MIND GAME

DOCUMENTO DE DISEÑO DEL SISTEMA

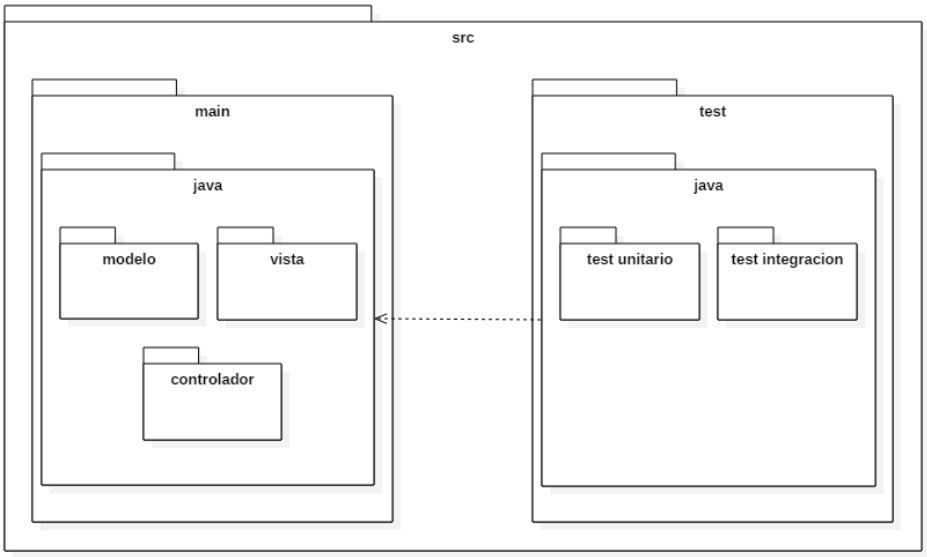
Autores: Sosa Ludueña Diego
Sleiman Mohamad
Choquevilca Gustavo

Versión del Documento: 1.0.0

Índice

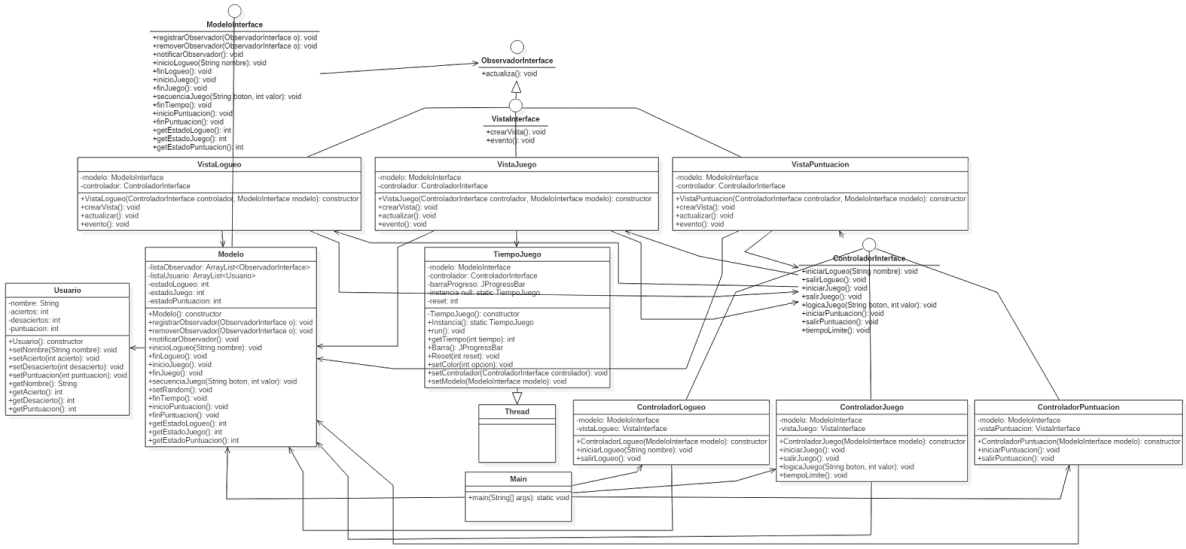
1. Diagrama de Paquetes

Este diagrama describe cómo se encuentra organizado el código y las dependencias.



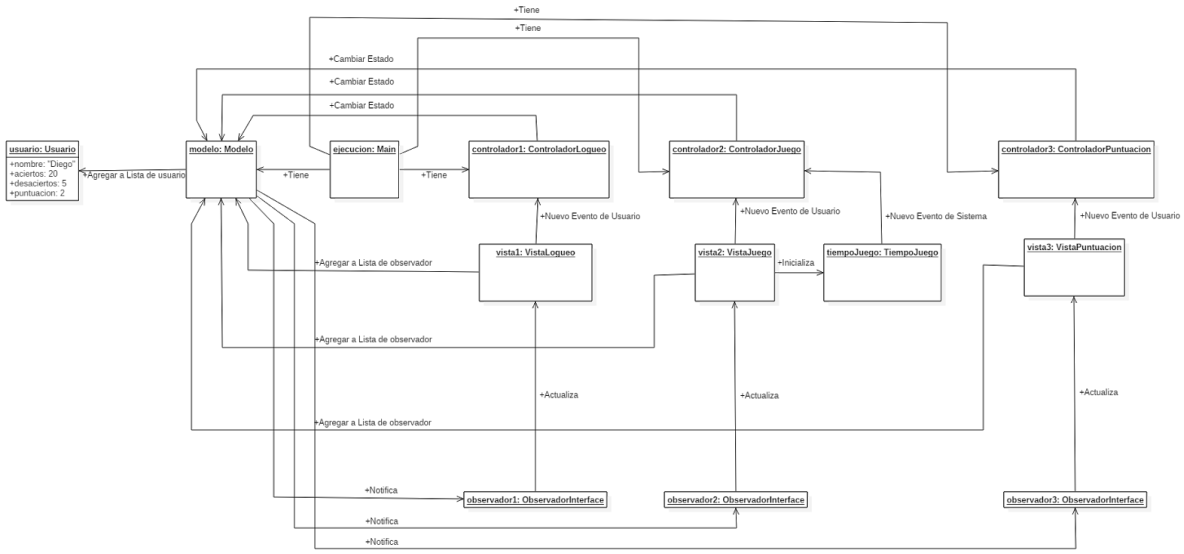
2. Diagrama de Clases

Este diagrama describe las clases de objetos y sus relaciones. Este diagrama de clases es un modelo estructural para representar la estructura estática de las clases.



3. Diagrama de Objetos

Un diagrama de objetos es un modelo de instancias, incluyendo objetos y datos. Un diagrama de objetos es una instancia de un diagrama de clases; muestra una 'foto' del estado de un sistema en un punto de tiempo determinado.

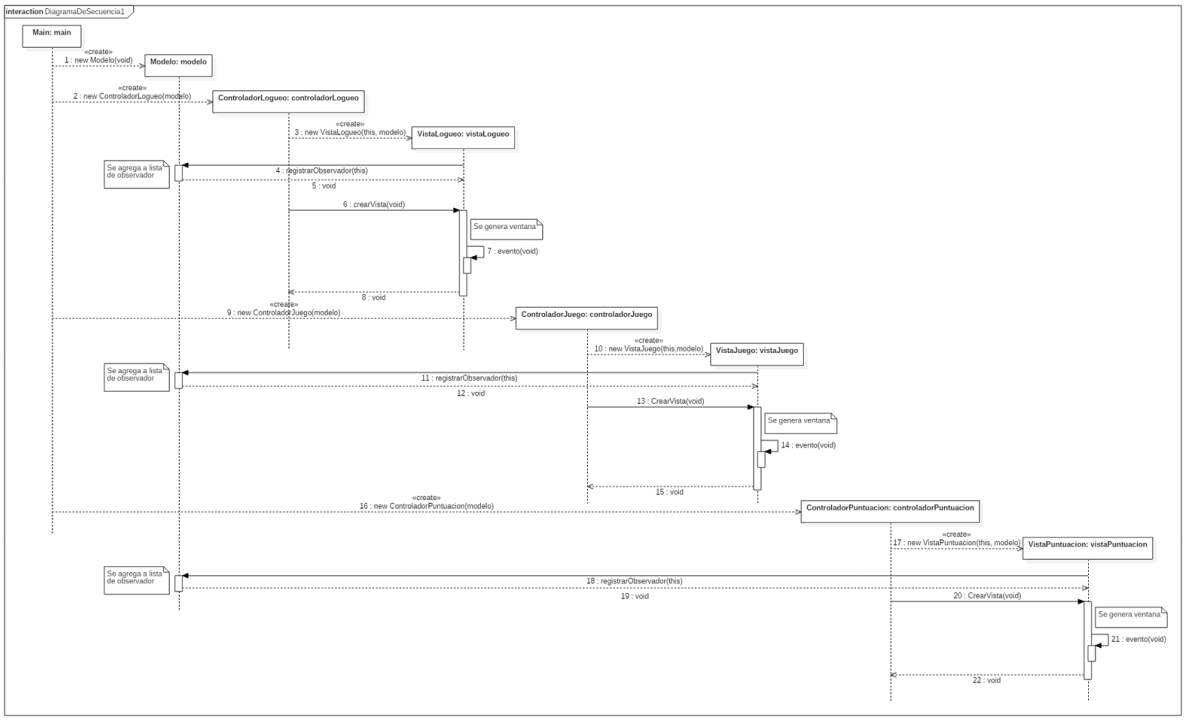


4. Diagrama de Secuencia

Este diagrama describe los mensajes que se envían los objetos para un escenario específico.

4.1 Creación de Objetos

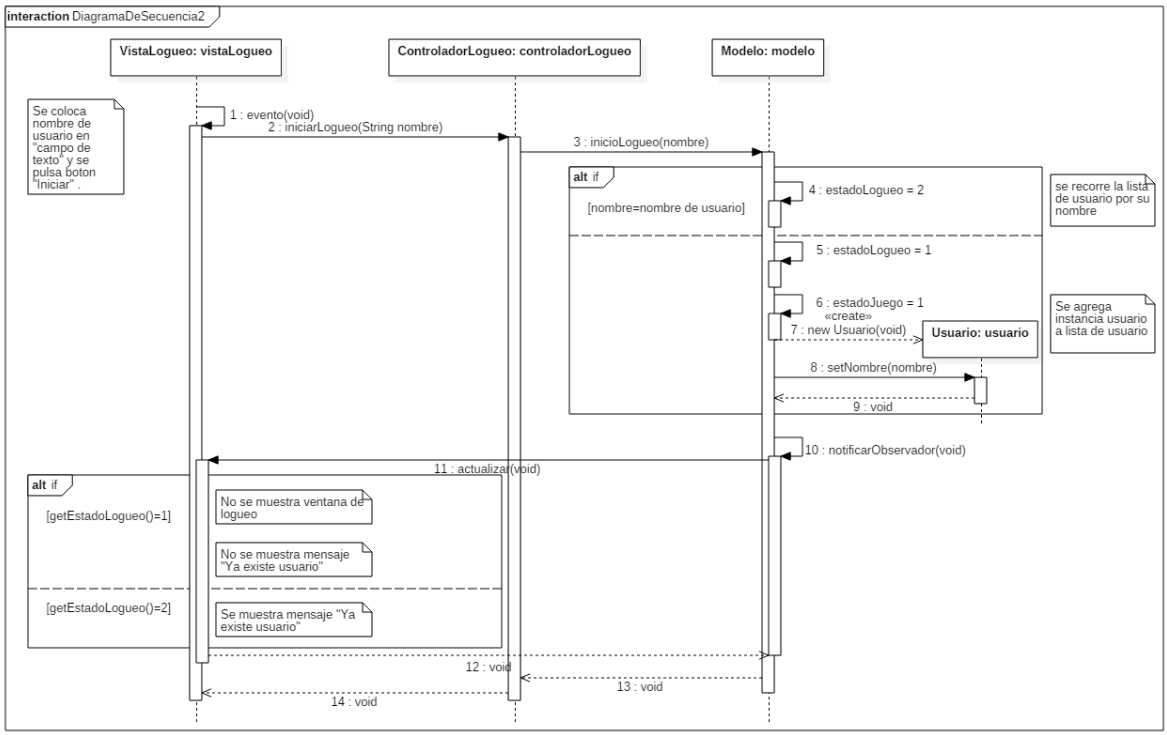
En este diagrama se muestra cómo cada objeto va creando otros objetos.



4.2 Ventana de Logueo ocurre evento de Usuario (pulsa tecla *Iniciar*)

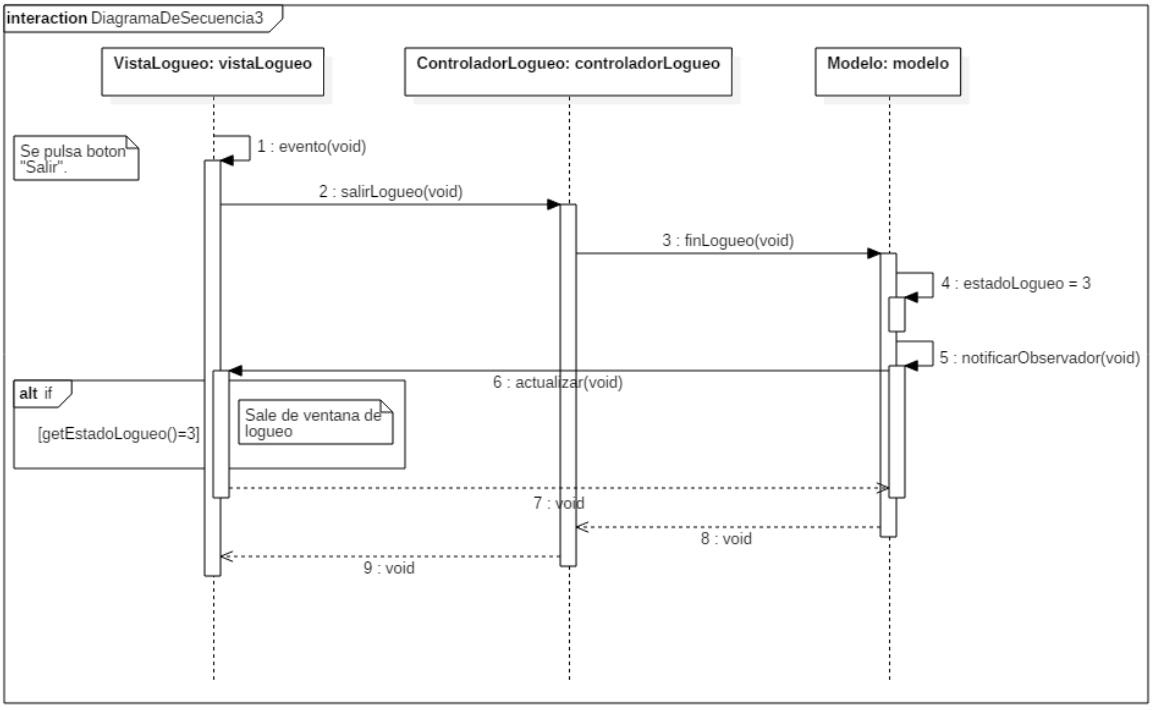
En este diagrama se muestra la interacción de los objetos, cuando se produce un evento de usuario en la ventana de logueo, donde el usuario coloca su *nombre de usuario* y pulsa la tecla

Iniciar.



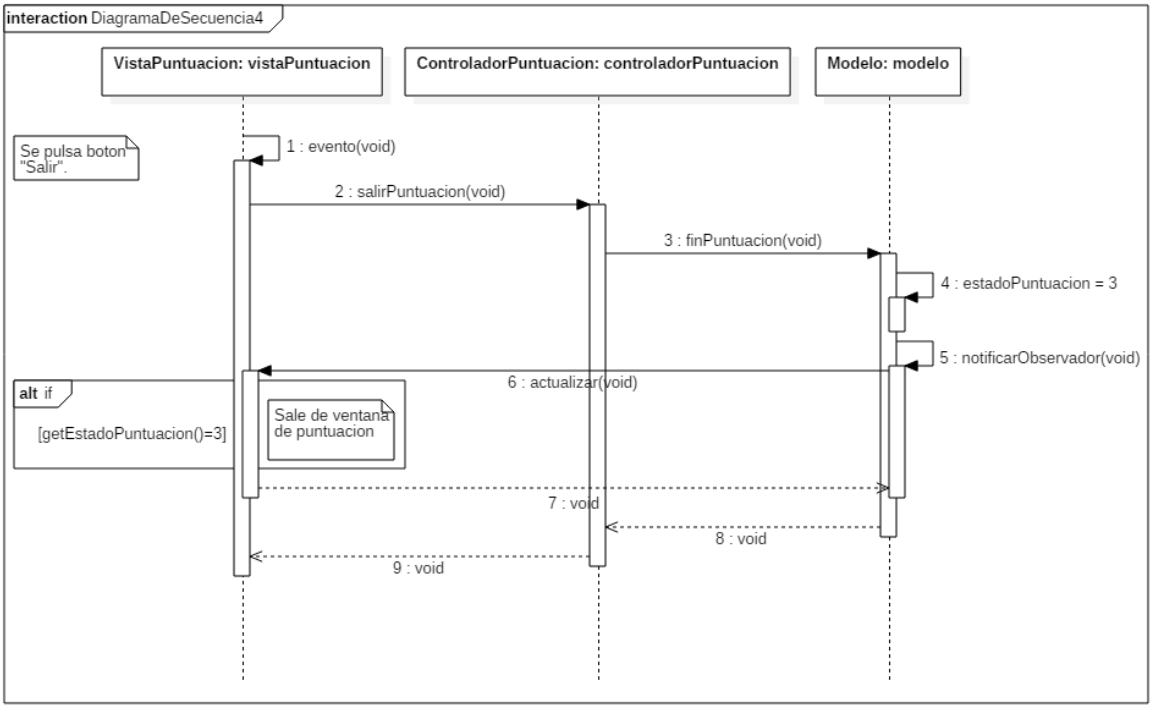
4.3 Ventana de Logueo ocurre evento de Usuario (pulsa tecla *Salir*)

En este diagrama se muestra la interacción de los objetos, cuando se produce un evento de usuario en la ventana de logueo, donde el usuario pulsa la tecla *Salir.*



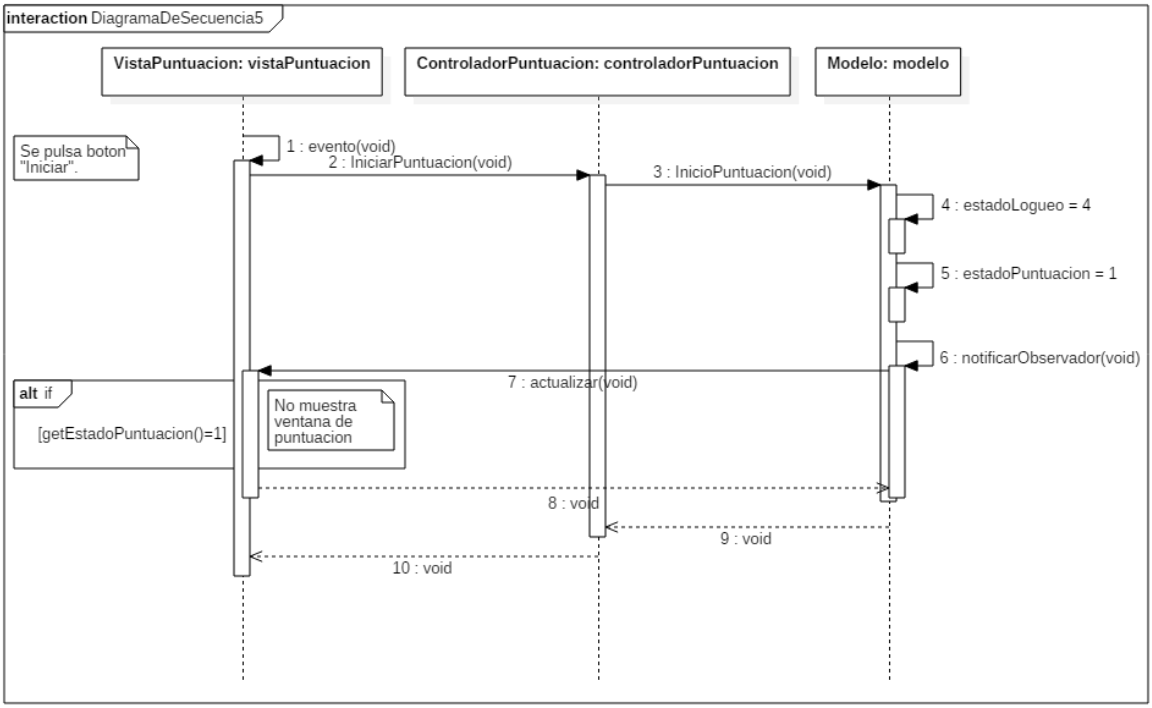
4.4 Ventana de Puntuación ocurre evento de Usuario (pulsa tecla *Salir*)

En este diagrama se muestra la interacción de los objetos, cuando se produce un evento de usuario en la ventana de puntuación, donde el usuario pulsa la tecla *Salir.*



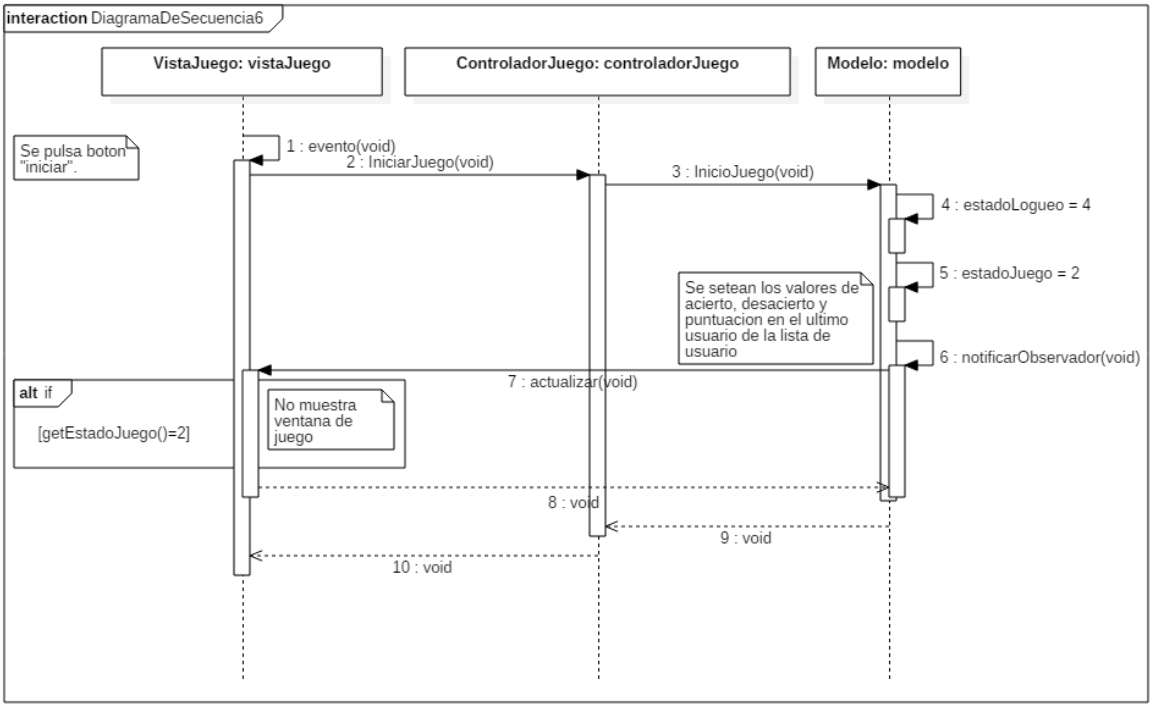
4.5 Ventana de Puntuación ocurre evento de Usuario (pulsa tecla *Iniciar*)

En este diagrama se muestra la interacción de los objetos, cuando se produce un evento de usuario en la ventana de puntuación, donde el usuario pulsa la tecla *Iniciar.*



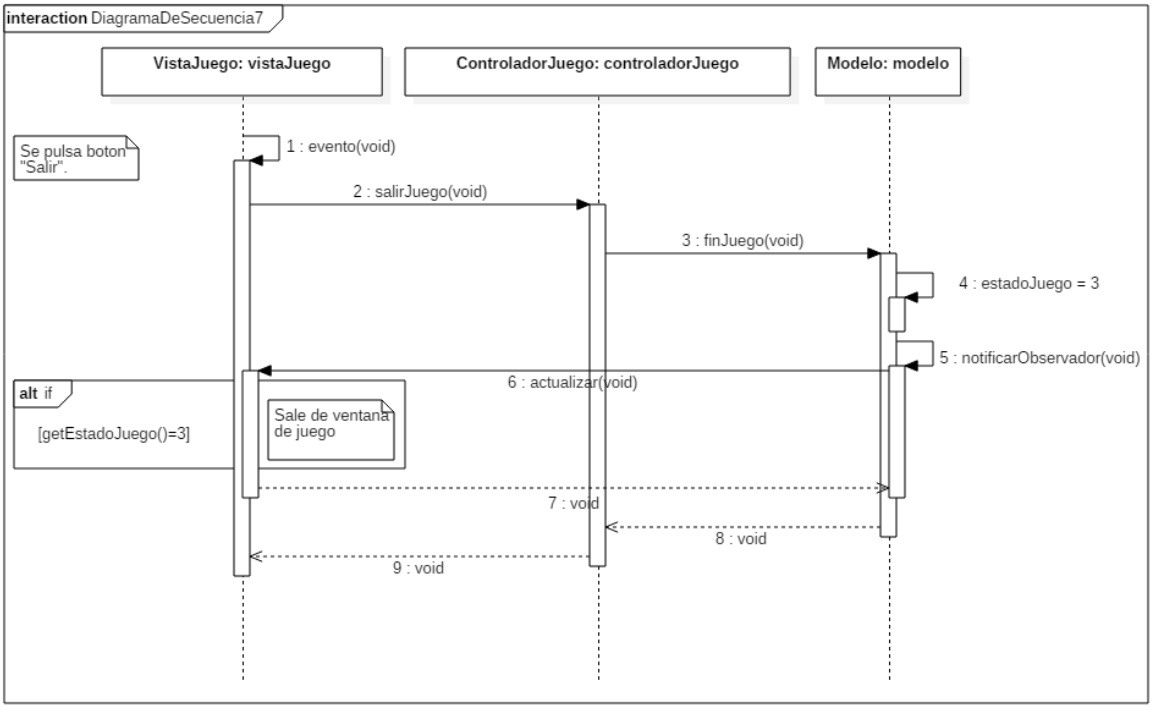
4.6 Ventana de Juego ocurre evento de Usuario (pulsa tecla *Iniciar*)

En este diagrama se muestra la interacción de los objetos, cuando se produce un evento de usuario en la ventana de juego, donde el usuario pulsa la tecla *Iniciar.*



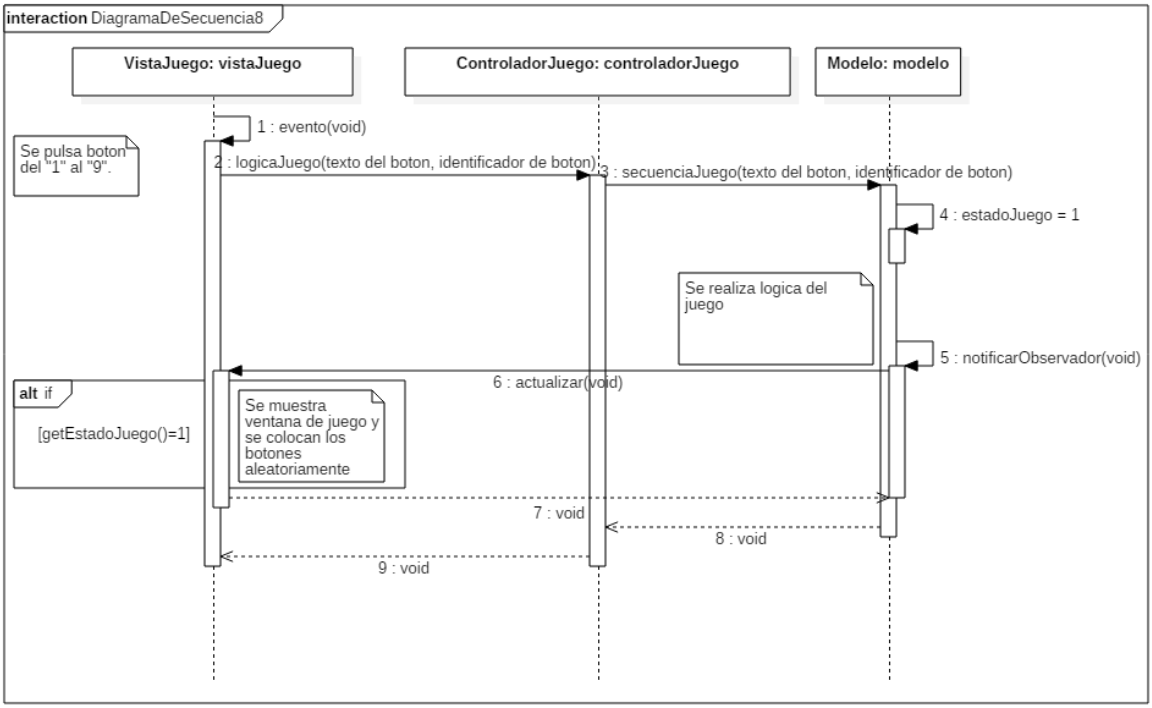
4.7 Ventana de Juego ocurre evento de Usuario (pulsa tecla *Salir*)

En este diagrama se muestra la interacción de los objetos, cuando se produce un evento de usuario en la ventana de juego, donde el usuario pulsa la tecla *Salir.*



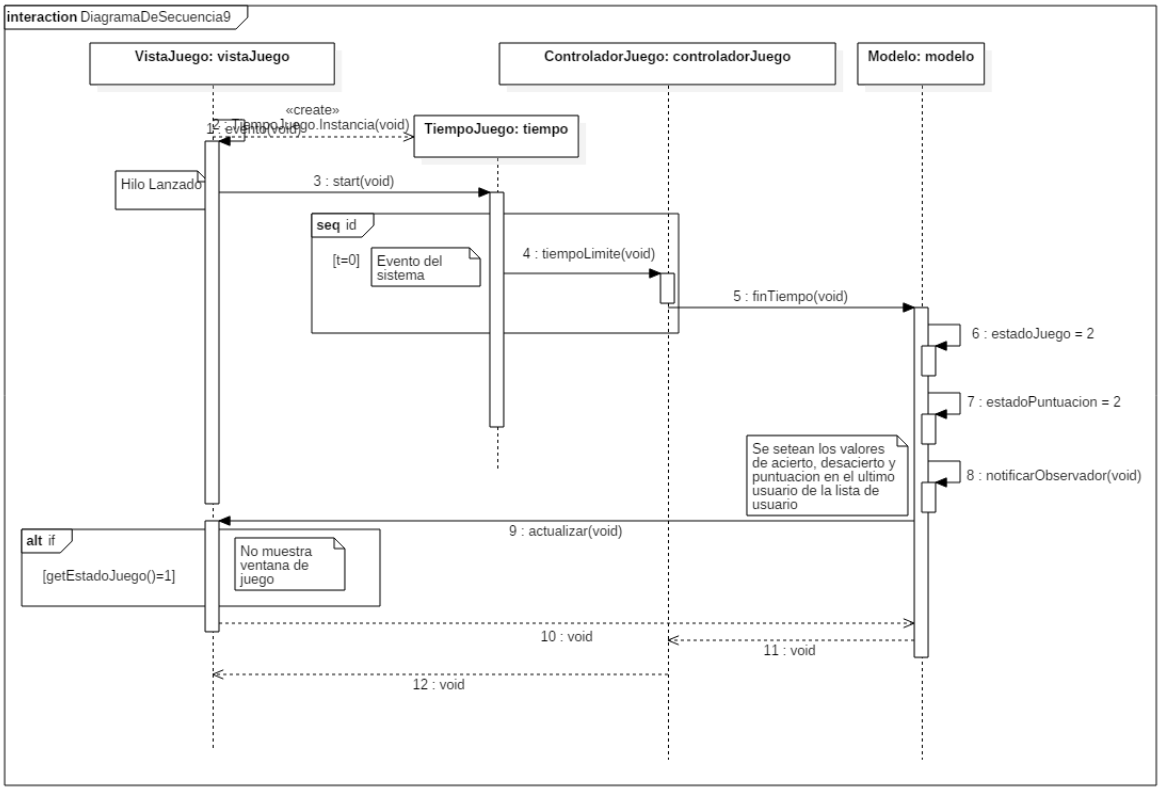
4.8 Ventana de Juego ocurre evento de Usuario (pulsa tecla *1,…, o 9*)

En este diagrama se muestra la interacción de los objetos, cuando se produce un evento de usuario en la ventana de juego, donde el usuario pulsa la tecla *1* a *9.*



4.9 Ventana de Juego ocurre evento de Sistema (tiempo=0)

En este diagrama se muestra la interacción de los objetos, cuando se produce un evento de sistema en la ventana de juego, donde el sistema produce un evento cuando la variable *tiempo=0.*

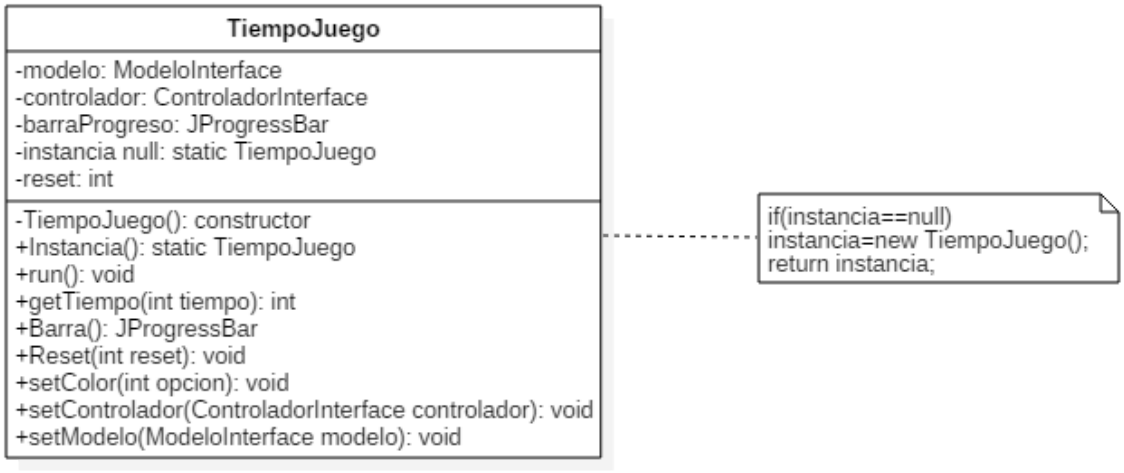


5. Patrones de Diseño

El patrón de diseño consiste en un diagrama de objetos que forma una solución a un problema conocido y frecuente. El diagrama de objetos está constituido por un conjunto de objetos descritos por clases y las relaciones que enlazan los objetos. Los patrones de diseño están basado en las buenas prácticas de la programación orientada a objetos. Por lo tanto, en este proyecto se utilizo 3 patrones de diseño esenciales que son el Singleton, Strategy y Observer. Cada uno se de estos patrones utilizados los describiremos a continuación:

5.1 Patrón de Diseño SINGLETON

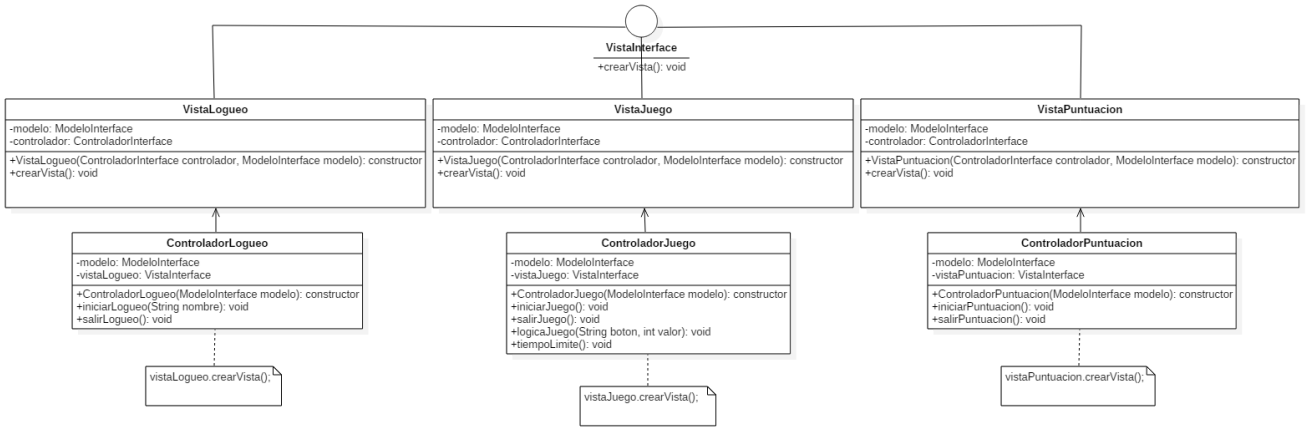
El patrón Singleton tiene como objetivo asegurar que una clase sólo posee una instancia y proporcionar un método de clase único que devuelva esta instancia.
En nuestra aplicación el patrón Singleton es utilizado en la Clase *TiempoJuego* donde creamos una única instancia de esta clase debido a que es necesario que solo haya un temporizador en nuestro juego. También es importante mencionar que debido a que dicha clase es manejada por un hilo secundario para que realice la tarea de chequear el tiempo. Por lo tanto, este patrón nos sirvió de ayuda para evitar que cada vez que se inicie una partida se lancen múltiples hilos. Para evitar este inconveniente se crea el objeto con el método *Instancia(),* donde logramos que dicho objeto se cree una única vez, por más que se quiera a volver a crear.. A continuación se muestra el diagrama de clases:



5.2 Patrón de Diseño STRATEGY

El patrón Strategy tiene como objetivo adaptar el comportamiento y los algoritmos de un objeto en función de una necesidad sin cambiar las interacciones de este objeto con los clientes.
En nuestra aplicación el patrón Strategy se implementa para asociar cada controlador con cada vista del patrón de arquitectura MVC. Debido a esto, MVC ofrece la posibilidad de cambiar, incluso en tiempo de ejecución, los componentes gráficos que administran las acciones del usuario. Por los tanto tendremos que la estrategia es la interface llamada *VistaInterface* cuyo método que implementan las estrategias concretas se llama *crearVista().* Estas estrategias concretas son las clases *VistaLogueo, VistaJuego y VistaPuntuacion.* Por lo tanto cada estrategia

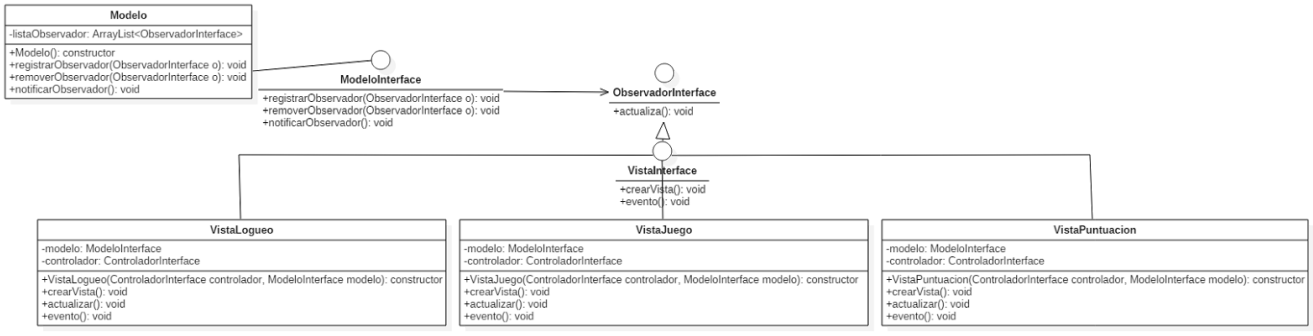
concreta será llamada desde las clases *ControladorLogueo, ControladorJuego y ControladorPuntuacion.* A continuación se muestra el diagrama de clases:



5.3 Patrón de Diseño OBSERVER

El patrón Observer tiene como objetivo construir una dependencia entre un sujeto y los observadores de modo que cada modificación del sujeto sea notificada a los observadores para que puedan actualizar su estado.

En nuestra aplicación el patrón Observer lo utilizamos para vincular el modelo con las vistas del patrón de arquitectura MVC. En este patrón, el modelo constituye el sujeto y cada vista es un observador. De este modo, cada actualización de datos que gestione el núcleo funcional genera una notificación a las distintas vistas. Éstas pueden, entonces, actualizar la información que muestran al usuario. Por lo tanto, se tiene que el sujeto es la interface *ModeloInterface* luego tenemos al sujeto concreto que es la clase *Modelo* que implementa los métodos *registrarObservador(ObservadorInterface o), removerObservador(ObservadorInterface o) y notificarObservador().* Por otra parte, se tiene al observador que es la interface *ObservadorInterface* donde el método *actualizar()* será implementado por los observadores concretos que son las clases *VistaLogueo, VistaJuego y VistaPuntuacion.* A continuación se muestra el diagrama de clases:



6. Pruebas Unitarias

Una vez realizado el diseño, se generaron en total 6 test unitarios que se realizaron con la herramienta *JUnit Test Case* provista por la *IDE Eclipse.* Estos test unitarios son:

6.1. TestNumeros

En este test unitario se prueba que el vector de números tenga sus elementos del 1 al 9 sin repetirse y pueden estar ordenados al azar.

6.2. TestNombreUsuario

En este test unitario se prueba que el listado de usuarios no tenga nombre de usuarios repetidos.

6.3. TestAciertosUsuario

En este test unitario se prueba que la variable de aciertos incremente cada vez que se pulsa el botón correcto.

6.4. TestDesaciertosUsuario

En este test unitario se prueba que la variable de desaciertos incremente cada vez que se pulsa el botón incorrecto.

6.5. TestPuntuacionUsuario

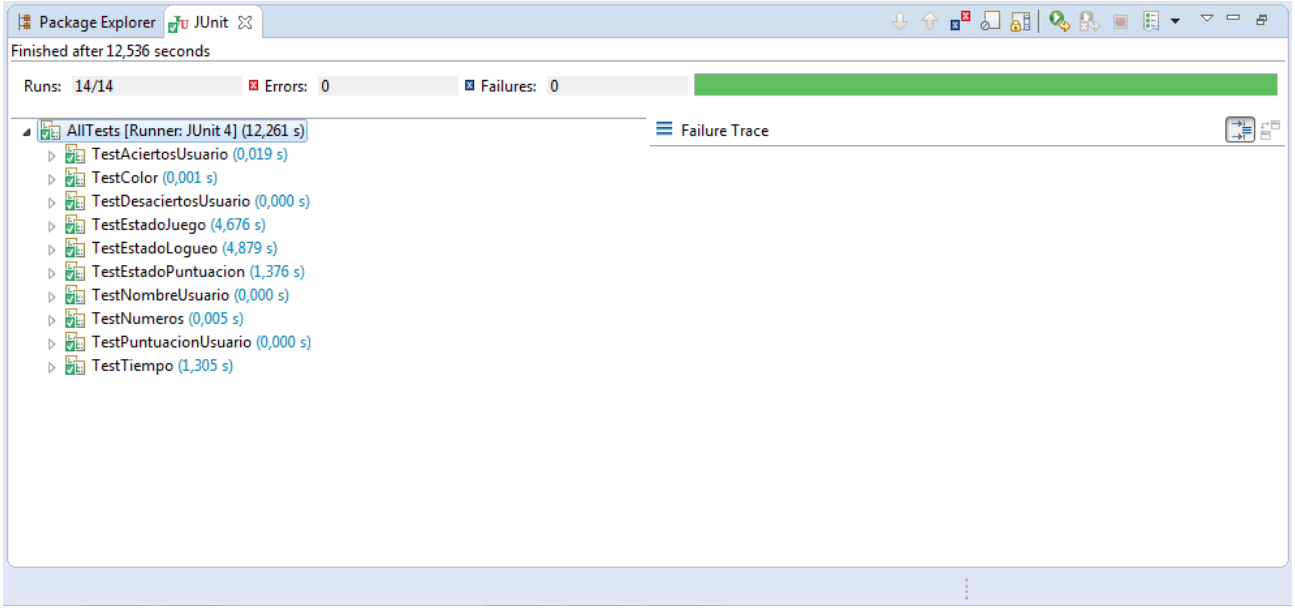
En este test unitario se prueba que la variable de puntuación incrementa cada vez que se completa la secuencia de números, ya sea, de forma incremental o decremental.

6.6. TestColor

En este test unitario se prueba que la variable de color indique cuando esta de color azul o de color naranja cada vez que se completa una secuencia de números, ya sea, de forma incremental o decremental.

6.7. AllTests

Para correr todos los test unitarios descritos anteriormente se encapsularon todos los test en un *JUnit Test Case Suite*. De esta manera cuando ejecutamos la clase *AllTests* el mismo se encarga de ejecutar todos los test que le hayamos indicado. A continuación mostramos una imagen:



Para ejecutar los test debemos iniciar *Eclipse*. Luego en el árbol de archivos del proyecto dirigirnos a la clase *AllTests.* Abrir la clase *AllTests*, ejecutar la clase y por último se observan los resultados mostrados en la imagen.

7. Pruebas de Integración

En esta apartado, se generaron en total 4 test de integración que se realizaron también como en el apartado anterior, con la herramienta *JUnit Test Case* provista por la *IDE Eclipse.* Estos test de integración son:

7.1. TestTiempo

En este test de integración se prueba por medio de la interacción de dos clases (modelo y controladorLogueo) que la variable de estado de puntuación vaya progresando hasta su último estado.

7.2. TestEstadoLogueo

En este test de integración se prueba que por medio de la interacción de dos clases (modelo y controladorLogueo) la variable de estado de logueo vaya progresando hasta su último estado.

7.3. TestEstadoJuego

En este test de integración como en el caso anterior se prueba por medio de la interacción de tres clases (modelo, controladorLogueo y controladorJuego) que la variable de estado de juego vaya progresando hasta su último estado.

7.4. TestEstadoPuntuacion

En este test de integración como en el caso anterior se prueba por medio de la interacción de cuatro clases (modelo, controladorLogueo, controladorJuego y controladorpuntuacion) que la variable de estado de puntuación vaya progresando hasta su último estado.

7.5. AllTests

Como en el caso de los test unitarios se tuvo en cuenta utilizar esta test (*AllTests*) para correr todas las pruebas.

8. Pruebas de Sistema

Teniendo en cuenta la generación de casos de pruebas del sistema que se hicieron en el *Documento De Requerimientos,* en este apartado se verificó el correcto funcionamiento de las mismas.

ID	Módulo a Probar	Pass/Fail
CP1	Barra de tiempo (temporizador)	**PASS**
CP2	Generación de matriz de números (Panel de números)	**PASS**
CP3	Señal indicativa de color (forma de ordenamiento ascendente o descendente).	**PASS**
CP4	Indicador de aciertos, desaciertos y nivel.	**PASS**
CP5	Ingreso del ID usuario	**PASS**
CP6	Finalización de partida	**PASS**
CP7	Tabla de puntuaciones	**PASS**
CP8	Presionar un número ya seleccionado previamente en la matriz de números generada.	**PASS**
CP9	Selección de un número incorrecto de la secuencia.	**PASS**
CP10	Descontar una unidad la barra de progreso de tiempo	**FAIL**
CP11	Inicio y Fin de partida.	**PASS**
CP12	Memoria Ram para el funcionamiento de la aplicación	**PASS**
CP13	Espacio libre en disco	**PASS**
CP14	Procesador mínimo	**PASS**
CP15	Tiempo límite inicial	**PASS**
CP16	Liberación de recursos al salir de la aplicación	**PASS**
CP17	Accesibilidad y legibilidad	**PASS**

9. Actualizado de Matriz de Trazabilidad

La matriz de trazabilidad que realizamos en el *Documento De Requerimientos* la actualizamos para anexar los test unitarios. Por lo tanto a continuación mostraremos la tabla generada:

	CU1	CU2	CU3	CU4	CU5	CP1	CP2	CP3	CP4	CP5	CP6	CP7	CP8	CP9	CP10	CP11	CP12	CP13	CP14	CP15	CP16	CP17	TU1	TU2	TU3	TU4	TU5	TU6
REQ F1	■					■																						
REQ F2	■						■																■					
REQ F3	■							■																				■
REQ F4	■								■																■	■	■	
REQ F5			■							■														■				
REQ F6											■																	
REQ F7												■																
REQ F8					■								■															
REQ F9				■										■														
REQ F10				■											■													
REQ F11	■	■														■												
REQ NO F1																	■											
REQ NO F2																		■										
REQ NO F3																			■									
REQ NO F4																				■								
REQ NO F5																												
REQ NO F6																						■	■	■	■	■	■	■
REQ NO F7		■																			■							
REQ NO F8																						■	■	■	■	■	■	■