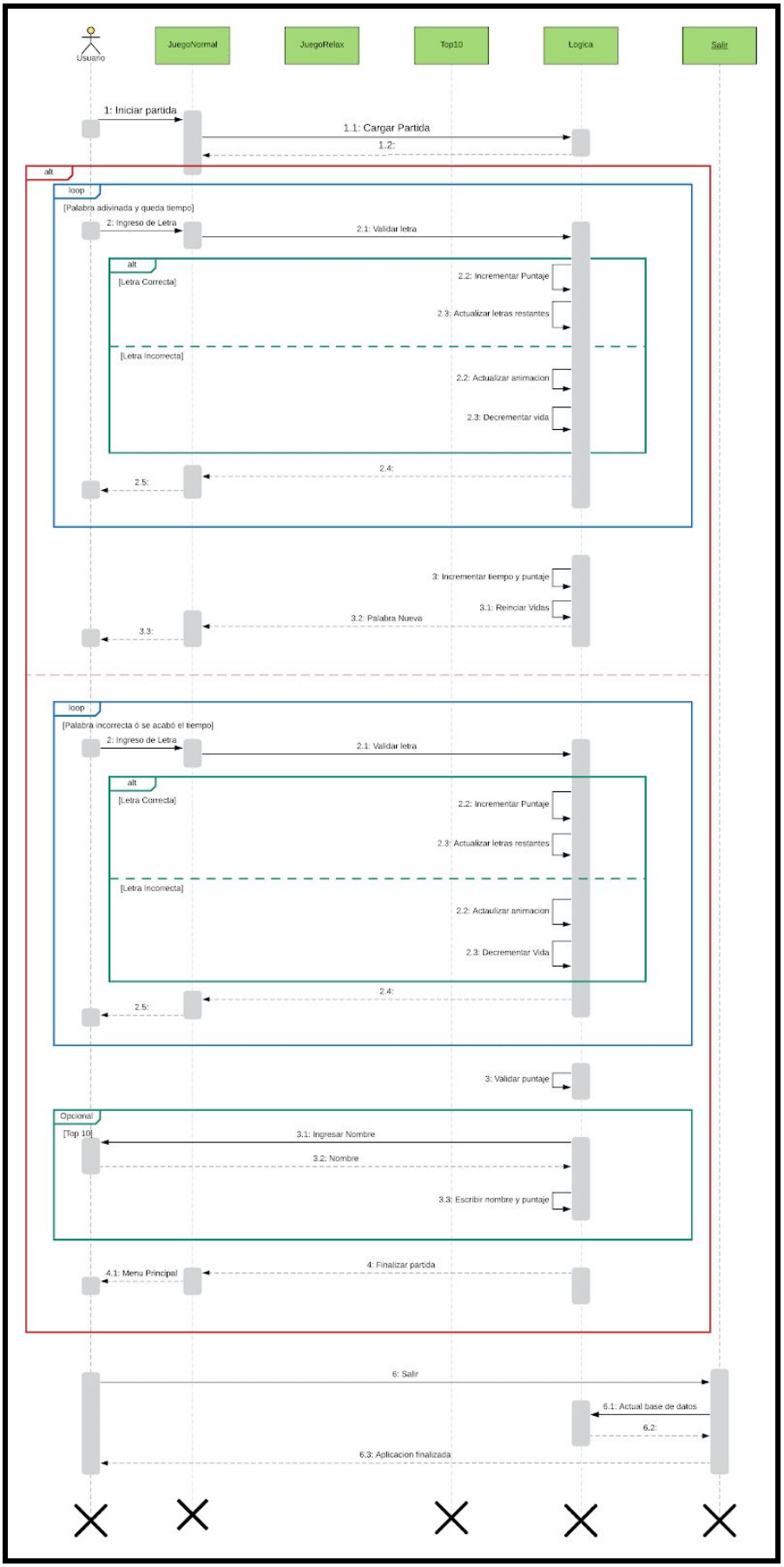


Diagramas de Secuencia

- Jugador inicia una partida, juega y sale.

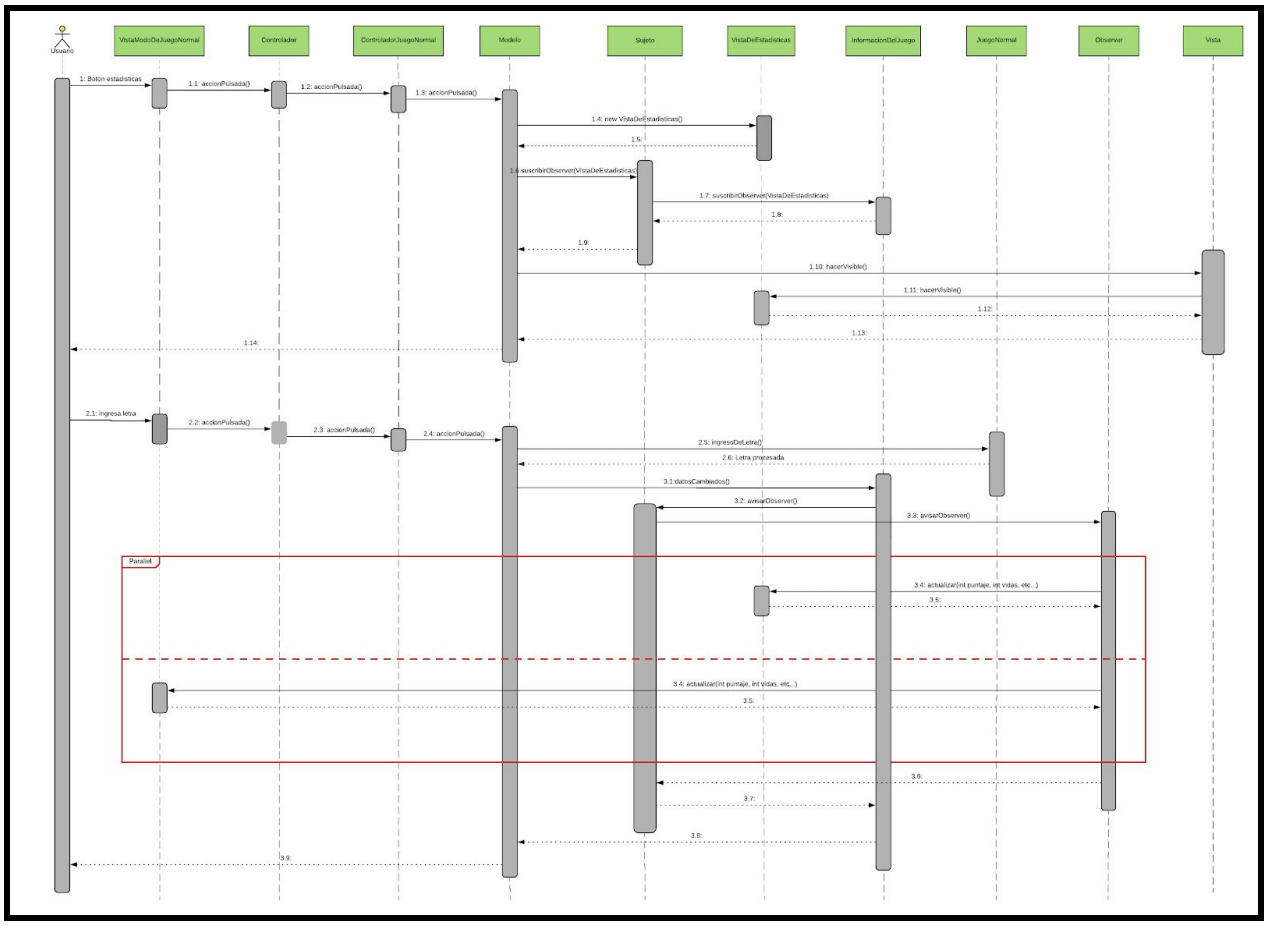
Interacción del usuario con el programa:

- El usuario selecciona el primer modo de juego
- Aquí se muestra dos posibles resultados
  - Adivina la palabra y sigue con el juego
  - Cumple una de las condiciones para perder
- El segundo modo de juego, dado que es igual al primero, pero sin la condición del tiempo, se ha decidido no incluirlo en el diagrama, sin embargo, se lo incluye al principio ya que forma parte del programa.
- Sin bien estando en el menú principal el usuario tiene la alternativa de elegir voluntariamente entre las 4 opciones que se le presentan, en esta ocasión se ha elegido un orden a seguir para mayor claridad. Este orden será visitar el Top10 luego de jugar, y finalmente salir del programa.
- En este diagrama se obviaron los detalles del programa y del modelo, para poder presentar de manera más sencilla la lógica que seguirá el juego.


Usuario
JuegoNormal
JuegoRelax
Top10
Logica
Salir
1: Iniciar partida
1.1: Cargar Partida
1.2:
alt
loop
[Palabra adivinada y queda tiempo]
2: Ingreso de Letra
2.1: Validar letra
alt
[Letra Correcta]
2.2: Incrementar Puntaje
2.3: Actualizar letras restantes
[Letra Incorrecta]
2.2: Actualizar animacion
2.3: Decrementar vida
2.4:
2.5:
3: Incrementar tiempo y puntaje
3.1: Reinciar Vidas
3.2: Palabra Nueva
3.3:
loop
[Palabra incorrecta ó se acabó el tiempo]
2: Ingreso de Letra
2.1: Validar letra
alt
[Letra Correcta]
2.2: Incrementar Puntaje
2.3: Actualizar letras restantes
[Letra Incorrecta]
2.2: Actaulizar animacion
2.3: Decrementar Vida
2.4:
2.5:
3: Validar puntaje
Opcional
[Top 10]
3.1: Ingresar Nombre
3.2: Nombre
3.3: Escribir nombre y puntaje
4: Finalizar partida
4.1: Menu Principal
6: Salir
6.1: Actual base de datos
6.2:
6.3: Aplicacion finalizada

- Jugador se encuentra en una partida y abre la ventana de estadísticas, ocurre un cambio y se actualizan las vistas.

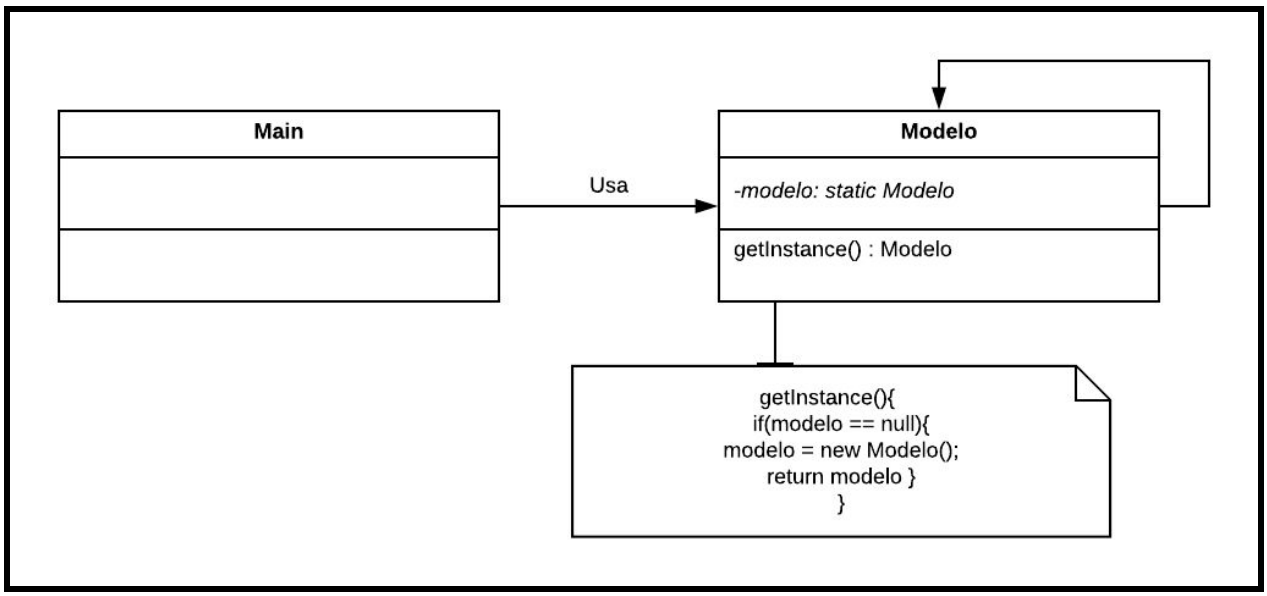
En este caso entendemos que es más importante representar las relaciones con el modelo, y por eso está enfocado a ello, y se abstrae de la lógica del juego (No tendrá en cuenta condición de victoria o derrota, ni verificar la validez de la letra)



Patrones de Diseño

- **Singleton**

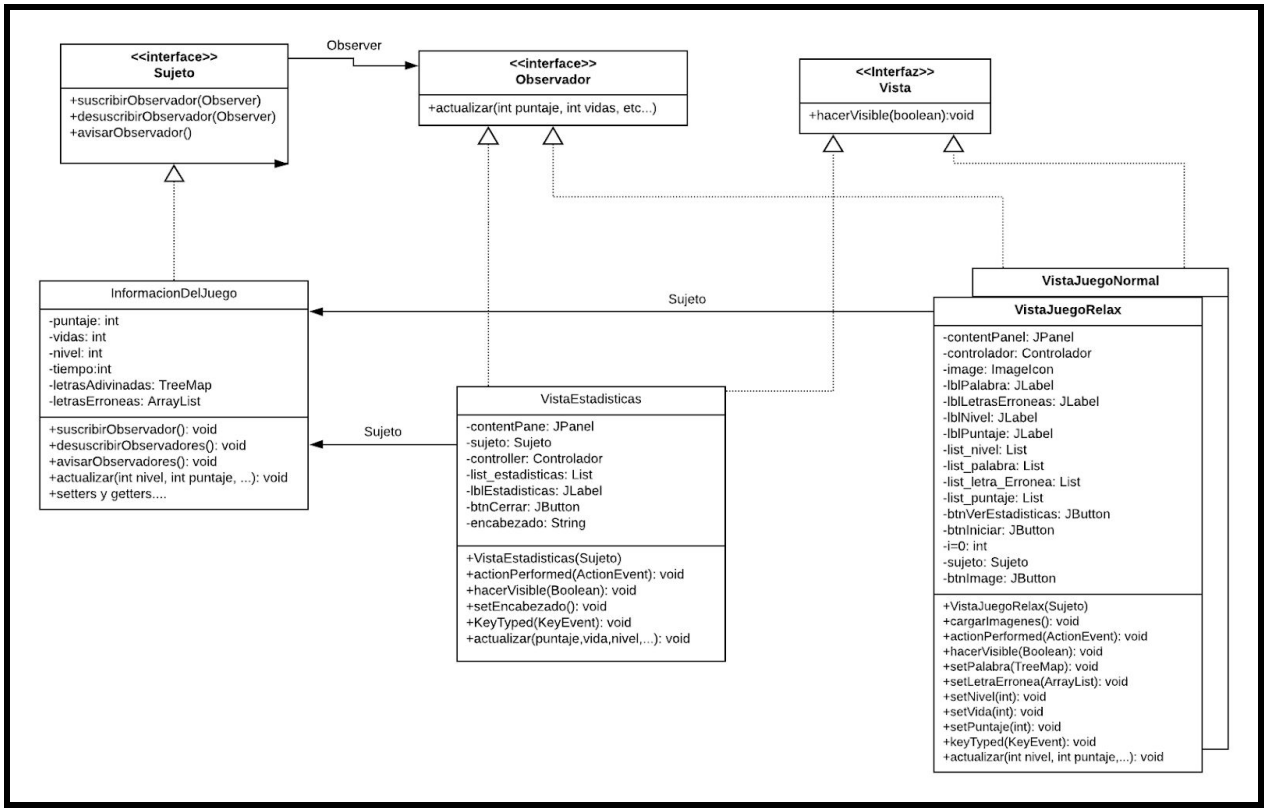
  Permite tener una sola instancia de una clase, en nuestro caso decidimos usar esta estrategia con Modelo que será compartida entre las diferentes instancias de Controlador. Así, se podrá interactuar al mismo Modelo desde cualquier Controlador. Adicionalmente, de este modo evitamos tener más de una instancia del programa, dado que el Modelo es en definitiva quien lo termina controlando en mayor medida.



- **Observer**

Permite actualizar la información de las diferentes vistas sólo cuando haya un cambio. Ahorra recursos, ya que no se tiene que estar preguntando a cada rato si hay información nueva o no.

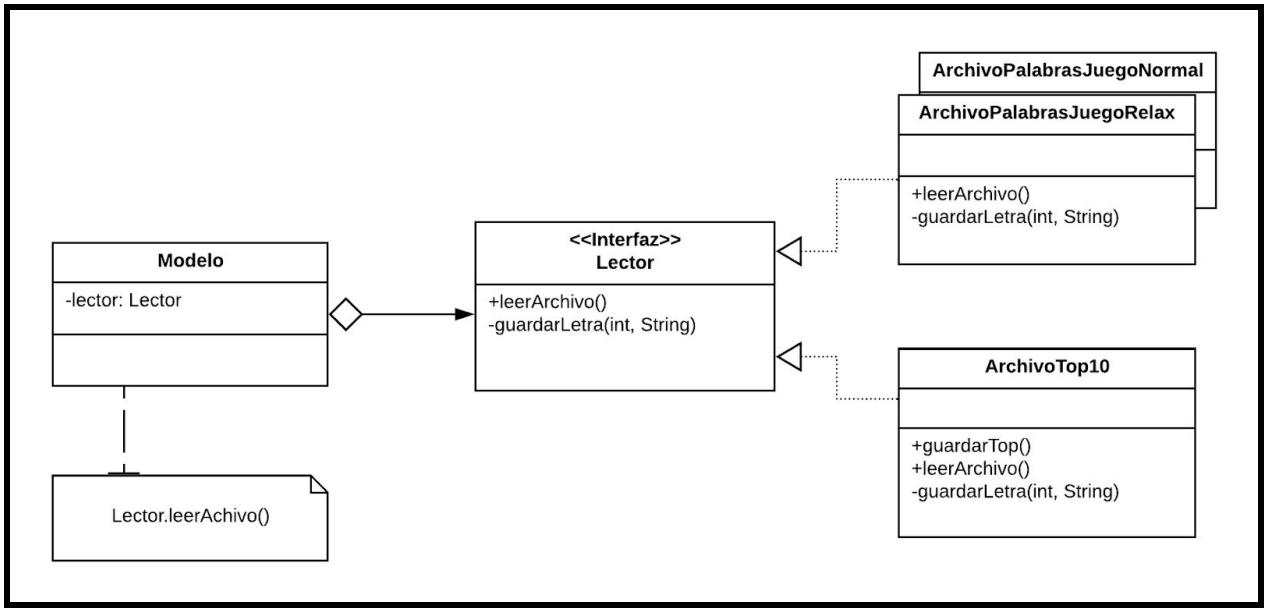
Este tipo de patrón de diseño implementado en el MVC permite desacoplar la vista del modelo y establecer una relación débil entre ambas clases para obtener un diseño más eficaz. Cada cambio de estado en el modelo implica una llamada al método actualizar de las vistas.

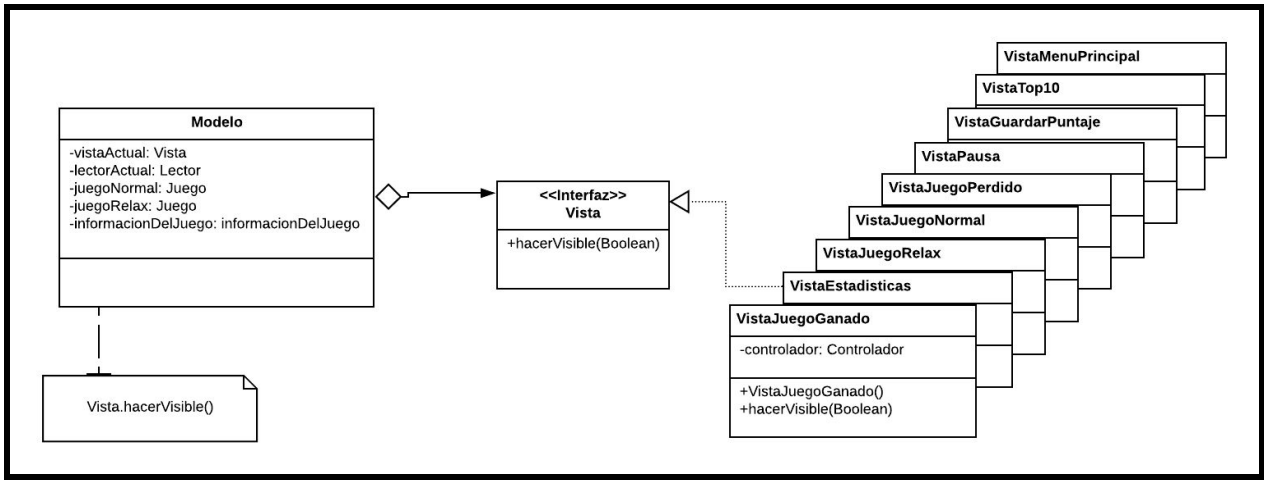


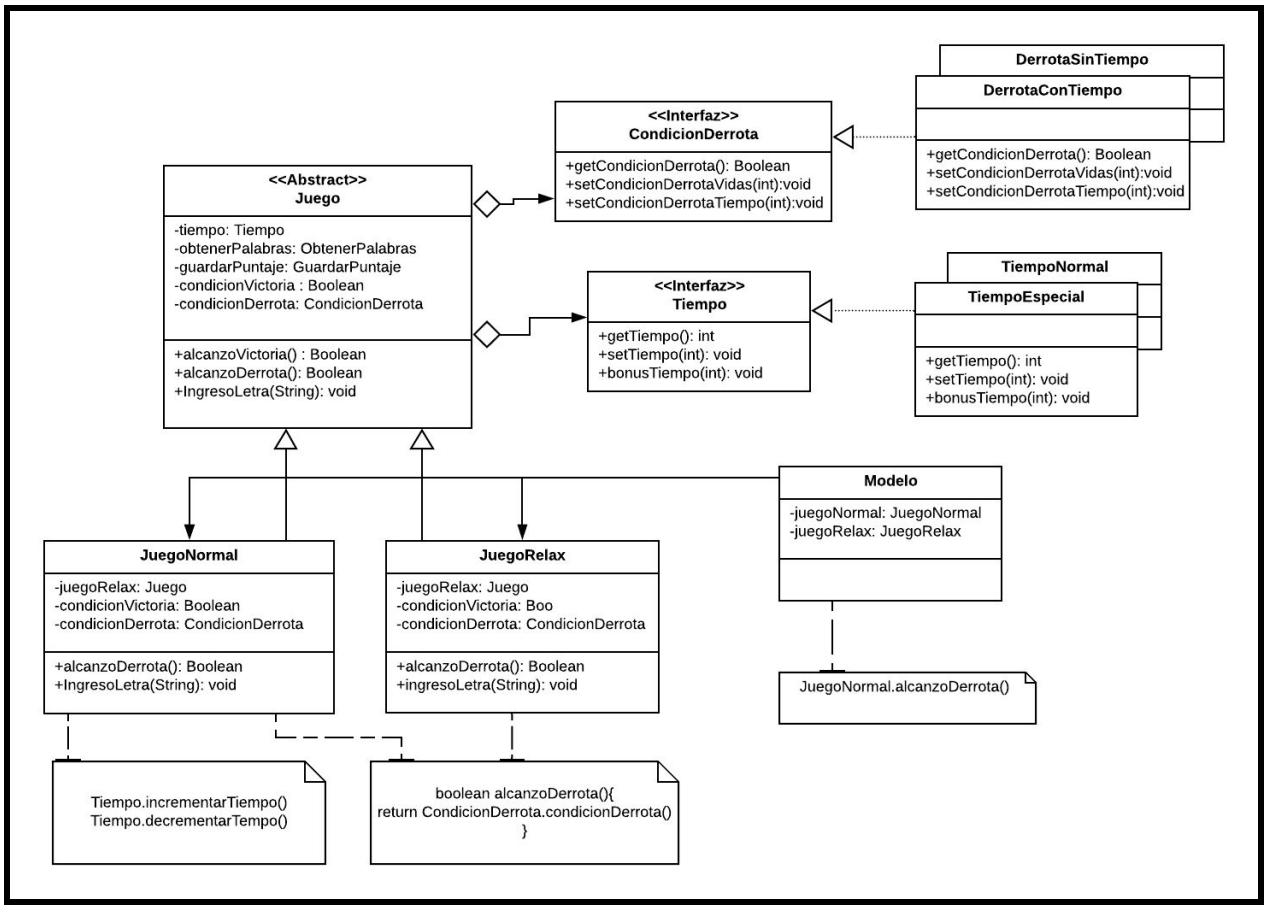
- **Strategy**

Se utiliza para cambiar entre varios comportamiento o vistas en tiempo de ejecución. En este proyecto se usa para los diferentes modos de juego, ya que la condición de derrota y el modo en que se maneja el tiempo es diferente para ambos. Además, como se obtendrán las palabras de distintas fuentes, se hace uso de este patrón nuevamente para la lectura y adquisición de las palabras.

Con este patrón podemos abstraernos de los detalles que conlleva trabajar con estas diferencias, y hacer uso de los resultados de forma general, sumando incluso a la posibilidad de cambiar en el futuro su comportamiento, o añadir un nuevo módulo que haga uso de ellas sin cambiar su forma de uso.






DerrotaSinTiempo
DerrotaConTiempo
+getCondicionDerrota(): Boolean
+setCondicionDerrotaVidas(int):void
+setCondicionDerrotaTiempo(int):void
<<Interfaz>>
CondicionDerrota
+getCondicionDerrota(): Boolean
+setCondicionDerrotaVidas(int):void
+setCondicionDerrotaTiempo(int):void
<<Abstract>>
Juego
-tiempo: Tiempo
-obtenerPalabras: ObtenerPalabras
-guardarPuntaje: GuardarPuntaje
-condicionVictoria : Boolean
-condicionDerrota: CondicionDerrota
+alcanzoVictoria() : Boolean
+alcanzoDerrota(): Boolean
+IngresoLetra(String): void
<<Interfaz>>
Tiempo
+getTiempo(): int
+setTiempo(int): void
+bonusTiempo(int): void
TiempoNormal
TiempoEspecial
+getTiempo(): int
+setTiempo(int): void
+bonusTiempo(int): void
Modelo
-juegoNormal: JuegoNormal
-juegoRelax: JuegoRelax
JuegoNormal
-juegoRelax: Juego
-condicionVictoria: Boolean
-condicionDerrota: CondicionDerrota
+alcanzoDerrota(): Boolean
+IngresoLetra(String): void
JuegoRelax
-juegoRelax: Juego
-condicionVictoria: Boo
-condicionDerrota: CondicionDerrota
+alcanzoDerrota(): Boolean
+ingresoLetra(String): void
JuegoNormal.alcanzoDerrota()
Tiempo.incrementarTiempo()
Tiempo.decrementarTempo()
boolean alcanzoDerrota(){
return CondicionDerrota.condicionDerrota()
}

Pruebas Unitarias

Las pruebas unitarias nos permitirán comprobar nuestro código a nivel de módulos individuales para asegurarnos que funcionan correctamente.

A continuación explicaremos 6 pruebas unitarias del total realizadas:

Prueba Unitaria 1 - TestCaseVentanaMenuPrincipal: Se generó esta prueba para comprobar el funcionamiento correcto del modelo con el Menú Principal, el test simula el resultado que debería provocar el controlador. Finalmente verifica que existe la ventana del Menú Principal. Este test se realizó haciendo uso de la clase Modelo.

Prueba Unitaria 2 - TestCaseBotonJuegoNormal: Se generó esta prueba para comprobar que, cuando se le comunica al modelo que se apreto el boton de iniciar una partida de Juego Normal, se reemplaza la ventana del Menú Principal, con la del Juego Normal. El test simula el aviso del controlador al modelo. Finalmente verifica que se realiza el aviso que, mediante Vista, abre la nueva ventana. Este test se realizó haciendo uso de la clase Modelo.

Prueba Unitaria 3 - TestCaseBotonJuegoRelax: Se generó esta prueba para comprobar que, cuando se le comunica al modelo que se apreto el boton de iniciar una partida de Juego Relax, se reemplaza la ventana del Menú Principal, con la del Juego Relax. El test simula el aviso del controlador al modelo. Finalmente verifica que se realiza el aviso que, mediante Vista, abre la nueva ventana .Este test se realizó haciendo uso de la clase Modelo.

Prueba Unitaria 4 - TestCaseVentanaApreturaDeObservadores: Se generó esta prueba para comprobar que se abra la venta de estadísticas al apretar el botón de estadísticas desde la ventana de juego. El test simula que el programa está en una partida activa. Finalmente verifica que existe la ventana de estadísticas. Este test se realizó haciendo uso de la clase Modelo.

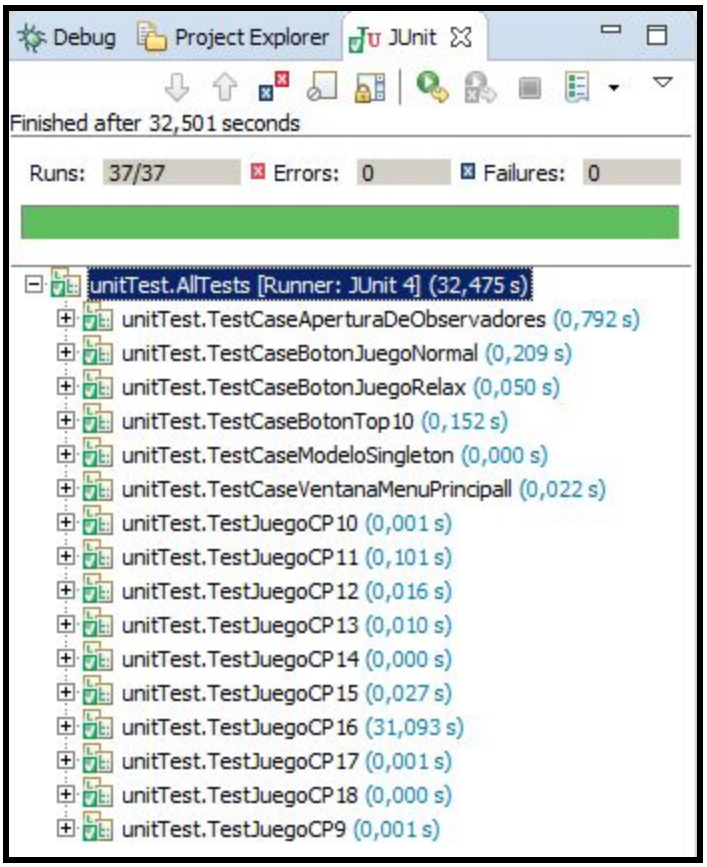
Prueba Unitaria 5 - TestCaseBotonTop10: Se generó esta prueba para comprobar que, cuando se le comunica al modelo que se apreto el boton de ver el Top10, se reemplaza la ventana del Menú Principal, con la del Top10. El test simula el aviso del controlador al modelo. Finalmente verifica que se realiza el aviso que, mediante Vista, abre la nueva ventana. Este test se realizó haciendo uso de la clase Modelo.

Prueba Unitaria 6 - TestCaseModeloSigleton: Se generó esta prueba para comprobar el funcionamiento correcto del patrón Singleton en la clase Modelo, el test simula el la existencia de una instancia del Modelo. Finalmente verifica que al intentar abrir una nueva instancia, sigue existiendo sólo uno. Este test se realizó haciendo uso de la clase Modelo.

Para ejecutar los test de unitarias, en la IDE Eclipse dirigirse a Ahorcado/Clases/testing/pruebasUnitarias -> AllTestCase

sobre este último paquete se hace click derecho y se selecciona la opción: run as -> JUnit test.

A continuación se muestra el resultado de la ejecución de los test unitarios:



Pruebas de Integración

Presentamos 4 pruebas unitarias realizadas para comprobar las Pruebas de Integración:

Prueba de Integración 1 - IntegracionBotonJuegoRelax: Se verifica que desde la selección del botón de Juego Relax en el Menú Principal hasta la la apertura del mismo, existe una correcta intercomunicación entre Modelo, Vista y Controlador.

Prueba Integración 2 - TestCaseBotonTop10: Se verifica que desde la selección del botón de Top10 en el Menú Principal hasta la la apertura del mismo, existe una correcta comunicación entre Modelo, Vista y Controlador.

Prueba Integración 3 - IntegracionBotonJuegoNormal: Se verifica que desde la selección del botón de Juego Normal en el Menú Principal hasta la la apertura del mismo, existe una correcta comunicación entre Modelo, Vista y Controlador.

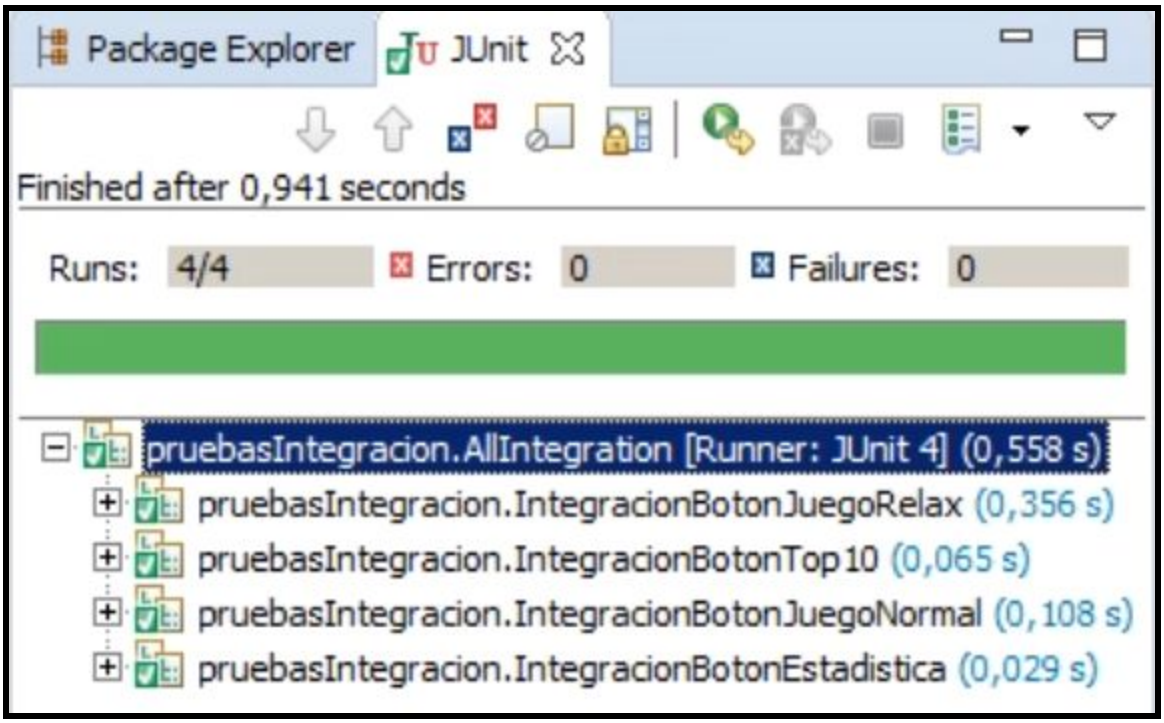
Prueba Integración 4 - IntegracionBotonEstadistica: Se verifica que desde la selección del botón de Estadística durante una partida hasta la la apertura del mismo, existe una correcta comunicación entre Modelo, Vista, Observador y Controlador.

Para ejecutar los test de integración, en la IDE Eclipse dirigirse a
Ahorcado/Clases/testing/pruebasIntegracion -> AllIntegrationTest

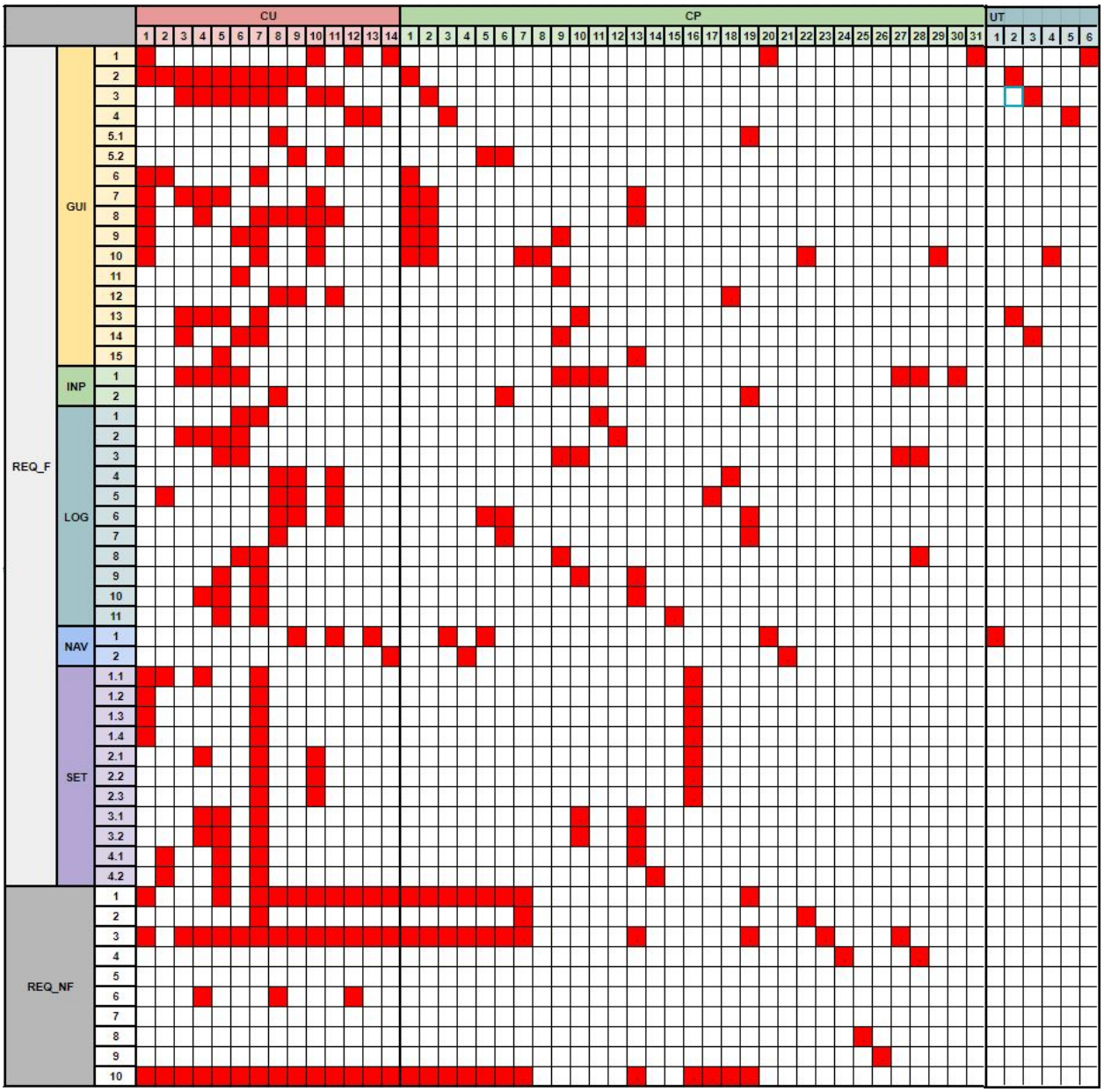
sobre este último

paquete se hace click derecho y se selecciona la opción: run as -> JUnit test.

A continuación se muestra el resultado de la ejecución de los test de integración:



Matriz de Trazabilidad Actualizada



A continuación se presenta una matriz donde puede ver como algunos test unitarios se relacionan con clases, aquí no se presentan para todos los casos de pruebas, solo algunos.

		Clases		
		Modelo	JuegoNormal	JuegoRelax
UT	1			
	2			
	3			
	4			
	5			
	6			
CP-UT	9			
	10			
	11			
	12			
	13			
	14			
	15			
	16			
	17			
	18			