ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN.
UNIVERSIDAD POLITÉCNICA DE CATAGENA.

PROYECTO FIN DE CARRERA

DISTRIBUCIÓN DE INTERFAZ GRÁFICA SEGÚN PATRÓN MVC USANDO JAVA RMI.

AUTOR: Manuel Esteban Juárez.
DIRECTOR: Juan Ángel Pastor Franco.
09 – 2008

Autor	Manuel Esteban Juárez
E-mail del Autor	manueldelosreyes@gmail.com
Director(es)	Juan Ángel Pastor Franco
E-mail del Director	juanangel.pastor@upct.es
Título del PFC	Distribución de interfaz grafica según patrón **MVC** usando Java **RMI**.
Descriptores	Ingenieria del software, patrones de diseño
Resúmen Este Proyecto Fin de Carrera trata de distribuir una aplicación para la simulación y comunicación con una maqueta de trenes. Se parte de un Proyecto anterior en el cual no se había podido conseguir distribuir la aplicación. Tras un estudio previo de las razones y posibles soluciones a este problema, se opta por aplicar patrones de diseño de la interfaz gráfica que faciliten su distribución: patrones Modelo-Vista-Controlador y Flyweight. Tras esto se rediseña la aplicación y se reimplementa de acuerdo al nuevo diseño.	
Titulación	Ingeniero Técnico de Telecomunicación, Especialidad Telemática.
Departamento	Tecnologías de la Información y las Comunicaciones
Fecha de Presentación	09 - 2008

Índice:

1. **Introducción** **4**
    1.1 Motivación 4
    1.2 Objetivos 4
    1.3 Entorno de desarrollo 5
        1.3.1 Eclipse 5
        1.3.2 Maqueta 5
        1.3.3 Java RMI 7
2. **Requisitos** **12**
    2.1 Introducción 12
    2.2 Requisitos de partida 12
    2.3 Capacidad de distribución 14
3. **Diseño del software** **16**
    3.1 Introducción 16
    3.2 Uso de middleware 16
    3.3 Patrones de diseño 17
        3.3.1 MVC 17
        3.3.2 Flyweight 18
    3.4 Implementación 21
4. **Paquetes y librerías** **26**
    4.1 Introducción 26
    4.2 Diferentes paquetes desarrollados 26
5. **Descripción de la aplicación** **29**
    5.1 Introducción 29
    5.2 Visión estática del código 29
        5.2.1 Los Nodos 29
        5.2.2 El tren 33
        5.2.3 Los controladores 36
        5.2.4 Las vistas 37
        5.2.5 Las clases de Inicios 42
    5.3 Visión dinámica del código 44
6. **Instalación y uso del software** **47**
    6.1 Introducción 47
    6.2 Instalación de la aplicación 47
    6.3 Uso de la aplicación 50
7. **Conclusiones y líneas de acción futuras** **53**
    7.1 Resultados obtenidos 53
    7.2 Conclusiones, aportaciones y líneas de acción futura 53

Bibliografía y referencias **55**

Capítulo 1.

Introducción

1.1 Motivación.

Este Proyecto Fin de Carrera surge de la necesidad de distribuir una interfaz gráfica dada usando para ello la tecnología Java[2] RMI y siguiendo el patrón de diseño MVC (Modelo – Vista – Controlador)[4]. Se parte de un Proyecto Fin de Carrera anterior, realizado por Jose Andrés Martínez Muñoz [3], en el cual se diseñaba e implementaba una aplicación para la simulación y comunicación con una maqueta de trenes.
Las características de dicha aplicación hacen muy difícil utilizar el mecanismo RMI[6][7] de Java para su distribución debido a un alto número de objetos altamente cohesionados, de forma que es complicado hacer una distribución de objetos compatible con el patrón MVC. Por ello, se plantea rediseñar la aplicación para que esa distribución sea posible reduciendo el número de objetos y optimizando las relaciones entres ellos.
La idea principal es construir un Servidor que creará y pondrá a disposición de los Clientes una instancia del simulador remoto de la maqueta y de los trenes que circularán por ella. Los Clientes podrán crear su propia vista de la maqueta y manejar los trenes haciendo uso de la información y los métodos remotos proporcionados por el Servidor. En última instancia, este será el encargado del control de los trenes y de comunicar a los Clientes los cambios realizados para que puedan actualizar su vista de la maqueta.
Todo esto se hará de forma coherente con el patrón MVC, separando claramente el modelo (simulador de la maqueta) de su control y representación (controlador y vista).

1.2 Objetivos.

Los objetivos que aborda este Proyecto Fin de Carrera son:

1. Encontrar un patrón de diseño de la interfaz gráfica que facilite su distribución.
2. Rediseñar la aplicación en consecuencia.
3. Reimplementar la aplicación de acuerdo con el nuevo diseño.

1.3 Entorno de desarrollo.

1.3.1 Eclipse

La implementación del Proyecto se ha realizado con el IDE Eclipse[1]. Eclipse es, en el fondo, únicamente un armazón sobre el que se pueden montar herramientas de desarrollo para cualquier lenguaje, mediante la implementación de los *plugins* adecuados. La arquitectura de *plugins* de Eclipse permite, además de integrar diversos lenguajes sobre un mismo IDE, introducir otras aplicaciones accesorias que pueden resultar útiles durante el proceso de desarrollo, como herramientas UML, editores visuales de interfaces, ayuda en línea para librerías, etc.
Eclipse ha proporcionado una perfecta organización y estructuración del código, localización de errores, y una interfaz muy amigable de utilización, tanto para el desarrollo como para la realización de otras prestaciones como es el *Javadoc*.

1.3.2 Maqueta.

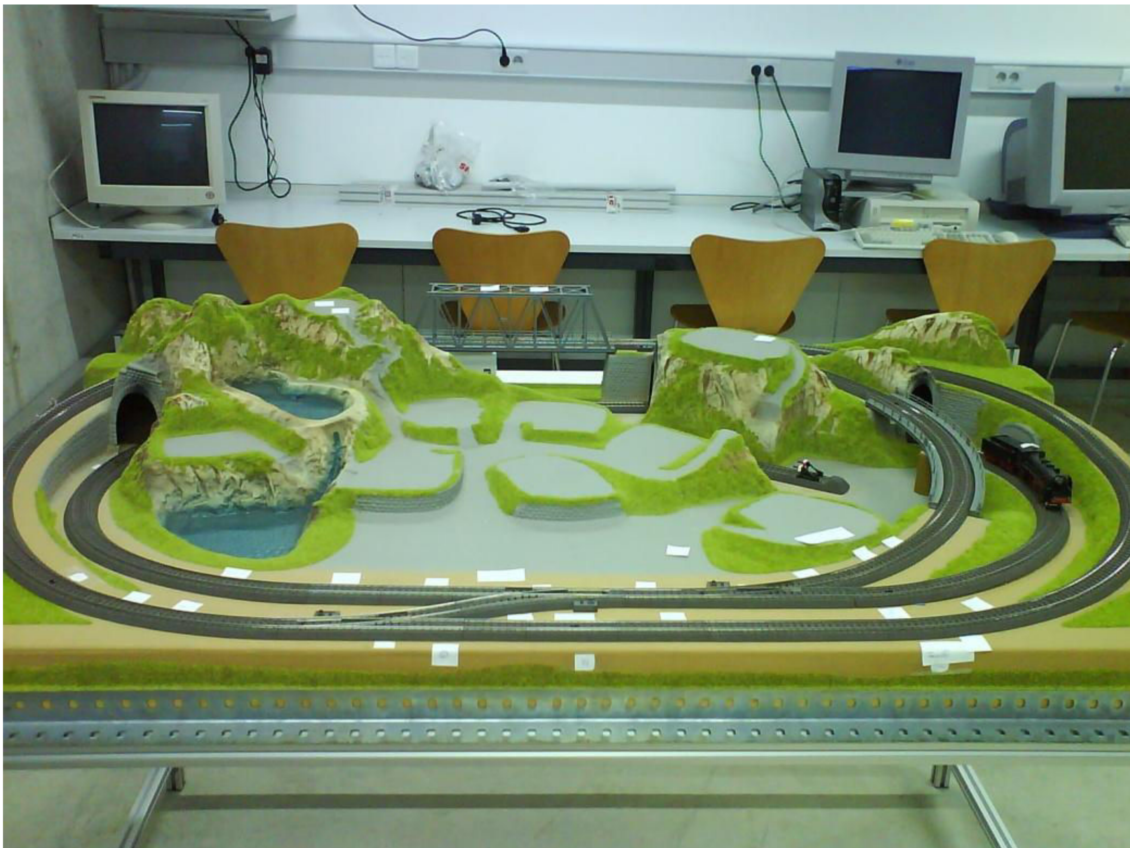


Figura 1: Maqueta de trenes. Laboratorio DSIE.

En la figura 1 se observa la maqueta que se encuentra en el laboratorio DSIE (División de Sistemas e Ingeniería Electrónica) en el edificio Cuartel de Antigones de la UPCT. Su instalación actual incluye cinco desvíos o agujas y una docena de patines o sensores de paso.

La maqueta está formada por un conjunto de elementos simples conectados para formar circuitos cerrados. Todos ellos son digitales y se pueden controlar mediante una consola central de mando o mediante la interfaz del puerto serie de una computadora.

Se realiza una clasificación de estos elementos presentes en la maqueta del laboratorio:

1. *Tramos de vía rectos.* De estos, cabe diferenciar, sin medir su longitud, entre los que tienen un sensor o patín de vía y los que no. Adicionalmente, hay algunos otros disponibles en catálogo, como vías de desenganche, etc., aunque no estén presentes en el laboratorio.

2. *Tramos de vía curvos.* De nuevo, se debe distinguir entre los que tienen sensor o los que no.

3. *Desvíos.* Pueden darse tanto entre un tramo recto y otro curvo, como entre dos tramos curvos.

4. *Vagones del tren.* Se diferencia entre vagones con motor (locomotoras) y vagones destinados a transporte de pasajeros o mercancías. En el Simulador, se ha diferenciado a los trenes que hay sobre la maqueta por el número de vagones que representa cada uno.

1.3.3 Java RMI.

RMI (*Java Remote Method Invocation*)[6][7] es un mecanismo ofrecido en Java para invocar un método remotamente. Al ser RMI parte estándar del entorno de ejecución Java, usarlo provee un mecanismo simple de distribución que solamente necesita comunicar servidores codificados para Java.

La invocación de métodos remotos permite que un objeto que se ejecuta bajo una máquina virtual de Java pueda invocar métodos de otro objeto en otra máquina virtual diferente. Con RMI se invocan métodos sobre objetos remotos, se pueden pasar objetos como parámetros y se retornan objetos.

Las aplicaciones RMI normalmente comprenden dos programas separados: un servidor y un cliente. Una aplicación servidor típica crea un montón de objetos remotos, hace accesibles unas referencias a dichos objetos remotos, y espera a que los clientes llamen a estos métodos u objetos remotos. Una aplicación cliente típica obtiene una referencia remota de uno o más objetos remotos en el servidor y llama a sus métodos. RMI proporciona el mecanismo por el que se comunican y se pasan información del cliente al servidor y viceversa. Cuando es una aplicación, algunas veces nos referimos a ella como *Aplicación de Objetos Distribuidos*.

Las aplicaciones de objetos distribuidos necesitan:

Localizar Objetos Remotos
Las aplicaciones pueden utilizar uno de los dos mecanismos para obtener referencias a objetos remotos. Puede registrar sus objetos remotos con la facilidad de nombrado de RMI **rmiregistry**. O puede pasar y devolver referencias de objetos remotos como parte de su operación normal.

Comunicar con Objetos Remotos
Los detalles de la comunicación entre objetos remotos son manejados por el RMI; para el programador, la comunicación remota se parecerá a una llamada estándar a un método Java.

Cargar Bytecodes para objetos que son enviados.
Como RMI permite al llamador pasar objetos Java a objetos remotos, RMI proporciona el mecanismo necesario para cargar el código del objeto, así como la transmisión de sus datos.

La arquitectura RMI puede verse como un modelo de cuatro niveles o capas, las cuales se explican a continuación:

1. Nivel de Aplicación

Se corresponde con la implementación real de las aplicaciones cliente y servidor. En esta capa se desarrollan todas las llamadas a alto nivel para acceder y exportar objetos remotos. Cualquier aplicación que quiera que sus métodos estén accesibles a clientes remotos debe declarar dichos métodos en una interfaz que extienda *java.rmi.Remote*. Dicha interfaz se usa para definir un objeto como remotamente accesible. Una vez que los métodos han sido implementados, el objeto debe ser exportado. Esto se puede hacer de forma implícita si el objeto extiende la clase *UnicastRemoteObject*, o puede hacerse de forma explícita con una llamada al método *exportObject( )*, situados ambos en el paquete *java.rmi.server*.

2. Nivel de resguardo/esqueleto

Es la que interactúa directamente con la capa de aplicación. Todas las llamadas a objetos remotos y acciones junto con sus parámetros y retorno de objetos tienen lugar en esta capa. También puede llamarse capa proxy o capa stub-skeleton.

3. Nivel de gestión de referencias

Es la responsable del manejo de referencias de las invocaciones remotas. También es responsable de la gestión de la replicación de objetos y realización de tareas específicas de la implementación con los objetos remotos. En esta capa se espera una conexión de tipo *stream* (stream-oriented connection) desde la capa de transporte.

4. Nivel de transporte

Se encarga de establecer las conexiones necesarias y establecer las comunicaciones necesarias para el transporte de los datos de una máquina a otra. El protocolo de transporte subyacente para RMI es JRMP (Java Remote Method Protocol), solamente "comprendido" por programas Java.

En la figura 2 se observa un esquema simplificado de lo anteriormente explicado, estableciéndose la correspondencia de cada capa con su nivel OSI adecuado. Se distinguen las capas que sigue esta arquitectura con las número 7, 6, 5 y 4, respectivamente, o también por el color salmón de fondo.

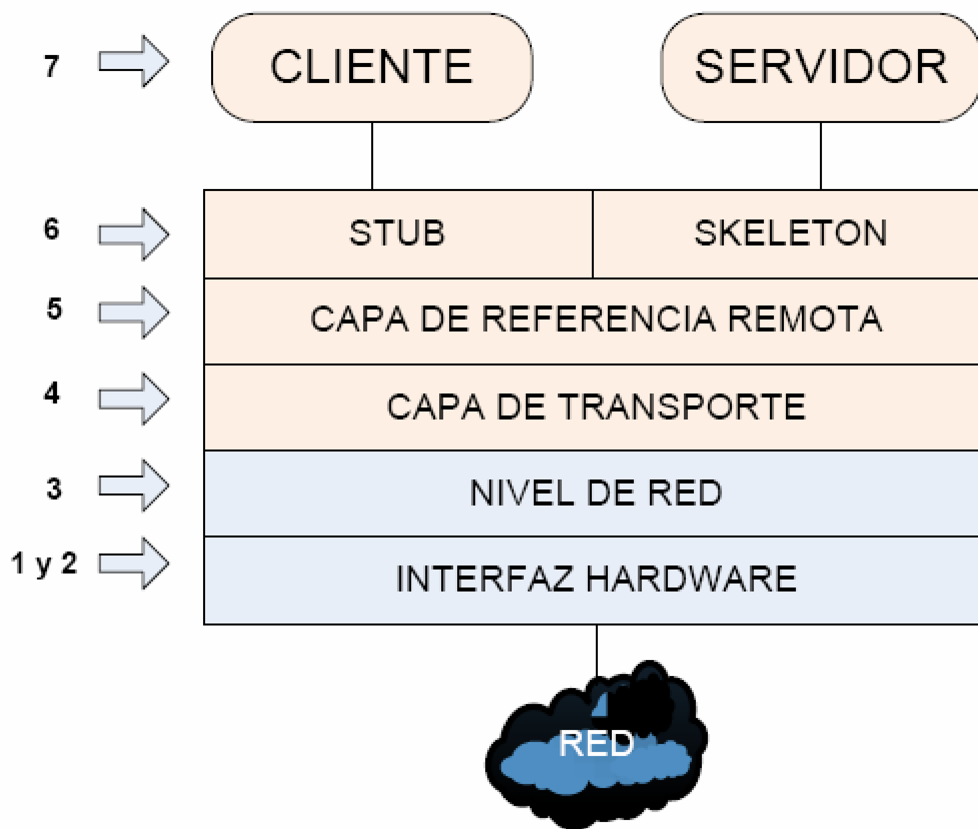


Figura 2: Arquitectura en cuanto a capas de la utilización del middleware RMI

Por medio de RMI, la aplicación puede exportar un objeto. A partir de esa operación, este objeto está disponible en la red, esperando conexiones en un puerto TCP. El proceso desde que se hace una invocación a un método remoto desde el cliente hasta que recibe un resultado, es el siguiente:

- El usuario/cliente que ejecute el simulador puede conectarse e invocar los métodos del resguardo local o *stub*. La invocación consiste en "*marshaling*" de los datos de salida, utilizando la funcionalidad de serialización/empaquetado que proporciona Java. Se notifica a la capa de referencia remota del cliente que la invocación debe hacerse. Mientras esto ocurre, el cliente se queda a la espera de la respuesta a su solicitud, realizándose ésta de manera transparente para el usuario.
- En el servidor se reciben las peticiones de los clientes, se realiza la invocación del método y se devuelven los resultados. De manera más específica, el *skeleton* desempaqueta los argumentos del mensaje de entrada (enviado por el cliente). Hace la llamada al objeto local (remoto para el cliente) ejecutándose la operación. Al obtenerse un resultado o una excepción, se empaqueta y se envía de vuelta por la red.
- Una vez llega al cliente, el *stub* desempaqueta los datos o excepción recibidos y se notifica a la capa de referencia remota que se ha completado la llamada. El código cliente recibe este valor como si la invocación hubiera sido local, y prosigue con su ejecución normalmente.

La figura 3 muestra de manera general la arquitectura de funcionamiento que se usa con middleware:

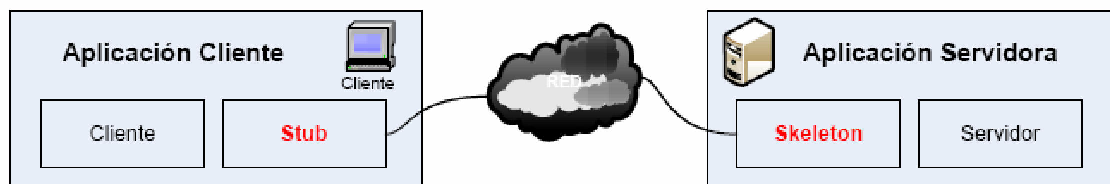


Figura 3: Representación de la arquitectura de funcionamiento en Java RMI

Objetivos de RMI:

1. Permitir invocación remota de métodos en objetos que residen en diferentes Máquinas Virtuales .
2. Permitir invocación de métodos remotos por *Applets*.
3. Integrar el Modelo de Objetos Distribuidos al lenguaje Java de modo natural y preservando en lo posible la semántica de objetos en Java .
4. Permitir la distinción entre objetos locales y remotos .

5. Preservar la seguridad de tipos (type safety) dada por el ambiente de ejecución Java.
6. Permitir diferentes semánticas en las referencias a objetos remotos: no persistentes (vivas), persistentes, de activación lenta.
7. Mantener la seguridad del ambiente dada por los Security Managers y Class Loaders.
8. Facilitar el desarrollo de aplicaciones distribuidas.

Una de las limitaciones debido a su estrecha integración con Java, es que esta tecnología no permite la interacción con aplicaciones escritas en otro lenguaje.

Capítulo 2

Requisitos.

2.1 Introducción.

A partir del Proyecto anterior, se hace un resumen, en forma de tablas, de las características que debe ofrecer el software desarrollado para este Proyecto. Se presentan los requisitos de partida, tanto funcionales como no funcionales, de los atributos de calidad del sistema, desde el punto de vista de su comportamiento, su desarrollo y su mantenimiento. Posteriormente se centra en la capacidad de distribución del sistema que es el punto central de este Proyecto.

2.2 Requisitos de partida.

Tabla 1: Requisitos funcionales

Requisito	Descripción
RF1 Funcionalidad	Debe disponerse de una interfaz desde la cual pueda actuarse sobre el funcionamiento de la maqueta de trenes y observar su estado.
RF2 Rendimiento	Tiempos de respuesta no excesivamente altos.
RF3 Disponibilidad/Fiabilidad	Si el equipo en el que se ejecuta el simulador, realizando tareas de control y monitorización, deja de funcionar, debe ser posible arrancar el sistema en cualquier otro equipo.
RF4 Usabilidad	La interfaz grafica de usuario debe ser sencilla y amigable.

Tabla 2: Requisitos no funcionales.	
Requisito	Descripción
RN1 Portabilidad	El software debe poder ejecutarse sobre cualquier versión de los sistemas operativos Linux y Windows.
RN2 Lenguaje de programación usado	Debe utilizarse Java para el desarrollo de este Proyecto.
RN3 Concurrencia y respuesta en tiempo real	Este punto hace referencia al tiempo de respuesta del sistema. Pueden existir tanto eventos con tratamiento de tiempo real estricto o activo, como eventos con tratamiento de tiempo real no estricto o pasivo.
RN4 Capacidad de distribución del sistema de monitorización y control	Punto central de este Proyecto. Se explica de una forma extensa más adelante.
RN5 Acceso a la información y a los servicios	Cualquier cliente puede controlar y consultar el estado de la aplicación.
RN6 Adaptabilidad y mantenimiento	Es fundamental en la etapa de planificación tener en cuenta los aspectos de mantenimiento, en el sentido de la reutilización del código y la posibilidad de refactorizar el diseño para mejorarlo y adaptarse a las nuevas necesidades que pueda ofrecer la tecnología.

2.3 Capacidad de distribución.

La característica principal de un sistema distribuido es la de compartir recursos situados en distintos lugares y con diferentes dominios de administración a través de la red. De esta manera se ahorran costes hardware.

1. El sistema de monitorización y control debe poder permitir a cualquier usuario ejecutarse en cualquier máquina de la red de área local del laboratorio DSIE, donde se encuentra actualmente la maqueta de trenes.
2. Debe existir homogeneidad entre llamadas locales y remotas, de forma que un cliente pueda invocar a un método remoto de forma similar a como lo haría con un local, siendo totalmente transparente para el.

En la figura 4 se muestra cual es el esquema a seguir si se utiliza un sistema distribuido:

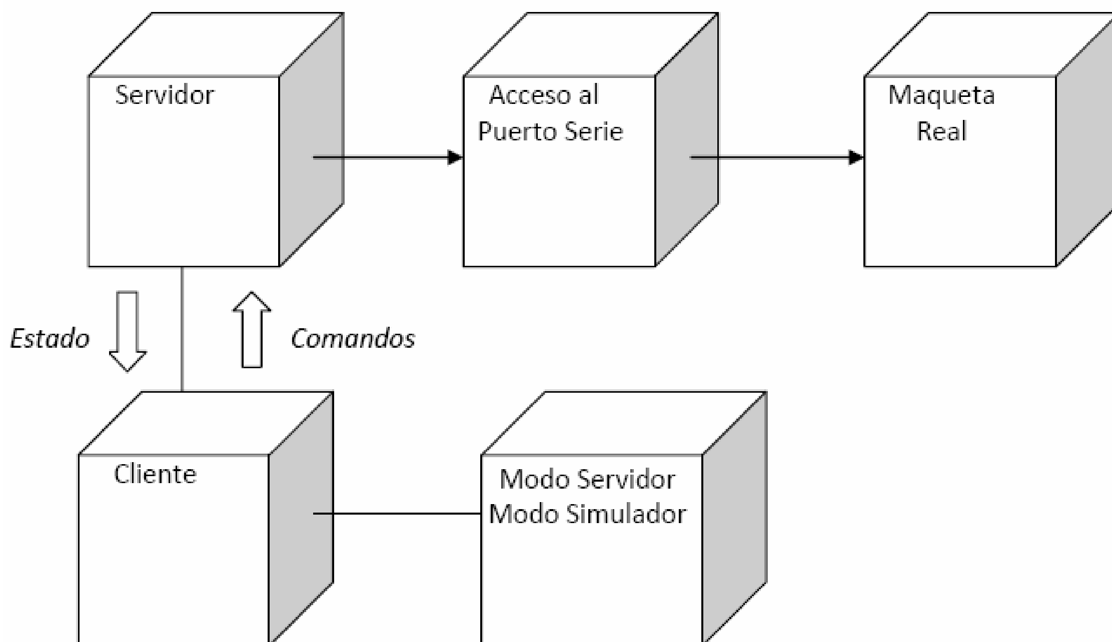


Figura 4: Diagrama de despliegue con la funcionalidad del Proyecto de manera distribuida entre varios equipos del laboratorio.

Este diagrama de despliegue modela cómo se distribuyen los nodos de procesamiento y la comunicación entre ellos. Se puede observar un nodo conectado a la maqueta a través del puerto serie. Este nodo interactúa con el cliente y viceversa, para el caso de que se quiera ejecutar la maqueta de tren real y monitorizar su estado para el usuario. Todo esto se puede ejecutar en diferentes equipos.

Se puede aplicar un sistema distribuido para que el/los clientes no tengan que estar en la misma máquina servidora del simulador, pudiendo estar en otras máquinas del mismo laboratorio, o mediante acceso vía Web.

Capitulo 3.

Diseño del software.

3.1 Introducción.

Habiendo expuesto anteriormente los requisitos de partida de este Proyecto, la planificación general del diseño debe adecuarse a los requisitos que hemos establecido. Para ello se explican las estrategias de diseño que se han empleado para cumplir los requisitos.

3.2 Uso de middleware.

Para cumplir el requisito de distribución se opta por utilizar middleware de comunicaciones. El uso de middleware permite precisamente conseguir homogeneidad entre llamadas locales y remotas permitiendo distribuir el sistema por toda la maqueta.

El middleware nos ofrece las siguientes ventajas:

- Permite usar máquinas de poca capacidad.
- Permite usar hardware ya existente, evitando comprar hardware nuevo.
- Ayuda a mantener y mejorar el tiempo de respuesta.
- Es rápido de implementar.

Si bien hay que contar también con algunos inconvenientes:

- Alto consumo de recursos, tanto de CPU como de memoria RAM.
- Debe desarrollarse de forma específica.
- Costo de desarrollo, implementación
- Capacidad limitada, no entrega todas las soluciones
- Requiere cierta complejidad logística a la hora de instalar en cada equipo.

Existen otras formas para cumplir estos requisitos, como hacer un Proxy a medida (oculta la información) o un pequeño Broker. Pero como hay middleware comercial que satisfacen nuestros requisitos no merece la pena. Ejemplos de middleware para implementar sistemas distribuidos son RMI, CORBA, .NET, etc. En este caso, se ha optado por el RMI que proporciona Java.

3.3 Patrones de diseño.

A continuación se resumen los patrones de diseño utilizados en el Proyecto, en especial Flyweight por constituir éste una novedad respecto del Proyecto de partida [3] y constituir uno de los fundamentos del trabajo realizado. El lector interesado puede encontrar una descripción más completa en [4][5].

3.3.1 MVC.

Considerando los requisitos contemplados anteriormente, la aplicación debe ser fácil de usar, ser fácilmente mantenible y adaptable a nuevas modificaciones o servicios solicitados por el cliente. Además debe tener una interfaz gráfica de usuario que sea fácil de sustituir y adaptar, donde las respuestas a las acciones del usuario deben de ser inmediatas.
Por todo esto, se estima que la mejor forma de cumplir los requisitos es mediante la utilización del patrón MVC. Este patrón es una de las herramientas básicas de cualquier desarrollador de GUI, estableciendo una independencia entre los módulos de la aplicación, para facilitar la mantenibilidad y la sustitución de diferentes componentes gráficos. Además, este patrón ofrece la posibilidad de componer diferentes vistas simultáneas sincronizadas de un mismo modelo.
De esta manera, se mantiene desacoplada la parte de las vistas y la parte del modelo, por lo que no hay que cambiar el modelo por el hecho de cambiar las vistas. La sustitución y extensión de los módulos de la aplicación no es tarea difícil, facilitando mucho los cambios y extensiones de la GUI sin afectar al modelo. La utilización de este patrón presenta algunos inconvenientes, debido a que requiere una implementación compleja, es difícil diseñar la jerarquía de componentes, y tanto la vista como el controlador están muy fuertemente acoplados, si bien no supone un gran problema ya que ambos módulos suelen tener una comunicación directa en esta aplicación.

De acuerdo con lo dicho, la aplicación se divide en tres áreas: procesamiento, entrada y salida, que se corresponden con los tres tipos de módulos del patrón arquitectónico base utilizado (modelo, vista y controlador). La capa del modelo mantiene referencias de todas las vistas y les notifica los cambios de estado, producidos por la solicitud de un servicio por medio de un controlador o de algún evento sincronizado interno, mediante el patrón Observador[4].
Y es que MVC se apoya a su vez en los patrones Observador y Estrategia[4], utilizándolos de manera implícita.

3.3.2 Flyweight.

A la hora de distribuir el sistema, el patrón MVC ayuda a diferenciar y separar el modelo, la vista y el controlar como anteriormente se ha expuesto. El Proyecto anterior fallaba a la hora de distribuir el sistema debido a la gran cantidad de objetos, en su mayoría repetidos o muy similares, altamente cohesionados. Por eso, el primer paso de este Proyecto ha sido reducir el número de objetos y las relaciones entre sí. Para ello, se ha hecho uso del patrón **Flyweight**[4][5] o "peso mosca" que permite reducir la cantidad de instancias de un objeto; esto se logra mediante la compartición de una misma instancia.
El patrón **Flyweight** es un patrón estructural que determina como combinar objetos y clases para definir estructuras complejas. Describe como almacenar un gran número de objetos sin un gran coste. Para conseguir esto se utilizan objetos que almacenan los estados compartidos y que pueden ser usados por varios objetos simultáneamente.

Este patrón se debe usar cuando:

- Se utiliza un gran número de objetos
- El coste de almacenamiento es alto debido a la cantidad de objetos
- La mayoría de los estados de los objetos pueden ser creados como comunes.
- Muchos objetos pueden ser reemplazados por unos pocos una vez que han sido borrados los estados no comunes.
- La aplicación no depende de la identidad de los objetos.

El funcionamiento de este patrón se explica con este diagrama de clases:

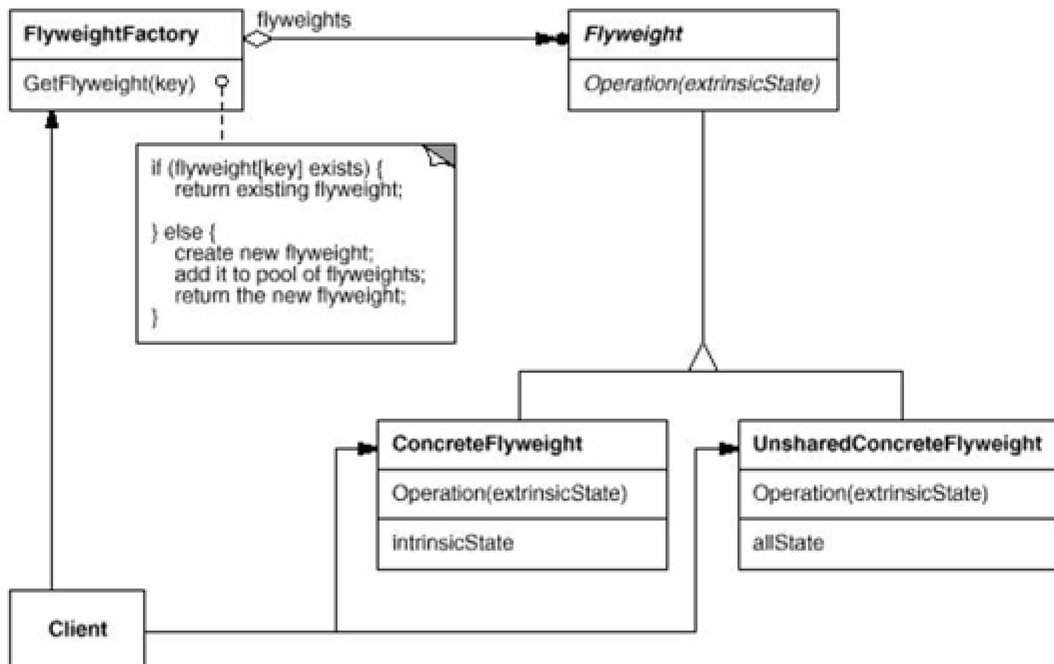


Figura 5: Diagrama de clases del patrón Flyweight

Flyweight

Declara una interfaz a través de la cual los objetos flyweights pueden recibir y actuar sobre los estados no compartidos.

ConcreteFlyweight

Implementa la interfaz Flyweight y almacena los estados compartidos, si los hay. Un objeto ConcreteFlyweight debe ser compartible. Cualquier estado que almacene debe ser intrínseco; es decir, debe ser independiente de su contexto.

UnsharedConcreteFlyweight

No todas las subclases de Flyweight tienen por qué ser compartidas. La interfaz Flyweight permite que se comparta; no lo fuerza. Es común que los objetos de esta clase tengan hijos de la clase ConcreteFlyweight en algún nivel de su estructura.

FlyweightFactory

Crea y gestiona los objetos flyweight.
Garantiza que los objetos flyweight se comparten de forma apropiada. Cuando un cliente solicita un flyweight, el objeto de la clase FlyweightFactory proporciona una instancia existente, o crea una.

Client

Contiene referencias a los objetos flyweights.
Calcula o almacena los estados no compartidos de los objetos flyweights.

La clave del asunto es poder diferenciar los atributos intrínsecos (aquellos que se mantienen igual en todas las instancias), de los atributos extrínsecos (aquellos que varían de una instancia a otra). El estado intrínseco se guarda en el propio objeto. Consiste en información que es independiente de su contexto y que puede ser compartida. El estado extrínseco depende del contexto y cambia con él, por lo que no puede ser compartido.
El cliente del patrón dejará de instanciar objetos directamente y pasará a hablar con una **Factory**[4]. **Factory** es una clase que mantiene una lista de las instancias que tenemos en memoria y le devuelve al cliente una instancia de un objeto que el solicite. Se encarga de reusar instancias y de crear otras nuevas (en el caso que no se haya instanciado la solicitada por el cliente).

Consecuencias de la aplicación de este patrón:

- Puede introducir costes *run-time* debido a la necesidad de calcular y transferir el estado extrínseco.
- La ganancia en espacio depende de varios factores:
  - la reducción en el número de instancias
  - el tamaño del estado intrínseco por objeto
  - si el estado extrínseco es calculado o almacenado

3.4 Implementación.

Los tipos de datos usados en la aplicación son los definidos en el Proyecto de partida. Los elementos principales son:

1. Las vías
2. Las agujas
3. Los sensores
4. Las vistas de las vías, agujas y sensores
5. Los trenes

A cada uno de los elementos de la maqueta del simulador se le establece su modelo (con cierta funcionalidad), su vista asociada, y si dispone de él, su controlador, para poder interactuar con el modelo.
Tanto la vía férrea como los trenes son implementados como listas enlazadas o grafos ya que facilitan mucho el cambio de topología y la adición/eliminación de elementos de la maqueta. Al necesitar establecer una relación de avance y retroceso de los trenes sobre los nodos, la vía férrea se ha implementado mediante listas doblemente enlazadas. Los nodos se organizan, de modo que cada uno apunta al siguiente y al anterior. El grafo resultante representa a la maqueta.
El cambio principal respecto al Proyecto anterior es en lo referentes a las vistas de los nodos y de la maqueta. Antes se generaba una vista por cada nodo que forma la maqueta, lo que acumula un número muy alto de vistas. Ahora, y gracias al uso del patrón Flyweight, sólo se tiene una vista por cada tipo de nodo (normal, sensor o aguja) y por el estado del mismo (ocupado o libre), lo que reduce considerablemente el número de objetos. Al contrario que en el Proyecto anterior que se definían las agujas y los sensores como un tipo de nodo especial, ahora se considera nodos normales pero con unos atributos extrínsecos que los diferencian como aguja o sensor en cada caso. Esto simplifica la operatividad con este tipo de nodos en lo referente al cambio de estado y la inclusión en la vía férrea resultante.
Otro criterio de diseño general importante es proporcionar las interfaces que se han definido. La implementación detallada se encuentra en el Capítulo 5 y en el CD adjunto en la carpeta *Javadoc*.

Los elementos antes mencionados se modelan en las siguientes interfaces y clases:

Trenes

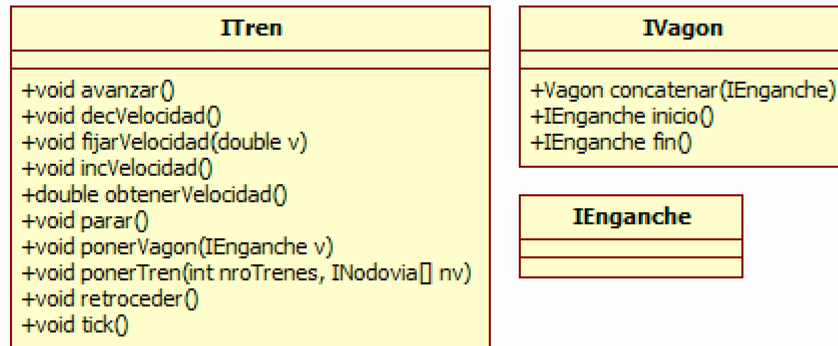


Figura 6: Las interfaces ITren, IVagon e IEnganche.

ITren: Interfaz común para modelar un tren.	
Servicios	**Descripción**
void avanzar ();	Permite avanzar a un objeto tren.
void decVelocidad ();	Decrementa la velocidad de un objeto tren.
void fijarVelocidad (double v);	Establece la velocidad de un objeto tren.
void incVelocidad ();	Incrementa la velocidad de un objeto tren.
double obtenerVelocidad ();	Obtiene la velocidad de un objeto tren.
void parar ();	Permite parar el movimiento de un objeto tren.
void ponerVagon (IEnganche v);	Método para agregar un vagón al tren.
void ponerTren (int nroTrenes, INodovia[] nv);	Método para poner un tren en la vía.
void retroceder ();	Permite retroceder a un objeto tren.
void tick ();	Método que permite hacer una determinada acción en cada tick de reloj.

IVagon: Interfaz común para modelar un vagón de tren.	
Servicios	**Descripción**
Vagon concatenar(IEnganche);	Permite enganchar un vagón con otro.
IEnganche inicio();	Obtiene el enganche frontal de un vagón.
IEnganche fin();	Obtiene el enganche trasero de un vagón.

IEnganche: Interfaz de marca para ocultar la implementación que hay por debajo.

Tabla 3: Resumen de las interfaces ITren, IVagon e IEnganche.

Vías, agujas y sensores

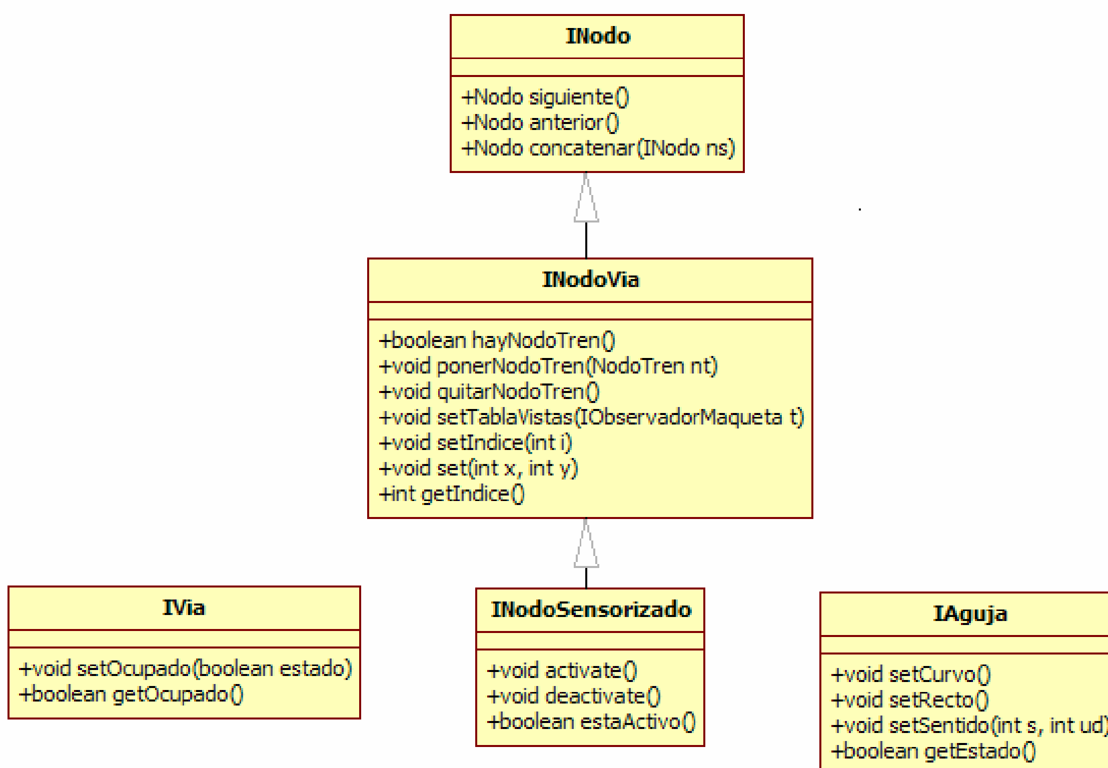


Figura 7: Las interfaces INodo, INodoVia, IVia, INodoSensorizado e IAguja.

INodo: Interfaz común para modelar los objetos de las clases que representan los elementos principales que forman la vía férrea.	
Servicios	**Descripción**
Nodo siguiente ();	Método que devuelve el nodo siguiente al actual.
Nodo anterior ();	Método que devuelve el nodo anterior al actual.
Nodo concatenar (INodo ns);	Método que concatena dos nodos.

INodoVia: Interfaz común para modelar los objetos de las clases que representan los elementos principales que forman la vía férrea.	
Servicios	**Descripción**
boolean hayNodoTren ();	Método para indicar si hay algún objeto tren ya colocado sobre un elemento de la vía férrea.
void ponerNodoTren (NodoTren nt);	Método para poner un nodo tren en un nodo de la vía.
void quitarNodoTren ();	Método para quitar un nodo tren de un nodo vía.
void setTablaVistas (IObservadorMaqueta t);	Método para asociar un observador a cada nodo.
void setIndice (int i);	Método para asignar un índice a cada nodo.
void set (int x, int y);	Método para asignar coordenadas a cada nodo.
int getIndice ();	Devuelve el índice de un nodo.

IVia: Interfaz común para modelar los objetos de las clases que representan los tramos de vía de la maqueta.	
Servicios	**Descripción**
void setOcupado (boolean estado);	Establece el estado de un nodo de vía.
boolean getOcupado ();	Devuelve el estado de un nodo de vía.

INodoSensorizado: Interfaz común para modelar los objetos de las clases que representan los sensores que se encuentran en la vía férrea de la maqueta.	
Servicios	**Descripción**
void activate ();	Método para activar el sensor.
void deactivate ();	Método para desactivar el sensor.
boolean estaActivo ();	Método para saber el estado del sensor.

IAguja: Interfaz común para modelar los objetos de las clases que representan las agujas que se encuentran en la vía férrea.	
Servicios	**Descripción**
void setCurvo ();	Método para enlazar la aguja con el siguiente tramo de la vía, con estado curvo.
void setRecto ();	Método para enlazar la aguja con el siguiente tramo de la vía, con estado recto.
void setSentido (int s, int ud);	Fija el sentido de la aguja, y la orientación que tendrá cuando esté en estado curvo.
boolean getEstado ();	Devuelve el estado en el que se encuentra la aguja, recto o curvo.

Tabla 4: Resumen de las interfaces INodoGeneral, INodoVia, IVia, IAguja e INodoSensorizado

Capitulo 4.

Paquetes y librerías.

4.1 Introducción.

En este capítulo se encarga de la presentación de los diferentes paquetes que conforman el software desarrollado en este Proyecto y su porqué. Surge de la necesidad de realizar un control de la estructura general que adopta el programa. Esto permite tener información de relevancia y global acerca del modelado aunque no tanto de la implementación.

4.2 Diferentes paquetes desarrollados.

Cada paquete proporciona un mecanismo de agrupación para organizar las clases e interfaces de la aplicación, además de proporcionar un espacio de nombrado sobre cada parte en la que está dividido el Proyecto. Al querer dar con la mayor claridad y elegancia posible el cumplimiento de características propias del patrón MVC los paquetes en los que se organiza el código tienden a reflejar la estructura de dicho patrón.

Dichos paquetes son los siguientes:

- Paquete **maqueta**: contiene las interfaces de la aplicación.
- Paquete **inicios**: contiene las aplicaciones principales que modelan el simulador.
- Paquete **vistas**: contiene las distintas vistas de los elementos de una vía, además de las interfaces gráficas que se muestran al usuario.
- Paquete **controladores**: contiene los distintos controladores en los que el usuario podrá interactuar con el simulador.
- Paquete **modelos**: contiene la funcionalidad de la aplicación, es decir, la implementación de cada uno de los tipos de elementos, así como el tratamiento de excepciones.

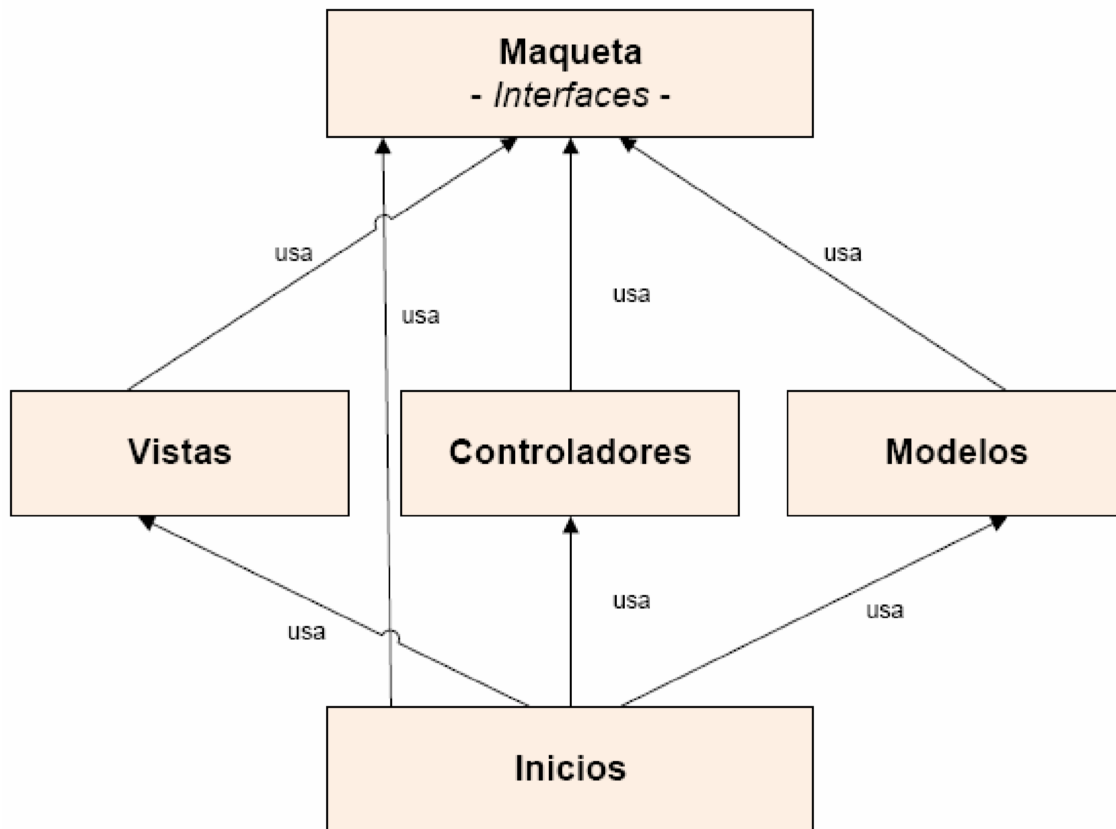


Figura 8: Representación de los paquetes

Se ha definido un paquete **Maqueta** para favorecer la ocultación de la implementación lo máximo posible. En **Maqueta** se encuentran todas las interfaces del Proyecto. De esta manera, el resto de paquetes lo usan para implementar la funcionalidad que mejor les convenga para desempeñar sus funciones sin que se establezcan dependencias mutuas entre implementaciones.

La biblioteca de clases para el iniciar el simulador, denominada **Inicios**, tiene una correspondencia directa con el resto de bibliotecas de clases, tanto para el control de la maqueta, como para la interacción del usuario, como para la visualización del estado de la maqueta. Éste paquete utiliza a todos los demás, debido a que además de contener el código de inicialización del simulador, contiene las clases que modelan la maqueta global creando sus componentes según el patrón MVC.

El paquete **Modelos**, tal y como su nombre referencia, se corresponde con la funcionalidad que presenta cada modelo. Está compuesto por diferentes clases para poder hacer posible llevar a cabo la funcionalidad de esta aplicación, registrar las vistas y controladores dependientes, y notificar a los componentes dependientes sobre los

cambios de datos. Se intenta establecer lo más independiente posible, por lo que no tiene referencia directa alguna sobre los paquetes **Controladores** y **Vistas**, que contienen implementaciones, sino con **Maqueta** que contiene las interfaces que implementan las clases del paquete. La idea es depender de interfaces, pero no de implementaciones.

Por otra parte, el paquete **Vistas** es aquel que incluye todas las vistas de cada uno de los elementos de los que está formada la maqueta de trenes del laboratorio, además de proporcionarnos una interfaz gráfica intuitiva para monitorizar el estado de cada tren, sensor, y aguja sobre la maqueta. Además, está formada por clases que a parte de realizar lo anteriormente reflejado, implementan el paquete **Maqueta** realizando el procedimiento de actualización y recuperación de los datos procedentes del modelo, según el patrón MVC. Este paquete tiene referencias a las interfaces de **Maqueta**, para la creación de los elementos de la vía férrea de la maqueta, sin necesidad de saber qué elementos de **Modelos** tendrán relación con este paquete.

En última instancia, queda el paquete **Controladores**, el cual gestiona las entradas del usuario para acceder a la funcionalidad del modelo. Su única clase se encarga de controlar los trenes que se van poniendo en la maqueta. En el Proyecto anterior, el estado de las agujas era gestionado por un controlador que se incluía en este paquete, pero ahora se hace de una manera más intuitiva pues el usuario solo tiene que “hacer click” sobre una aguja para que esta cambie su estado. Para esto se ha hecho uso del manejador de eventos proporcionado por Java con lo que no es necesario una clase especifica para el manejo de las agujas. Con respecto a la relación que hay entre este paquete y el paquete **Modelos** es que reflejan una correspondencia directa unilateral por medio de interfaces, para poder informar a las clases del modelo, los cambios que se deben llevar a cabo. También usa el paquete **Maqueta** para implementar una interfaz de su tipo.

Capitulo 5.

Descripción de la aplicación.

5.1 Introducción.

El objetivo de este capítulo es realizar un estudio detallado sobre el modelado y la implementación de los componentes. Se pretende distinguir el código desde dos puntos de vista: una vista estática, que englobará la estructura de clases, y una vista dinámica, que representa el comportamiento de la aplicación en tiempo de ejecución.
Desde el punto de vista del funcionamiento, deben distinguirse tres partes, totalmente diferenciadas pero que cooperan entre sí: el cliente, el servidor y la maqueta. Es posible ejecutar cada una de las partes en una maquina distinta, es más, el propósito de este Proyecto es que esto sea así.

5.2 Visión estática del código.

La funcionalidad de cada uno de los componentes de la aplicación de este Proyecto se modela en una interfaz, proporcionándose además, para cada una de ellas, una o varias clases que implementan el comportamiento por defecto común de cada tipo de componente. A continuación se explicará la funcionalidad, e implementación para cada una de las clases implementadas en este Proyecto. La documentación específica de la aplicación está en el *Javadoc* incluida en el CD.

5.2.1 Los nodos.

En este Proyecto se necesita establecer una relación de avance y retroceso de los trenes sobre los nodos, por lo que la vía férrea se ha implementado mediante listas doblemente enlazadas o grafos, facilitando el cambio de topología y la adición/eliminación de elementos de la maqueta. Estos nodos se organizan, de modo que cada uno apunta al siguiente y al anterior nodo. El grafo resultante de enlazar cada uno de los tramos representa a la maqueta global.

Existe una interfaz que representa a cada uno de los nodos, definiendo la funcionalidad común que tiene cada uno de ellos.
La aguja se implementa como un tipo de nodo especial ya que en vez de tener un sucesor, puede tener dos posibles sucesores o anteriores, dependiendo del tipo de aguja que sea.
Existe una clase **Nodo** que implementa **INodo**. Éstos definen la funcionalidad común que tienen los nodos, sea cual sea cada uno de ellos. En este caso será la implementación de una lista doblemente enlazada, por lo que de ahí vienen sus métodos.
Las clases de los nodos de vía son **NodoVia**, **NodoAguja2A**, **NodoAguja2P** y **NodoViaSensorizado**. Éstos son los elementos de la vía férrea que componen la maqueta. Cabe destacar que tanto las clases **NodoViaSensorizado** como **NodoAguja2A** y **NodoAguja2P** heredan directamente de **NodoVia**.
La clase **NodoVia** permite modelar la posición de un tren sobre la vía de la maqueta. Hereda la funcionalidad de una lista, de **Nodo**, e implementar la interfaz **IVia**, recogiendo la funcionalidad básica de todo **NodoVia**.
Un **NodoViaSensorizado** adquiere la funcionalidad recogida en **NodoVia**, al heredar de ella, e implementa la funcionalidad básica de un sensor, de la interfaz **NodoSensorizado**, tal como activar y desactivar un sensor, comprobar su estado, etc.
Como tipo de nodo especial se encuentran las agujas. Se definen dos clases para su implementación: **NodoAguja2A** (cuando la aguja tiene dos nodos anteriores) y **NodoAguja2P** (para cuando tiene dos nodos posteriores). Al formar parte también de una vía, adquiere la funcionalidad de **NodoVia**, y por consiguiente, hereda de esta clase.
Destacar también la clase **NodoTren**, utilizada para ayudar a los objetos de las clases de los elementos de la vía férrea, proporcionándoles una serie de métodos para el modelado de la posición del tren sobre la maqueta del simulador, tales como indicar y fijar el nodo de la maqueta en el que se encuentra el tren en un momento determinado.
Las clases antes mencionadas se resumen a continuación:

Nodo: Clase para modelar los elementos de una maqueta.	
Servicios	**Descripción**
Nodo siguiente ();	Método que devuelve el nodo siguiente al actual.
Nodo anterior ();	Método que devuelve el nodo anterior al actual.
Nodo concatenar (INodo ns);	Método que concatena dos nodos.

NodoVia: Clase para modelar los elementos de los tramos de vía de una maqueta.	
Servicios	**Descripción**
boolean hayNodoTren ();	Método para indicar si hay algún objeto tren ya colocado sobre un elemento de la vía férrea.
void ponerNodoTren (NodoTren nt);	Método para poner un nodo tren en un nodo de la vía.
void quitarNodoTren ();	Método para quitar un nodo tren de un nodo vía.
void setTablaVistas (IObservadorMaqueta t);	Método para asociar un observador a cada nodo.
void setIndice (int i);	Método para asignar un índice a cada nodo.
void set (int x, int y);	Método para asignar coordenadas a cada nodo.
int getIndice ();	Devuelve el índice de un nodo.
void setOcupado (boolean estado);	Método para establecer el estado en el que estará el nodo de la vía, ocupado o no.
boolean getOcupado ();	Método para devolver el estado de cada nodo Vía.
void actualizarVista ();	Método que indica al observador de la vía los cambios de estado de los nodos.

NodoSensorizado: Clase para modelar los elementos de los tramos de vía que están sensorizados.	
Servicios	**Descripción**
void activate ();	Método para activar el sensor.
void deactivate ();	Método para desactivar el sensor.
boolean estaActivo ();	Método para saber el estado del sensor.

NodoAguja2A: Clase para modelar los elementos de las agujas con dos nodos anteriores.	
Servicios	**Descripción**
void setCurvo ();	Método para enlazar la aguja con el siguiente tramo de la vía, con estado curvo.
void setRecto ();	Método para enlazar la aguja con el siguiente tramo de la vía, con estado recto.
void setSentido (int s, int ud);	Fija el sentido de la aguja, y la orientación que tendrá cuando esté en estado curvo.
boolean getEstado ();	Devuelve el estado en el que se encuentra la aguja, recto o curvo.

NodoAguja2P: Clase para modelar los elementos de las agujas con dos nodos anteriores.	
Servicios	**Descripción**
void setCurvo ();	Método para enlazar la aguja con el siguiente tramo de la vía, con estado curvo.
void setRecto ();	Método para enlazar la aguja con el siguiente tramo de la vía, con estado recto.
void setSentido (int s, int ud);	Fija el sentido de la aguja, y la orientación que tendrá cuando esté en estado curvo.
boolean getEstado ();	Devuelve el estado en el que se encuentra la aguja, recto o curvo.

Tabla 5: Resumen de las clases de los nodos

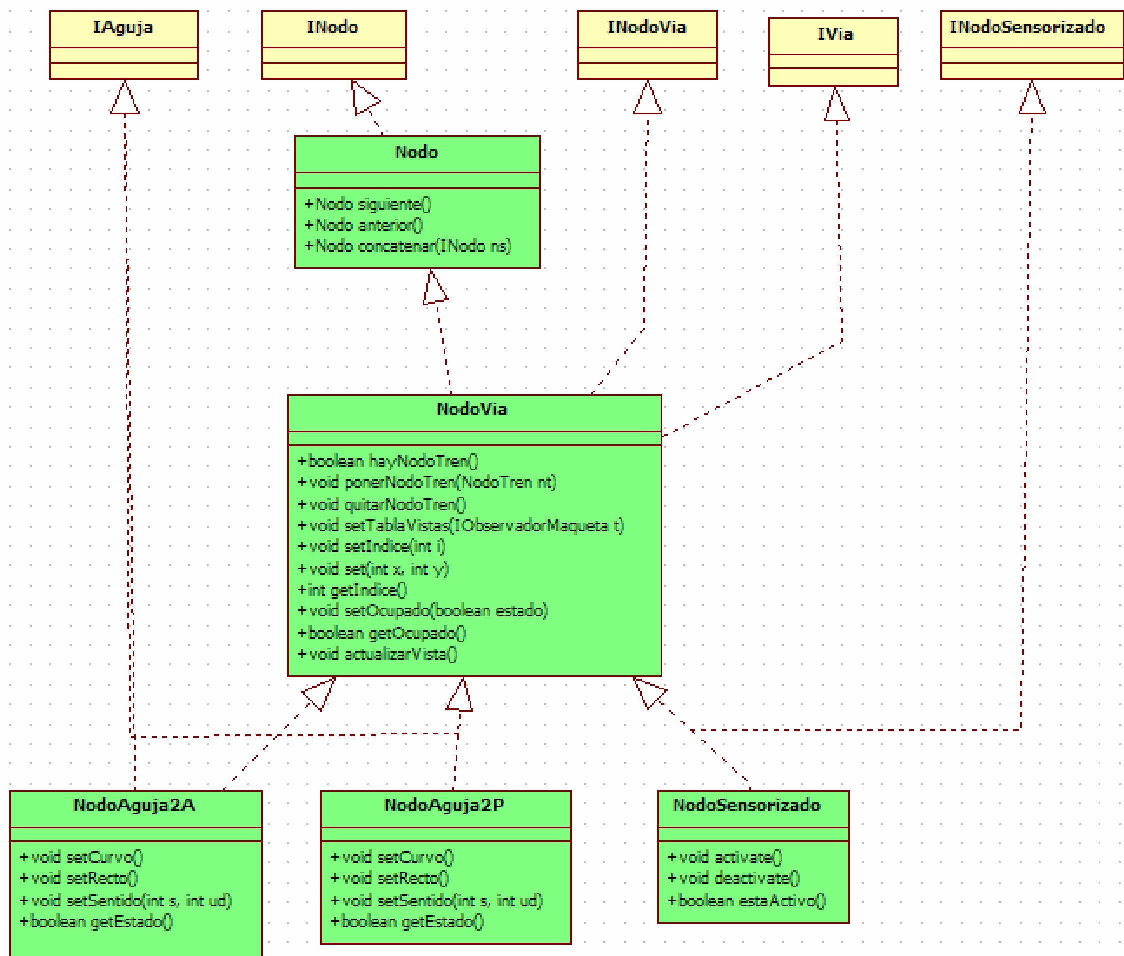


Figura 9: Relaciones de las clases e interfaces de los nodos.

5.2.2 El tren.

El comportamiento de un tren está definido en la interfaz **ITren**. La clase que lo implementa es **TrenSimulado**. Dos de los métodos más importantes que definen el comportamiento de un tren, y por lo tanto, los implementan, son **avanzar( )** y **retroceder( )**. Se realiza un estudio detallado sobre la implementación de cada uno de ellos:

1. Método **avanzar()**:

El código desarrollado en este método es el siguiente:

```
public void avanzar() throws java.rmi.RemoteException{
    NodoTren indice = (NodoTren)primero;
    while (indice != null){
        indice.getNodo().quitarNodoTren();
        try{
            NodoVia nv = (NodoVia) indice.getNodo().siguiente();
            nv.ponerNodoTren(indice);
        }
        catch(ColisionTrenesException e){
            System.out.println("Colisión");
            parar();
            JOptionPane.showMessageDialog(null, "Se ha producido una colisión entre trenes.\n" +
                    "Todos los trenes implicados se detienen.",
                    "COLISIÓN DE TRENES",JOptionPane.WARNING_MESSAGE);
        }
        indice.setNodo((NodoVia)(indice.getNodo().siguiente()));
        indice = (NodoTren) indice.siguiente();
    }
}
```

Figura 10: Código del método avanzar().

La explicación de esta implementación es la siguiente:

- Hay un nodo de referencia que define la posición actual del tren en la vía y a partir del cual se avanza o retrocede. Este nodo, llamado **índice**, inicialmente es el primer nodo donde se coloca el tren en la maqueta.
- Al avanzar, y por tanto, cambiar de un nodo a su siguiente, se le quita la asociación directa que tiene con el nodo vía en el que se encuentra, actualizando su vista para que se refleje que ya ha abandonado ese nodo de la vía.
- Posteriormente se necesita obtener una referencia del nodo de la vía que está concatenado con el que estaba antes, que será el que se encuentra inmediatamente después. Los métodos de la clase **Nodo** permiten realizar esto.
- Al obtener esta referencia, el nodo del tren se coloca en ese mismo nodo de la vía y se actualiza la vista para reflejar que el tren se ha colocado en dicho nodo. En el caso de que se intente ocupar un nodo por dos trenes al mismo tiempo, se produce una excepción, que se trata parando los trenes.
- Para continuar con el ciclo que permite ir pasando de un nodo de vía a otro nodo de vía colocando el tren, se obtiene una referencia del nodo siguiente al actual, donde se encuentra el tren, y se le establece al nodo utilizado en la condición (al nodo de referencia). Por lo tanto, de esta manera, siempre se van obteniendo referencia al nodo siguiente, hasta que no haya ninguno, y por lo tanto, el tren no pueda avanzar más.

2. Método **retroceder()**:

Se expone a continuación el código desarrollado en este método:

```
public void retroceder() throws java.rmi.RemoteException{
    NodoTren indice = (NodoTren)ultimo;
    while (indice != null){
        indice.getNodo().quitarNodoTren();
        try{
            NodoVia nv = (NodoVia) indice.getNodo().anterior();
            nv.ponerNodoTren(indice);
        }
        catch(ColisionTrenesException e){
            System.out.println("Colisión");
            parar();
            JOptionPane.showMessageDialog(null, "Se ha producido una colisión entre trenes.\n" +
                    "Todos los trenes implicados se detienen.",
                    "COLISIÓN DE TRENES",JOptionPane.WARNING_MESSAGE);
        }
        indice.setNodo((NodoVia)(indice.getNodo().anterior()));
        indice = (NodoTren) indice.anterior();
    }
}
```

Figura 11: Código del método retroceder().

El procedimiento implementado en este método, es igual que el que se ha explicado para el método avanzar( ), con la salvedad de que ahora el nodo de referencia no tiene que saber cual es su nodo siguiente, sino su nodo anterior, provocando así que cada nodo enlace con su nodo anterior concatenado, haciendo que el tren pueda retroceder satisfactoriamente.

Las clases antes mencionadas se resumen a continuación:

TrenSimulado: Clase que define el comportamiento de un tren simulado.	
Servicios	**Descripción**
void avanzar ();	Permite avanzar a un objeto tren.
void decVelocidad ();	Decrementa la velocidad de un objeto tren.
void fijarVelocidad (double v);	Establece la velocidad de un objeto tren.
void incVelocidad ();	Incrementa la velocidad de un objeto tren.
double obtenerVelocidad ();	Obtiene la velocidad de un objeto tren.
void parar ();	Permite parar el movimiento de un objeto tren.
void ponerVagon (IEnganche v);	Método para agregar un vagón al tren.
void ponerTren (int nroTrenes, INodovia[] nv);	Método para poner un tren en la vía.
void retroceder ();	Permite retroceder a un objeto tren.
void tick ();	Método que permite hacer una determinada acción en cada tick de reloj.

Vagon: Clase que define el comportamiento de un vagón.	
Servicios	**Descripción**
Vagon concatenar(IEnganche);	Permite enganchar un vagón con otro.
IEnganche inicio();	Obtiene el enganche frontal de un vagón.
IEnganche fin();	Obtiene el enganche trasero de un vagón.

Tabla 6: Resumen de las clases de los nodos

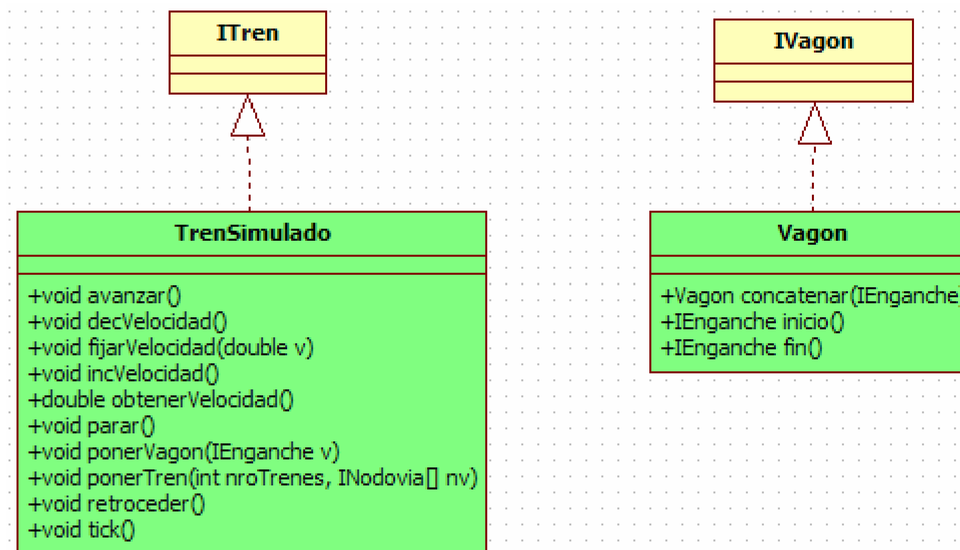


Figura 12: Relaciones de las clases e interfaces de los trenes.

5.2.3 Los controladores.

Los controladores se encargan de aceptar entradas del usuario para acceder a la funcionalidad del modelo, es decir, proporciona la funcionalidad (servicios) de la aplicación.

Como se ha explicado anteriormente, ahora sólo se tiene un controlador para cada tren, **ControladorTren**, que permite al usuario seleccionar la velocidad del mismo, pararlo o ponerlo en marcha. Este controlador es un panel con los correspondientes botones, amigable e intuitivo.

También se ha indicado anteriormente que el control del estado de las agujas se realiza con el manejador de eventos proporcionado por Java, con lo que se ahorra el tener un controlador para cada aguja. Al hacer click sobre cada aguja, esta cambia alternativamente de estado recto a estado curvo.

Las clases antes mencionadas se resumen a continuación:

ControladorTren: Clase que controla el movimiento de un tren.	
Servicios	**Descripción**
void actionPerformed (ActionEvent e);	Controla los botones para Arrancar y Parar el tren.
void stateChanged (ChangeEvent ce);	Método para controlar la barra de velocidad del tren.

Tabla 7: Resumen de la clase ControladorTren.

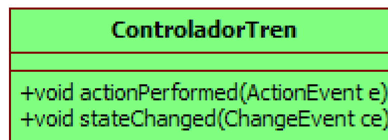


Figura 13: Clase ContraladorTren

5.2.4 Las vistas.

Las vistas se encargan de desplegar en una interfaz de usuario la información procedente del modelo. Se hace de manera diferente al anterior Proyecto.

Se ha definido una interfaz común, denominada **IVistaNodoVia**, la cual presenta la funcionalidad que tendrán las vistas de los diferentes elementos que tiene la vía férrea, tales como nodos de vía, agujas y sensores. Como ya se ha indicado, en vez de existir una vista por cada nodo, hay una vista por cada tipo y estado del nodo. Hay tres clases que implementan **IVistaNodoVia** que son las encargadas de dibujar los nodos en sus diferentes combinaciones de estado. Estas clases son: **VistaNodoVia_0** para la vista de un nodo simple o un nodo sensor, **VistaNodoVia_1** para la vista de una aguja en estado recto y **VistaNodoVia_2** para la vista de una aguja en estado curvo. Una clase factoría, **VistaNodoViaFactory**, es la encargada de determinar que vista de nodo genera en función de los parámetros que se le pasan.

Para facilitar la distribución de las vistas se ha optado por incluir en una tabla toda la información de los nodos (coordenadas, tipo,…) y su estado (ocupado, libre, curvo, recto, activo,…). Esta tabla es un array de objetos y tiene el nombre de **TablaVistas**. Se

ha creado un nuevo tipo de dato con toda esta información llamado **DatoVista** que es utilizado por **TablaVistas.**

Los clientes con esta información son capaces de crear una vista de la maqueta, para ello crean una instancia de la clase **AreaDibujo** que es la encargada de dibujar (pintar) cada uno de los nodos. Esta clase proporciona además un método para actualizar (repintar) cada vez que se produzca un cambio de estado. La clase **TablaVistas** proporciona unos métodos estáticos para alinear correctamente las vistas de los nodos. **TablaVistas** implementa la interfaz **IObservadorMaqueta**.

La clase **VistaMaqueta** es la encargada de gestionar las vistas sobre la maqueta. Asocia el área donde se dibuja la maqueta y actualiza el área en caso de que se produzcan cambios. Implementa **IVistaMaqueta**.

Las clases e interfaces antes mencionadas se resumen a continuación:

IVistaNodoVia: Interfaz común para las vistas de los nodos.	
Servicios	**Descripción**
void draw (Graphics g, int x, int y);	Método para dibujar los nodos.

VistaNodoVia_0: Clase para dibujar la vista de un nodo simple o sensor.	
Servicios	**Descripción**
void draw (Graphics g, int x, int y);	Método para dibujar los nodos.

VistaNodoVia_1: Clase para dibujar la vista de un nodo aguja en estado recto.	
Servicios	**Descripción**
void draw (Graphics g, int x, int y);	Método para dibujar los nodos.

VistaNodoVia_2: Clase para dibujar la vista de un nodo aguja en estado curvo.	
Servicios	**Descripción**
void draw (Graphics g, int x, int y);	Método para dibujar los nodos.

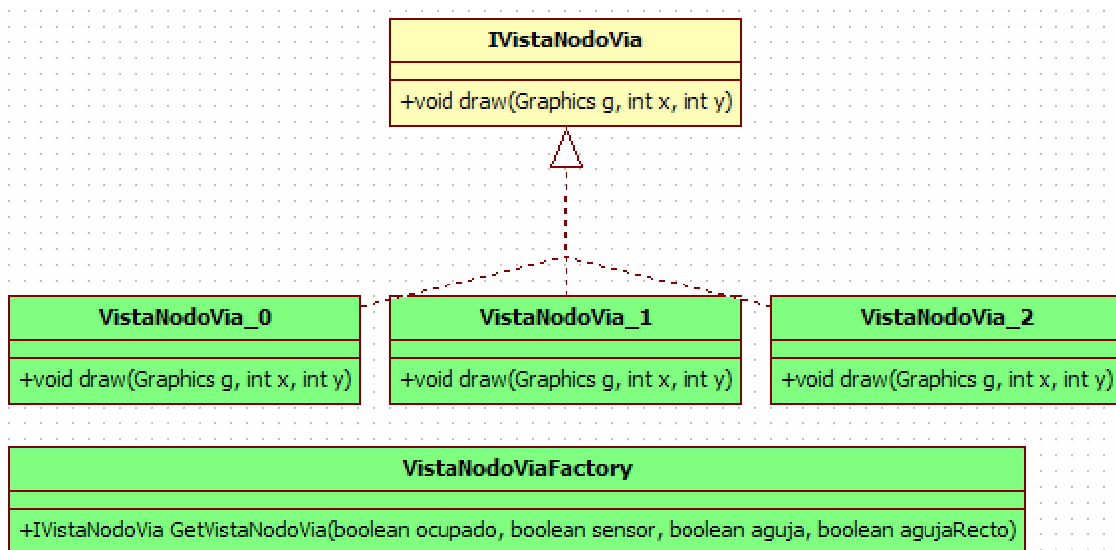
VistaNodoViaFactory: Clase factoría que genera una vista diferente en función del estado y tipo de nodo.	
Servicios	**Descripción**
IVistaNodoVia GetVistaNodoVia (boolean ocupado, boolean sensor, boolean aguja, boolean agujaRecto);	Dependiendo de los parámetros, devuelve uno de los tipos de vista de nodos.

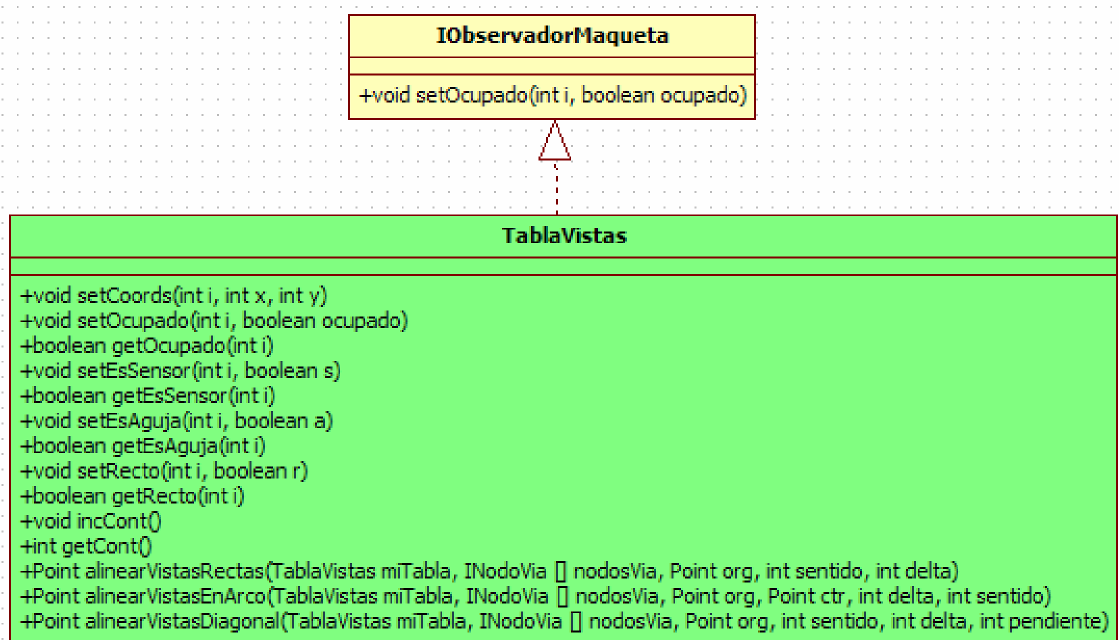
TablaVistas: Clase que guarda la información y el estado de los nodos que forman la maqueta.	
Servicios	**Descripción**
void setCoords (int i, int x, int y);	Fija coordenadas de un nodo.
void setOcupado (int i, boolean ocupado);	Pone un nodo en estado ocupado.
boolean getOcupado (int i);	Devuelve el estado de un nodo.
void setEsSensor (int i, boolean s);	Establece si un nodo es sensor.
boolean getEsSensor (int i);	Indica si un nodo es sensor.
void setEsAguja (int i, boolean a);	Establece si un nodo es aguja.
boolean getEsAguja (int i);	Indica si un nodo es aguja.
void setRecto (int i, boolean r);	Establece si una aguja se encuentra en estado recto.
boolean getRecto (int i);	Indica si una aguja esta recta.
void incCont ();	Incrementa el número de elementos de la tabla.
int getCont ();	Devuelve el número de elementos de la tabla.
Point alinearVistasRectas (TablaVistas miTabla, INodoVia [] nodosVia, Point org, int sentido, int delta);	Método para alinear vistas en tramos rectos.
Point alinearVistasEnArco (TablaVistas miTabla, INodoVia [] nodosVia, Point org, Point ctr, int delta, int sentido);	Método para alinear vistas en tramos curvos.
Point alinearVistasDiagonal (TablaVistas miTabla, INodoVia [] nodosVia, Point org, int sentido, int delta, int pendiente);	Método para alinear vistas en tramos rectos en diagonal.

AreaDibujo: Clase encargada de dibujar (pintar) la maqueta.	
Servicios	**Descripción**
void paintComponent (Graphics g);	Método encargado de dibujar la maqueta en función del estado de los nodos que la forman.
void refrescar (TablaVistas t);	Método para actualizar la maqueta en función de los cambios en la tabla de vistas.

VistaMaqueta: Clase encargada de gestionar las vistas sobre la maqueta.	
Servicios	**Descripción**
void setAreaDibujo (IAreaDibujo area);	Se asocia un área donde dibujar la maqueta.
void update (TablaVistas t);	Actualiza el área de dibujo.

Tabla 8: Resumen de las clases e interfaces que constituyen las vistas.





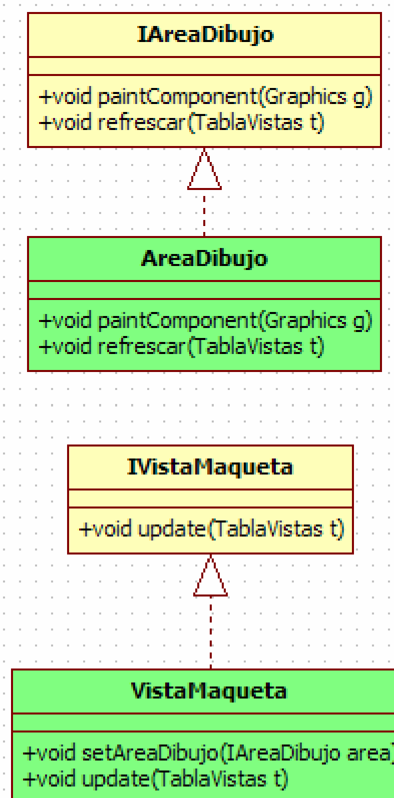


Figura 14: Relaciones de las clases e interfaces de las vistas.

5.2.5 Las clases de Inicios.

Las clases del paquete **Inicios** son las que establecen el inicio de la aplicación desarrollada en este Proyecto.

Por un lado esta el servidor de la aplicación, **GUI_Maq_Servidor**, encargado de crear una instancia de la maqueta y poniéndola a disposición para que los clientes puedan solicitarla de forma remota. Como se ha explicado anteriormente, lo que se distribuye en realidad es una tabla con la información de los nodos, **TablaVistas**.

La clase encargada de crear la maqueta es **MaquetaRemota**, que implementa **IMaquetaRemota** y tiene la función de crear los nodos que forman la vía. Se encarga de controlar el estado de los trenes y de las agujas de la vía. Actúa como servidor poniendo a disposición de los clientes las instancias de los distintos trenes. A su vez, actúa como cliente obteniendo una referencia de las vistas de las maquetas que son distribuidas para poder comunicarles los cambios realizados y así poder actualizarlas.

Por otro lado está el cliente de la aplicación, **GUI_Maq_Cliente**, que obtiene del servidor una referencia de la maqueta y de los trenes y crea una vista con esa información como se ha explicado anteriormente. Maneja el control de los trenes así como el estado de las agujas (mejor dicho indica al servidor que cambie el estado del tren o aguja, el control propiamente dicho lo realiza **MaquetaRemota** en el lado del servidor). También hace la función de servidor proporcionando una instancia de la vista de la maqueta para que el programa servidor le comunique los cambios y pueda actualizar.

Las clases antes mencionadas se resumen a continuación:

GUI_Maq_Servidor: Clase principal, es el servidor de la aplicación aunque también actúa como cliente. Crea una instancia de MaquetaRemota y la distribuye.

MaquetaRemota: Clase principal, es la encargada de crear todos los nodos que forman la vía y de alinearlos para formar la maqueta. Es a la vez servidor y cliente; como servidor pone a disposición de los clientes las instancias de los distintos trenes. Como cliente, obtiene del servidor una referencia de las vistas para actualizarlas.

Servicios	Descripción
TablaVistas getTablaVistas ();	Devuelve una tabla donde esta toda la información de los nodos y su estado.
ITren ponerTren ();	Método para poner un tren en la maqueta.
void agujaRecta (int a);	Cambia una aguja a estado "recto".
void agujaCurva (int a);	Cambia una aguja a estado "curvo".
void suscribirVista ();	Método que recoge las vistas de los clientes para poder comunicarle los cambios posteriormente.
void actualizarVistas ();	Actualiza las vistas de los clientes y del servidor.

GUI_Maq_Cliente: Clase principal, es el cliente de la aplicación aunque también actúa como servidor. Obtiene una referencia de la maqueta y de los trenes y la dibuja. Proporciona una referencia de la vista creada para su actualización.

Servicios	Descripción
void cambiarAguja (boolean r, int a);	Avisa al servidor que cambie el estado de una aguja.
void mouseClicked (MouseEvent evento);	Maneja los eventos del ratón para cambiar el estado de las agujas. Al hacer click sobre las coordenadas donde se encuentra una aguja, cambia el estado de esta.

Tabla 9: Resumen de las clases de Inicios.

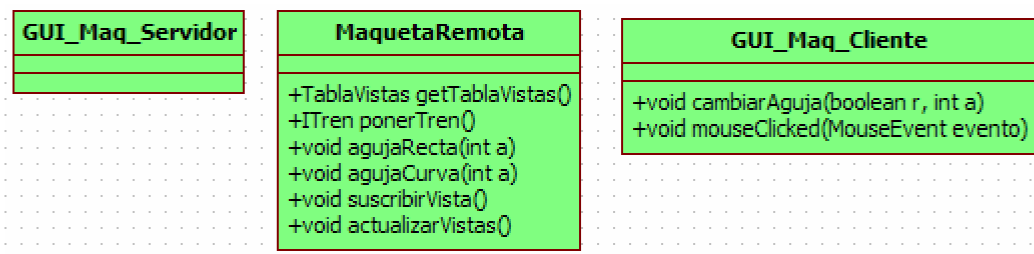


Figura 15: Clases de Inicios.

5.3 Visión dinámica del código.

Se expone a continuación el funcionamiento de la aplicación en forma de pasos:

Paso 1. Se empieza iniciando el Servidor de la aplicación, **GUI_Maq_Servidor**, que lo primero que hace es crear una instancia de **MaquetaRemota**. (Véase en figura 15: llamada 1).

Paso 2. **MaquetaRemota** se encarga de crear todos los nodos que forman la maqueta. Guarda toda la información y estado de los nodos en una **TablaVistas**. Asigna coordenadas a los nodos alineándolos para formar una estructura de vías similar a la maqueta del laboratorio DSIE. También crea tres instancias de trenes y los pone a disposición para que los clientes de la aplicación los invoquen remotamente. (Figura 15: llamadas 2, 3, 5 y 7).

Paso 3. Ahora el Servidor de la aplicación distribuye de forma remota la maqueta instanciada. A su vez, crea una vista con los datos de la maqueta (los consigue de **TablaVistas** creada por MaquetaRemota). Para ello hace uso de la clase **AreaDibujo** y le pasa como parámetro la tabla con la información de los nodos. Suscribe esta vista a la maqueta remota para poder actualizar. (Figura 15: llamadas 4, 6 y 8).

Paso 4. A continuación se inicia el Cliente de la aplicación, **GUI_Maq_Cliente**, que obtiene del Servidor una referencia de la maqueta remota y de los trenes remotos. Con esta información, crea su propia vista de la maqueta, haciendo uso de nuevo de la clase **AreaDibujo**. Esta vista es puesta a disposición del Servidor para que **MaquetaRemota** se pueda suscribir a ella de forma remota y comunicarle los cambios realizados. Para conseguir esto, el Cliente invoca el método remoto **void suscribirVista ();** de **MaquetaRemota** que se conecta al Cliente y obtiene de él una referencia de su vista. (Figura 15: llamadas de 9 a 14).

Paso 5. En la vista del Cliente se incluye también un botón “Poner tren” que, como su propio nombre indica, llama al método remoto de **MaquetaRemota** encargado de poner un tren en la vía. Cuando se pone el primer tren en la vía, se inicia un timer, **tick();** encargado de llevar el tiempo de reloj. Cada tren podrá realizar una acción por cada tick

de reloj. Por cada tren que se sitúa en la maqueta se crea una instancia de **ControladorTren** encargada de controlar dicho tren. Este controlador es un panel con dos botones: "Arrancar", para poner en marcha el tren y "Parar" para detener el tren, más una barra para fijar la velocidad del tren, de -1 a 10 (cuando la velocidad es -1 el tren retrocede). Tanto la barra de velocidad como el botón "Arrancar" llaman al método **void fijarVelocidad (double v);** del tren remoto. El botón "Parar" hace uso del método **void parar ();**. (Figura 15: llamada 15).

Paso 6. Cada vez que un tren avanza o retrocede, o cambia el estado de una aguja, el estado de los nodos cambia. **MaquetaRemota** se encarga de comunicar estos cambios realizados a los clientes. Cuando un nodo cambia de estado, llama al método **void actualizarVistas();** de **MaquetaRemota**. Este método actualiza la vista de la maqueta de cada uno de los clientes a los que esta suscrita con **void update(TablaVistas t);**, es decir, envía de nuevo a los clientes la tabla con la información de los nodos actualizada. Cuando se recibe la tabla con la información de los nodos actualizada, cada Cliente refresca su vista de la maqueta con el método **void refrescar(TablaVistas t);** de **AreaDibujo**. (Figura 15: llamadas 16, 17 y 18).

Paso 7. El Cliente tiene la posibilidad de cambiar el estado de las agujas de la maqueta haciendo click sobre ellas como se ha explicado en capítulos anteriores. Si dos trenes llegan a ocupar un mismo nodo, se lanza una excepción, *ColisionTrenesException*, y se detiene el movimiento de los trenes implicados en la colisión.

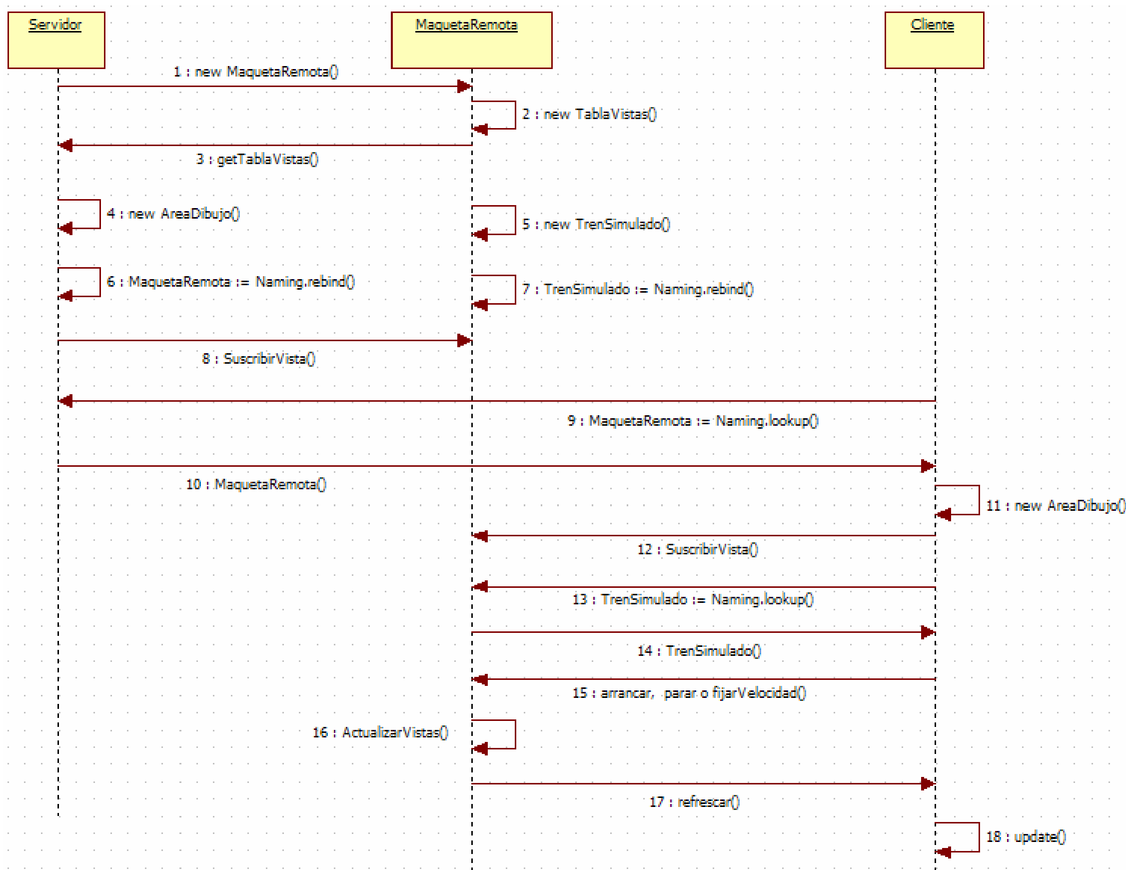


Figura 16: Diagrama de secuencia de inicio de la aplicación.

Capitulo 6.

Instalación y uso del software.

6.1 Introducción.

Todo proyecto software debe incluir una especificación de documentación incluyendo cómo se instala el producto y cómo se utiliza de forma eficaz.
En este capítulo se pretende seguir con esta especificación, indicando en primer lugar cuáles son los pasos o pautas a seguir para instalar y ejecutar el programa, y posteriormente, mostrar cuál es la manera más adecuada de usar el programa en tiempo de ejecución.

6.2 Instalación de la aplicación.

La aplicación se puede instalar y ejecutar de diversas maneras. Todo ello depende de la existencia de un entorno de desarrollo en el PC donde se quiera instalar.

Antes de ejecutar nada, hay que dar unas cuantas indicaciones:

- Se ha de modificar el fichero de configuración de política de seguridad de java, **java.policy**, para permitir la conexión de sockets. Habría que incluir lo siguiente:
  grant {
  permission java.net.SocketPermission ":1024-65535", "connect,accept";*
  permission java.net.SocketPermission ":80", "connect";*
  };
- Lo más sencillo es crear un archivo llamado java.policy con este código y pasarlo como parámetro de seguridad al ejecutar el programa con la directiva:
  “-Djava.security.policy”

- Para el correcto funcionamiento del protocolo RMI, es necesario generar los "Stubs" y "Skeletons" como ya se ha indicado en capítulos anteriores. Java proporciona una herramienta que se encarga de hacer esto mismo, **rmic**.

  rmic -v1.2 Inicios.MaquetaRemota

- **rmiregistry** es un programa proporcionado por Java que admite registrar en él objetos para que puedan ser invocados remotamente y admite peticiones de clientes para ejecutar métodos de estos objetos. Como parámetro se le pasa el puerto donde queremos registrar los objetos.

  start rmiregistry 3015

Lo primero que hay que hacer es guardar los paquetes, con las clases e interfaces en una carpeta. Desde una consola de comandos, nos situamos en el directorio donde están los archivos de la aplicación.

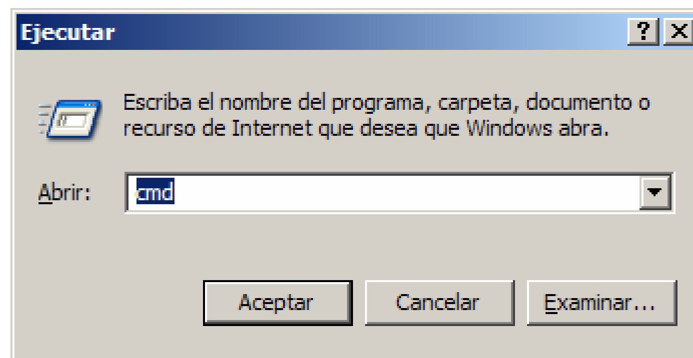


Figura 17: Acceder a una consola de comandos.

Figura 18: Directorio con los archivos y paquetes de la aplicación.

Primero la parte del servidor, se ejecuta por este orden:

rmic -v1.2 Inicios.MaquetaRemota

start rmiregistry 3015

java -Djava.security.policy=java.policy Inicios.GUI_Maq_Servidor

Para facilitar y agilizar la ejecución de estos comandos, se ha creado un archivo bat llamado servidor.bat:

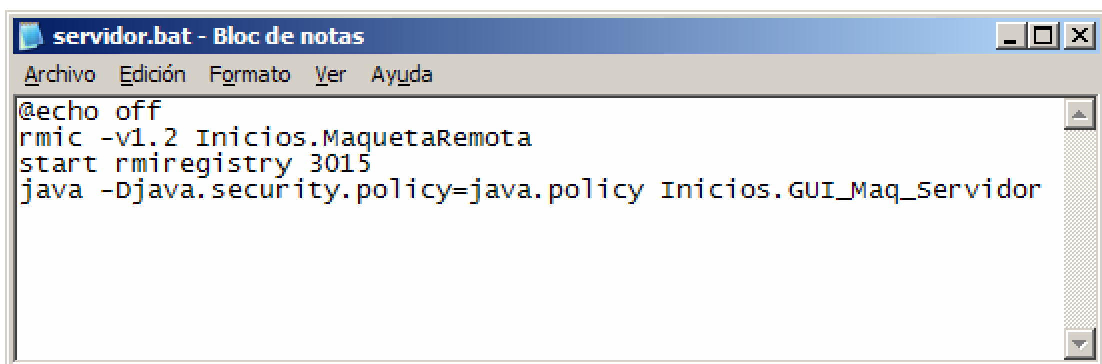


Figura 19: Archivo servidor.bat

Con esto, ya esta en marcha el servidor, ahora se inicia el cliente en otra consola de comandos. Se ejecuta lo siguiente:

java -Djava.security.policy=java.policy Inicios.GUI_Maq_Cliente

También se ha creado un archivo bat para el cliente con el nombre cliente.bat:

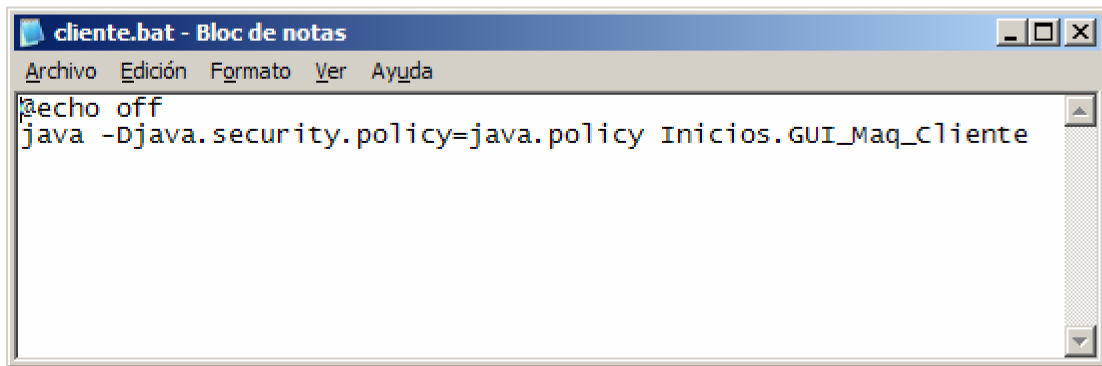


Figura 20: Archivo cliente.bat

6.3 Uso de la aplicación.

Una vez se ejecuta el Servidor, aparece una ventana con la vista de la maqueta tal que así:

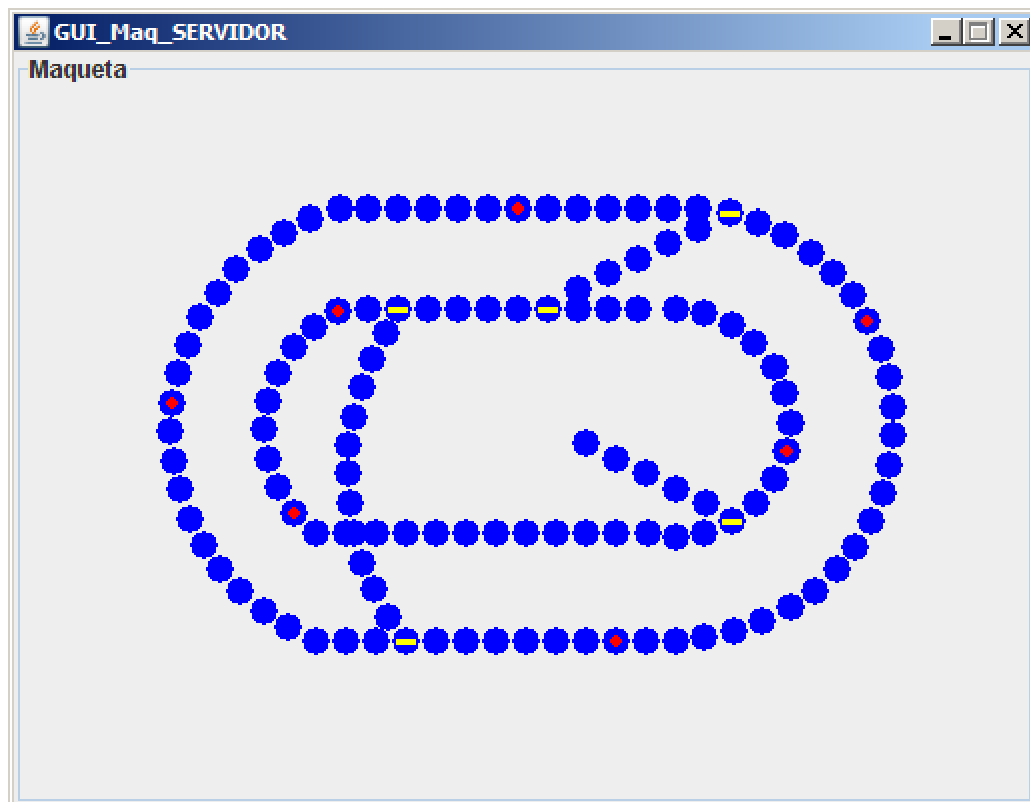


Figura 21: Vista de la maqueta del Servidor.

Al ejecutar el Cliente, aparece una ventana con la vista de la maqueta más un botón con el que ir incluyendo trenes. A modo de nota informativa se indica la forma de cambiar el estado de las agujas:

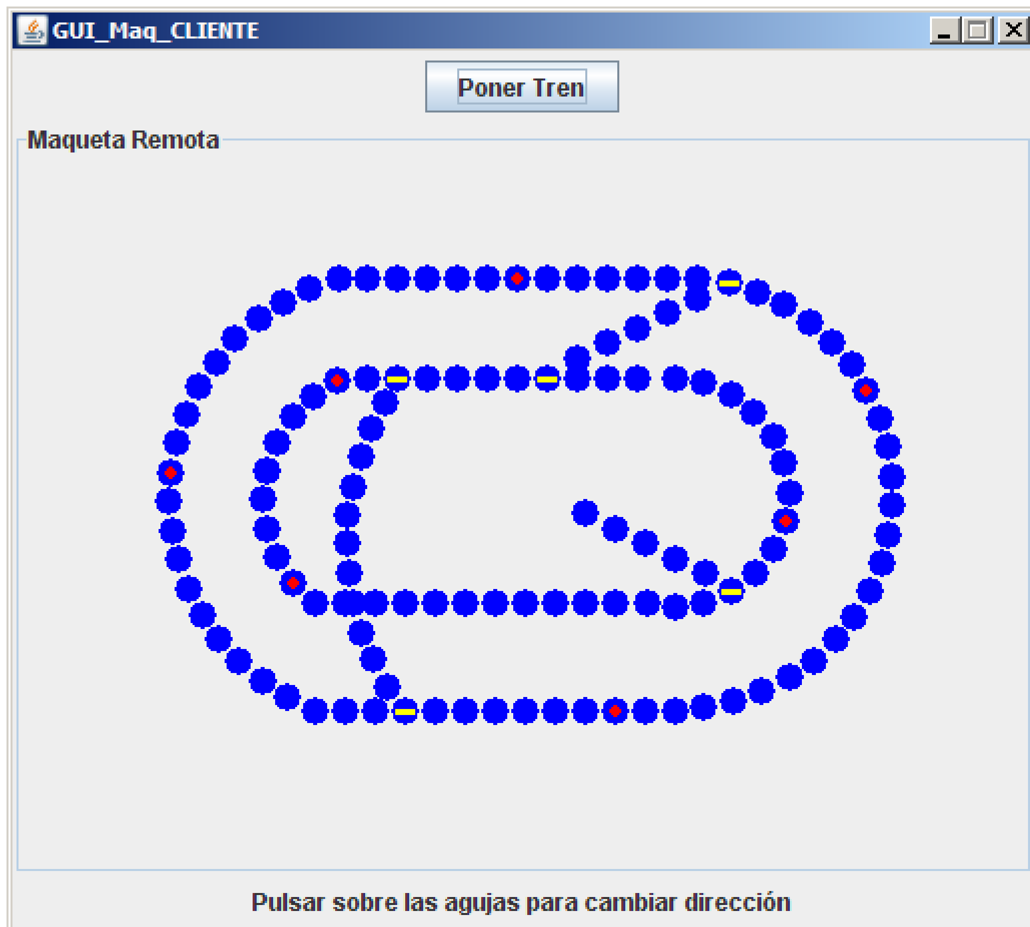


Figura 22: Vista de la maqueta del Cliente.

Cada vez que el cliente pulsa sobre el botón "Poner Tren", se pone un tren en la maqueta y aparece el correspondiente panel de control para ese tren, hasta un máximo de tres:

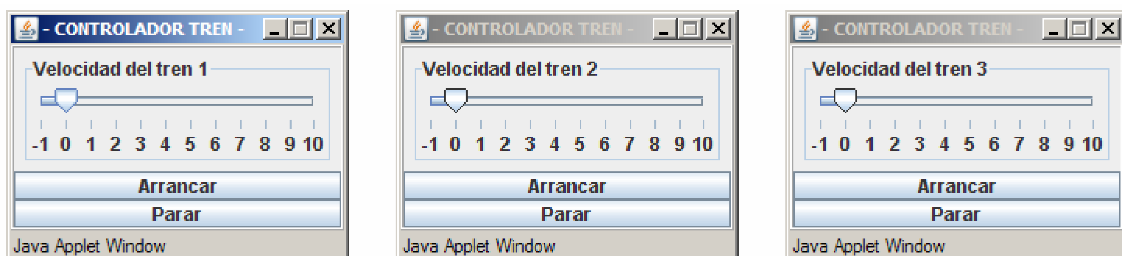


Figura 23: Paneles de control para los trenes.

El resultado total de la aplicación es el siguiente:

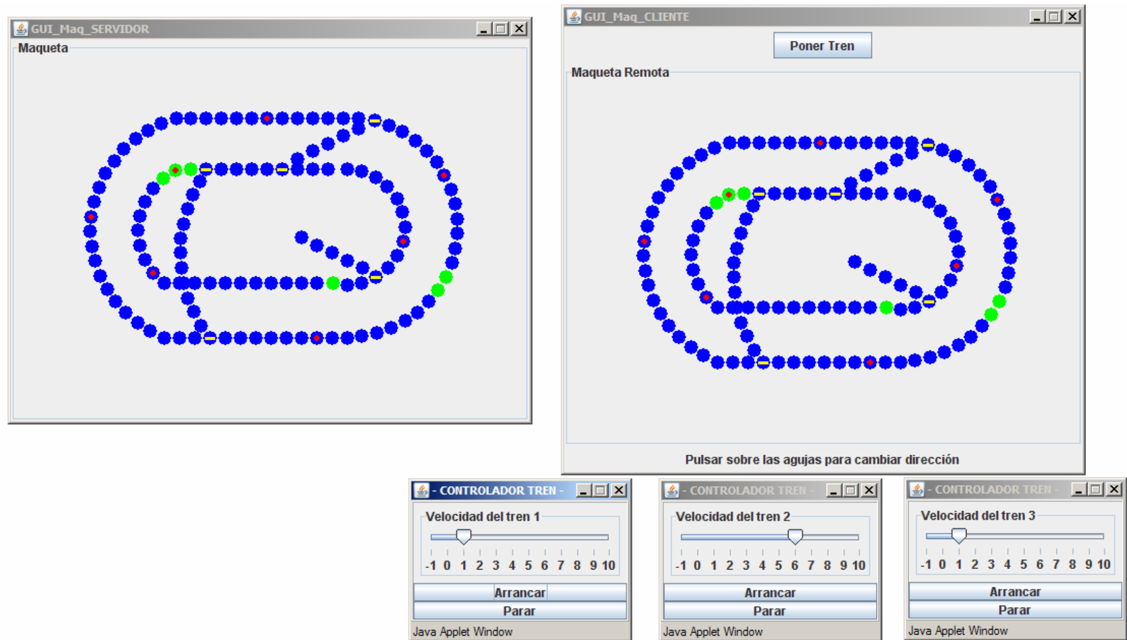


Figura 24: Resultado total de la aplicación.

Capitulo 7.

Conclusiones y líneas de acción futuras.

7.1 Resultados obtenidos.

El resultado final es un simulador de una maqueta de tren con la capacidad de ser distribuido a través de una red.
Es una aplicación consistente, fiable y robusta respetando los requisitos del software enunciados en el capítulo 2. Es multiplataforma al trabajar sobre maquinas virtuales de Java.
Posee una interfaz gráfica amigable e fácilmente manejable, que proporciona a los Clientes, tanto locales como remotos, la información con el estado y situación de los trenes dentro de la maqueta. Actualiza las vistas acorde a los cambios de estado.
Realiza un control independiente de agujas y trenes lo que facilita la mantenibilidad y realización de futuros cambios.

7.2 Conclusiones, aportaciones y líneas de acción futura.

Al término de este proyecto se han conseguido todos los objetivos presentados al principio del mismo. Manteniendo el modelo de componentes software creados en el Proyecto anterior, se ha conseguido reducir drásticamente el número de objetos totales que componían la aplicación lo que ha facilitado enormemente su distribución. Gracias al uso del patrón Flyweight se ha podido prescindir de un gran número de instancias de objetos repetidos o muy similares.
El éxito a la hora de distribuir la aplicación, y donde fallaba el proyecto anterior, radica en que en este caso no se ha tratado de distribuir cada uno de los objetos que la forman, sino distribuir, en forma de array, la información y el estado de los objetos. El modelado y control de la aplicación corren a cargo de Servidor, dejando la representación visual de la misma a los Clientes, separando claramente cada uno de bloques de acuerdo al patrón MVC. Si bien, en el Proyecto anterior ya se hacía uso del mismo, ahora se ha optimizado, sobre todo en el tema de relación entre objetos, al existir un número mucho

menor y estar las partes mas diferenciadas. Una vez hecho esto, se ha rediseñado la aplicación en consecuencia y se ha reimplementado de acuerdo al nuevo diseño.
Se abre ahora un amplio abanico de posibles mejoras concretas sobre este Proyecto, como puede ser incrementar el número de trenes que se distribuyen. Concretamente en este caso, como ya se ha explicado, los Clientes pueden controlar hasta un máximo de tres trenes. Esto es debido básicamente a tema de espacio, pero no habría inconveniente que en una posible futura ampliación de la maqueta, se pudiera incrementar el número de trenes.
También sería interesante un estudio más en profundidad de la forma en que los datos de los objetos y su estado son almacenados, y que al fin y al cabo es lo que pasa a ser distribuido. En este caso se ha hecho uso de un array de objetos debido a que el simulador de la maqueta no es muy grande, pero para una maqueta de mayor tamaño, con el correspondiente incremento de objetos que la forman, se podría haber usado un **hashmap**[1], lo que supondría un incremento de la velocidad. Además, en lugar de enviar a los Clientes todo el array con la información de los objetos cada vez que se actualiza uno de ellos como se hace actualmente, sería más productivo enviar sólo la información de los objetos que cambian de estado. El uso de un hashmap facilitaría esto.
A un nivel mayor, este Proyecto abre multitud de nuevas líneas de desarrollo para diferentes futuros proyectos. Cuando estas tareas son transparentes, se permite abordar mejor los retos. Entre las líneas futuras, cabría destacar, por ser las más relacionadas con este proyecto, las dos siguientes:

1. Automatizar la creación de maquetas, con una herramienta de paletas de elementos.
2. Interfaz gráfica más lograda, incluyendo tal vez afectos de sonido.

[1] Un **hashmap** es una estructura de datos que permite crear un mapa en memoria para la rápida identificación de elementos a partir de un dato usado como llave.

Bibliografía y referencias.

[1] **eclipse.org**

[2] **java.sun.com**

[3] **Simulador Java de una maqueta de trenes.**
José Andrés Martínez Muñoz.
Proyecto Fin de Carrera, Escuela Técnica Superior de Ingeniería de Telecomunicaciones. Universidad Politécnica de Cartagena, 07-2007.

[4] ***Design Patterns: Abstraction and Reuse of Object-Oriented Design.***
Erich Gamma et al.
Springer-Verlag.

[5] ***J2EE Design Patterns.***
William Crawford, Jonathan Kaplan.
O'Reilly September 2003.

[6] ***Java™ Remote Method Invocation Specification.***
Sun Microsystems, Inc.

[7] ***Java RMI.***
William Grosso.
O'Reilly.

[8] **Pattern-Oriented Software Architecture Volume 1: A System of Patterns**
Frank Buschmann et al
Hardcover