***Manejo de las configuraciones***

**Introducción**

***Sobre este documento***

El presente documento muestra el plan de administración de las configuraciones para el proyecto denominado “Walkers of the city” que se realiza como trabajo práctico integrador de las materias Ingeniería de Software y Programación Concurrente. Se realiza con el fin de definir y dar a conocer un repositorio de versiones, la forma de acceso al código fuente y los documentos ubicados dentro del mismo, definir el equipo y las políticas generales de trabajo.

**Control de versiones**

Se utilizó el servicio de Project Hosting que ofrece Google Code para proyectos de software libre. El hosting se configuró para que trabaje con repositorios del sistema de control de versiones Subversion (svn).

El proyecto se puede acceder a través siguiente link:

<http://code.google.com/p/juego-walkers-of-the-city>/

Para acceder al servicio de google se utiliza el cliente TortoiseSVN. Se sincroniza el proyecto con los ordenadores de cada miembro haciendo un checkout desde el TortoiseSVN con la siguiente URL de repositorio: <https://juego-walkers-of-the-city.googlecode.com/svn/trunk/.> A partir de este punto cada miembro puede crear sus propias ramas del proyecto (branch) a partir de la rama principal (trunk).

**Esquema de directorios**

Los repositorios Subversion (svn) respetan un esquema de directorios determinado que organiza el proyecto para optimizar el desarrollo en paralelo del proyecto. El esquema de directorios de nuestro proyecto es el siguiente:

• svn/

doc/

branches/

tito/

marcos/

icaio/

tags/

trunk/

En la carpeta documentación se incluyen los documentos de diseño y referidos al código.

En branches se encuentran las carpetas que va a utilizar cada usuario para desarrollar nuevas ramas del proyecto (ver punto d).

En la carpeta tags están aquellos avances del proyecto principal que se consideren estables y se decidan etiquetar.

En trunk va el desarrollo de la rama principal del proyecto.

**Política de etiquetado**

Para el etiquetado vamos a utilizar el siguiente sistema:

tags/

internos/

i1.0.0

i1.0.1

releases/

Beta/

1.0.0/

1.0.1/

1.1.0/

Final/

1.0.2/

1.1.1/

En el subdirectorio “internos” colocaremos las versiones del proyecto estables para el uso interno en el proyecto, no orientadas a una entrega. Estas versiones no serán entregables al cliente puesto que aún no cumplen con todos los requerimientos exigidos por el mismo, pero cumplen con ciertas funcionalidades que los programadores desean fijar. Agregamos una i antes del número, por ejemplo: i1.0.0

El subdirectorio “releases” contiene las versiones entregables al cliente, que cumplen la gran mayoría o todos los requerimientos exigidos por el mismo para esa version del desarrollo. Este directorio se divide en dos subdirectorios más:

* Final en donde se encontrarán las versiones entregables que cumplen absolutamente todos los requerimientos del cliente y además están testeadas.
* Beta en donde se encontrarán las versiones entregables que cumplen algunos o todos los requerimientos del cliente y no están completamente testeadas. Las versiones ubicadas en este subdirectorio tienen el fin de mostrarle funcionalidades al cliente y son candidatas a convertirse en una versión final.

**Plan de Esquemas de ramas y Políticas de Fusión**

Para el presente trabajo hemos seleccionado como plan de esquemas de ramas, el conocido como UCM (unified configuration management) este método se basa en “rebase before deliver”.

La principal característica de este esquema es que antes de realizar un commit de un branch determinado en la carpeta trunks\ primero se verifica si se ha modifica el trunks del cual se realizó el branch en un primer momento. Si la versión del trunks fue modificada, el commit no puede realizarse directo, sino que primero debe realizarse un “rebase”, este consiste en actualizar nuestro branch y fusionarlo con el nuevo trunks, y luego realizar el commit.

Otra de las características de este esquema es que cada usuario posee su propio branch, de tal manera que cada uno trabaja en su propia rama.

**Entrega de releases.**

Se le entregará al cliente un ejecutable de los distintos releases de las versiones del proyecto a medida que se va desarrollando el mismo. Estas pueden ser tanto versiones finales como betas. Los códigos fuente de los releases entregables se encontrarán en el directorio /svn/tags/releases, tanto en el subdirectorio Final como en el Beta.

Además junto con cada release se le hará entrega de manuales de usuario en formato pdf tanto para el uso, como para la instalación y la configuración. Los archivos de configuración necesarios para ese release tambien seran entregados.

Para la entrega de versiones finales, todo lo mencionado anteriormente se grabará dentro de un DVD y se le entregará al cliente, ofreciendo adicionalmente servicio de soporte para la instalacion y configuracion del sistema. Para los fallos encontrados o reportados sobre una cierta versión se ofrecerán parches descargables desde internet.

Para la entrega de versiones betas, todo lo mencionado anteriormente se comprimirá y se subirá a un sitio web en donde sea descargable por el cliente, ofreciendo adicionalmente servicio de soporte online para la instalacion y configuracion del sistema.

La instalación de cada release no dependerá nunca de instalaciones de versiones anteriores, para que el cliente pueda obtener y utilizar cualquier versión independientemente de la que tenga.

**Integrantes y Roles**

Los integrantes del grupo son:

* Altman Quaranta, Augusto D.
* Diaz, Marcos
* Fretes, Ricardo G.

En caso de necesitar una comunicación entre los miembros estas se realizarán vía chat o video-llamada brindada por la pagina www.facebook.com , asi como también el uso de telefonia celular.

Tendremos dos cargos destacados, los cuales son:

* CCB (change control Board) estara a cargo del señor Altman Quaranta, Augusto y Fretes, Ricardo
* Presidente de la CCB estará a cargo del señor Diaz, Marcos

Estos cargos están divididos por su responsabilidad, la responsabilidad de cada uno esta dado por:

* CCB: Toma las decisiones si los cambios a un proyecto deben ser implementados o no. Las decisiones tomadas por el change control board son tomadas como definitivas y obligatorias
* Presidente de la CCB: Establece dias y horarios de cuando se realizarán las reuniones correspondientes y dirige las mismas.

Las reuniones se realizarán 2 veces por semana los dias jueves y viernes en el horario de 15 hs, definido por el presidente de la CCB.

**Herramienta de seguimiento de bugs**

Para el seguimiento de errores e inconvenientes vamos a utilizar la herramienta que nos proporciona el repositorio de Google Code, para reportarlos y posteriormente hacer un seguimiento de los mismos (ver el estado, quien lo reporto, si fue resuelto, por quien, etc.).

La direccion del repositorio de issues es la siguiente:

<http://code.google.com/p/juego-walkers-of-the-city/issues/list>

***Requerimientos***

**Introducción**

***Sobre este documento***

En esta sección se describe la especificación de los requerimientos para el sistema “Walkers of thecity”, juego multijugador en línea desarrollado como trabajo práctico final de las materias Programación Concurrente e Ingeniería del Software. Se provee una descripción de alto nivel del sistema, mostrando los requerimientos de usuario y de sistema y haciendo una distinción entre los requerimientos funcionales y no funcionales del mismo.

***Glosario***

A continuación se listan una serie de términos con sus respectivas definiciones que serán utilizados con frecuencia a lo largo del documento. Se recomienda familiarizarse con los términos para evitar confusiones al leer el presente.

Partida: Una instancia del juego compuesta por una lista de jugadores que participan en ella.

Partida creada: (Createdgame) Es un estado en el que se puede encontrar una partida. Es una partida no comenzada pero lista para iniciar, cuyos parámetros iniciales han sido determinados por el Game Master.

Crear partida: (Creategame) Proceso para instanciar una partida creada

Partida iniciada: (Startedgame) Es un estado en el que se puede encontrar una partida. Partida creada que ha comenzado a desarrollarse.

Iniciar partida: (Startgame) Proceso para instanciar una partida iniciada.

Unirse a una partida: (Joingame) Dada una partida creada un jugador pasa a formar parte de la lista de jugadores que van a participar de dicha partida cuando se inicie.

Jugador: (Player) Un participante que desarrolla el juego dentro de una partida desde un client.

Game Master: Jugador que ha determinado los parámetros iniciales de una partida (creador de una partida). Es el primer jugador que se une a la partida.

Client: Subsistema de software perteneciente al sistema “Walkers of thecity” que hace las veces de cliente en un esquema cliente-servidor proveyendo conectividad con el server.

Server: Subsistema de software perteneciente al sistema “Walkers of thecity” que hace las veces de servidor en un esquema cliente-servidor proveyendo toda la información necesaria, el estado y la lógica del juego.

Graphic Interface: Subsistema de software perteneciente al sistema “Walkers of thecity” cuya función es mostrarle al jugador el estado del juego y proveer un método de acceso para cambiar dicho estado.

Mapa: (Mapa de la ciudad) Mapa virtual que los jugadores deben recorrer. Está compuesto por calles, veredas, autos y repositorios.

Repositorio: Punto específico del mapa a donde el jugador debe llevar la palabra encontrada en la sopa de letras.

***Descripción general del sistema***

El sistema “Walkers of thecity” es un juego multijugador en línea que se desarrolla en partidas creadas por los usuarios.

Cada partida es básicamente una carrera entre los jugadores participantes, que se compone de dos etapas que deben completarse en orden. La primera etapa consiste en buscar una palabra determinada en un sopa de letras. Una vez completada la primera etapa, el jugador accede a la segunda que consiste en llevar la palabra encontrada a través de una ciudad hasta un repositorio. La ciudad consiste de un mapa transitado por autos y por los demás jugadores que hayan completado la primera etapa. Una vez completada esta última etapa el juego termina y el jugador debe esperar a que el resto termine o se cumpla el tiempo límite.

El ganador será el primero en llevar su palabra al repositorio.

**Definición de requerimientos de usuario**

***Requerimientos iniciales***

A continuación se listan una serie de requerimientos de usuario que son tomados como punto de partida para el desarrollo del sistema “Walkers of thecity”.

Dichos requerimientos son el producto de sucesivas entrevistas con el cliente, el Ingeniero Orlando Micolini, quien firmó, junto con nosotros los desarrolladores, un documento que detalla la misma al haber llegado a un acuerdo.

1. Todo usuario del sistema debe ser capaz de conectarse al servidor para crear la partida.
2. Al crearse la partida debe esperarse que se unan la cantidad de jugadores que haya establecido el usuario que creó la partida.
3. La partida iniciará cuando: se hayan unido tantos jugadores como lo indicó el Game Master al crear la partida, o cuando el Game master lo decida.
4. Al comenzar la partida el tiempo general de juego comienza a correr, se le entrega a cada jugador la palabra a buscar, las coordenadas del repositorio y la sopa de letras.
5. El tiempo máximo que puede durar una partida es de 10 minutos.
6. Debe existir un método por el cual el usuario pueda señalar o indicar al sistema la posición de la palabra que le corresponde buscar.
7. Cuando un jugador resuelve correctamente la de la sopa de letras se lo debe ubicar en el mapa y mostrarselo.
8. Los usuarios deben poder moverse a través del mapa y, solo en las esquinas, poder cruzar la calle hacia una de las 2 direcciones posibles.
9. En el mapa los semáforos y vehículos deben ser controlados por el sistema, cumpliendo que el 5% de los autos no respetan a los semáforos, pudiendo chocar a los peatones (usuario-jugador)
10. En el mapa Los cruces de calles se establecen en algunas áreas para dar prioridad a los autos (80%) y en otras para dar prioridad a los peatones (20%)
11. Si un jugador es atropellado debe comenzar en el punto de inicio correspondiente en el mapa.
12. Los autos no colisionan entre sí.
13. Los autos son 50, estos tienen una velocidad al menos 10 veces superior a los peatones, y en las boca calles pueden seguir de largo, doblar derecha e izquierda
14. No puede haber inanición.
15. Al llegar un jugador al repositorio termina su partida, quedándose en una pantalla de espera con la tabla de posiciones hasta el momento, y marcando su tiempo con respecto al tiempo general de juego.
16. El ganador es el que haya llevado la palabra al repositorio en el menor tiempo, y las posiciones se van ocupando a medida que los jugadores entregan las palabra en el repositorio.
17. Una vez llegados a los 10 min de tiempo general de juego, la partida termina instantáneamente para todos los jugadores. Conformándose la tabla general de juego final y mostrándola a todos los jugadores .

**Especificación del sistema**

***Diagrama de casos de uso***

Para completar y extender la información que proporciona el anterior diagrama se han confeccionado las tarjetas de casos de uso para los mostrados.

Las tarjetas de casos de uso se presentan en la siguiente sección.

***Tarjetas de casos de uso***

1) Caso de uso: Unirse a la partida

|  |  |  |
| --- | --- | --- |
| **Nombre:** | Unirse a partida |  |
| **Actor:** | Usuario |  |
| **Descripción:** | el usuario se agrega a una partida creada y queda a la espera de empezar la misma |  |
| **Flujo Principal:** | **Eventos ACTOR** | **Eventos SISTEMA** |
|  | 1. El usuario peticiona unirse a una partida aleatoria |  |
|  |  | 2. Realiza la conexión necesaria |
|  |  | 3.Le muestra al usuario la partida con la conexión establecida. o el error en la conexión |
| **Precondición:** |  |  |
| **Poscondición:** | Usuario con informe de conectividad y a la espera del inicio de la misma |  |

2) Caso de uso: Crear la partida

|  |  |  |
| --- | --- | --- |
| **Nombre:** | Crear partida |  |
| **Actor:** | Usuario |  |
| **Descripción:** | El usuario crea la partida |  |
| **Flujo Principal:** | **Eventos ACTOR** | **Eventos SISTEMA** |
|  | 1.Se conecta al servidor |  |
|  | 2.Solicita crear partida determinando en esta solicitud la cantidad de jugadores máxima y otros parametros |  |
|  |  | 3.Crea la partida con sopa de letras, mapa, e iniciando tiempo general de juego en 0 min 0 seg |
|  | 4.Espera por otros jugadores. |  |
| **Flujo alternativo:** | En paso 4, si se completa el numero de usuarios conectados, el sistema inicia la partida,enviando nombre a buscar, mapas y sopas de letras a los clientes. pone a correr el contador |  |
| **Precondición:** | El sistema tiene un servidor habilitado |  |
| **Poscondición:** | Partida creada y espera por otros jugadores, o bien partida iniciada |  |

3) Caso de uso: Iniciar la partida

|  |  |  |
| --- | --- | --- |
| **Nombre:** | Iniciar partida |  |
| **Actor:** | Usuario |  |
| **Descripción:** | El usuario que creo inicia la partida. |  |
| **Flujo Principal:** | **Eventos ACTOR** | **Eventos SISTEMA** |
|  | 1.Solicita iniciar partida |  |
|  |  | 2.El servidor inicia la partida, enviando nombre a buscar, mapas y sopas de letras a los clientes. pone a correr el contador |
| **Precondición:** | El usuario creo la partida |  |
| **Poscondición:** | Comienzo del juego |  |

4) Caso de uso: Abandonar la partida

|  |  |  |
| --- | --- | --- |
| **Nombre:** | Abandonar partida |  |
| **Actor:** | Usuario |  |
| **Descripción:** | El usuario en medio de una partida, decide abandonarla |  |
| **Flujo Principal:** | **Eventos ACTOR** | **Eventos SISTEMA** |
|  | 1. El usuario realiza el abandono a una partida |  |
|  |  | 2. Marca registro de abandono del jugador . |
|  |  | 3.Realiza desconexión del servidor |
|  |  | 4. Muestra pantalla de abandono de partida |
| **Precondición:** | Partida creada y en ejecución |  |
| **Poscondición:** | Usuario con informe de abandono. |  |

5) Caso de uso: Jugar con sopa de letras

|  |  |  |
| --- | --- | --- |
| **Nombre:** | Jugar con sopa de letras |  |
| **Actor:** | Usuario |  |
| **Descripción:** | El usuario busca el nombre en la sopa de letras y lo manda al servidor |  |
| **Flujo Principal:** | **Eventos ACTOR** | **Eventos SISTEMA** |
|  | 1.El usuario busca el nombre en la sopa de letra. |  |
|  | 2.Entrega de las posiciones inicial y final donde se encuentra el nombre encontradas. |  |
|  |  | 3.validez de respuesta, si es incorrecta vuelta al punto 2. , si es correcta , lo lleva a ciudad |
| **Precondición:** | Partida creada e iniciada con sopa de letras y nombre a buscar cargado en el cliente |  |
| **Poscondición:** | El usuario pasa a jugar a la ciudad |  |

6) Caso de uso: Jugar en la ciudad

|  |  |  |
| --- | --- | --- |
| **Nombre:** | Jugar en la ciudad |  |
| **Actor:** | Usuario |  |
| **Descripción:** | El usuario se mueve por el mapa de la ciudad para llevar el nombre al repositorio |  |
| **Flujo Principal:** | **Eventos ACTOR** | **Eventos SISTEMA** |
|  | 1. El usuario se mueve por el mapa |  |
|  |  | 2. Realiza verificación de movimiento y acciones que desencadena |
|  |  | 3.Informa el movimiento del usuario en el mapa |
|  | 4. Vuelta al punto 1 |  |
| **Precondición:** | El usuario completo el juego de la sopa de letra. |  |
|  | Se le muestra el mapa al usuario con su posicion y posicion de repositorio |  |
|  | El usuario se encuentra en el punto inicial del mapa |  |
| **Poscondición:** | Usuario con el informe de posición actual en el mapa |  |

7) Caso de uso: Crear y configurar semáforos

|  |  |  |
| --- | --- | --- |
| **Nombre:** | Completar partida |  |
| **Actor:** | Usuario |  |
| **Descripción:** | El usuario completa la partida, cumpliendo con los objetivos de ambos juegos antes de que transcurra el tiempo límite. |  |
| **Flujo Principal:** | **Eventos ACTOR** | **Eventos SISTEMA** |
|  | 1. El usuario llega al repositorio |  |
|  |  | 2. Informa al usuario que ha completado la partida satisfactoriamente. |
|  |  | 3. Muestra tabla de resultados parciales. |
| **Precondición:** | El usuario completo el juego de la sopa de letra. |  |
|  | El usuario está jugando en la ciudad. |  |
| **Poscondición:** | Usuario solo viendo la tabla de resultados parciales. |  |

8) Caso de uso: Crear y configurar semáforos.

|  |  |  |
| --- | --- | --- |
| **Nombre:** | Crear y configurar semáforos |  |
| **Actor:** | Semáforos |  |
| **Descripción:** | Se crean dos semáforos relacionados por cada cruce del mapa y se configuran adecuadamente. |  |
| **Flujo Principal:** | **Eventos ACTOR** | **Eventos SISTEMA** |
|  |  | 1. Se crea un par de semáforos relacionado para un cruce determinado del mapa. |
|  |  | 2. Se lo configura. |
|  |  | 3. Se lo mapea. |
|  |  | 4. Se vuelve al punto 1 hasta que se cumpla la poscondición. |
| **Precondición:** | No se ha iniciado la partida. |  |
| **Poscondición:** | Hay un par de semáforos por cada cruce del mapa. |  |

9) Caso de uso: Control de semáforos por cruce.

|  |  |  |
| --- | --- | --- |
| **Nombre:** | Control de semáforos por cruce. |  |
| **Actor:** | Semáforos |  |
| **Descripción:** | Se controlan los semáforos de a pares en cada cruce del mapa. |  |
| **Flujo Principal:** | **Eventos ACTOR** | **Eventos SISTEMA** |
|  |  | 1. En cada cruce del mapa el semáforo que estaba en verde se pone a rojo. |
|  |  | 2. El otro semáforo del mismo cruce del mapa se pone en verde. |
|  |  | 3. Cada cierto tiempo se vuelve al punto 1. |
| **Precondición:** | Hay al menos un Usuario jugando sobre el mapa de la ciudad. |  |
| **Poscondición:** |  |  |

10) Caso de uso: Crear y configurar autos.

|  |  |  |
| --- | --- | --- |
| **Nombre:** | Crear y configurar autos |  |
| **Actor:** | Autos |  |
| **Descripción:** | Se crean y se configuran todos los autos del sistema. |  |
| **Flujo Principal:** | **Eventos ACTOR** | **Eventos SISTEMA** |
|  |  | 1. Se crea un auto. |
|  |  | 2. Se le configura la velocidad, posición inicial y otros elementos de importancia. |
|  |  | 3. Se vuelve al punto 1 hasta que se cumpla la poscondición. |
| **Precondición:** | No se ha iniciado la partida. |  |
| **Poscondición:** | Se han creado todos los autos necesarios del sistema. |  |

***Diagramas de actividad***

Diagrama general del sistema:

Especifica el desarrollo del juego desde el comienzo hasta el final.

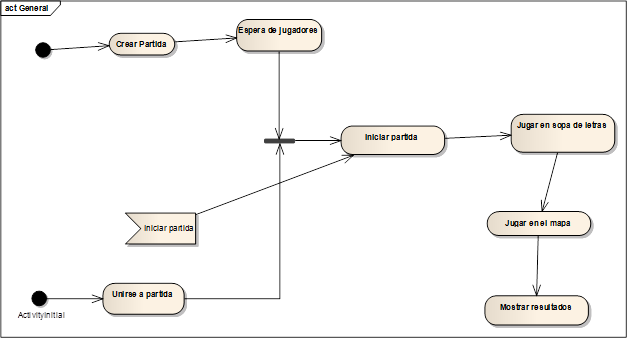


Diagrama de juego sobre el mapa:

Especifica el desarrollo de la segunda etapa del juego que se lleva a cabo sobre el mapa de la ciudad.

***Diagramas de secuencia***

Crear partida / Inicio automático:

Especifica el escenario donde un usuario crea una partida definiendo un número máximo de jugadores. Los jugadores se van uniendo hasta que se alcanza el número máximo permitido (especificado por el Game master), punto en el cual comienzo la partida.



Crear partida / Inicio no automático: Especifica el escenario donde un usuario crea una partida definiendo un número máximo de jugadores. La partida, sin embargo, comienza antes de que se alcance ese número, ya que el Game master dio la orden.



Juego sobre la sopa de letras: Especifica el desarrollo de la primera etapa del juego en donde el jugador debe encontrar una palabra determinada dentro de una sopa de letras.



Movimientos sobre mapa: Especifica el desarrollo de la segunda etapa del juego en donde el jugador se mueve sobre el mapa de la ciudad.



Refresco de interfaz gráfica: Especifica los sucesivos refrescos de la interfaz gráfica que se dan en escenarios donde el jugador está jugando sobre la ciudad, permitiendo que el mismo contenga un estado actualizado del mapa.



Movimiento de un auto: Especifica como el sistema hace que un auto se desplace por la ciudad sin chocar con los demás. Muestra la presencia de monitores que funcionan en base al funcionamiento de una red de Petri.



*Figure:* Dynamic View

**Requerimientos**

***Análisis de requerimientos***

Gracias al análisis de requerimientos que se hizo en las secciones anteriores es posible generar (se descubren nuevos), ordenar y clasificar los mismos, para luego verificar su consistencia (que ningún requerimiento se contradiga con otro) y su validez (que todo el sistema propuesto cumpla con los requerimientos iniciales que impone el cliente) a través de un estudio de los mismos.

A partir de este estudio hemos obtenido los siguientes requerimientos funcionales y no funcionales que se detallan en las siguientes dos secciones.

## Modelo de requerimientos



Requerimientos funcionales



*Figure:* Functional Requirements

Partidas



*Figure:* Partidas

|  |  |  |  |
| --- | --- | --- | --- |
| REQ 1.1.1 Crear la partida | | | |
| *«Functional»* | *Status:*Proposed | *Priority:*Medium | *Difficulty:*Medium |
| *Phase: 1.0* | *Version: 1.0* |  |
| Todo usuario del sistema debe ser capaz de conectarse al servidor para crear la partida. | | |

|  |  |  |  |
| --- | --- | --- | --- |
| REQ 1.1.10 Ganador y tabla de posiciones | | | |
| *«Functional»* | *Status:*Proposed | *Priority:*Medium | *Difficulty:*Medium |
| *Phase: 1.0* | *Version: 1.0* |  |
| El ganador es el que haya llevado la palabra al repositorio en el menor tiempo, y las posiciones se van ocupan a medida que los jugadores entregan la palabra en el repositorio. | | |

|  |  |  |  |
| --- | --- | --- | --- |
| REQ 1.1.2 Establecer la capacidad de jugadores que admite la partida | | | |
| *«Functional»* | *Status:*Proposed | *Priority:*Medium | *Difficulty:*Medium |
| *Phase: 1.0* | *Version: 1.0* |  |
| Al crearse la partida, el Game Master establece la cantidad maxima de jugadores que admite esa partida. | | |

|  |  |  |  |
| --- | --- | --- | --- |
| REQ 1.1.3 Cuando se inicia una partida | | | |
| *«Functional»* | *Status:*Proposed | *Priority:*Medium | *Difficulty:*Medium |
| *Phase: 1.0* | *Version: 1.0* |  |
| La partida iniciara cuando el Game Master lo decida o cuando la cantidad de jugadores que se han unido a la partida alcance el limite maximo establecido para la misma. | | |

|  |  |  |  |
| --- | --- | --- | --- |
| REQ 1.1.4 Que ocurre al iniciar una partida | | | |
| *«Functional»* | *Status:*Proposed | *Priority:*Medium | *Difficulty:*Medium |
| *Phase: 1.0* | *Version: 1.0* |  |
| Cuando se inicia una partida el tiempo general de juego comienza a correr, se le entrega a cada jugador la palabra a buscar, la sopa de letras y las corrdenadas del repositorio. | | |

|  |  |  |  |
| --- | --- | --- | --- |
| REQ 1.1.5 Unirse a una partida | | | |
| *«Functional»* | *Status:*Proposed | *Priority:*Medium | *Difficulty:*Medium |
| *Phase: 1.0* | *Version: 1.0* |  |
| Todo usuario del sistema debe ser capaz de conectarse al servidor para unirse a una partida creada. | | |

|  |  |  |  |
| --- | --- | --- | --- |
| REQ 1.1.6 Abandonar una partida | | | |
| *«Functional»* | *Status:*Proposed | *Priority:*Medium | *Difficulty:*Medium |
| *Phase: 1.0* | *Version: 1.0* |  |
| Todo usuario del sistema debe contar con la posibilidad de retirarse de una partida en cualquier momento sin interferir con el juego de los demas jugadores de la misma partida. | | |

|  |  |  |  |
| --- | --- | --- | --- |
| REQ 1.1.7 Que ocurre cuando un jugador termina una partida | | | |
| *«Functional»* | *Status:*Proposed | *Priority:*Medium | *Difficulty:*Medium |
| *Phase: 1.0* | *Version: 1.0* |  |
| Cuando un jugador termina una partida se le muestra al mismo la tabla de resultados parciales hasta el momento y se la mantiene actualizada. | | |

|  |  |  |  |
| --- | --- | --- | --- |
| REQ 1.1.8 Cuando un jugador termina una partida | | | |
| *«Functional»* | *Status:*Proposed | *Priority:*Medium | *Difficulty:*Medium |
| *Phase: 1.0* | *Version: 1.0* |  |
| Un jugador termina una partida cuando completa la segunda etapa del juego, llevando la palabra al repositorio correspondiente. | | |

|  |  |  |  |
| --- | --- | --- | --- |
| REQ 1.1.9 Cuando finaliza una partida | | | |
| *«Functional»* | *Status:*Proposed | *Priority:*Medium | *Difficulty:*Medium |
| *Phase: 1.0* | *Version: 1.0* |  |
| Una partida finaliza completamente cuando todos los jugadores han terminado la partida o cuando haya transcurrido el tiempo general de juego. | | |

Ciudad



*Figure:* Ciudad

|  |  |  |  |
| --- | --- | --- | --- |
| REQ 1.3.2 Responsabilidades del sistema | | | |
| *«Functional»* | *Status:*Proposed | *Priority:*High | *Difficulty:*High |
| *Phase: 1.0* | *Version: 1.0* |  |
| En el mapa los semaforos y veiculos deben ser controlados por el sistema. | | |

|  |  |  |  |
| --- | --- | --- | --- |
| REQ 1.3.1 Interfaz | | | |
| *«Functional»* | *Status:*Proposed | *Priority:*Medium | *Difficulty:*Medium |
| *Phase: 1.0* | *Version: 1.0* |  |
| Los usuarios deben poder moverse a traves del mapa y, solo en las esquinas, poder cruzar la calle hacia una de las 2 direcciones posibles. | | |

|  |  |  |  |
| --- | --- | --- | --- |
| REQ 1.3.10 Cuando se gana la etapa | | | |
| *«Functional»* | *Status:*Proposed | *Priority:*Medium | *Difficulty:*Medium |
| *Phase: 1.0* | *Version: 1.0* |  |
| Al llegar un jugador al repositorio se completa la segunda etapa satisfactoriamente, y por el REQ 1.1.8 el jugador termina la partida. | | |

|  |  |  |  |
| --- | --- | --- | --- |
| REQ 1.3.11 Cuando un jugador es atropellado | | | |
| *«Functional»* | *Status:*Proposed | *Priority:*High | *Difficulty:*High |
| *Phase: 1.0* | *Version: 1.0* |  |
| La situacion de choque entre un jugador y un auto se da cuando, en un instante dado, un auto pasa por una posicion que estaba siendo ocupada por un jugador (roba el recurso) o un jugador intenta pasar por una posicion que esta siendo ocupada por un auto (intento de robo de recurso). | | |

|  |  |  |  |
| --- | --- | --- | --- |
| REQ 1.3.3 Prioridad en los cruces | | | |
| *«Functional»* | *Status:*Proposed | *Priority:*Medium | *Difficulty:*Medium |
| *Phase: 1.0* | *Version: 1.0* |  |
| En un 80% de las esquinas existentes, cuando un semaforo esta en verde en un instante dado, los autos pueden pasar por una posicion que esta siendo ocupada por un jugador (puede robarle el recurso) y en el 20% restante esto no se permite. | | |

|  |  |  |  |
| --- | --- | --- | --- |
| REQ 1.3.4 Autos / Autos | | | |
| *«Functional»* | *Status:*Proposed | *Priority:*Medium | *Difficulty:*High |
| *Phase: 1.0* | *Version: 1.0* |  |
| El sistema debera controlar que los autos no colisionen entre si. | | |

|  |  |  |  |
| --- | --- | --- | --- |
| REQ 1.3.5 Autos / Semaforos | | | |
| *«Functional»* | *Status:*Proposed | *Priority:*Medium | *Difficulty:*Medium |
| *Phase: 1.0* | *Version: 1.0* |  |
| El 5% de los autos no respetan a los semaforos, pudiendo chocar a los peatones. | | |

|  |  |  |  |
| --- | --- | --- | --- |
| REQ 1.3.6 Que ocurre cuando un jugador es atropellado | | | |
| *«Functional»* | *Status:*Proposed | *Priority:*High | *Difficulty:*Low |
| *Phase: 1.0* | *Version: 1.0* |  |
| Si un jugador es atropellado debe comenzar en el punto de inicio correspondiente en el mapa. | | |

|  |  |  |  |
| --- | --- | --- | --- |
| REQ 1.3.7 Cantidad de autos | | | |
| *«Functional»* | *Status:*Proposed | *Priority:*Medium | *Difficulty:*Low |
| *Phase: 1.0* | *Version: 1.0* |  |
| El sistema debera generar exactamente 50 autos. | | |

|  |  |  |  |
| --- | --- | --- | --- |
| REQ 1.3.8 Velocidad de autos | | | |
| *«Functional»* | *Status:*Proposed | *Priority:*Low | *Difficulty:*Low |
| *Phase: 1.0* | *Version: 1.0* |  |
| El sistema debera controlar que los autos tengan una velicodad al menos 10 veces superior a los peatones. | | |

|  |  |  |  |
| --- | --- | --- | --- |
| REQ 1.3.9 Movimientos de los autos | | | |
| *«Functional»* | *Status:*Proposed | *Priority:*High | *Difficulty:*High |
| *Phase: 1.0* | *Version: 1.0* |  |
| En cada curce de calle del mapa los autos pueden doblar a la izquierda o a la derecha o seguir de largo. El sistema sera quien decida el camino que siguen. | | |

Sopa de letras



*Figure:* Sopa de letras

|  |  |  |  |
| --- | --- | --- | --- |
| REQ 1.2.1 Interfaz | | | |
| *«Functional»* | *Status:*Proposed | *Priority:*High | *Difficulty:*Medium |
|  | *Phase: 1.0* | *Version: 1.0* |  |
|  | Debe existir un metodo por el cual el usuario pueda señalar o indicar al sistema la posicion de la palabra que le corresponde buscar. Ademas de un sistema a traves del cual el jugador pueda visualizar la sopa de letras. | | |
| REQ 1.2.2 Que ocurre al resolver la sopa de letras | | | |
| *«Functional»* | *Status:*Proposed | *Priority:*Medium | *Difficulty:*Medium |
|  | *Phase: 1.0* | *Version: 1.0* |  |
|  | Cuando un jugador resuelve correctamente la sopa de letras pasa a la siguiente etapa del juego. Se lo debe ubicar en el mapa y mostrárselo. | | |

|  |  |  |  |
| --- | --- | --- | --- |
| REQ 1.2.3 Contenido de la sopa de letras | | | |
| *«Functional»* | *Status:*Proposed | *Priority:*High | *Difficulty:*Medium |
| *Phase: 1.0* | *Version: 1.0* |  |
| La sopa de letras que se le proporciona al jugador al iniciar la partida debe contener dentro de ella el palabra que el jugador tiene que encontrar. | | |

|  |  |  |  |
| --- | --- | --- | --- |
| REQ 1.2.4 Cuando se resuleve la sopa de letras | | | |
| *«Functional»* | *Status:*Proposed | *Priority:*Medium | *Difficulty:*Medium |
| *Phase: 1.0* | *Version: 1.0* |  |
| La sopa de letras se resuelve cuando el jugador indica donde se encuentra la palabra que está buscando. | | |

#### Cliente



*Figure:* Cliente

|  |  |  |  |
| --- | --- | --- | --- |
| Interfaz grafica | | | |
| *«Functional»* | *Status:* Proposed | *Priority:* Medium | *Difficulty:* Medium |
| *Phase: 1.0* | *Version: 1.0* |  |
| El cliente SOLO muestra la interfaz grafica y no tiene nada de la lógica del juego ni los datos del mismo. | | |

#### Servidor



*Figure:* Servidor

|  |  |  |  |
| --- | --- | --- | --- |
| Control de logica de autos y semaforos | | | |
| *«Functional»* | *Status:* Proposed | *Priority:* Medium | *Difficulty:* Medium |
| *Phase: 1.0* | *Version: 1.0* |  |
| Todo el control de la lógica de los autos y los semáforos se ejecutara en el servidor. | | |

|  |  |  |  |
| --- | --- | --- | --- |
| Resguardo de datos de todo el juego en general | | | |
| *«Functional»* | *Status:* Proposed | *Priority:* Medium | *Difficulty:* Medium |
| *Phase: 1.0* | *Version: 1.0* |  |
| El servidor será el responsable de contener y mantener actualizados todos los datos del juego, ya sea los datos de partidas, de la ciudad y de la sopa de letras. | | |

Requerimientos no funcionales



*Figure:* Non-Functional Requirements

Concurrencia



*Figure:* Concurrencia

|  |  |  |  |
| --- | --- | --- | --- |
| REQ 2.1.2 Inanicion | | | |
| *«Performance»* | *Status:*Proposed | *Priority:*High | *Difficulty:*High |
| *Phase: 1.0* | *Version: 1.0* |  |
| No puede haber inanición entre hilos. | | |

|  |  |  |  |
| --- | --- | --- | --- |
| REQ 2.1.3 Interbloqueos | | | |
| *«Functional»* | *Status:*Proposed | *Priority:*High | *Difficulty:*High |
| *Phase: 1.0* | *Version: 1.0* |  |
| Para que el sistema funcione correctamente no debe haber interbloqueos entre hilos. | | |

General



*Figure:* General

|  |  |  |  |
| --- | --- | --- | --- |
| REQ 2.1.1 Duracion maxima de una partida | | | |
| *«Performance»* | *Status:*Proposed | *Priority:*Medium | *Difficulty:*Medium |
| *Phase: 1.0* | *Version: 1.0* |  |
| El tiempo general de juego nunca puede ser mayor a 10 minutos. | | |

***Matriz de trazabilidad entre casos de uso y requerimientos***

Para simplificar la matriz se asignan los siguientes acrónimos a los casos de uso:

* Crear partida: UC1
* Unirse a partida: UC2
* Iniciar partida: UC3
* Abandonar partida: UC4
* Jugar con sopa de letras: UC5
* Jugar en la ciudad: UC6
* Completar partida: UC7
* Definir parámetros de partida: UC8
* Indicar dónde se encuentra la palabra: UC9
* Moverse por la ciudad: UC10
* Ver tabla de resultados: UC11

Partidas:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **REQ** | **UC1** | **UC2** | **UC3** | **UC4** | **UC5** | **UC6** | **UC7** | **UC8** | **UC9** | **UC10** | **UC11** |
| **1.1.1** | **x** |  |  |  |  |  |  |  |  |  |  |
| **1.1.2** | **x** |  |  |  |  |  |  | **x** |  |  |  |
| **1.1.3** |  |  | **x** |  |  |  |  | **x** |  |  |  |
| **1.1.4** |  |  | **x** |  |  |  |  |  |  |  |  |
| **1.1.5** |  | **x** |  |  |  |  |  |  |  |  |  |
| **1.1.6** |  |  |  | **x** |  |  |  |  |  |  |  |
| **1.1.7** |  |  |  |  |  |  | **x** |  |  |  | **x** |
| **1.1.8** |  |  |  |  |  |  | **x** |  |  |  |  |
| **1.1.9** |  |  |  |  |  |  | **x** |  |  |  |  |
| **1.1.10** |  |  |  |  |  |  | **x** |  |  |  | **x** |

Sopa de letras:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **REQ** | **UC1** | **UC2** | **UC3** | **UC4** | **UC5** | **UC6** | **UC7** | **UC8** | **UC9** | **UC10** | **UC11** |
| **1.2.1** |  |  |  |  | **x** |  |  |  | **x** |  |  |
| **1.2.2** |  |  |  |  | **x** |  |  |  |  |  |  |
| **1.2.3** |  |  |  |  | **x** |  |  |  |  |  |  |
| **1.2.4** |  |  |  |  | **x** |  |  |  | **x** |  |  |

Ciudad:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **REQ** | **UC1** | **UC2** | **UC3** | **UC4** | **UC5** | **UC6** | **UC7** | **UC8** | **UC9** | **UC10** | **UC11** |
| **1.3.1** |  |  |  |  |  | **x** |  |  |  | **x** |  |
| **1.3.2** |  |  |  |  |  | **x** |  |  |  |  |  |
| **1.3.3** |  |  |  |  |  | **x** |  |  |  |  |  |
| **1.3.4** |  |  |  |  |  | **x** |  |  |  |  |  |
| **1.3.5** |  |  |  |  |  | **x** |  |  |  |  |  |
| **1.3.6** |  |  |  |  |  | **x** |  |  |  |  |  |
| **1.3.7** |  |  |  |  |  | **x** |  |  |  |  |  |
| **1.3.8** |  |  |  |  |  | **x** |  |  |  |  |  |
| **1.3.9** |  |  |  |  |  | **x** |  |  |  |  |  |
| **1.3.10** |  |  |  |  |  | **x** | **x** |  |  |  |  |
| **1.3.11** |  |  |  |  |  | **x** |  |  |  |  |  |

***Matriz de trazabilidad entre requerimientos***

(ver aparte)

**Arquitectura preliminar del sistema**



***Arquitectura***

**Introducción**

***Sobre este documento***

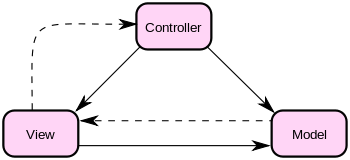
En esta sección se describe la arquitectura del sistema “Walkers of the city”, juego multijugador en línea desarrollado como trabajo práctico final de las materias Programación Concurrente e Ingeniería del Software. Se provee una descripción de alto nivel del sistema, mostrando los subsistemas que lo integran y los componentes de cada subsistema mediante diferentes diagramas. Se define un patrón de organización para la estructura del sistema y a partir de allí se continúa con el diseño de la arquitectura basado en esa estructura.

**Patrón de diseño**

***Descripción del patrón elegido***

El patrón de diseño elegido es el MVC (Model View Controller), un modelo organizacional compuesto por 3 capas que se relacionan entre sí. Cada capa tiene una responsabilidad bien definida, y el objetivo de las mismas es separar la parte la lógica, los datos y la presentación.

El modelo general respeta el siguiente esquema:



Se distinguen 3 subsistemas en el diagrama anterior. Cada uno, como se aclaro arriba, cumple con una función bien definida.

* Model (modelo): Este subsistema mantiene el estado de la aplicación. Contiene los datos principales del sistema con sus valores actualizados.
* Controller (controlador): Este subsistema responde a los eventos disparados desde la vista y se ocupa de toda la lógica que se le aplica a los datos. Es el encargado de cambiar el estado del modelo. También puede hacerle peticiones a la vista.
* View (vista): Este subsistema responde a los eventos disparados por el usuario. Se ocupa de la presentación del modelo en un formato adecuado para interactuar con el usuario.

***Justificación de la elección***

Dados los requerimientos obtenidos en la sección B de este proyecto, necesitamos un sistema en donde el estado (los datos y sus valores) de una partida específica esté compartido y sea accesible por todos los jugadores participantes en ella, ya que los mismos se mueven sobre el mismo mapa, todos deben poder ver el resultado de los demás, etc. Esto implica que la información necesariamente debe estar centralizada y todos los jugadores deben poder acceder a ella en el momento que lo requieran. A su vez existe toda una lógica de control, que define el desarrollo del juego interactuando con el estado de la partida, conoce el formato de los datos y sabe tratar con ellos. Esta lógica es la misma para todos los jugadores.

Por otra parte el estado de una partida debe cambiar cuando los jugadores interactúan con el juego a través de una interfaz, pero toda interacción de los jugadores con los datos debe respetar la lógica del juego.

Es factible pensar que se podría centralizar toda la lógica de control del juego como un subsistema aparte, ya que su existencia es claramente necesaria, todos los jugadores utilizan la misma lógica y no forma parte de los datos, solo se remite a interactuar con los mismos. Luego, bajo este esquema, toda interacción jugador/interfaz debe disparar ciertos eventos en la lógica de control que cambien de manera apropiada los datos de la partida.

Por último, sabemos por lo dicho arriba (y por los requerimientos) que todos los jugadores deben poder acceder al estado de la partida en la cual participan, en el momento que lo requieran. Esto trae aparejado dos consecuencias: una es que debe existir una forma de mostrar este estado al usuario de manera apropiada, es decir debe existir una interfaz con la cual el jugador pueda interactuar; la otra consecuencia es que para acceder al estado no es necesario ninguna lógica intermedia, ya que solo se requiere información y no se modifica nada, por ende el acceso a los datos puede ser directo.

Al elegir un patrón de diseño para la arquitectura de nuestro sistema, vemos que se adapta a la perfección con un MCV. Los requerimientos que definen nuestro sistema nos han llevado a concluir que una estructura organizacional del mismo que respete un patrón MCV

es la indicada para encarar este desarrollo.

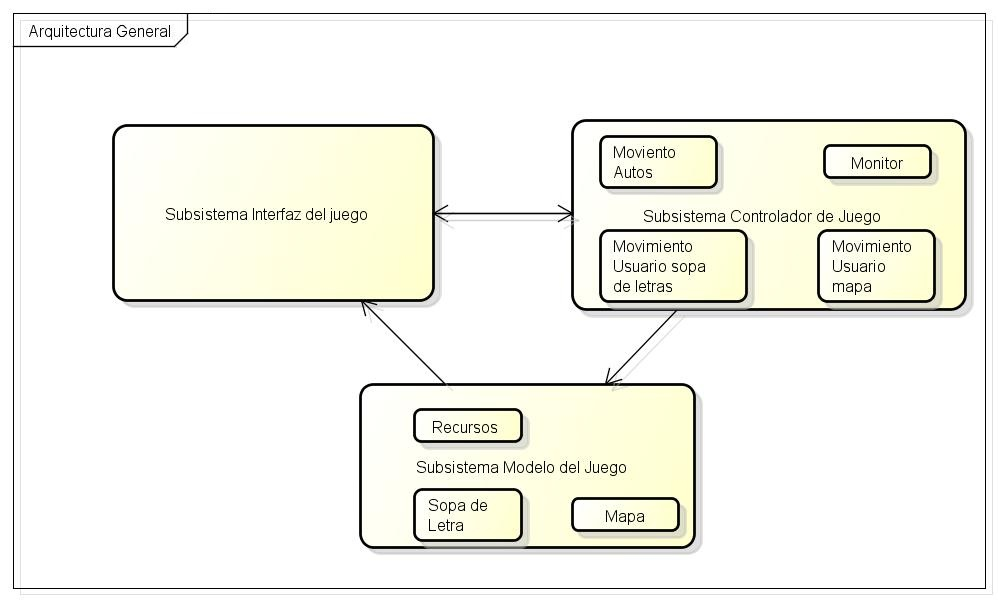
En la elección del modelo también se debatió la posibilidad de utilizar el modelo de capas ( con 3 capas - capa de presentación, control y modelo), con el cual también era posible abordar el problema. Hemos seleccionado el MCV en consecuencia de que el modelo de capas, señala que una capa presta servicios a las consiguientes, cosa que para nuestro caso, pensando en lo que se dijo anteriormente, no es necesario. Esto se ve claramente en el caso en que para mostrar la interfaz no es necesario ninguna lógica, por lo que, si se realizara con un modelo de capas, el control debería intervenir en estas situaciones, no siendo necesario. Por otro lado, si se analiza con más detenimiento, en la capa de control, se realizan diferentes procesos, pudiendo aplicar un modelo multicapa de esta misma, pero no es conveniente ni lógico ya que estos subsistemas de control, no se prestan servicios entre ellos.

El modelo elegido (MVC), además, nos permite facilitar la resolución de los requerimientos no funcionales de la siguiente forma:

* Para el requerimiento 2.1.1 (Duración máxima de una partida) al tener el controlador como una unidad aparte nos permite llevar una cuenta de todo el juego separada de cada jugador, y así podemos llevar una cuenta unificada.
* Para los requerimientos 2.1.2 (inanición) y 2.1.3 (interbloqueos), es mucho más fácil realizar esto si tenemos los componentes necesarios (como los monitores) agrupados en el mismo subsistema, en este caso en el controlador.
* Para el 2.1.6 (responsabilidad del cliente) vemos que tener el subsistema vista separado e independiente nos facilita mucho la tarea de que el cliente puede mostrar el estado del modelo actualizado y, además, la forma en que el jugador actúa sobre el sistema es más clara y ordenada al tener por un lado la interfaz o vista y por otro el controlador que efectúe la acción en el modelo.
* Para el 2.1.5 (ejecución alojada en el servidor), al tener los datos y la lógica de control independiente, es más simple alojarlo en el servidor, y que los clientes hagan solicitudes al mismo, de este modo se centraliza el funcionamiento del juego.

***Modelo de Arquitectura***

Este es el gráfico de arquitectura general en el cual nos basaremos para la realizacion del software solicitado.



***Diagrama de componentes***



***Diagrama de Desplazamiento***

***Diseño e Implementación***

**Introducción**

***Sobre este documento***

Aquí vamos a mostrar los diagramas de clase, primero separados en los tres subsistemas principales: Controlador, Modelo y Vista; y luego mostrando la relación entre las principales clases de cada uno de estos.

**Especificación de los subsistemas**

***Subsistema controlador***

Diagrama de clases



Creación y Unirse



Aquí podemos ver como es la interacción necesaria para la creación y para unirse a partidas.

**Movimiento\_jugador\_mapa** -



Aquí se observa las interacciones que se realizan cuando un jugador se mueve en la etapa 2 del juego, cuando se encuentra en el mapa.

**Verificar\_palabra** - *(Sequence diagram)*



Este diagrama muestra el proceso que se realiza cuando se recibe un mensaje para la verificación de la ubicación de la palabra en la sopa de letras.

***Subsistema vista***

Diagrama de clases



Interacción entre el cliente y las primeras etapas del juego



En este diagrama se especifica cómo son las interacciones entre el cliente y las primeras pantallas del juego, hasta llegar a la etapa de la sopa de letras. Además se puede observar como es el proceso de conexión y el paso de mensajes entre el cliente y el servidor, y como a partir de estos mensajes el cliente va cambiando las pantallas que muestra.Interacción entre el cliente y y el juego en la etapa de la ciudad



En este diagrama se especifica cómo son las interacciones entre el cliente y las ultimas pantallas del juego.

***Subsistema modelo***

Diagrama de clases



***Relaciones principales entre subsistemas***

Diagrama de clases



Como podemos observar, utilizamos el patrón Observer en este diagrama, entre la clase ControladorCambios (del subsistema controlador) y la clase Observer (de interfaz de juego), con el fin de mantener actualizado el estado del juego en cada cliente (en cualquiera de sus etapas y con todos sus elementos). Para que, de esta forma, pueda ser visualizado en la pantalla de todos los usuarios que estén jugando le juego con la información mas actualizada.

Utilizamos este patrón ya que veíamos que en nuestro sistema existía la posibilidad de que el servidor tuviera que actualizar el mapa de la ciudad a varios clientes, y el patrón Observer facilita esta tarea. Simplemente se suscribe al usuario correspondiente a la lista que tiene ControladorCambios. El método notify de este se encarga de ejecutar el método update correspondiente a cada objeto de la clase Observer (que tiene cada cliente). En caso que cierto cliente haya terminado la partida, se lo retira de la lista y deja de recibir actualizaciones del mapa.

***Testing***

**Introducción**

***Sobre este documento***

En esta sección se listan una serie de pruebas de regresión que se le aplican al sistema “Walkers of the city” para verificar su correcto funcionamiento. Para realizar dichas pruebas se aplico Unit testing a través de la librería JUnit de java. Aunque muchos de los test mostrados no están implementados, se listan para que se implementen cuando sea el momento; ya que se considera que son esenciales para mantener un mínimo de control sobre el funcionamiento del sistema.

**Modelo general de testing con JUnit**

***Template genérico***

Para hacer Unit Testing con JUnit desarrollamos el siguiente template que nos sirve como modelo general para desarrollar los casos de test, además de especificar ciertas convenciones de escritura y de programación.

**import** junit.framework.TestCase;

**public** **class** test\_[NOMBRE\_CLASE] **extends** TestCase {

[OBJETOS NECESARIOS PARA REALIZAR EL TEST]

**public** **void** setUp() {

[INSTACIAR LOS OBJETOS NECESARIOS]

[LLEVAR LAS INSTACIAS AL ESTADO ADECUADO]

}

**public** **void** test[NOMBRE\_METODO]() {

[PROCESO DE TESTING]

[USAR ALGUN TIPO DE *ASSERT*]

}

}

***Tipos de assert***

* assertEquals(expected, actual)
* assertEquals(message, expected, actual)
* assertEquals(expected, actual, delta) - used on doubles or floats, where delta is the difference in precision
* assertEquals(message, expected, actual, delta) - used on doubles or floats, where delta is the difference in precision
* assertFalse(condition)
* assertFalse(message, condition)
* assertNotNull(object)
* assertNotNull(message, object)
* assertNotSame(expected, actual)
* assertNotSame(message, expected, actual)
* assertNull(object)
* assertNull(message, object)
* assertSame(expected, actual)
* assertSame(message, expected, actual)
* assertTrue(condition)
* assertTrue(message, condition)

**Unit Testing**

***Caso N***

Objeto del test: [que se está testeando]

Estado: [implementado / no implementado]

Descripción del test: [como se está testeando (en lo referente a la implementación)]

Condiciones de ejecución: [condiciones previas a la ejecución del test]

Resultado: [éxito / falla]

**Unit Testing**

**A continuación se presentan los diferentes casos de testing realizados:**

Caso 1

Objeto del test: Conexión tcp de parte del cliente al servidor Localhost y puerto 5000(estado de conexión de un objeto de la clase Conexión)

Estado: implementado

Descripción del test: se testea la dirección del puerto de origen del socket correspondiente, de ser 0 es porque no se produjo la conexión.

Condiciones de ejecución: Se debe tener un puerto abierto por parte del servidor

Resultado: Falla (no esta habilitado el puerto)

**import** junit.framework.TestCase;

**public** **class** test\_Conexion **extends** TestCase {

Conexion c;

**public** **void** setUp()

{

c = **new** Conexion();

}

**public** **void** testconectar() {

//fail("Not yet implemented");

c.conectar("localhost", 5000);

*assertEquals*("no hay conexion", 0, c.getSocket().getPort());

}

/\*De no producirse la conexion el puerto en el socket queda como 0

getPort() Returns:

the remote port number to which this socket is connected,

or 0 if the socket is not connected yet.\*/

}

***Caso 2***  
Objeto del test: Se comprueba que se cree una nueva partida en espera.  
Estado: Implementado.  
Descripción del test: Luego de ejecutar el método agregar\_partida\_en\_espera de la clase ControladorMasterServer se verifica que el objeto devuelto por el hashMap con la key correspondiente no sea NULL.  
Condiciones de ejecución: Ninguna.  
Resultado: Falla por falta de implementación de la clase ControladorMasterServer

**import** junit.framework.TestCase;  
  
**public** **class** test\_ControladorMasterServer **extends** TestCase {

ControladorMasterServer a;

**public** **void** setUp() {

   a = new ControladorMasterServer(7170); //puerto = 7170

Sopa\_letras sopa = new Sopa\_letras();  
 sopa.asignar\_palabra(1);  
 ControladorSopaDeLetras cs= new ControladorSopaDeLetras(sopa);  
 ControladorMovimientoUsuario cm = new ControladorMovimientoUsuario(new mapa\_ciudad());  
 Partida nueva = new Partida(cs,cm,new Jugador(1), 1); //id = 1 → partida

   a.agregar\_partida\_en\_espera(nueva);

   }  
  
   **public** **void** testAgregar\_partida\_en\_espera() {

   assertNotNull(“El hashMap no deberia devolver una referencia nula a la key correspondiente”, a.get\_partidas\_en\_espera().get(1));

   }  
}

***Caso 3***  
Objeto del test: Se comprueba que se verifique correctamente la palabra ingresada por un jugador.  
Estado: No implementado.  
Descripción del test: Luego de que el jugador ingrese un par de coordenadas correctas se espera que el metodo comprobarPalabra de la clase ControladorSopaLetras retorne true.  
Condiciones de ejecución: Las coordenadas ingresadas son correctas.  
Resultado: Falla por falta de implementación de las clases involucradas.

***Caso 4***  
Objeto del test: Se comprueba que un auto luego de moverse en línea recta no se haya salido de los valores permitidos en el mapa  
Estado: No Implementado.  
Descripción del test: Luego de ejecutar el método move\_forward de la clase auto, se verifica que los valores de sus coordenadas hayan aumentado en 1 en la dirección correspondiente.  
Condiciones de ejecución: El auto estaba sobre una calle (en donde es correcto su paso).  
Resultado: Falla por falta de implementación.  
  
  
***Caso 5***  
Objeto del test: Conexión tcp de parte del cliente al servidor Localhost y puerto 5000(estado de conexión de un objeto de la clase Conexión)  
Estado: implementado  
Descripción del test: se testea la dirección del puerto de destino del socket correspondiente, de ser 0 es porque no se produjo la conexión.  
Condiciones de ejecución: Se debe tener un puerto abierto por parte del servidor  
Resultado: Falla (no esta habilitado el puerto)  
  
**import** junit.framework.TestCase;  
**public** **class** test\_Conexion **extends** TestCase {  
   Conexion c;  
   **public** **void** setUp()  
   {  
         c = **new** Conexion();  
   }  
   **public** **void** testconectar() {  
         //fail("Not yet implemented");  
         c.conectar("localhost", 5000);  
         *assertEquals*("no hay conexion", 0, c.getSocket().getPort());  
   }

/\*De no producirse la conexion el puerto en el socket queda como 0  
   getPort() Returns:  
         the remote port number to which this socket is connected,  
   or 0 if the socket is not connected yet.\*/  
        
}

***Caso 6***  
Objeto del test: Se comprueba que el jugador y el ControlCambios se encuentran agregados al hashmap correspondiente de MasterServer y se cree correctamente el AnalizadorMaster  
Estado: Implementado.  
Descripción del test: Se crea una instancia de AnalizadorMaster a traves del metodo ControladorMasterServer::crear\_analizador() y se controla que los campos del mismo sean distintos de null e iguales a los correspondientes en el hashmap de ControladorMasterServer  
Condiciones de ejecución: Ninguna  
Resultado: Falla por falta de implementación de la clase ControladorMasterServer

**import** junit.framework.TestCase;  
**public** **class** test\_ControladorMasterServer **extends** TestCase {

ControladorMasterServer a;

Jugador jugador;

ControlCambios control;

ControladorMovimientoJugador cm;

ControladorSopadeLetras cs;

string comando;

**public** **void** setUp() {

   a = new ControladorMasterServer(7170); //puerto = 7170

Sopa\_letras sopa = new Sopa\_letras();  
 sopa.asignar\_palabra(1);  
 ControladorSopaDeLetras cs= new ControladorSopaDeLetras(sopa);  
 ControladorMovimientoUsuario cm = new ControladorMovimientoUsuario(new mapa\_ciudad());

jugador=new Jugador(cs,cm,1,”192.168.0.2”);// partida id =1

comando = “arriba”;

control = new ControlCambios();

   }  
   **public** **void** test() {  
 AnalizadorMaster am;

   a.controlador\_notificadores.put(1,control);

a.mapeo\_ip\_jugador.put(jugador.get\_ip(),jugador);

am = a.crear\_analizador(jugador, comando, control);

// comprobación de nulidad de los objetos

assertNotNull(“el campo AnalizadorMaster::jugador no debería ser nulo”, am.get\_jugador());

assertNotNull(“el campo AnalizadorMaster::cambios no debería ser nulo”, am.get\_ControlCambios());

// comprobación de igualdad entre instancias de objetos

    assertEquals(“El jugador que posee el AnalizadorMaster debería ser el mismo que posee el ControladorMasterServer según el ip recibido”, a.mapeo\_ip\_jugador.get(jugador.get\_ip()), am.get\_jugador());  
 assertEquals(“La instancia de ControlCambios que posee el AnalizadorMaster debería ser el mismo que posee el ControladorMasterServer según el ip recibido”, a.controladores\_notificadores.get(1), am.get\_cambios());   }

}

***Release Note***

**Walkers of the city 1.0.0 Release Note**  
  
**Breve descripción**  
Primer entrega del ejecutable del juego que consiste en encontrar una palabra en una sopa de letras y llevarla a un repositorio de palabras en una ciudad evitando ser chocado por los autos.  
   
  
**Datos de la Entrega**  
  
**Publicación: 1.0.0**  
Fecha de la nota y publicación: 18 de diciembre de 2012  
  
  
  
**Requerimientos del Sistema**  
Sistema Operativo: Cualquiera que tenga instalado Java(TM) 2 SDK & Java Runtime Environment.  
  
**Funcionalidad Incluida**  
La funcionalidad actual del sistema consta de la simulación de 20 autos que se interceptan en una esquina, donde podemos apreciar que no hay colisiones (en la esquina), ni inanición de los autos esperando para cruzar. Todo esto acompañado de una primitiva interfaz gráfica que permite apreciar lo anterior  
  
**Impactos sobre el usuario final**  
Permite al usuario final apreciar cómo va a ser la interfaz y también ver cómo va a ser el movimiento de los autos por el mapa.  
  
**Notas**  
Con este release esperamos obtener la devolución por parte de los clientes sobre posibles cambios a realizar (puede ser de la interfaz o de la velocidad de los autos), así también sobre posibles bugs encontrados, todo esto para poder realizar un mejor producto final.

***Estimaciones y Datos Históricos***

A continuación realizaremos una estimación de los tiempos empleados por los integrantes para la realización del trabajo:  
  
Hubo una etapa previa de investigación y estudio de las materias Ingeniería de Software y Programación Concurrente, con el fin de realizar un trabajo más ordenado.

**Tiempo estimado 5\*3=15 horas.**  
  
Plan de manejo de las configuraciones: todo el grupo decidio y luego aprendio a utilizar las herramientas para el manejo.

**Tiempo estimado 4\*3=12 horas.**  
  
Requerimientos: Todo el grupo discutio esta parte.

**Tiempo estimado 10\*3=30 horas.**  
  
Arquitectura: Tambien realizado y discutido por todos los integrantes.

**Tiempo estimado 3\*3=9 horas.**  
  
Diseño e Implementación: Realizado por todo el grupo.

**Tiempo estimado 15\*3=45horas.**  
  
Pruebas Unitarias y de Sistemas.

**Tiempo estimado 6\*3=18 horas.**  
  
Entrega, Estimaciones y datos históricos, e información adicional:

**Tiempo estimado 3\*3=9horas.**  
  
**Tiempo total estimado: 138 horas.**

***Información adicional***

**Informe Trabajo Final de la materia Programación concurrente**  
Por archivo separado se adjunta el informe con el correspondiente trabajo final realizado para la materia Programación concurrente, con consigna dada en el año 2010.  
En esta entrega de información adicional se entrega:

* Informe final
* Código relativo a la concurrencia en una esquina
* Código relativo a la concurrencia en una esquina modificado con presentación de control de errores por pantalla en runtime

**Lecciones aprendidas**  
Es meritorio comenzar diciendo que los 3 integrantes del grupo estamos de acuerdo en decir que con este trabajo hemos aprendido la altísima utilidad e importancia de la ingeniería de software en la planificación y desarrollo de programas.  
Es trivial decir que se aprendió el manejo de las herramientas necesarias así como también el propósito de cada tipo de diagrama.  
Creemos y estamos convencidos que una de las lecciones que más valor agregado nos aporta, es aquella en donde se entiende y se ve a los diagramas como una herramienta de descripción del software, encontrando cuáles son los puntos críticos a focalizar y reconocer con qué tipo de diagrama se representa de manera más representativa según el enfoque de dicho punto. Esto es un conocimiento que vimos como evolución a lo largo del trabajo, debido que en un primer momento se veía a los diagramas como una consigna a cumplir y a través de la evolución del trabajo y el aprendizaje nuestro, se convirtió en una herramienta **necesaria** para poder representar, diseñar y comprender el software a desarrollar.  
Por otro lado, este trabajo requirió el aprendizaje de herramientas de sistemas versionado, así como el trabajo en paralelo, lo cual constituyó una experiencia nueva y conocimientos de estas herramientas así como también las comunicaciones y la utilización de estas, con respecto a los demás integrantes del grupo. Fue otra de las lecciones que merecen ser tenidas en cuenta.

Diaz, Marcos

Fretes, Ricardo G.

Altman Quaranta,   
Augusto C.