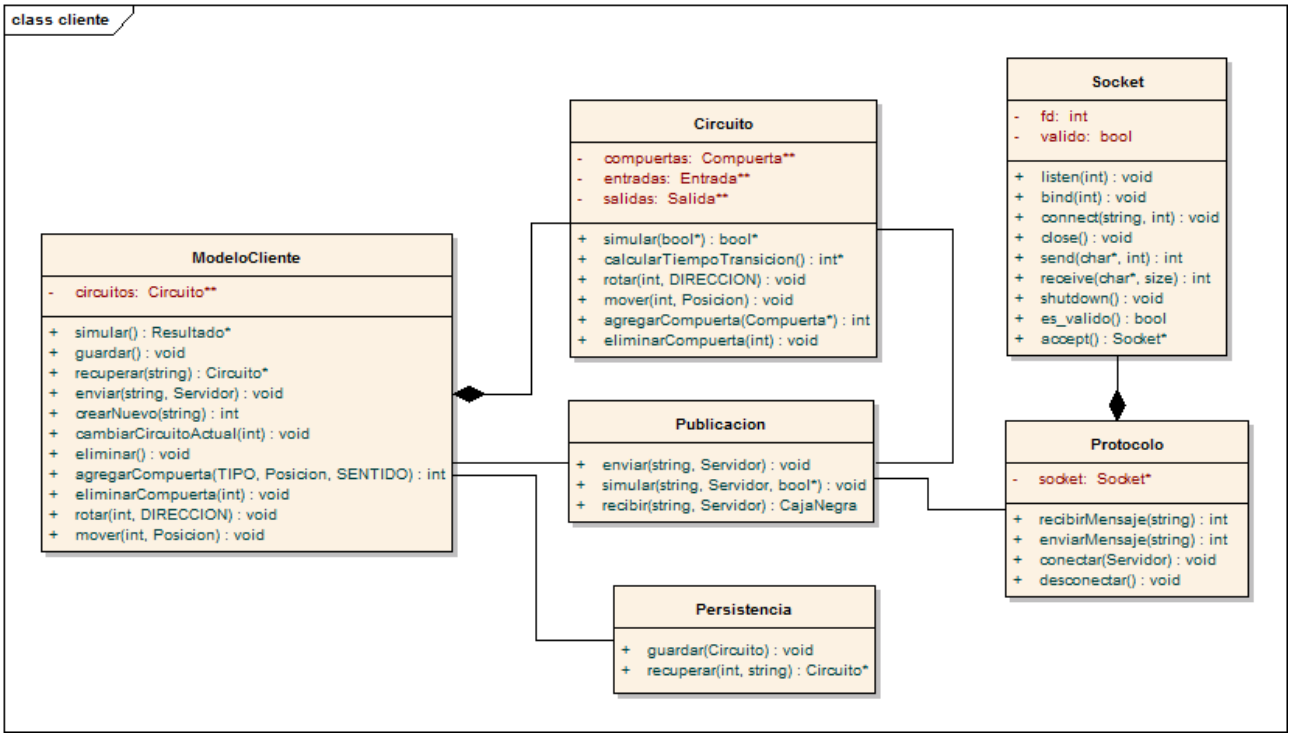
Manual Técnico
Simulador de circuitos lógicos distribuidos

Índice de contenido

ModeloCliente



Circuito

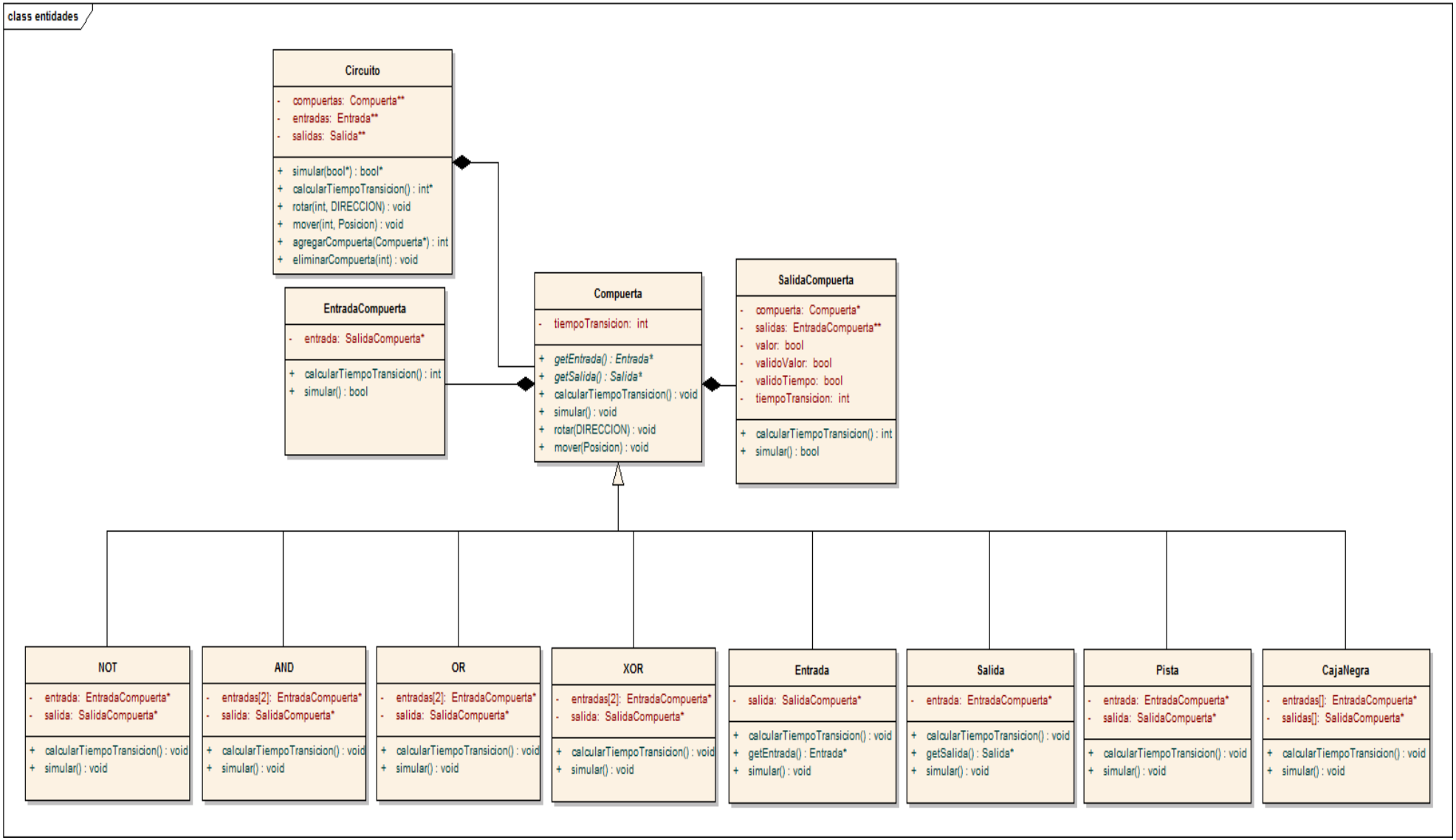
El circuito esta formado por las clase Circuito, Compuerta, EntradaCompuerta y SalidaCompuerta.

La clase Compuerta es abstracta y heredan de ella las compuertas especificas (NOT,AND,OR,XOR,PISTA,ENTRADA,SALIDA,CAJA NEGRA). Cada compuerta esta formada por EntradaCompuerta y SalidaCompuerta. Por ejemplo la AND tiene dos EntradaCompuerta y una SalidaCompuerta y la NOT tiene una de cada uno.

Las clases EntradaCompuerta y SalidaCompuerta se utilizan para poder llevar a cabo la conexión entre las compuertas. Una EntradaCompuerta se puede conectar a una unica SalidaCompuerta, pero una SalidaCompuerta puede conectarse a varias EntradaCompuerta.

Las compuertas tienen un id, para poder ser identificadas desde el módulo vista/controlador, una posición y un sentido. Con estos últimos dos atributos se pueden llevar a cabo las conexiones entre compuertas y además son utilizados por la vista para dibujar a las compuertas.

Por último la clase Circuito contiene una lista de Compuertas, y tiene el comportamiento de administrarlas.


class entidades
Circuito
- compuertas: Compuerta**
- entradas: Entrada**
- salidas: Salida**
+ simular(bool*) : bool*
+ calcularTiempoTransicion() : int*
+ rotar(int, DIRECCION) : void
+ mover(int, Posicion) : void
+ agregarCompuerta(Compuerta*) : int
+ eliminarCompuerta(int) : void
EntradaCompuerta
- entrada: SalidaCompuerta*
+ calcularTiempoTransicion() : int
+ simular() : bool
Compuerta
- tiempoTransicion: int
+ getEntrada() : Entrada*
+ getSalida() : Salida*
+ calcularTiempoTransicion() : void
+ simular() : void
+ rotar(DIRECCION) : void
+ mover(Posicion) : void
SalidaCompuerta
- compuerta: Compuerta*
- salidas: EntradaCompuerta**
- valor: bool
- validoValor: bool
- validoTiempo: bool
- tiempoTransicion: int
+ calcularTiempoTransicion() : int
+ simular() : bool
NOT
- entrada: EntradaCompuerta*
- salida: SalidaCompuerta*
+ calcularTiempoTransicion() : void
+ simular() : void
AND
- entradas[2]: EntradaCompuerta*
- salida: SalidaCompuerta*
+ calcularTiempoTransicion() : void
+ simular() : void
OR
- entradas[2]: EntradaCompuerta*
- salida: SalidaCompuerta*
+ calcularTiempoTransicion() : void
+ simular() : void
XOR
- entradas[2]: EntradaCompuerta*
- salida: SalidaCompuerta*
+ calcularTiempoTransicion() : void
+ simular() : void
Entrada
- salida: SalidaCompuerta*
+ calcularTiempoTransicion() : void
+ getEntrada() : Entrada*
+ simular() : void
Salida
- entrada: EntradaCompuerta*
+ calcularTiempoTransicion() : void
+ getSalida() : Salida*
+ simular() : void
Pista
- entrada: EntradaCompuerta*
- salida: SalidaCompuerta*
+ calcularTiempoTransicion() : void
+ simular() : void
CajaNegra
- entradas[]: EntradaCompuerta*
- salidas[]: SalidaCompuerta*
+ calcularTiempoTransicion() : void
+ simular() : void

Simulación

Para la simulación se creo una clase Simulador que se encarga de generar las entradas y simular el circuito para cada combinación de entradas.

Para llevar a cabo la simulación cada compuerta tiene un método simular, el cual llama al simular de las compuertas conectadas a sus entradas obteniendo los valores de entrada y así transformarlos en la salida. De esta manera el circuito va a recorrer la lista de Salida llamando al simular de cada uno. Cada salida va a llamar al simular de la compuerta conectada a su entrada, y ésta va a llamar al simular de la compuerta conectada a su entrada y así sucesivamente hasta llegar a la Entrada.

Para implementar el método simular se utilizó el patrón Template. La clase Compuerta va a recorrer las entradas llamando al simular y luego va a llamar al método actuarSimular que va a estar definido en las clases concretas y se va a encargar de transformar las entradas en la salida

Persistencia

La persistencia esta desarrollada utilizando la librería de Apache Xerces C++, que es la encargada de guardar los archivos en formato XML.

La clase Persistencia es la encargada de cargar todos los archivos a memoria y de cualquier accion que requiera de identificar elementos en formato XML en un archivo. Por lo tanto para levantar un circuito en memoria, se utiliza la funcion *recuperarCircuito(),* la cual parsea el archivo, crea el circuito y devuelve un puntero al mismo.

A la hora de guardar un circuito en memoria, la funcion *guardarCircuito()* crea el documento y llama al metodo *guardar()* de la clase *Circuito*. Este metodo recorre la lista de compuertas y llama al *guardar()* de cada una, implementado polimorficamente ya que la clase *Compuerta* lo define como virtual puro. Por lo tanto cada compuerta sabe como debe persistirse y la clase *Persistencia* queda desligada de esta obligacion.

En la clase *Persistencia* tambien se encuentran los metodos para generar el XML en formato SOAP, para cumplir asi el protocolo y poder adjuntarlo al codigo HTML.

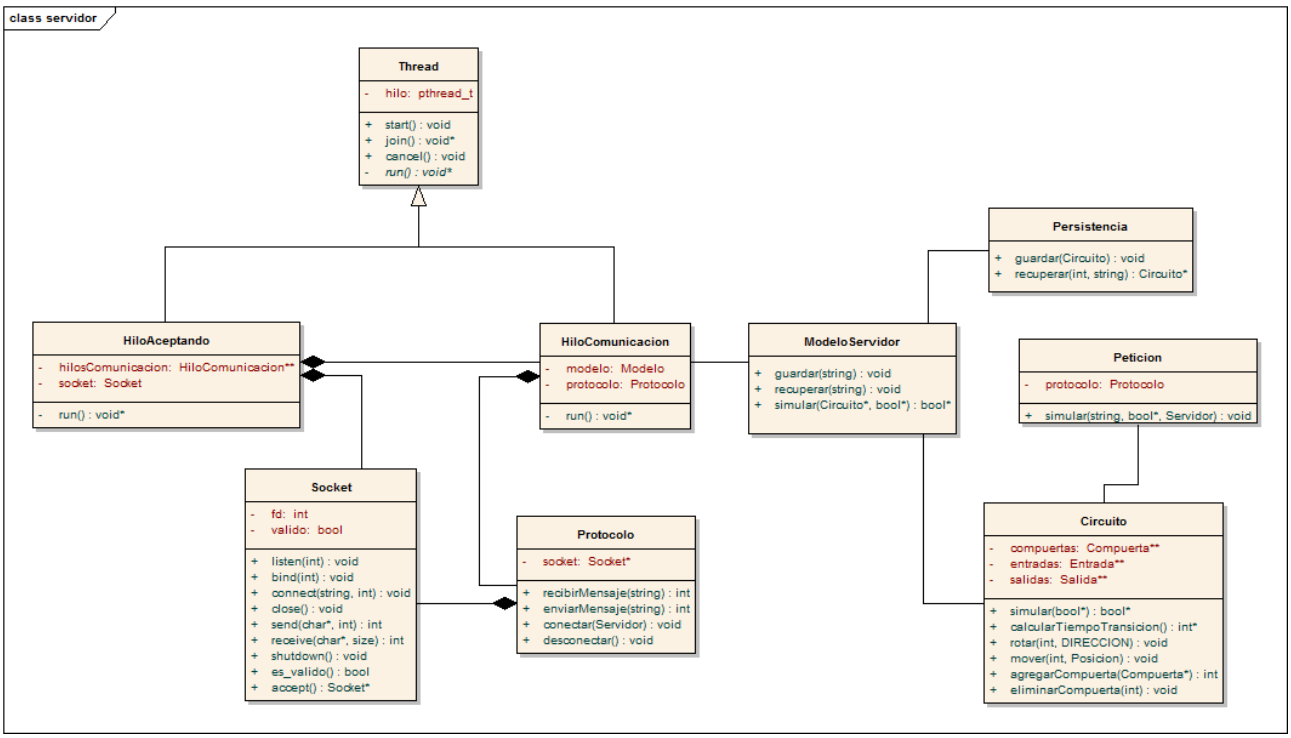
Publicación

La *Publicacion* esta compuesta por la clase *Mensajes, Publicacion* y *Servidor.* La clase *Mensajes* es la encargada de generar los mensajes en un elemento de formato XML, para que luego la clase *Publicacion* genere los mensajes SOAP, para enviarlos al servidor. Luego de enviado un mensaje, espera su respuesta e identifica el mensaje respuesto por el Servidor, para asi continuar con la aplicación de la forma requerida.

El *Cliente* tiene la posibilidad de publicar sus circuitos guardados, de descargar circuitos del servidor subidos por otros clientes, o sus propios circuitos. Todos los circuitos descargados se visualizan como una caja negra con sus respectivas entradas y salidas. Una vez conectada una caja negra de la forma adecuada, se procede a la simulacion.

La simulacion de una caja negra esta a cargo del servidor que la contiene, por lo tanto el *Cliente* debe enviar un mensaje por cada conjunto de entradas asignados, para poder obtener las salidas de la misma. Lo mismo sucede con los tiempos de la simulacion, pero es un solo mensaje ya que el tiempo no depende de los valores tomados en las entradas.

Servidor



El *Servidor* es el encargado de responder a las peticiones de los distintos *Clientes,* siendo las distintas posibilidades: publicaciones de circuitos, solicitudes de circuitos, solicitudes de simulacion. Y a su vez debe poder funcionar como cliente para poder pedir la simulacion a otro servidor en caso de tener un circuito guardado que contenta una caja negra publicada en otro servidor o en si mismo.

Para soportar la concurrencia se utiliza un hilo por cada comunicación establecida, el *Servidor* tiene un hilo de acpetaciones, que esta constantemente aceptando peticiones y generando un hilo para cada una.

Existe un manager de archivos, el cual maneja la cantidad de archivos recibidos y enviados, asignandoles nombres distintos para que no ocurran problemas entre clientes. El manager de archivos tambien es el encargado de borrar los archivos generados temporalmente.

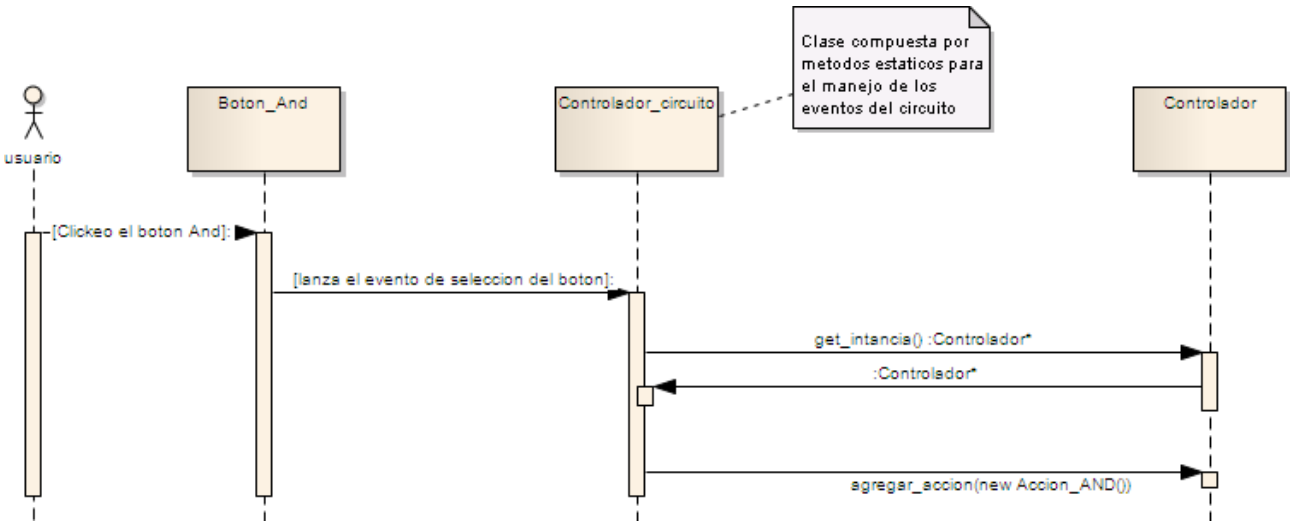
Controlador

El controlador definirá el modo en que la interfaz reacciona a la entrada del usuario, por lo que será el encargado de manejar los eventos emitidos por la vista de la aplicación.

Esta clase esta compuesta principalmente por el modeloCliente, que es el objeto de la aplicación y por la fachadaVista, que es la una interfaz para la comunicación con la vista.

También entre sus atributos encuentra el modelo de la vista (Modelo_Vista_circuito), que facilitará la decisión del procedimiento a seguir a la hora de recibir un evento producido por la entrada del usuario.

El atributo acción, también juega un papel muy importante en el control del programa. Este representa la estrategia a seguir al recibir eventos provenientes del sector de la vista donde se diseñan los circuitos. Para explicar la utilidad de este componente incluimos el siguiente diagrama de secuencias:

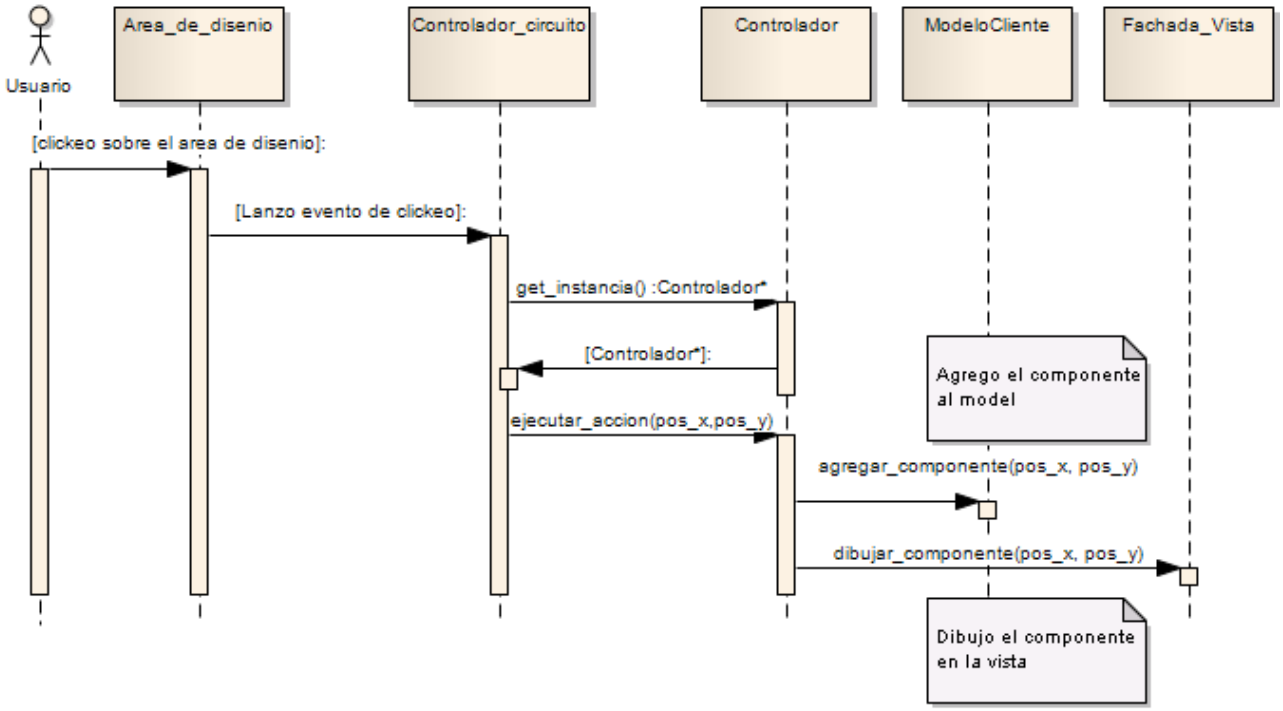


Como se puede apreciar en el diagrama, el usuario utiliza la herramienta para agregar una compuerta del tipo AND, la cual lanza un evento que es capturado por el el controlador del mismo.

Antes de continuar con la explicación vale la pena mencionar el objetivo de la clase controlador_circuito, dado a que no se incluyo en el diagrama de clases. Esta clase contiene todos los manejadores de los eventos relacionados al diseño del circuito.

Luego de capturar el evento, se obtiene la instancia del controlador y se ejecuta el método para agregarle una nueva acción al controlador, la acción de incluir una compuerta AND al modelo en este caso.

A partir de este momento, el controlador incorpora la estrategia a seguir al recibir la orden de ejecutar_acción, que es la requerida por el controlador del evento de clickeo sobre el área de diseño. A continuación mostraremos la secuencia que se sigue cuando el usuario clickea sobre el área de diseño.



El usuario clickea sobre el área de diseño, y el controlador_circuito captura el evento que esto produce, pide una instancia de la clase controlador e invoca al método ejecutar acción.

Debido a que anteriormente le agregamos la acción de incorporar una compuerta AND, el controlador agrega la compuerta en el modelo y la dibuja en la vista, en el caso de que se haya podido incorporar el componente con éxito.

Por ultimo, vale la pena mencionar, la decisión de implementar la clase controlador como un singleton. Esto fue principalmente debido a dos cuestiones, primero, la aplicación solo tendrá un controlador que se encargara de ser el intermediario entre el modelo del programa y sus posibles vistas, y segundo, la necesidad de que el acceso a esa única instancia sea global, para poder ser utilizada desde las funciones controladoras de eventos.

Vista

La vista, es la representación de la aplicación en pantalla, con la cual el usuario va a tener la posibilidad de interactuar con el programa.

Como detalle de implementación, mencionaremos primero la utilización de la clase componente_Visual, que representará todos los componentes que se utilizaron para la vista. De esta manera podemos utilizar cualquier elemento visual como una objeto de esta clase.

Luego de definir esta clase, implementamos el patrón de diseño decorador, esto lo logramos a partir de heredar de componente visual la clase decorador, la cual tendrá como composición a otro componente_Visual. Esta decisión nos facilito decorar componentes de la vista, como por ejemplo con barras de scroll, o cajas contenedoras.

Como ultimo detalle de la vista, destacaremos el uso de la clase fachada, la cual tiene como objetivo, proporcionar una interfaz para la comunicación con cualquier objeto de la vista. Facilitando la interacción del controlador con la representación del programa en pantalla.