Implementación de una interfaz de usuario sin contacto

Autor: Javier Onielfa Belenguer
Director: Francisco José Abad Cerdá
Fecha: Septiembre 2011

Índice general

Resumen

En este proyecto final de carrera se va investigar una posible evolución de las interfaces de usuario existentes. En una época donde el auge de las pantallas táctiles ha copado el sector de los dispositivos móviles, como los teléfonos y las tabletas, este proyecto pretende dar un paso más hacía un sistema que no necesite del contacto físico como entrada. Gracias a este proyecto se desarrollará una herramienta que permite integrar diferentes dispositivos de entrada, como el Microsoft Kinect, en nuestras aplicaciones.

Durante el desarrollo del proyecto se tomó como punto de partida la tecnología en la que se iba a apoyar el mismo. Se investigaron dos dispositivos diferentes. El primero de ellos fue el Microsoft Kinect. Este es un dispositivo diseñado por Microsoft para su uso con la videoconsola Xbox 360. Permite la utilización de la misma sin necesidad de ningún mando. En segundo caso está una plataforma llamada Arduino. Éste no es un dispositivo por si mismo, si no que consiste en un microcontrolador que puede conectarse a una gran variedad de sensores.

El proyecto se basa en una estructura cliente/servidor. El cliente en el dispositivo. que reconocerá el gesto que ha realizado el usuario. A continuación, enviará el gesto reconocido al servidor. En la parte del servidor se encontrará un sistema de reglas. Una vez recibido el gesto que ha realizado el usuario buscará la regla que valida las condiciones en las que se encuentra el sistema (gesto del usuario, aplicación con el foco). Una vez encontrada se ejecutarán las acciones asociadas a la regla.

En conclusión, se desarrollará un sistema que permitirá combinar diferentes dispositivos de entrada para desarrollar interfaces de usuario avanzadas. Este sistema está basado en reglas. Además este proyecto proporciona un editor de reglas para que el usuario pueda personalizar su instalación de forma sencilla.

Capítulo 1

Introducción

1.1 Presentación del problema

Desde que se introdujeron al mundo los ordenadores personales, (en inglés PC Personal Computer), uno de los principales aspectos de diseño de los mismos era el referido a la interfaz entre el usuario y el ordenador personal.

Antes de la introducción de los ordenadores personales, que fue propiciado por avances como el microprocesador, los ordenadores eran sistemas grandes y muy caros. El público al que iban dirigidos estos sistemas eran grandes corporaciones, gobiernos y universidades. Debido a que estos usuarios no demandaban un uso continuo de interacción con el sistema, sino que preparaban tareas para el computador en procesos fuera de línea, la interacción con el mismo no era un área dónde se hubieran realizado avances e investigaciones.

A medida que los sistemas mainframe comenzaron a proliferar entre las grandes empresas, universidades y gobiernos, apareció un problema de utilización del mismo. En estos ambientes el número de personas que necesitaban hacer uso del computador era cada vez mayor. Cada uno de ellos necesitaba realizar una tarea distinta y era físicamente difícil hacer que todas las personas introdujeran las tareas en el sistema.

Para solucionar éste problema se inventaron las terminales o consolas. La idea era muy sencilla: si el sistema iba a ser accedido por mucha gente y estas personas no iban a estar cerca del sistema, la mejor solución es instalar un sistema externo desde el cual los usuarios pudieran enviar sus tareas desde sus puestos de trabajo.

Básicamente las terminales o consolas son sistemas hardware que sirven de unión entre el sistema principal o mainframe y el usuario. Las terminales permitían introducir las tareas para ser ejecutadas en los sistemas mainframe. En algunos casos eran tarjetas perforadas, en otras terminales también se incluía un teclado para introducir las ordenes tecleándolas.

En este tipo de casos, las tareas no eran unos procesos opacos en los cuales el usuario mandaba el conjunto de instrucciones a realizar y se finalizaba la interacción con el sistema. Éste debía, en la mayoría de los trabajos, devolver un feedback o resultado de la tarea. En este sentido, las terminales también tenían que tener en cuenta cómo mostrar el resultado obtenido. Al principio simplemente imprimían el resultado en papel pero a medida que la tecnología avanzaba se introdujo una pantalla para mostrar los resultados.

Alcanzado este punto, las máquinas mainframe se convirtieron en unos sistemas de tiempo compartido. Durante el tiempo que realizaban tareas, recibían otras muchas provenientes de otros usuarios. Se empezaron a desarrollar todos los

algoritmos de administración de tareas para alternar entre las distintas solicitudes de los usuarios.

Figura 1: IBM 3279. Terminal utilizada para comunicarse con mainframes IBM.
Fuente: http://en.wikipedia.org

Dado que este tipo de organización se popularizó, el mundo de la informática empezó a investigar e invertir recursos en como diseñar las terminales o consolas. En la Figura 1 se puede ver el modelo IBM 3279. Éste modelo se trata de una terminal de entrada mediante teclado. El diseño de las terminales estaba profundamente ligado al avance de las tecnologías en los mainframes. Obviamente un terminal debía implementar todo tipo de mecanismos que permitiera al usuario utilizar el 100% de las posibilidades del mainframe. Por otra parte el ordenador debía ser capaz de reconocer, interpretar y procesar todas las posibles entradas procedentes de las terminales.

La verdadera revolución se produjo cuando se empezaron a diseñar e introducir los primeros microprocesadores. Este cambio supuso que esas terminales, que anteriormente solo servían para enviar y recibir datos al mainframe, comenzaron

a tener capacidad de procesamiento. Debido a este avance la tupla terminal-mainframe fue evolucionando hasta lo que hoy en día conocemos por PC (Personal Computer).

Los ordenadores personales se volvieron independientes. Estos no requerían de ningún mainframe al que conectarse para procesar las peticiones de los usuarios. A medida que los ordenadores personales aumentaban su potencial estos se popularizaban.

Llegados a este punto, el mundo de las interfaces de usuario tenía que evolucionar para adaptarse al ritmo de crecimiento de los PC.

Figura 2: Apple II. Uno de los PC más famosos de los 80.
Fuente: http://en.wikipedia.org

Durante mucho tiempo los únicos interfaces de los que disponía el usuario eran un teclado y una pantalla. Mediante el teclado el usuario podía introducir las órdenes para que el ordenador las ejecutase. A su vez, éste último, devolvía el resultado o los posibles errores a través del uso de la pantalla. En la Figura 2 se muestra el ordenador personal Apple II. Como se puede observar su método de entrada está basado en un teclado. Además posee dos entradas para disquetes.

Éste paradigma, basado en una linea de comandos, estuvo presente durante décadas y consiguió establecerse como el estándar entre los ordenadores personales. Requería que el usuario aprendiese de memoria los comandos a utilizar y exigía un conocimiento del uso del sistema operativo en general.

El siguiente paradigma de interfaz de usuario es el conocido como entorno de escritorio. Ésta interfaz de usuario está directamente relacionada con un nuevo dispositivo que fue introducido en 1968, el ratón.

En un sistema con una interfaz de usuario de tipo escritorio, ya no se interactúa mediante comandos introducidos por el teclado. El escritorio es un entorno gráfico donde cada elemento o acción que se pueda realizar en el ordenador está representada gráficamente. Gracias al ratón se puede seleccionar una posición en pantalla e indicar qué acciones a realizar.

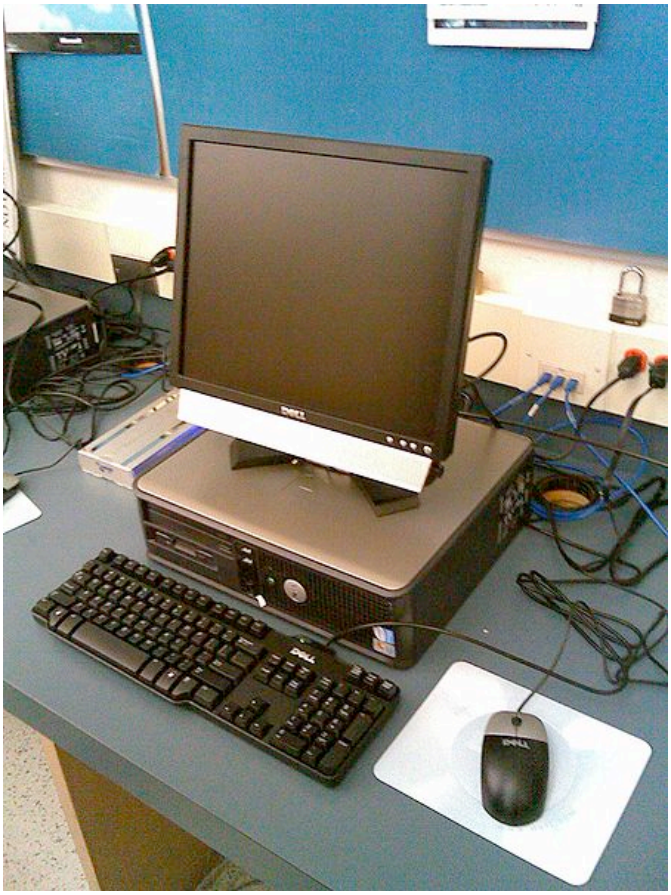

Figura 3: Dell OptiPlex. Ordenador personal con ratón y teclado.
Fuente: http://en.wikipedia.org

En la Figura 3 vemos una configuración de ordenador personal que posee un teclado así como un ratón. Éste avance supuso una gran revolución en el mundo de los ordenadores personales. Permitió que muchas personas ajenas al mundo de la informática pudiesen utilizar un ordenador fácilmente. Principalmente se debe a que el usuario ya no debía aprenderse de memoria todos los comandos posibles. Simplemente debía hacer uso de su memoria gráfica y acordarse de los elementos con los que interactuar para llevar acabo sus propósitos.

Por último han empezado a extenderse las soluciones táctiles para la interacción con los sistemas. En el mundo de la telefonía móvil cada vez existen más terminales que prescinden de los teclados e incorporan una única pantalla táctil. Gracias a ella el usuario puede realizar las mismas tareas que realizaba anteriormente, pero usando únicamente sus dedos. Se ha demostrado que este sistema es bastante cómodo si el diseño de la interfaz de usuario es el adecuado. Gracias al éxito de este tipo de dispositivos, se introdujo al mercado otro producto que se encuentra a medio camino entre los ordenadores personales y la telefonía móvil: las tabletas.

Estos aparatos, como el iPad de la figura 4, en la actualidad incluyen una pantalla multitáctil de entre 7 y 10 pulgadas y su utilización se asemeja a la de un dispositivo móvil pero aprovechando las bondades que le concede el poseer una pantalla considerablemente más grande.

Figura 4: Apple iPad
Fuente: http://en.wikipedia.org

Estos dispositivos táctiles empezaron a incluir un método de entrada importante para este proyecto final de carrera: los gestos.

Un ejemplo de sistema basado en gestos es Graffiti para el sistema operativo Palm OS. Éste está diseñado para utilizarse sobre PDAs que tienen como entrada una pantalla táctil. El usuario a través del uso de un lápiz táctil podía indicar las acciones que deseaba realizar. Gracias al sistema Graffiti se podía introducir información haciendo uso de gestos.

Figura 5: Gestos reconocidos por la aplicación Graffiti 2 para Palm.
Fuente: http://en.wikipedia.org

En la Figura 5 podemos ver una imagen con los gestos que son reconocidos por la versión 2 del sistema Graffiti. Cada uno de los trazos representa o bien una letra de vocabulario o bien un número.

A partir de la proliferación de estos sistemas, se incluyó el reconocimiento de los gestos realizados por el usuario sobre la pantalla táctil dentro de los sistemas operativos. De esta forma, el usuario podía realizar acciones tales como cerrar aplicaciones, cambiar entre aplicación, mostrar todas las aplicaciones, etc, realizando gestos con los dedos. Éste método también se ha incorporado a sistemas como los portátiles, donde los trackpad son sensibles a varios dedos y permiten gestos.

1.2 Objetivos

En este proyecto final de carrera se explorará una de las posibles evoluciones de las interfaces, buscando maneras de simplificar el uso de los sistemas informáticos.

Como se ha visto en el pequeño recorrido que se ha realizado sobre la evolución de las interfaces de usuario, estas han evolucionado hacía modelo más cómodos para el usuario. Al principio había que tener un conocimiento muy profundo de la máquina a utilizar. Lo que se pretende es que el conjunto de acciones que puede realizar el usuario, se acerque lo máximo posible a un lenguaje más natural para el mismo.

En la búsqueda de un nuevo modelo de interfaz de usuario que cambie la experiencia en el uso de un ordenador, el paso más obvio sería una interfaz de usuario sin contacto. Como su propio nombre indica, este modelo de interacción no precisa de ningún contacto físico entre el sistema y el usuario. Para su funcionamiento hay que diseñar y establecer un “lenguaje” gracias al cual el ordenador sea capaz de comprender qué acción desea realizar el usuario. Para ello el sistema usará sensores de entrada, que evalúen e interpreten las acciones de el usuario.

Un programa cliente en el dispositivo se encargará de administrarlo y de comunicar al sistema los gestos que sea reconocidos. El sistema realizará una búsqueda de las acciones a realizar y se las comunicará al sistema operativo.

El sistema estará basado en gestos, gracias a los cuales el usuario podrá comunicarse con la máquina. El sistema, una vez haya reconocido el gesto, realizará las acciones correspondientes al mismo. Por lo tanto, constará de dos partes bien diferenciadas: reconocimiento de gestos y administración de reglas y acciones.

El objetivo de este proyecto es proporcionar al usuario un sistema donde se incluyan estas dos partes. En la administración de reglas, el usuario debe ser capaz de definir qué acciones realizar al reconocer un gesto en concreto y en qué situación se debe realizar.

Por otra parte, el reconocimiento de gestos debe realizar una lectura del gesto realizado por el usuario y lanzar todas las acciones asociadas.

1.3 Descripción del sistema

A continuación se dará una descripción a alto nivel de cómo ha sido diseñado el sistema y qué ventajas conllevan la elecciones tomadas.

Como se puede apreciar en la Figura 6, el sistema ha sido diseñado de tal forma que sirve de nexo entre los clientes que reconocerán los gestos y el sistema operativo. Windows 7 [1] ha sido el SO elegido siendo también compatible con Windows Vista [2].

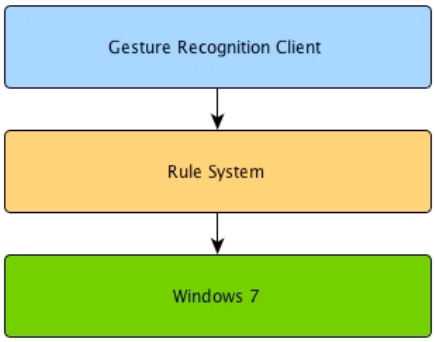

Figura 6: Interacción cliente, sistema de reglas y sistema operativo.

Como dispositivos de entrada se puede utilizar cualquier sistema que sea capaz de reconocer acciones de usuario. Como se puede apreciar en la Figura 7, el sistema es compatible con Microsoft Kinect [3], Arduino [4] o un sistema de reconocimiento de voz, entre otros.

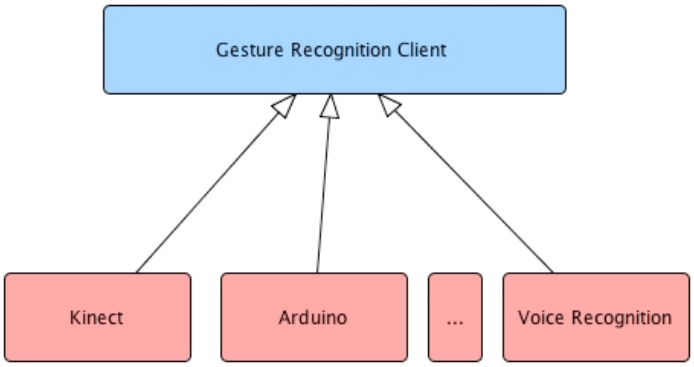

Figura 7: Diferentes tipos de dispositivos compatibles

Si se desea utilizar únicamente un dispositivo de entrada la mejor solución es incluir el sistema de reglas dentro del cliente del dispositivo. De esta forma la comunicación entre el dispositivo y el sistema de reglas es directa. No es necesario montar ningún tipo de servidor. Esta configuración es la más sencilla si se dispone de un dispositivo que está conectado al mismo ordenador en el cual se van a ejecutar las acciones.

Una vez el sistema de reglas ha recibido la cadena de texto que contiene el gesto a procesar, este realiza una búsqueda en la base de reglas que tiene almacenadas. Comprobará también que se cumplen otras posibles condiciones, por ejemplo, que una aplicación en concreto tenga el foco del sistema operativo en el momento de la ejecución del gesto.

Cuando el sistema ha encontrado la regla a ejecutar, entonces recorre todas las acciones asociadas a la regla y las va ejecutando una a una. En ese momento es cuando el sistema de reglas envía mensajes al sistema operativo para que éste ejecute las acciones.

Las acciones incluidas son:

- Abrir, cerrar, maximizar y minimizar aplicaciones.
- Control del volumen del sistema operativo.
- Mostrar escritorio.
- Envío de pulsaciones de teclas y atajos de teclado.

Este es el proceso normal de ejecución de una regla perteneciente al sistema. A continuación se explicará qué ventajas supone el haber elegido esta estructura o modelo.

Como se ha podido observar, este sistema tiene una estructura separada por capas lo que permite que se puedan utilizar distintas versiones y que el resto del sistema siga funcionando. Esto aporta una gran flexibilidad en la elección del hardware ya que lo único que se necesita es que el código que controla el dispositivo llame al sistema de reglas con el gesto reconocido.

El sistema proporciona una serie de funciones para la administración de toda la información contenida en la base de reglas. Así el usuario podrá realizar acciones como:

- Añadir, modificar y borrar reglas.
- Añadir, modificar y borrar acciones asociadas a una regla.

Otro de los aspectos importantes en lo referentes a las acciones, es que el sistema está diseñado de tal forma que sería relativamente sencillo adaptarlo a otro sistema operativo. Esto es así ya que las acciones tienen sus funciones localizadas en fichero único. Adaptando éste fichero al sistema operativo deseado y recompilando el código fuente, el sistema sería totalmente funcional.

1.4 Estructura del resto de la memoria

Hasta aquí se ha explicado el funcionamiento general del sistema. En las siguientes páginas de la memoria se dará una visión general de los antecedentes en este área de investigación. Asimismo se comparará con la solución proporcionada.

A continuación, se dará detalle de todo el proceso de análisis que se ha llevado a cabo. Se explicará cómo los requerimientos se han sustentado en las ventajas e inconvenientes de la tecnología de los dispositivos. Se dará detalle sobre cómo fluye la información en el diseño final así como todas las decisiones que de alguna forma dieron o eliminaron flexibilidad. Se determinará de qué será capaz el sistema.

Después del análisis de la solución propuesta y de haber establecido sus límites, se procederá al diseño del mismo. Las clases más relevantes se describirán mediante diagramas UML. Se explicará en detalle cómo fluye la información entre las distintas capas. Asimismo, se dará una visión de la persistencia de las reglas y las razones por las cuales se ha elegido XML como el sistema de almacenamiento.

Una vez el diseño está acabado se procederá a evaluar el resultado final. Para ello se implementará el sistema en varios ejemplos prácticos. El primero de ellos es desarrollar una interfaz de usuario que permita administrar las reglas y las acciones fácilmente. También se implementarán varios ejemplos donde se usarán dos dispositivos (Kinect y Arduino) dentro de nuestro sistema.

Por último se ofrecerá una visión general de lo que se ha podido concluir de este proyecto. Asimismo se analizarán los problemas que se han encontrado y cómo se han solucionado. Además se propondrán posibles ampliaciones y nuevos usos del sistema.

Capítulo 2

Antecedentes

2.1 Reconocimiento de voz

Diseñar un sistema de reconocimiento de voz es un problema que es muy estudiado. Esto es debido a que su aplicación en el mundo real es muy grande y variada. Durante mucho tiempo se ha intentado encontrar un sistema que sea realmente usable en un entorno real. Lamentablemente esto no ha podido ser así debido a que una serie de problemas hacen que este sistema sea muy complejo de implementar.

Cuando nos comunicamos con el sistema operativo se realizan básicamente dos tipos de entradas. La primera de ellas es la utilización del ratón para indicar al sistema operativo que deseamos realizar una operación especifica. Dentro de este tipo de operaciones puede estar abrir un programa, cerrarlo, maximizar una ventana, cambiar de escritorio, etc. La otra gran operación es la escritura de texto en el ordenador.

Como se puede deducir esta serie de operaciones, sobre todo la escritura de texto, son fácilmente modelables en un sistema de reconocimiento de voz. Para las operaciones que se realizan con el ratón se les dará un nombre unívoco que los represente.

Sin embargo los sistemas que reconocen voz no son muy precisos debido una serie de características que lo hacen complejo de resolver.

En los sistemas de entrada tradicionales, como el ratón y el teclado, el sistema operativo recibe la misma entrada con diferentes usuario. Sin embargo, cuando se trata de un sistema de voz, las entradas son diferentes entre distintos usuarios. Cada uno de ellos tiene una voz característica, lo que hace que un mismo comando pueda ser representado con diversos sonidos. Además de la dificultad añadida si se desea incluir diferentes idiomas al sistema [5]. Esto implica que el sistema de reconocimiento de voz debe de ser lo suficientemente flexible como para poder reconocer el mismo comando proveniente de distintos usuarios.

Otro de los aspectos a tener en cuenta, es que la comunicación usando el lenguaje entre humanos es una característica que se aprende desde pequeños y que es parte de nuestra vida. El hombre es un ser expresivo, y cuando se comunica con otros humanos no lo hace de una forma simple y llana. La pronunciación de palabras va acompañada de una serie de factores que aportan mucha información a la comunicación. Aspectos como la entonación o la ironía son factores clave que hay que tener en cuenta en el proceso de reconocimiento [5].

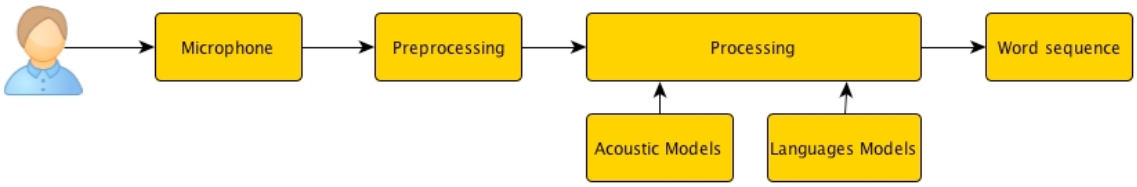

Figura 8: Diseño de un sistema de reconocimiento de voz.

En la figura 8 podemos ver la estructura de un sistema de reconocimiento de voz. El sistema funciona de la siguiente forma:

El usuario a través del micrófono comunica cual es la orden que desea ejecutar. Éste se encarga de grabar el sonido y enviarlo al preprocesado. En este etapa, se trata el sonido para eliminar imperfecciones así como partes que no sean necesarias, como silencios. A continuación comienza el verdadero proceso del sistema de reconocimiento. En esta etapa se utilizan modelos acústicos y modelos del lenguaje para poder construir la secuencia de palabras articuladas por el usuario.

Figura 9: Sistema Ford Sync en un automóvil Ford.

Una de las aplicaciones que se puede encontrar en la vida real es la que se muestra en la Figura 9. El sistema Ford Sync permite que el conductor utilice todos los elementos del coche como radio, GPS, móvil, etc, sin quitar las manos del volante. Para ello incorpora un sistema de reconocimiento de voz mediante el cual el usuario se comunica con dichos dispositivos.

Para finalizar, el sistema de reconocimiento de voz es una de las áreas más investigadas porque el encontrar una solución robusta abriría nuevos campos en muchas otras áreas. Personas con problemas para utilizar ratón y teclado así

como otros ámbitos fuera del uso de un computador podrían beneficiarse de esta solución.

2.2 Reconocimiento de gestos mediante movimientos oculares

El ser humano dispone de cinco sentidos para percibir el mundo y poder relacionarse con él. De entre todos ellos el que está más desarrollado es sin duda el de la vista. Gracias a él podemos percibir muchas características del mundo real y de los elementos que lo conforman. El color, tamaño, forma y textura son ejemplos de factores que podemos identificar de los elementos que nos rodean.

Junto con el reconocimiento del habla, que tratamos en el punto anterior, el reconocimiento de movimientos oculares ha sido un área de investigación muy importante. No sólo en el mundo de la computación se investiga si no también en el mundo de la psicología, neurociencia y la publicidad.
En estas últimas áreas la investigación se ha realizado buscando entender cómo el ser humano recibe la información del mundo real a través de la vista. De esta forma se puede investigar como el ser humano procesa la información mediante la vista y adaptar los productos a el mismo [6].
Por otra parte, en el mundo de la computación se ha buscado otra finalidad distinta a la descrita anteriormente. Así como antes se busca entender como funciona el sentido de la vista, en la computación se busca un modo de hacer que el ser humano se comunique con el ordenador a través de la vista. Para ello se apoyará en el movimiento de los ojos así como en el parpadeo.

Para realizar ésta comunicación entre el usuario y el ordenador debe existir alguna forma de captar los movimientos que realice el ser humano. Este es un problema para el cual existen una gran variedad de soluciones.

Una de las soluciones es la llamada electrooculograma. En este sistema se implantan una serie de electrodos cerca de los músculos del ojo. De esta forma se puede medir la diferencia de potencial que se aplica en los músculos. Gracias a esta información se infiere cual ha sido el movimiento realizado por el usuario [6].

Otra solución menos intrusiva consiste en montar un detector de movimiento en unas lentes de contacto. Por lo tanto el usuario no necesita ningún elemento que le limite el movimiento de la cabeza. Es una de las soluciones más precisas que existen pero tiene inconvenientes. Resulta un sistema muy incomodo para el usuario además de que este sistema proporciona el movimiento del ojo respecto a la cabeza [6].

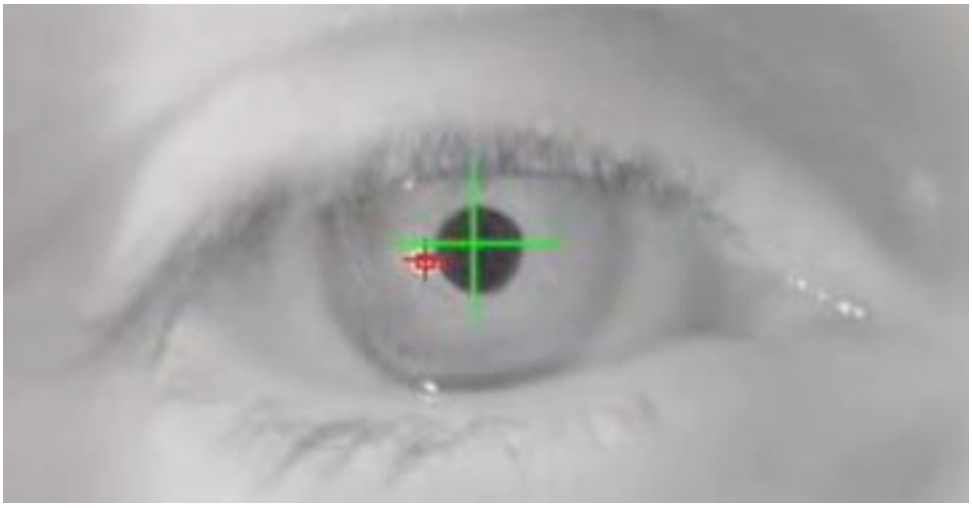

Figura 10: Sistema de visión por computador que reconoce la posición de la pupila del usuario.

Por último, entre las soluciones intrusivas, está la que consiste en montar un sistema en la cabeza del usuario. Éste tendrá como parte de él un cámara apuntando a hacia el ojo del usuario. En algunos casos se pueden montar una cámara por ojo. Gracias a técnicas de reconocimiento de imágenes, se puede detectar hacia donde está apuntando el usuario, así como el parpadeo del mismo. En la Figura 10 se observa una imagen del sistema de reconocimiento que indica la posición de pupila.

Como se ha podido ver, estos sistemas son bastante intrusivos en el físico del usuario, lo que implica que el usuario puede no sentirse cómodo utilizando este sistema diariamente. Sin embargo también existen otros sistemas que no son intrusivos.

Figura 11: Sistema Tobii TX300

En la Figura 11 vemos el sistema Tobii TX300 [7] que permite el seguimiento de los ojos sin ningún elemento intrusivo. El Tobi TX300 consiste en una serie de cámaras que trabajan a una velocidad de 300Hz. De esta forma se pueden capturar cada uno de los movimientos de los ojos. Todo esto se realiza a través de un sistema de reconocimiento de pupilas.

Las ventajas de este sistema es que no es para nada intrusivo, lo que permite que se pueda utilizar fácilmente por distintos usuarios. Además se puede integrar fácilmente con la pantalla.

El reconocimiento de movimientos del ojo se utiliza mucho cuando el usuario presenta alguna deficiencia física que le impida utilizar el ordenador con normalidad. Se abrió un nuevo mundo cuando éstas técnicas se aplicaron a personas que sufrían de invalidez severa que no les permitía mover ninguna parte del cuerpo excepto los ojos.

2.3 Elliptic Labs

Elliptic es una empresa noruega que se dedica a diseñar y crear nuevas formas de interactuar con un ordenador. Proceden del grupo de investigación sobre procesamiento de señal de la Universidad de Oslo [8]. La compañía se especializado en el diseño de sistemas de interacción sin contacto. Para ello utilizan un sistema mediante ultrasonidos que les permite detectar en cada momento el movimiento de la mano.

Figura 12: Dock para iPad que utiliza la tecnología de Elliptic Labs

Como se puede observar en la Figura 12, la empresa ha diseñado un prototipo para Apple iPad que permite al usuario realizar gestos que se envían al dispositivo. Este utiliza una banda que emite ultrasonidos. Midiendo la perturbación en los mismos el sistema es capaz de detectar cuales son los movimientos realizados por el usuario.

De momento es un prototipo y no se tiene más información acerca de si existirá una API o si se comercializará la base del iPad.

2.4 Sony EyeToy

Sony EyeToy [9] es un dispositivo diseñado por Sony e ideado para ser utilizado sobre la plataforma PlayStation 2. Como se puede observar en la Figura 13, el hardware de este dispositivo es muy sencillo ya que consta de una cámara web conectada a través de un puerto USB 1.0.

Figura 13: Sony EyeToy
Fuente: http://en.wikipedia.org

Las características técnicas son muy básicas. Posee una resolución de 320x240 pixeles así como dos LEDs que indican al usuario si se dan ciertas situaciones de luz. Además incluye un micrófono interno que proporciona la opción de realizar comandos de voz. El potencial de este dispositivo no reside en su hardware, si no en el software que lo utiliza.

EyeToy utiliza tecnología de visión por computador así como de reconocimiento de gestos para explotar su potencial. Gracias a este tipo de algoritmo el sistema es capaz de ofrecer al usuario una gran variedad de elementos de interacción.

Para poder ejecutar todo esto el sistema necesita que exista una cantidad de luz constante y suficiente en la habitación. Sin ésta las imágenes capturadas por el sensor no tendrán las suficiente calidad como para poder ser utilizadas. Para informar sobre ello EyeToy utiliza unos de los LEDs disponibles. Si este LED se enciende significa que no hay suficiente luz.

Figura 14: Imagen del juego EyeToy Play Sports
Fuente: http://es.playstation.com

En la Figura 14 podemos observar una captura de pantalla del videojuego EyeToy Sport. En este juego podremos realizar una serie de pruebas físicas utilizando nuestro cuerpo como forma de interactuar con el sistema. En este juego se permite hasta cuatro jugadores simultáneos.

El éxito de este dispositivo ha sido muy grande. Se han vendido más de 10.5 millones de unidades en el mundo y se han desarrollado más de 26 juegos solamente para EyeToy y más de 40 compatibles con el dispositivo.

2.5 Microsoft Kinect

El dispositivo Microsoft Kinect [3] da un paso más allá dentro del mundo de los videojuegos. Este dispositivo está diseñado para ser utilizado sobre la plataforma Xbox 360. Utiliza el mismo concepto que la Eyetoy de Sony pero sobre una tecnología mucho más potente.

Figura 14: Microsoft Kinect para Xbox 360
Fuente: http://www.xbox.com

En la figura 14 se puede ver una imagen del dispositivo Microsoft Kinect. Es un elemento que está formado por los siguientes componentes:

- Cámara de video
- Proyector de luz infrarroja
- Cámara de video infrarroja
- Matriz de cuatro micrófonos

Gracias a estos elementos el sistema es capaz de extraer información de la escena. El proyector de luz infrarroja emite un patrón de luz sobre el escenario que capta la cámara de infrarrojos. A partir de esta información es capaz de calcular un mapa de profundidades de la escena. Por otra parte, al incorporar un matriz de micrófonos el dispositivo es capaz de recibir comandos de voz. Gracias a estos micrófonos puede realizar cancelación del ruido ambiente así como localizar el punto exacto de emisión del sonido.

Figura 15: Imagen del videojuego Just Dance 3

Una de las opciones más utilizadas es el reconocimiento del cuerpo humano. En la figura 15 vemos una imagen del juego Just Dance 3. Este videojuego consiste en utilizar tu cuerpo para bailar según el ritmo de la música. Dependiendo de lo bien que realices los movimientos tendrás más o menos puntuación.

El gran avance de este dispositivo es que permite que personas que normalmente no se acercaban al mundo de las videoconsolas, lo hagan. También permite jugar en grupo.

Capítulo 3

Dispositivos de entrada

Una de las tareas iniciales de este proyecto final de carrera, fue hacer un estudio de qué dispositivos había disponibles para la implementación de nuestro sistema. Estos debían presentar una serie de características comunes que permitieran ser utilizados mediante conjunto de reglas. Se analizarán en profundidad los dispositivos Microsoft Kinect y Arduino.

Cada dispositivo de entrada debe ser capaz de reconocer algún gesto realizado por el usuario e identificarlo unívocamente. Como se describirá más adelante, cada evento se identificará mediante una cadena de texto única.

3.1 API en los dispositivos de entrada

Una API (Application Programming Interface) es un conjunto funciones y procedimientos que proporciona al usuario una capa de abstracción en la utilización de un software o hardware. Cualquier dispositivo de salida debe proporcionar una API que permita la comunicación con el mismo.

Gracias a esta API se puede crear un programa que interprete la información generada por el dispositivo. A través de ellos se deducirá qué gesto ha realizado el usuario, además de información adicional que puede ser de utilidad. Un ejemplo sería la velocidad con la que se ha ejecutado un gesto o con cuál de las dos manos se ha realizado.

3.2 Microsoft Kinect

Microsoft Kinect es un dispositivo desarrollado por Microsoft y diseñado para ser utilizado en la videoconsola Xbox 360.

Figura 16: Microsoft Kinect para Xbox 360
Fuente: http://en.wikipedia.org

Es un sistema que permite al jugador interactuar con la máquina sin necesidad de tener ningún contacto físico con la misma. Kinect está basado en una interfaz natural de usuario que permite reconocer gestos, comandos de voz y objetos.

En la Figura 16 se puede observar cual es el aspecto del Microsoft Kinect. En él se agrupan todos los sensores que posee y que se detallarán a continuación.

Kinect captura imágenes VGA gracias a una cámara de video integrada de 640 x 480 píxeles y 8 bits de profundidad de color por canal.
Éste dispositivo es capaz de reconstruir la escena en 3D gracias a dos elementos incluidos: un proyector infrarrojo y una cámara de infrarrojos.
El proyector de infrarrojos emite un patrón estructurado de puntos.

Figura 17: Imagen del patrón infrarrojo proyectado por Kinect
Fuente: http://en.wikipedia.org

Una vez proyectado el patrón sobre el escenario, éste rebota en los objetos y es captado por la cámara de infrarrojos, como se puede ver en la Figura 17. La cámara posee una profundidad de 11 bits lo que proporciona hasta 2048 niveles.

Una vez Kinect tiene la imagen infrarroja se procesa en el propio hardware. El procedimiento es el siguiente: se compara el patrón con la imagen que ha sido capturada por la cámara. Observando las perturbaciones en las distancias entre puntos contiguos se puede calcular cual es la diferencia en profundidad. Gracias a esto se puede construir una mapa de profundidad de la habitación.

Para completar los elementos que conforman el Kinect, éste proporciona una matriz de 4 micrófonos además de un motor que permite levantar o bajar el punto de vista de las cámaras. El hecho de incluir 4 micrófonos permite que se pueda realizar una localización de la fuente del sonido así como una cancelación del ruido de fondo.

A continuación se describirán dos API de libre distribución que se pueden usar para implementar aplicaciones que usan Kinect.

3.2.1 OpenKinect

OpenKinect [10] es una libreria de código abierto que permite comunicarse con el Kinect. Es multiplataforma, lo que permite que se puede utilizar bajo sistemas Windows, Linux y Mac.

Está escrita en C pero existen una gran variedad de wrappers que permiten su uso bajo otros lenguajes como: Python, Actionscript, C++, C#, Java JNI, Java JNA, Javascript y Common Lisp. Este aspecto le proporciona una gran versatilidad a la librería porque permite que sea utilizada en distintos ambientes, como el web.

Respecto a las funciones que proporciona la API, estas son unas funciones de alto nivel que proporcionan accesos a los datos generados por el Kinect.

```
public class KinectDevice {
     public KinectDevice();
     public void setLEDStatus(LEDStatus option);
     public LLEDSTATUS getLEDStatus();
     public void setMotorPosition(float);
     public float getMotorPosition();
     public IntImage getRGBImage();
     public IntImage getDepthImage();
}
```

Figura 18: Clases públicas para el acceso a Kinect mediante OpenKinect

Como se puede observar en la Figura 18, OpenKinect proporciona acceso a la imagen RGB de la cámara y al mapa de profundidades. También proporciona acceso al LED y al motor que controla la inclinación.

3.2.2 OpenNI

El framework OpenNI [11] es un conjunto de APIs que proporcionan acceso a dispositivos denominados “de interacción natural”. OpenNI no es una librería que esté diseñada para el uso y control de Kinect sino que está diseñada para funcionar sobre cualquier dispositivo que proporcione ciertas características. A diferencia de OpenKinect, OpenNI proporciona una capa de abstracción que va más allá de la información relacionada con los sensores.

OpenNI utiliza un concepto llamado “production node” (nodo de producción) que representa un objeto que produce información de un tipo especifico. En OpenNI se incluyen los siguientes nodos de producción relacionados con la información del sensor:

- Device: Este nodo es el utilizado para representar un dispositivo, como Kinect, y poder configurarlo.
- Depth Generator: Se encarga de generar información relacionada con un mapa de profundidades.
- Image Generator: Genera información relacionada con una cámara instalada en el sistema.
- IR Generator: Nodo que genera imágenes capturadas por la cámara infrarroja

• Audio Generator: Nodo que permite la obtención de datos de audio.

OpenNI proporciona una capa de abstracción por encima de los “production node” enumerados anteriormente. Ésta implementa una serie de algoritmos que son esenciales para el desarrollo de interfaces de usuario naturales.

OpenNI denominada a estos,que también son “production node”, con el calificativo de “middleware”. Se incluyen cuatro nodos de producción que representan cuatro grandes áreas. Éstas son las que se detallan a continuación:

• Gestures Alert Generator: Genera callbacks cada vez que se reconoce un gesto específico.
• Scene Analyzer: Analiza la escena, siendo capaz de diferenciar entre el fondo, primer plano, suelo y figuras humanas.
• Hand Point Generator: Genera callbacks para el seguimiento de una mano.
• User Generator: Genera una representación en 3D del cuerpo del usuario en la escena.

Como se puede observar, estos elementos permite una gran versatilidad en el desarrollo de interfaces naturales. Si se desea manipular la información proveniente de los sensores de Kinect directamente o bien se puede hacer uso de los nodos de producción middleware que nos proporcionan herramientas ya implementadas.

En este proyecto se utilizará principalmente el nodo de producción “Gestures alert generator” ya que permite obtener cuál de los gestos registrados en el sistema ha realizado el usuario.
Su uso está basado en callbacks, gracias a las cuales solo nos tenemos que preocupar de implementar la función que va a ejecutar cuando se detecte el gesto.

A continuación se describe el uso de los callbacks en el caso de la detección de gestos:

Dentro de OpenNI existe una clase llamada xn::GestureGenerator que es la encargada de controlar todo lo relacionado con la generación de eventos de gestos. Para su utilización hay que seguir los siguientes pasos:

• Se crea una variable de tipo xn::GestureGenerator, por ejemplo:

```
xn::HandsGenerator g_HandsGenerator;
```

• Se inicializa el contexto de la variable utilizando el siguiente método:

```
g_GestureGenerator.Create(context);
```

• A continuación se registra la función que se ejecutará cuando se detecten los gestos:

```
g_GestureGenerator.RegisterGestureCallback
                                (Gesture_Recognized,
                                 Gesture_Process,
                                 NULL, h1);
```

• Por último se agregan los gesto a reconocer:

```
g_GestureGenerator.AddGesture("Click",NULL);
```

El primer parámetro es la función que se ejecutará cuando el gesto ha sido reconocido. El segundo parámetro es la función que se ejecuta mientras se procesa el gesto pero aún no ha sido reconocido.

Como conclusión, podemos observar que esta API proporciona un buen número de funciones que pueden ahorrar mucho trabajo si se está trabajando a un nivel más alto. Además con la ventajas de ser un sistema que no depende exclusivamente del Microsoft Kinect si no que puede ser utilizado bajo cualquier hardware compatible.

Ésta API es la que se utilizará para los ejemplos de uso con el sistema de reglas.

3.2.3. Microsoft Kinect SDK

Poco antes de escribir esta memoria, Microsoft anuncio la salida de un SDK (Software Development Kit) [12] oficial. Está aún en estado beta, por lo tanto, se recomienda su uso únicamente en experimentos y no en productos finales.

Microsoft Kinect SDK da soporte para la utilización del array de micrófonos, lo que permitiría la utilización de comandos de voz.

Lamentablemente, este SDK no incluye ninguna opción respecto al reconocimiento de gestos. Únicamente da acceso a los datos de imagen y profundidad, por lo que habría que implementar los algoritmos de detección.

A pesar de estar en fase beta, la ventaja de este producto es que hay una gran compañía detrás que conoce a la perfección el hardware del Microsoft Kinect.

3.3 Arduino

Arduino [4] es una plataforma de hardware libre que consta de una placa con un microcontrolador y además un entorno de desarrollo que facilita el uso de la misma.

Básicamente es un sistema que consta de un microcontrolador así como de unos puertos de entrada y salida. Es capaz de tomar información, a través de sus pins, de una gran variedad de sensores. Además es posible controlar a través de ella LEDs, actuadores, etc.

Existen en total 10 modelos de Arduino, los cuales tienen características diferentes que los hacen adecuados para tareas muy específicas.

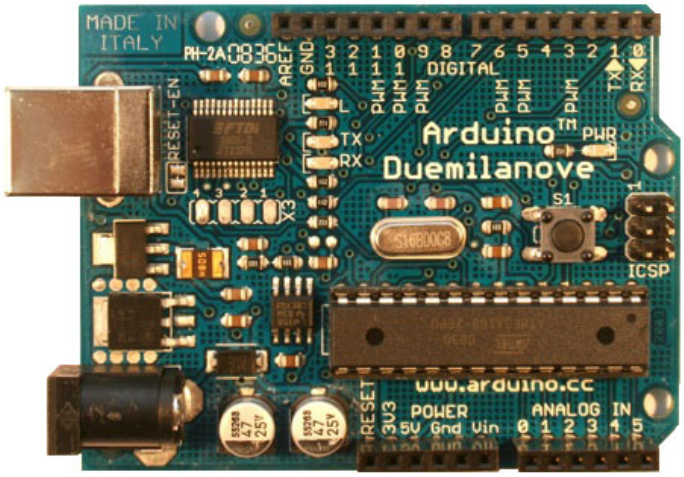

Figura 19: Arduino Duemilanove.

En la Figura 19, podemos ver una imagen del modelo Arduino Duemilanove. Como se puede observar este consta de un microcontrolador, en este caso un ATmega168, y de 20 pins. 14 de estos son pins de entrada y salida digitales y 6 son de entrada analógica.

La mayoría de los modelos de Arduino se comunican con el ordenador a través de un puerto USB. En realidad, este funciona como un puerto virtual serie, por lo tanto para el programador toda la comunicación es en serie.

La potencia del Arduino reside en la gran cantidad de accesorios que se le pueden conectar.

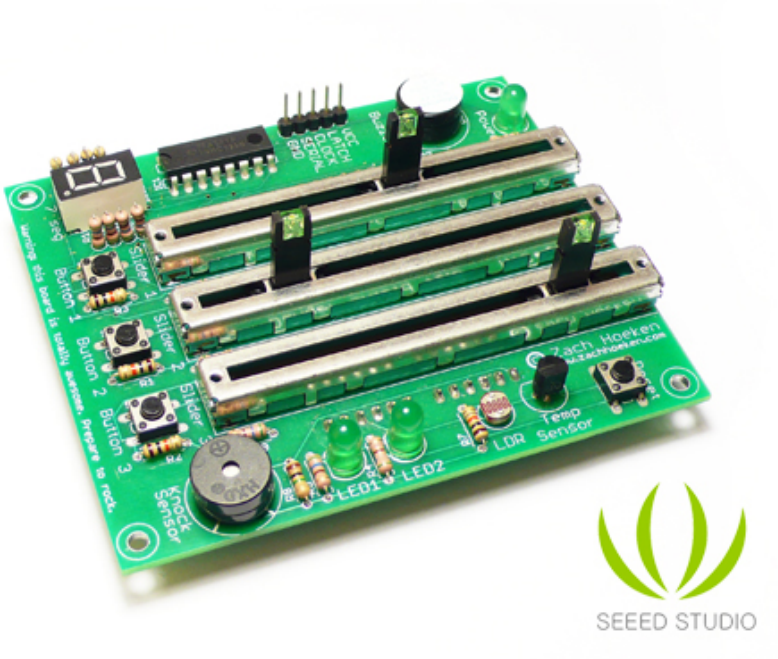

Figura 20: Danger Shield 1.0

En la Figura 20 y podemos ver una placa de sensores Danger Shield 1.0. Esta placa tiene conectada una gran variedad de sensores y actuadores. Entre otros, hay LEDs, zumbadores, botones, un sensor de luz, un sensor de temperatura, etc.

Se pueden comprar y conectar sensores o actuadores específicos, pero este tipo de placas nos dan la posibilidad de tener toda una serie de elementos reunidos.

3.3.1 Programación de Arduino

Antes de empezar a explicar cómo se programa en Arduino, se dará una pequeña visión de cómo funciona.

Arduino cuenta con una memoria flash interna donde almacena el programa a ser ejecutado. Cuando Arduino reciba corriente eléctrica, ya sea mediante el USB o por la alimentación, el programa empezará a ejecutarse. También cuenta con un botón de reset, que permite reiniciar la placa como si se acabara de conectar.

Arduino también incluye un IDE desde el cual se puede escribir, compilar y transferir código al microcontrolador.

El código mínimo de una aplicación para Arduino, consta de dos funciones: `setup()` y `loop()`. La función `setup()` se ejecuta cada vez que Arduino arranca y únicamente una vez. Cuando `setup()` ha finalizado su ejecución, se empieza a ejecutar la funcion `loop()` indefinidamente. Por lo tanto un programa básico en Arduino quedaría como sigue:

```
void setup() {
   // put your setup code here, to run once:

}

void loop() {
   // put your main code here, to run repeatedly:

}
```

El siguiente paso, es el uso de los pines para comunicarse con los elementos que estén conectados a la placa. Se pueden realizar dos operaciones: lectura y escritura. Antes de poder realizar cualquier operación mediante un pin hay que establecer su modo. Esta operación se realizará haciendo uso de la funcion `pinMode(pin, MODE)`. Solamente hará falta configurar los pines una vez, por lo tanto éstas operaciones se incluirán en la función `setup()`.

Una vez se han configurado los pines que se van a utilizar, se podrá hacer uso de las funciones `digitalRead(pin)` y `digitalWrite(pin,value)`. A continuación (Figura 21), se muestra en el caso de utilizar uno de los 14 pines disponibles:

```
int ledPin = 13; // LED connected to digital pin 13
int inPin = 7;   // pushbutton connected to digital pin 7
int val = 0;     // variable to store the read value

void setup()
{
  pinMode(ledPin, OUTPUT); // sets the pin 13 as output
  pinMode(inPin, INPUT);   // sets the pin 7 as input
}

void loop()
{
  val = digitalRead(inPin);   // read the input pin
 digitalWrite(ledPin, val); //sets the LED
}
```

Figura 21: Ejemplo básico de uso de un LED en Arduino.
Fuente: http://arduino.cc/en/Tutorial/Button

El programa de este ejemplo lee un valor de entrada mediante el pin `inPin` y escribe este mismo a `ledPin`. En el pin `inPin` hay conectado un botón y en `ledPin` hay conectado un LED. Con lo cual este programa consigue encender y apagar un LED mediante la pulsación de un botón. En la Figura 22 tenemos el esquema físico de los elementos a conectar.

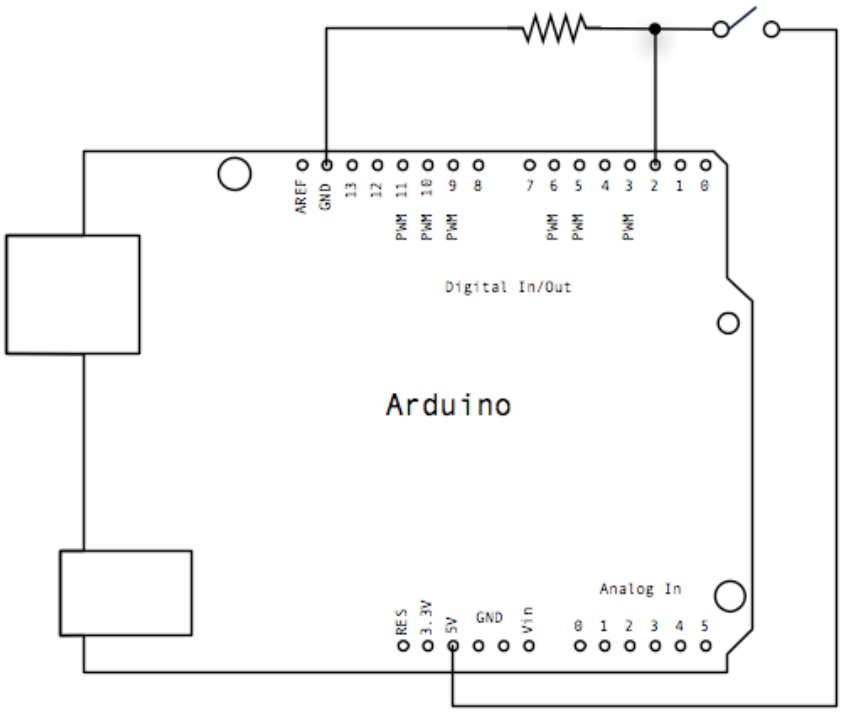

Figura 22: Esquema de conexión del LED, el botón y la placa Arduino.
Fuente: http://arduino.cc/en/Tutorial/Button

Arduino también permite la comunicación con el ordenador al que está conectado mediante USB. Como se ha explicado anteriormente, Arduino se comunica mediante un puerto serie virtual. Para su uso únicamente hay que inicializar el puerto serie utilizando la función `Serial.begin(bauds)` especificando los baudios para la comunicación. Para enviar cualquier información se pueden utilizar las funciones:

- `Serial.print(value);`
- `Serial.println(value);`
- `Serial.write(value);`

Las dos primeras están diseñadas para enviar cadenas de texto y la última envía bytes en binario.

Como se puede observar, el uso de Arduino es muy sencillo, lo cual lo hace muy recomendable para la lectura de sensores. Como se verá más adelante, escribiendo un pequeño programa que lea de un sensor y aplicando una pequeña lógica, se puede conseguir un sistema que reconozca gestos. Este ejemplo se puede encontrar en el anexo C.

3.4 Otros dispositivos de entrada

En los puntos anteriores hemos dado un repaso a dos sistemas que se usan como dispositivos de entrada. Una de las grandes virtudes de este proyecto es que puede utilizar cualquier dispositivo de entrada, pudiendo ser compatible con sistemas que aún no existen. Por ejemplo, un sistema de reconocimiento de voz podría tener asociada reglas a cada uno de los comandos.

Capítulo 4

Análisis

4.1 Objetivos

Como ya se ha explicado en la introducción, el objetivo de este proyecto es el de diseñar un sistema que sirva como nexo entre unos dispositivos de entrada y nuestras aplicaciones. Este debe de ser lo suficientemente flexible para que permita usar diferentes dispositivos de entrada y ser extensible para permitir integrar nuevos dispositivos de reconocimiento de gestos que aparezcan en el mercado.

Debe basarse en un sistema de reglas. Esto permitirá al usuario definir cuál será el comportamiento del sistema dependiendo del estado actual. Para ello se debe proporcionar al usuario herramientas para el manejo de las mismas.

Por último, el sistema debe proporcionar una seria de acciones con las que controlar las aplicaciones. En este caso se han incluido operaciones como abrir y cerrar aplicaciones, controlar el sonido o enviar pulsaciones de teclado.

4.2 Requisitos del sistema respecto a los dispositivos

El sistema debe cumplir varios requisitos en su relación con los dispositivos de entrada. A continuación se enumeran:

- El sistema debe funcionar con distintos dispositivos de entrada.
- Pueden existir distintos dispositivos funcionando sobre el mismo sistema.
- Deben poder tener una comunicación remota.

Como se puede observar en la enumeración anterior, el sistema debe tener un alto grado de independencia frente a los dispositivos de entrada. No solamente debe poder funcionar sobre diferentes dispositivos de entrada, si no que debe hacerlo simultáneamente. Además, para añadir más flexibilidad al sistema, se debe permitir el uso remoto del mismo, es decir, tener el sistema instalado en una máquina y el dispositivo en otra máquina distinta.

4.3 Flujo de trabajo

El flujo de trabajo del sistema consistirá en tres partes. La primera parte es la correspondiente al cliente en el dispositivo de entrada. Éste se encargará de reconocer el gesto realizado por el usuario. Una vez identificado se procederá a comunicarlo al servidor. Para ello se enviará el identificador del gesto a través del canal especificado, por ejemplo un paquete TCP.
En la segunda parte está el servidor. Éste se encarga de escuchar los gestos enviados por los clientes de los dispositivos. Cuando recibe un gesto se

procesará. Para ello el servidor buscará en la base de datos si alguna de las reglas se valida. Una vez encontrada se ejecutará.
Por último, el sistema ejecutará cada una de las acciones asociadas a la regla que ha sido validada

4.3.1 Envío de información entre dispositivos y sistema de reglas

El envío de información entre el dispositivo de entrada y el sistema de reglas debe ser unívoco. Esto quiere decir que el sistema de reglas debe recibir cuál es el gesto que ha realizado el usuario, junto a toda la información necesaria (Figura 23).

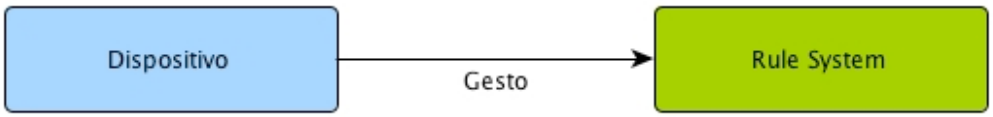

Figura 23: Comunicación entre el dispositivo y el sistema de reglas

Por lo tanto, uno de los puntos claves a analizar es cómo se codifica el gesto que ha realizado el usuario. Una de las formas más sencillas y que menos recursos utilizaría es codificarlas con un identificador entero. El problema que presenta ésta solución es que no es reconocible a qué gesto corresponde cada identificador.

Como solución se ha propuesto que los gestos se identifiquen con una cadena de texto. Con miras a la legibilidad del código, el nombre debe de representar el gesto que se realiza. Por ejemplo, si el usuario agita la mano el gesto se podrá denominar como “wave” o si el usuario levanta la mano se podrá identificar el gesto con el texto “raisehand”.

Finalmente, para comunicar al sistema de reglas que el usuario ha realizado un gesto en concreto, se deberá transmitir el nombre del gesto y éste ya se encargará de realizar las acciones asociadas.

4.3.2 Reglas y acciones

El servidor contará con dos entes y que representan lo más básico del sistema: reglas y acciones.

Una regla describe una situación que puede darse durante la ejecución del sistema. Si la situación actual es igual a la que representa la regla, ésta se validará. Cuando una regla es validada significa, que será ejecutada.

Para modelar una situación dentro de la regla se utilizarán tres variables:

- Gesture: Cada regla tendrá obligatoriamente un gesto asociado.
- Application: Una regla puede tener asociada una aplicación que deberá tener el foco del sistema operativo o no. Dependerá de la siguiente variable.

- Foreground: Será una variable booleana que representará si la aplicación indicada en la variable application tiene que tener el foco del sistema operativo o no.

Para que una regla sea validada se debe cumplir que el gesto de la regla debe ser igual al gesto realizado por el usuario. Si la variable foreground es verdadera, entonces el foco lo deberá tener la aplicación indicada en la regla. Si foreground es falso, únicamente se comprobará la coincidencia en el gesto.

El segundo elemento importante son las acciones. Una acción indica al sistema operativo la tarea a realizar. Una regla puede tener asociadas tantas acciones como se quiera. Cuando una regla se valide se ejecutarán todas las acciones asociadas, en orden FIFO. En la Figura 24 se muestra un ejemplo de la estructura de un base de datos con reglas y acciones.

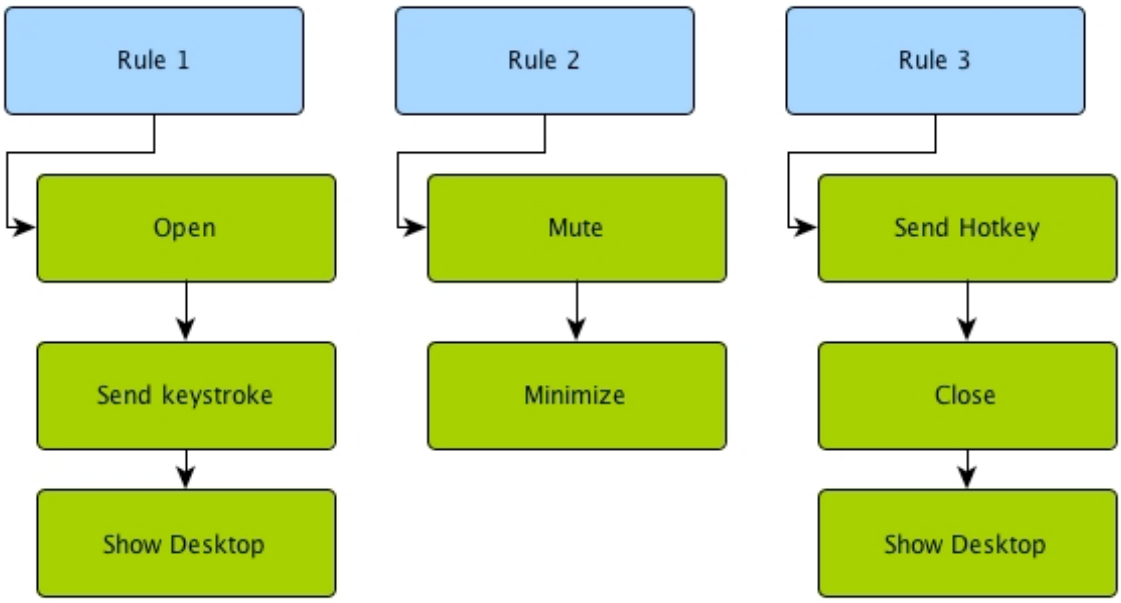

Figura 24 : Ejemplo de la estructura de un sistema de reglas.

Cada acción tendrá un nombre identificativo. En algunos casos, las acciones incluirán parámetros necesarios para su ejecución.

Las acciones permitidas en el estado actual del proyecto son:

- Abrir aplicaciones. Se indicará que aplicación abrir y podrá pasarse argumentos.
- Cerrar aplicaciones. Se cerrará la aplicación con el foco.
- Maximizar y minimizar la aplicación con el foco.
- Mostrar el escritorio.
- Aumentar, disminuir y asignar el volumen del sistema. Recibirá cual es la cantidad a aumentar, disminuir o asignar.
- Enviar pulsaciones de teclado. Recibirá qué tecla se ha pulsado. Se enviará a la aplicación con el foco.
- Enviar atajos de teclado. Recibirá qué combinación de teclas se ha pulsado. Se enviará a la aplicación con el foco.

4.3.3 Flujo de datos en la ejecución de un gesto

A continuación se detalla el procesamiento que realiza el servidor para responder a los eventos que se producen.

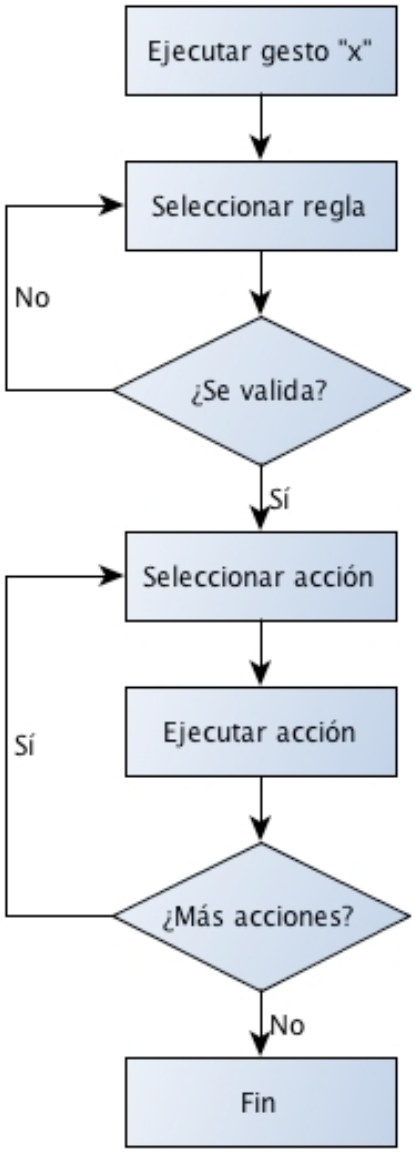

Figura 25: Flujo de ejecución de un gesto

En la Figura 25 se muestra un diagrama de flujo de ejecución de una regla. El sistema recibirá un identificador de texto que indicará cual es el gesto que se ha detectado. Inmediatamente el sistema comenzará a buscar cuál es la primera regla que se cumple y la ejecutará. El orden de las reglas define su preferencia.

Pueden haber dos casos en los que se valide una regla:

- El gesto coincide y el foreground está a false. En este caso se ha encontrado una regla que representa un estado del sistema en el cual da igual qué aplicación tenga el foco, solamente se comprobará que se ha realizado el gesto.

- El gesto coincide, la aplicación coincide y foreground está a true. Se dan dos coincidencias en este caso: el gesto y la aplicación que tiene el foco.

Una vez se ha encontrado la regla que se valida, entonces se procederá a su ejecución. Para ello se ejecutará la lista de acciones que tiene asociada, por orden. Dentro de las acciones está almacenada toda la información que se necesita para su ejecución.

Cuando una acción recibe la orden de ejecutarse, comprobará que sus parámetros, si los tuviera, son correctos y llamará a un método propio que se encargará de comunicar al sistema operativo la acción a realizar.

4.3.4 Administración de reglas y acciones

Aparte de la ejecución de gestos, se proporcionan operaciones para la administración de las reglas y las acciones. Se incluirán como operaciones las siguientes:

- Añadir regla.
- Modificar regla.
- Borrar regla.
- Añadir acción a una regla.
- Modificar acción
- Borrar acción.

También se incluirán todas las operaciones de consulta.

4.4 Limitaciones del sistema

Como se ha descrito anteriormente, el sistema es muy flexible y abierto. Se proveen como acciones las operaciones más comunes que se pueden realizar en un entorno de escritorio. Por otra parte el sistema tiene algunas limitaciones. Éstas van a ser abordadas en los siguientes apartados.

4.4.1 Dependencia del sistema operativo

En el diseño de las acciones se tomó como decisión que las funciones que son llamadas en cada una de las acciones estén almacenadas en un fichero. Se ha diseñado una versión donde solamente se da soporte para Windows 7 y Vista. Para soportar otro sistema operativo, habría que portar las funciones de dicho fichero. En este caso se debe compilar el sistema de nuevo.

4.4.2 Fiabilidad de los dispositivos de entrada

Otro de los aspectos que no se ha mencionado pero es importante tener en cuenta es la fiabilidad de los dispositivos de entrada. Cuando el sistema recibe que se ha detectado un gesto, éste ejecutará las acciones asociadas a la regla que se valide. Por lo tanto, la fiabilidad del dispositivo debe ser alta. Pongamos un ejemplo:

Imaginemos que tenemos un dispositivo de entrada que está mal calibrado. El usuario realiza un gesto, levantar la mano, pero el dispositivo entiende que este gesto se está activando cada vez que el usuario tiene la mano levantada. Si el usuario la tiene levantada durante 0.5 segundos, y durante ese tiempo el dispositivo manda que se ha realizado el gesto “raisehand” 10 veces, la regla asociada se ejecutará 10 veces.

Esto puede causar problemas si la regla asociada, por ejemplo, lanza una aplicación que consume muchos recursos.

Capítulo 5

Diseño

5.1 Visión general del diseño

En el apartado anterior se ha perfilado la solución propuesta para el problema al que nos enfrentamos. Teniendo en cuenta todas las decisiones tomadas y el modelo de sistema a desarrollar, en este apartado se explicará cómo ha sido el diseño e implementación del mismo.

El sistema de reglas estará compuesto por reglas y acciones. En la Figura 26 podemos observar el diagrama UML que representa las tres clases principales del sistema.

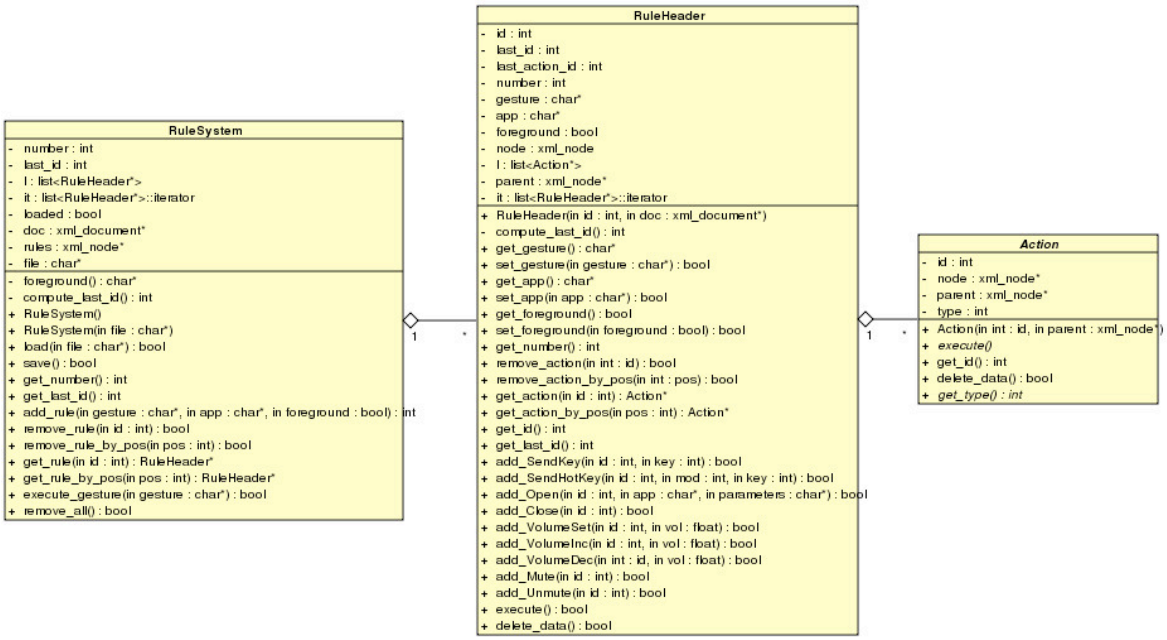

Figura 26: Diagrama UML de las clases RuleSystem, RuleHeader y Action.

En el sistema existen tres clases principales RuleSystem, RuleHeader y Action.

La primera clase, RuleSystem, representa un ente que administra un sistema de reglas y por tanto esta clase proporcionará, como se verá después, métodos para el acceso a todas las opciones de administración y ejecución. A través de él se puede acceder a toda la información del sistema, así como a los objetos de las otras dos clases principales que contiene. Desde él se podrá cargar archivos de reglas previamente almacenados.

La segunda clase, y una de las más importantes, es RuleHeader. En un objeto de tipo RuleHeader se almacenará toda la información referente a una regla. La

clase proporciona a través de sus métodos todos las operaciones que se han descrito en el análisis del sistema, así como otras que no son utilizadas por el usuario a través de RuleSystem, como por ejemplo el método para ejecutar una regla.

La tercera y última clase es Action y representa una acción en el sistema. Como se analizó en el capítulo anterior, el sistema de reglas debe ser lo suficientemente flexible y ampliable para que se puedan añadir nuevas acciones al mismo. Para poder cumplir este requisito se ha diseñado la clase haciendo uso del polimorfismo.

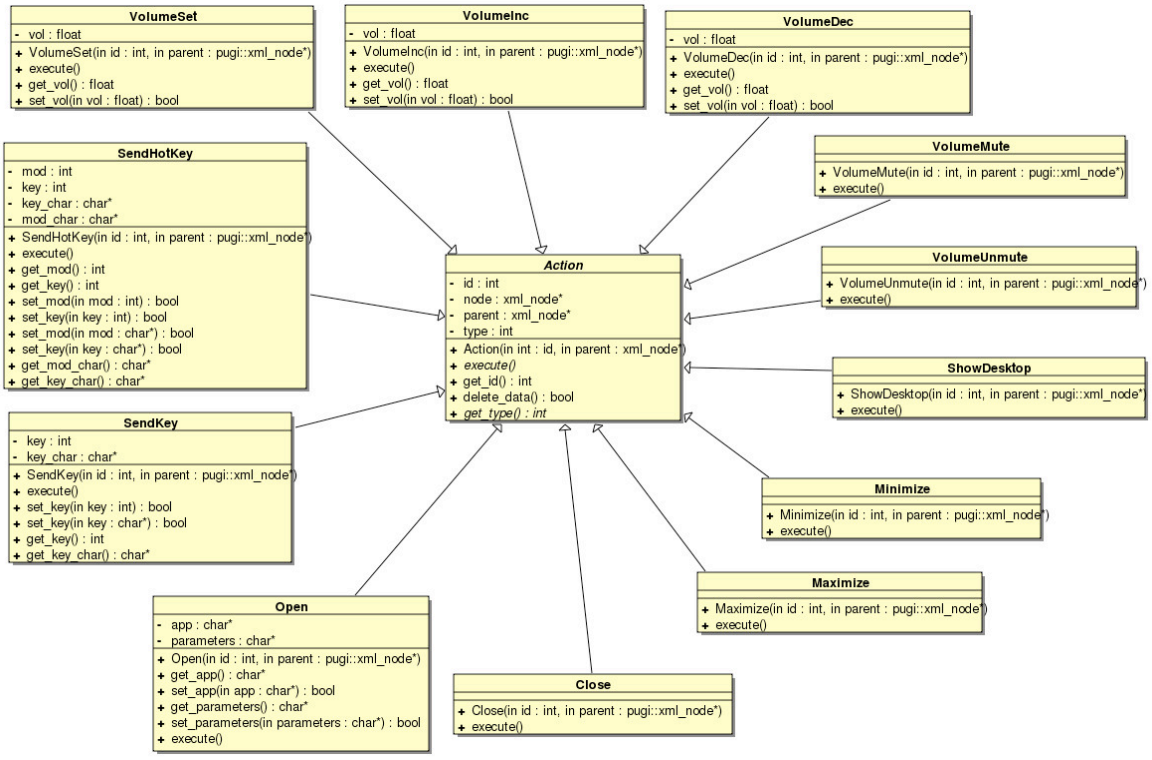

Figura 27: Diagrama UML de la clase Action y sus derivadas.

Cuando el sistema de reglas se esté ejecutando no existirán instancias de la clase Action. Esto es así porque Action únicamente representa una clase padre de la cual van a heredar las acciones concretas. En esta clase se encontrarán todas las características que son comunes a todas las acciones además de una serie de métodos que todas las clases derivadas deben implementar. Es decir, será una clase virtual. En la Figura 27 vemos el diseño UML de las acciones.

Una vez se ha dado una visión general de las tres clases principales, se detallará la relación que existe entre ellas.

Para crear un sistema de reglas se instanciará un objeto de tipo RuleSystem que estará vacío y no contendrá ninguna regla. Un objeto de tipo RuleSystem puede contener tantas reglas como se quiera o ninguna. Cada vez que se agregue una regla al sistema se creará un objeto de tipo RuleHeader.

Al crearse un objeto RuleHeader este no contiene regla alguna. Si se intentase ejecutar simplemente no se realizaría ninguna acción. Un objeto de tipo RuleHeader puede contener multiples acciones. Si se añade una acción a una regla el sistema creará un objeto del tipo de acción que se desee.

Cuando se añade una acción a una regla se realiza a través de un método especifico para cada tipo de regla. Se debe realizar de esta forma ya que se debe saber que tipo de acción se desea añadir para crear una instancia de la clase adecuada.

Cada una de las acciones y reglas que se crean tienen un identificador único. De esta forma, podemos identificar fácilmente las reglas y acciones cuando se vayan a realizar operaciones sobre ellas.

5.2 Persistencia de las reglas usando Pugixml

Una de las características vitales para el sistema es que todas las reglas y acciones que se han ido diseñando se almacenarán para su posterior uso. Para poder ofrecer esta posibilidad al usuario se debe buscar alguna forma de almacenamiento que sea adecuada.

Se tuvieron en cuenta tres opciones a la hora de diseñar el almacenamiento de reglas: fichero de texto plano, base datos o XML. Para elegir entre ellos al más adecuado se tuvieron en cuenta las siguientes cuestiones.

Al ser un sistema de reglas que está basado en los gestos que puede realizar el usuario no se probable un sistema con más de diez gestos. Por lo tanto, el número de reglas y acciones no será un problema para el ordenador. Aunque hay información diversa, las consultas a realizar son muy sencillas.

Por lo tanto, se descarta la base de datos porque está orientado a sistemas más complejo además de un coste computacional. También se descarta un fichero de texto plano, aunque sería el más rápido, al necesitar almacenar información diversa (reglas y distintos tipos de acciones) haría falta implementar un parser. Ésta implementación supondría un coste temporal en la implantación del proyecto.

Finalmente se optó por escoger XML como el formato a utilizar en la persistencia de las reglas y las acciones.

La utilización de un sistema basado en el formato XML presenta muchas ventajas respecto a las otras opciones consideradas. No hace falta la creación de un parser para su utilización, ya que hay disponibles diversas opciones. Es un sistema ligero, no precisa de ningún programa funcionando para su acceso, como si lo necesitaría una base de datos, y es fácil representar en un diseño de clases sencillo.

5.2.1 Pugixml

En el apartado anterior se ha explicado el por qué de la elección de XML como el sistema de representación y almacenamiento de las reglas. Una de las ventajas a la que se hacía referencia es que no hace falta crear un parser para su utilización ya que existen en el mercado diversas opciones. De entre todas las posibles opciones se ha escogido Pugixml [14].

Pugixml es un parser de xml muy ligero implementado en C++. Es capaz de trabajar tanto con fichero como buffers en memoria. Soporta Xpath 1.0 además de texto en Unicode. Es fácilmente integrable y usable en un proyecto.

La instalación más sencilla de Pugixml en un proyecto es la inclusión tanto de la cabecera como del código fuente. De esta forma se compilará Pugixml a la vez que nuestro proyecto. Como inconveniente está que hay que compilarlo cada vez. En algunos compiladores de C++ existe la posibilidad de precompilar las cabeceras. En el anexo E podemos encontrar un ejemplo de como usar Pugixml en un proyecto.

5.2.2 Diseño del modelo XML

Uno de los puntos críticos del diseño del sistema es cómo modelar toda la información que se debe almacenar. El diseño que se realice del XML deberá soportar todas y cada una de las posibles opciones que tiene el usuario. Por lo tanto, se deberá tener en cuenta todas las posibles operaciones que se puedan realizar sobre el sistema de reglas. Además el diseño debe ser lo suficiente flexible para que sea fácil agregar nuevas opciones en el futuro.

Toda la información del sistema de reglas está contenida entre la etiqueta Rules. Por lo tanto la unidad mínima del sistema, será un archivo XML que contenga:

```
<Rules>
</Rules>
```

Un fichero que contenga esa información significará un sistema de reglas donde no hay regla alguna. Es un sistema válido, de hecho es el punto de partida cuando se crea uno nuevo.

Entre esas dos etiquetas se encontrarán todas las reglas y acciones que pertenecen al sistema. El siguiente ejemplo muestra una base de datos con una regla sin atributos ni acciones:

```
<Rules>
     <rule>
     </rule>
</Rules>
```

Cada regla recibe un identificador que permite que posteriormente se le puedan aplicar operaciones sobre las mismas. El siguiente ejemplo muestra un sistema de reglas con una regla identificada con el 1.

```
<Rules>
     <rule id=”1”>
     </rule>
</Rules>
```

El siguiente ejemplo muestra como asignar un gesto a una regla:

```
<Rules>
     <rule id=”1” gesture=”Wave”>
     </rule>
</Rules>
```

Otra de las posibilidades, la cual da mucha versatilidad al sistema, es que las reglas no sólo dependan del gesto que se ha realizado, si no también del estado del ordenador en ese momento. Para ello se incluye la posibilidad de que la validez de la regla dependa del programa que tiene el foco del sistema operativo. Modificamos nuestro modelo XML para que sea acorde con esta propuesta.

```
<Rules>
     <rule id=”1” gesture=”Wave” app=”Spotify.exe”>
     </rule>
</Rules>
```

De esta forma para que se valide la regla se tienen que dar dos condiciones: el gesto que ha realizado el usuario debe ser el mismo que el que contiene la regla y el programa que tiene el foco en ese momento del sistema operativo debe coincidir con el de la regla. Dándose estas dos condiciones, la regla se validaría y por lo tanto se ejecutaría.

Con el fin de darle más versatilidad, se pensó que el usuario podía querer que el sistema ejecutara una regla siempre que se da un gesto, sin dar importancia a la aplicación que tiene el foco. Con ese fin se agregó un atributo más a la regla.

```
<Rules>
     <rule id=”1” gesture=”Wave” app=”Spotify.exe”
foreground=”false”>
     </rule>
</Rules>
```

Si el atributo `foreground` contiene como valor true, entonces se comprobará si la aplicación que tiene el foco y la aplicación que indica la regla coincide. Si el atributo `foreground` está a false, solamente será suficiente para validar la regla que coincidan los gestos y se ignorará el valor de app.

Llegados a este punto, podemos tener un sistema de reglas con tantas reglas como se quiera. Se han añadido todas las opciones posibles para cada una de

las mismas, haciendo que el sistema sea tan versátil como nos los propusimos en el análisis.

En este punto el sistema sigue siendo valido. Se tienen reglas que están bien formadas y que se pueden validar si se dan las condiciones. Si una regla se valida en este punto, simplemente la regla se ejecutaría pero como no contiene ni una acción, no realizaría nada.

El siguiente paso es añadir soporte para las acciones que se ejecutarán en cada una de las reglas.

En el caso de las acciones existen distintos tipos de las mismas. Aunque todas tienen características en común, cada una tiene aspectos particulares. Una de las primeras posibilidades que se puede pensar es diseñar una acción tal que así:

```
<Rules>
     <rule id=”1” gesture=”Wave” app=”Spotify.exe”
foreground=”false”>
          <action type=”ShowDesktop”/>
     </rule>
</Rules>
```

Estas es una de las muchas posibilidades a la que se puede llegar en un primer acercamiento. Como se puede observar hay información redundante ya que se puede modelar el tipo de la acción como el nombre del nodo:

```
<Rules>
     <rule id=”1” gesture=”Wave” app=”Spotify.exe”
foreground=”false”>
          <ShowDesktop/>
     </rule>
</Rules>
```

Al igual que pasa con las reglas, las acciones pueden ser objeto de operaciones sobre ellas. Por lo tanto se les deberá identificar con el mismo sistema que el de las reglas.

```
<Rules>
     <rule id=”1” gesture=”Wave” app=”Spotify.exe”
foreground=”false”>
          <ShowDesktop id =”1”/>
     </rule>
</Rules>
```

En este caso tenemos una acción asociada a una regla, la cual no tiene ningún parámetro porque no lo necesita. A continuación se mostrarán todas las posibles acciones y cómo se representan en el archivo XML.

- `<Open id=”” app=”” parameters=””/>`
- `<Close id=””/>`
- `<Maximize id=””/>`

- <Minimize id=””/>
- <VolumeSet id=”” vol=””/>
- <VolumeInc id=”” vol=””/>
- <VolumeDec id=”” vol=””/>
- <Mute id=””/>
- <Unmute id=””/>
- <SendKey id=”” key=””/>
- <SendHotKey id=”” mod=”” key=””/>

En la lista anterior están modeladas todas las posibles acciones que pueden existir en el sistema. Para hacer uso de ellas, simplemente hay que incluirlas dentro de una regla. Cuando esta se valide, se ejecutarán todas las acciones que contenga.

5.3 Tipos de acciones e implementación

En el apartado anterior se dio cuenta de cómo se modelaba todo el sistema de reglas en un archivo XML. En la última parte se explicaron las acciones y se dio una pequeña visión de cada una de las posibles opciones.

Cuando se analizó el problema y se propusieron posibles acciones se tuvo en cuenta la tecnología que soporta el sistema. En este caso Windows7/Vista proporciona una API [13] que da acceso a la interacción con el sistema operativo.

En este apartado empezaremos explicando cada una de las clases que forman la parte de nuestro sistema que controla las acciones.

Una de las clases más importantes para modelar una acción es Action. Como ya se dijo anteriormente, esta clase representa las características que deben tener todas las acciones que se creen en el sistema. Por lo tanto, todas las acciones del sistemas deben heredar de Action.

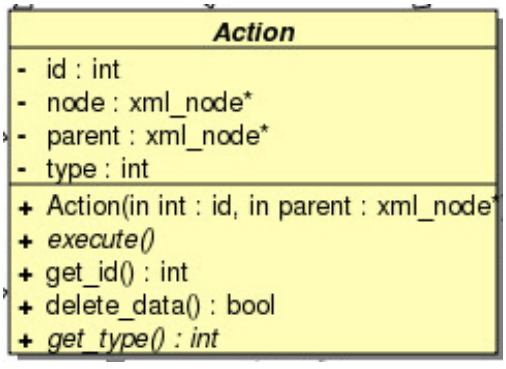

Figura 28: Diagrama de clase de Action.

En la figura 28 está representado el diagrama de clase de Action.

La clase Action representa cual es la unidad mínima de información que deben de tener todas las acciones. En la parte superior podemos observar los miembros privados de la clase.

Para poder identificar las acciones, como ya se dijo, se utilizará un identificador. Para ello se ha hecho uso de una variable de tipo entero.

Como las clases son representaciones de lo que se encuentra en el XML, éstas tendrán un puntero a los nodos XML con los que están relacionados. En el caso de las acciones tendrán un puntero al nodo XML que las representa, además de un puntero al nodo XML que representa la regla en la que están contenidas. Gracias a esta decisión de diseño, el propio nodo acción será el encargado de acceder a toda la información del XML que le representa.

Por último, en lo que respecta a los miembros privados, está la variable entera `type`. Esta variable contiene un valor entero que representa el tipo de clase. Ya que vamos a usar polimorfismo, se explicará más adelante, esta variable es muy importante. Se utilizará más adelante para calcular la clase real del objeto en tiempo de ejecución.

A continuación se dará una visión de para que sirven y como están implementados cada uno de los métodos públicos de la clase Action.

El constructor tiene dos parámetros de entrada. El primero es un parámetro de tipo entero que se denomina id. Como su propio nombre indica es el identificador que se le dará a la acción. Hay que tener en cuenta, aunque aquí aún no sea visible, que las acciones serán creadas por una clase RuleHeader. Es la propia clase RuleHeader la que controla los identificadores de cada una de las acciones que se agregan.

El segundo parámetro es de tipo `xml_node*` y su nombre es `parent`. Es un puntero que apunta al nodo XML de la regla en la que estará contenida la acción.

El código contenido en el constructor es muy sencillo, simplemente inicializa las variables privadas `id` y `parent` con los parámetros de entrada.

El siguiente método es uno de los más importantes en la clase Action. El método execute() permite ejecutar las acciones asociadas al objeto. No recibe ningún parámetro, porque como ya veremos a continuación, toda la información necesaria para su ejecución se encuentra en la misma clase.

Los métodos `get_id()` y `get_type()` devuelven, como su propio nombre indica, las variables `id` y `type`.

Por último está el método `delete_data()`. Éste método se utiliza para eliminar toda la información del XML que esté representando la acción. En su interior simplemente ejecuta la siguiente instrucción:

```
parent->remove_child(node);
```

Este es uno de los usos de `parent` dentro de una acción.

En los siguientes subapartados se explicará cada una de las clases que representan los diferentes tipos de acción. Todas ellas derivan de la clase Action

y por lo tanto heredan todas su características. Es muy importante que implementen el método execute.

5.3.1 Open

Una de las primeras acciones que se pensó en el análisis del problema era la de que el sistema fuera capaz de abrir aplicaciones. Se ha utilizado COM para abrir aplicaciones en Windows 7/Vista. Este sistema permite que además de lanzar aplicaciones también que se pudiera especificar que parámetros podía recibir.

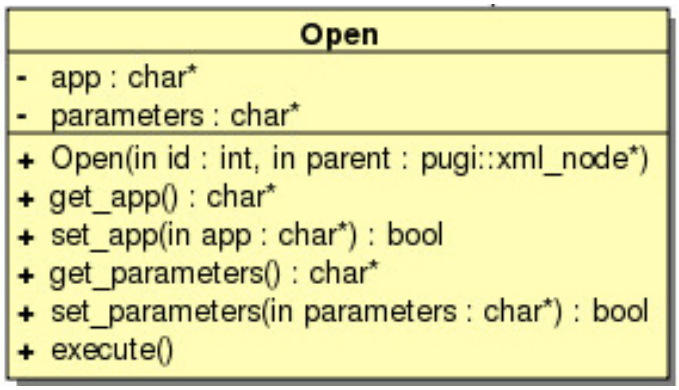

Figura 29: Diagrama de clase de Open.

En la Figura 29 vemos el modelo UML. La clase Open contiene dos miembros privados. Éstos almacenan la información necesaria para poder ejecutar la acción. En el primero de ellos, `app`, se almacena el nombre y la ruta de la aplicación que va a ser ejecutada. El segundo, `parameters`, contiene una cadena de texto con los parámetros que se pasarán a la aplicación en su ejecución.

En la parte de los métodos, el primer elemento que se ha definido es el constructor. Una acción puede crearse por dos razones:

La primera de ellas, y la más obvia, es que estamos agregando una nueva acción a una regla. Para ello tenemos que crear un elemento XML que represente la nueva acción agregada. En principio éste nodo se creará sin ningún valor en sus parametros.

La segunda posibilidad es que estemos creando un objeto de tipo acción porque estamos leyendo un archivo XML. Este caso se da en la reconstrucción del modelo de objetos a partir del XML. Obviamente no queremos agregar un nuevo nodo al XML pero accederemos a la información del nodo y la almacenaremos en el objeto.

Además existen todos los métodos de lectura y escritura de los miembros privados, `app` y `parameters`. Éstos métodos actualizan el estado del nodo en el archivo XML. De esta forma las estructura de clases y el XML será consistentes.

Por último está la operación más importante dentro de la clase Open, `execute()`. En esta parte reescribiremos la función que hereda de Action. Dentro de ella

llamaremos a una función que encapsula una serie de operaciones mediante COM que permite la ejecución de programas.

5.3.2 Close

En contraposición al elemento anterior, la acción Close permite al usuario cerrar aplicaciones. En este caso la clase Close no contiene ningún miembro privado, ya que no es necesario almacenar ninguna información adicional para su ejecución. Esto se debe a que la aplicación que se cerrará será siempre la que tenga el foco en el momento de la ejecución.

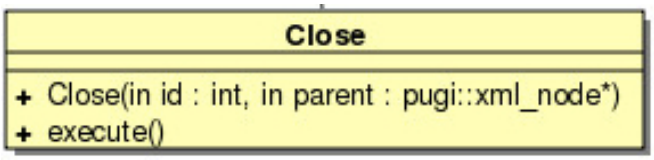

Figura 30: Diagrama de clase de Close.

Com se puede ver en la Figura 30, en la parte pública de la clase, se encuentran únicamente dos métodos. El primero de ellos es el constructor de la clase. Al igual que en el caso anterior, y en todos los que suceden, el constructor realiza la misma operación: inicializa las variables `id` y `parent` a los valores recibidos como parámetros.

El método `execute()` contendrá las acciones a realizar para poder cerrar la aplicación que tiene el foco. Para ello se utilizará la API de Windows, que permite interactura con el sistema operativo. La función `execute()` llamará a una función que contiene el siguiente código:

```
HWND x = GetForegroundWindow();
PostMessage(x,WM_CLOSE,0,0);
```

En la primera de ella se crea una variable `x` de tipo `HWND`. Este tipo es un controlador de ventana. Contiene un valor que identifica la ventana dentro del sistema operativo. X se inicializa a un valor que devuelve la función `GetForegroundWindow()`. Ésta se encarga de consultar al sistema operativo cual es la ventana que tiene el foco y devolver su identificador.

La función PostMessage es una de las funciones más importantes para interactuar con el sistema operativo. De hecho, muchas de las acciones que incorpora el sistema de reglas están basadas en la función PostMessage. Ésta función encola un mensaje en la cola de procesamiento de una ventana, sin esperar a que éste sea procesado. La función PostMessage recibe cuatro parámetros. El primero de ellos es el HWND de la ventana a la que se desea enviar el mensaje. El segundo de ellos es el propio mensaje. És un código que representa el mensaje que se va a enviar a la ventana. Existen una gran variedad de mensajes que se pueden enviar. Gracias a ellos se puede simular cualquier comportamiento del usuario sobre la ventana. En nuestro caso usaremos el mensaje WM_CLOSE que indica al sistema operativo que debe cerrar la aplicación.

El tercer y cuarto parámetro son parámetros adicionales que puede necesitar el mensaje para poderse ejecutar. En este caso no hace falta ninguno pero en otros, como enviar pulsaciones de teclado, es necesario.

5.3.3 Maximize

La acción Maximize funciona de forma similar a como lo hace la acción Close. Al ser una acción que no necesita parámetros, esta clase carece de miembros privados.

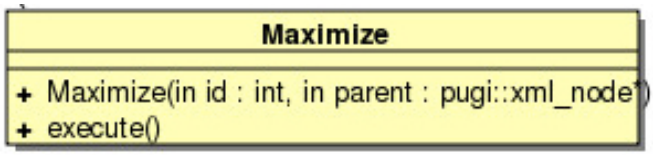

Figura 31: Diagrama de clase de Maximize.

En la Figura 31 vemos su modelo UML. Su constructor funciona de la misma forma que los anteriores. Dependiendo de si estamos añadiendo una acción Maximize nueva o estamos reconstruyendo el estado del sistema, se realizarán unas operaciones u otras.

Respecto al método execute(), éste es parecido al método de la clase Close.
Se ejecutará el siguiente código:

```
HWND x = GetForegroundWindow();
PostMessage(x,WM_SYSCOMMAND,SC_MAXIMIZE,0);
```

Como se puede observar hay un pequeño cambio en el envío del mensaje. En este caso será WM_SYSCOMMAND el mensaje enviado pero se necesitará un parámetro adicional. El mensaje WM_SYSCOMMAND representa todas las posibles acciones que puede realizar un usuario desde el menú Ventana, o desde los mismos botones que existen en la ventana. El parámetro adicional indica cual de todas estas posibles acciones es la que se debe ejecutar. En nuestro caso es SC_MAXIMIZE que indica al sistema operativo que debe maximizar la ventana.

5.3.4 Minimize

La acción Minimize es análoga a la acción tratada en el apartado anterior. En este caso al ejecutar la acción, el sistema operativo minimizará la ventana que tiene el foco. Como se puede deducir en este caso tampoco necesitamos ningún miembro privado para almacenar información necesaria para la ejecución.

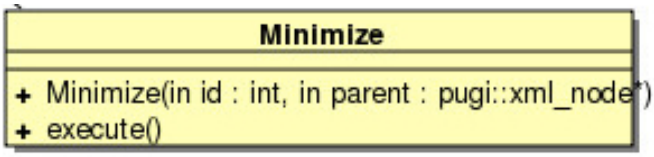

Figura 32: Diagrama de clase de Minimize.

En la Figura 32 vemos su modelo UML. El constructor de la clase Minimize funciona exactamente igual que los anteriores. Análogamente al miembro execute() de la clase Maximize, en este caso el parámetro que se envía al mensaje cambia, quedando así:

```
HWND x = GetForegroundWindow();
PostMessage(x,WM_SYSCOMMAND,SC_MINIMIZE,0);
```

Se ejecutará sobre la ventana que tiene el foco del sistema operativo.

5.3.5 VolumeSet

Una de las opciones interesantes en la que se pensó cuando se analizaba la solución era la de incluir un control sobre el volumen del sistema operativo. Una de las operaciones más habituales dentro del control del volumen es la de establecer un valor determinado del mismo. Para ello se creó la acción VolumeSet.

A diferencia de las anteriores, esta clase sí tiene un miembro privado para el almacenamiento: el nivel de volumen a establecer. Para ello se creó una variable de tipo coma flotante, donde se almacenará el valor del volumen. Éste debe de estar comprendido entre 0 y 1. Valores que significarán silencio y máximo volumen, respectivamente.

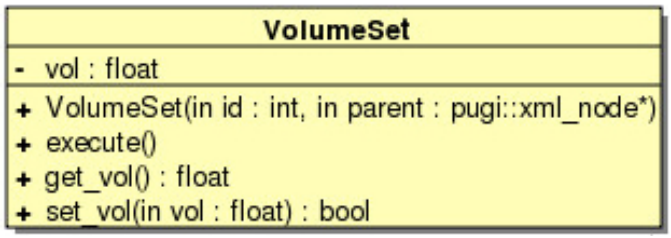

Figura 33: Diagrama de clase de VolumeSet.

En la Figura 33 vemos su modelo UML. El constructor de la clase realiza las mismas operaciones que los constructores de las otras clases que se han explicado anteriormente. Paralelamente los getter y setter funcionan de la misma forma.

Para la implantación del método público `execute()` se utilizo una API que permite el acceso a los dispositivos de control de volumen. No se va a exponer todo el código que realiza esta acción porque es muy extenso y no es el objetivo del proyecto.

5.3.6 VolumeInc

Esta acción permite al usuario aumentar el volumen del sistema en una cantidad determinada. Al igual que la acción `VolumeSet`, `VolumeInc` necesita de un miembro privado que almacene la información necesaria para la ejecución. En el miembro `vol`, que es de tipo coma flotante, se guarda en cuánto debe aumentar el volumen del sistema operativo.

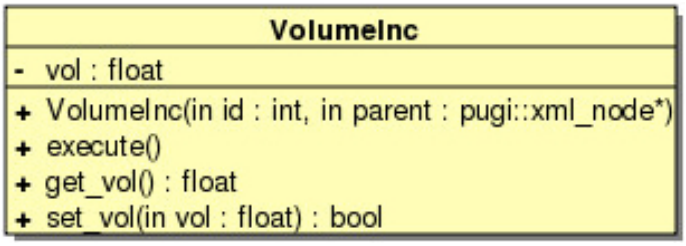

Figura 34: Diagrama de clase de VolumeInc.

En la Figura 34 vemos su modelo UML. Todos los métodos públicos funcionan exactamente igual que lo hacen en `VolumeSet`. La única diferencia está en el método `execute()`. En este, antes de poner el volumen a un valor determinado, se consulta cuál es el valor actual del volumen en el sistema operativo. Una vez se sabe el volumen actual, se suma a la variable `vol` y se establece el volumen del sistema al resultado de la suma. Si el volumen final supera el 100% se deja en 100%.

5.3.7 VolumeDec

VolumeDec funciona de forma exactamente igual a como lo hace VolumeInc. La única diferencia reside en que cuando se calcula el nuevo valor, en vez de sumar vol, se resta. Lo podemos ver en la Figura 35.

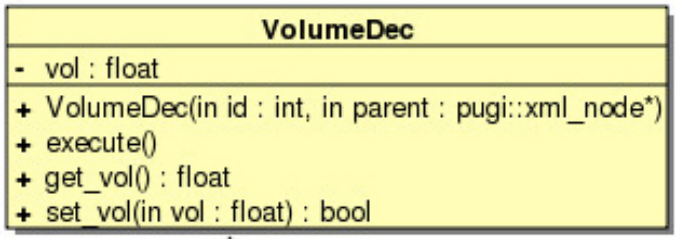

Figura 35: Diagrama de clase de VolumeDec.

5.3.8 ShowDesktop

Como su propio nombre indica, esta acción es capaz de simular que el usuario ha pulsado el atajo de teclado para mostrar el escritorio. Ésta acción no requiere de ningún valor introducido por el usuario, lo que nos lleva a una clase sin ningún miembro privado.

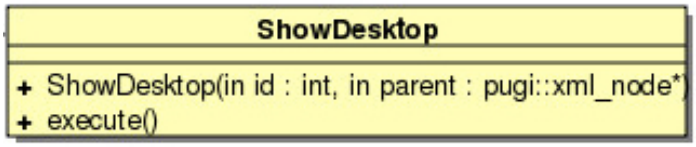

Figura 36: Diagrama de clase de ShowDesktop.

En la Figura 36 vemos su modelo UML. El método execute mandará un mensaje al sistema operativo para que active la opción de mostrar el escritorio. A diferencia de las clases anteriores donde se enviaba un mensaje a través de la función PostMessage, en este caso no será así. Una de las razones es que no estamos enviando el mensaje a ninguna aplicación en concreto, si no al sistema operativo.

Obviando la parte donde se reserva memoria y se inicializan ciertas variables la parte importante sería esta:

```
pShellDisp->ToggleDesktop();
```

La variable pShellDisp es de tipo IShellDispatch4 [15]. Éste da acceso a toda las posibles opciones relacionadas con el Shell del sistema operativo.

5.3.9 SendKey

La acción SendKey permitirá simular que ha ocurrido una pulsación de teclado. Antes de mostrar como se ha implementado la acción, se explicará como funciona la lectura de las pulsaciones de teclado.

Primero hemos de mencionar que la pulsación de teclado se enviará siempre a la aplicación que tenga el foco.
Cada uno de los dispositivos de entrada de un ordenado disponen de una cola que almacena cada una de las ordenes que va recibiendo. Por ejemplo, en la cola del ratón podemos encontrar lo siguiente:

- Aumentar la coordenada x en 1 unidad.
- Pulsación del botón izquierdo.
- Disminuir la coordenada y en 3 unidades.

Todas las operaciones se realizarán en un estricto orden FIFO. De esta forma se asegura que todas las operaciones que realiza el usuario son procesadas en el mismo orden en el que fueron introducidas.

Para simular que se ha pulsado una tecla se ha de introducir en la cola del teclado el evento correspondiente. Cada una de las teclas que existen en un teclado tiene asociado un código virtual de tecla. Gracias a él se puede identificar una tecla en concreto entre distintos teclados de diversos idiomas.

Una pulsación de teclado consta de dos partes, la pulsación y la liberación de la tecla. Para realizar una simulación correcta debemos enviar una pulsación de la tecla y una liberación de la misma.

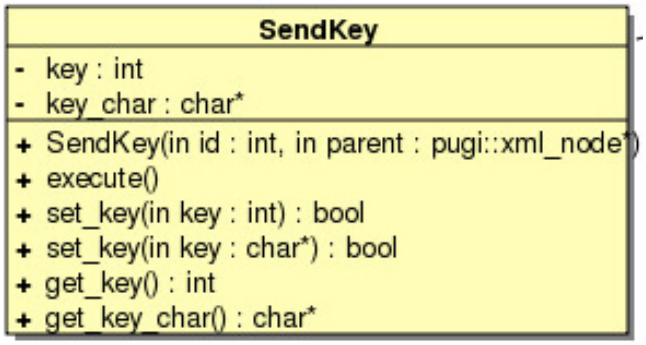

Figura 37: Diagrama de clase de SendKey.

En la Figura 37 vemos su modelo UML. Una vez explicado cómo funciona la recepción de los eventos de teclado por parte del sistema operativo, se procederá a explicar la clase SendKey.

La clase SendKey tiene que almacenar la tecla: Para ello nos hemos servido de dos miembros privados.

El primero de ellos, `key`, es una variable de tipo entera. En ella se almacenará el código de tecla virtual que pertenece a la tecla a pulsar. El segundo, `key_char`, es una variable de tipo `char*`. En ella se almacenará el mismo valor que en `key` pero en forma de cadena de texto. Esta decisión se ha tomado porque el usuario puede no saber cual es el código de tecla virtual de una tecla.

En la parte de los miembros públicos, encontramos el constructor que realiza la misma función que la comentada en todos los casos anteriores. Obviando el miembro `execute()`, que lo trataremos al final de este apartado, hay cuatro miembros más. Éstos permiten la lectura y escritura de los miembros privados que almacenan la tecla a ser enviada. Como hemos decidido que la tecla se almacenará de dos formas, en forma de entero y de carácter, entonces tenemos que proporcionar los setter y getter para cada una de ellas. El uso de cualquiera de los setters actualiza ambas variables, para que sean consistentes.

Por último, está la función `execute()` que es la parte más importante de la clase. Cuando se ejecuta se deberá enviar una pulsación de teclado con el código virtual almacenada en la clase.

El código siguiente muestra exactamente como se realiza a través de la API de Windows.

```
INPUT input[2];
ZeroMemory(input, sizeof(input));
input[0].type = input[1].type = INPUT_KEYBOARD;
input[0].ki.wVk = input[1].ki.wVk = key;
input[1].ki.dwFlags = KEYEVENTF_KEYUP;
SendInput(2, input, sizeof(INPUT));
```

Este código consta de tres partes importantes.

En la primera de ellas, se crean dos estructuras de tipo INPUT. Éstas almacenarán cada uno de los eventos que hay que enviar a la cola del teclado, en este caso dos: pulsación y liberación.

En la segunda inicializamos cada uno de los valores necesarios en las estructuras. Indicamos el código virtual, a qué cola va a ser enviado, así como si es una pulsación o un evento de liberación.

En la tercera y última enviaremos los eventos. Gracias a la función `SendInput` podremos enviar los dos eventos. El primero de los parámetros indica cuántos eventos van a ser enviados, el segundo es el vector de eventos y el tercero es el tamaño del mismo.

5.3.10 SendHotKey

La acción SendHotKey es una evolución de la acción tratada en el apartado anterior. En él se diseñaba una acción que fuera capaz de enviar pulsaciones de teclado a la ventana que tiene el foco. SendHotKey amplía estas opciones y permite al usuario enviar combinaciones de tecla.

La gran mayoría de las aplicaciones que existen en el mercado, y sobre todo aquellas basadas en un sistema de ventanas, proporcionan atajos de teclado a los usuarios. Éstos permiten al usuario indicar al programa que realice una operación cuando se pulsa una combinación teclas.

Una vez explicado en qué consiste un atajo de teclado veremos cómo se ha implementado en nuestro sistema de reglas.
La clase SendHotKey deberá almacenar la información necesaria para poder realizar la acción. Paralelamente a como se ha realizado en el apartado anterior, se tendrán dos miembros privados por cada uno de los códigos virtuales que hay que almacenar.
A la primera tecla que se pulsa y se mantiene pulsada la llamaremos mod y a la segunda key.

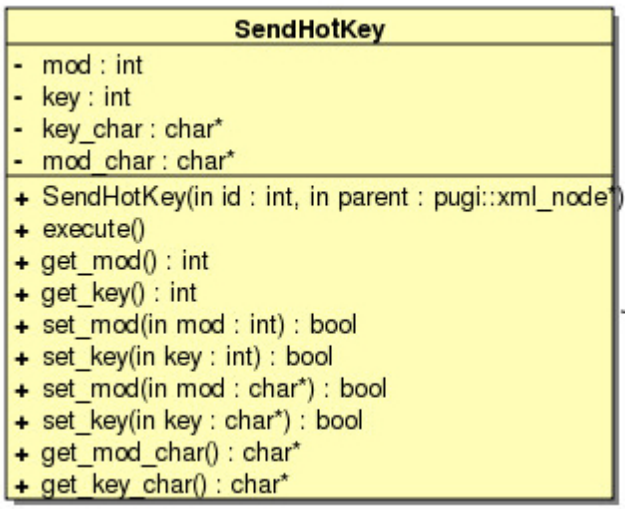

Figura 38: Diagrama de clase de SendHotKey.

En la Figura 38 vemos su modelo UML. Todas las operaciones para el manejo de los miembros privados tienen la misma forma que en los casos anteriores. Todos ellos modifican el modelo XML cuando realizan su operación.

La parte de ejecución tiene muchas partes en común con SendKey. Como se ha explicado anteriormente, en este caso hay que enviar cuatro eventos al sistema operativo. Los pasos a seguir serán:

- Envío de la pulsación de mod.
- Envío de la pulsación de key.
- Envío de la liberación de key.
- Envío de la liberación de mod.

El envío de las liberaciones de las teclas es muy importante. Si se nos olvidase enviar este evento, el sistema operativa se quedaría en un estado donde la tecla sigue estando pulsada por el usuario.

Al igual que el caso anterior todos los envíos de señales se realizarán usando la función SendInput.

5.4 Implementación de una regla

La unidad básica del sistema es la acción, pero una de las partes más importantes, si no la que más, son las reglas. Gracias a ellas se puede modelar cada una de las situaciones de las cuales queremos dar cuenta. Como ya se ha explicado, cada una de las reglas, generalmente, tendrá una serie de acciones asociadas las cuales son ejecutadas cuando la regla se valide.

Para poder representar una situación en concreto se tendrán en cuenta dos variables. La primera de ellas es el gesto que ha realizado el usuario. La segunda, y opcional, es la aplicación que tiene el foco.

Con el uso de estas dos variables se pueden describir muchas y diversas situaciones. La opción de controlar cuál es el programa que tiene el foco xes la que proporciona gran flexibilidad al sistema.

A continuación se explicará la clase RuleHeader. Cada una de las instancias de la clase RuleHeader representará una regla en el sistema.

La clase RuleHeader deberá almacenar toda la información necesaria para la representación de una regla. De igual forma, deberá proporcionar todos los métodos necesarios para la administración tanto de la regla, como de las acciones que contiene.

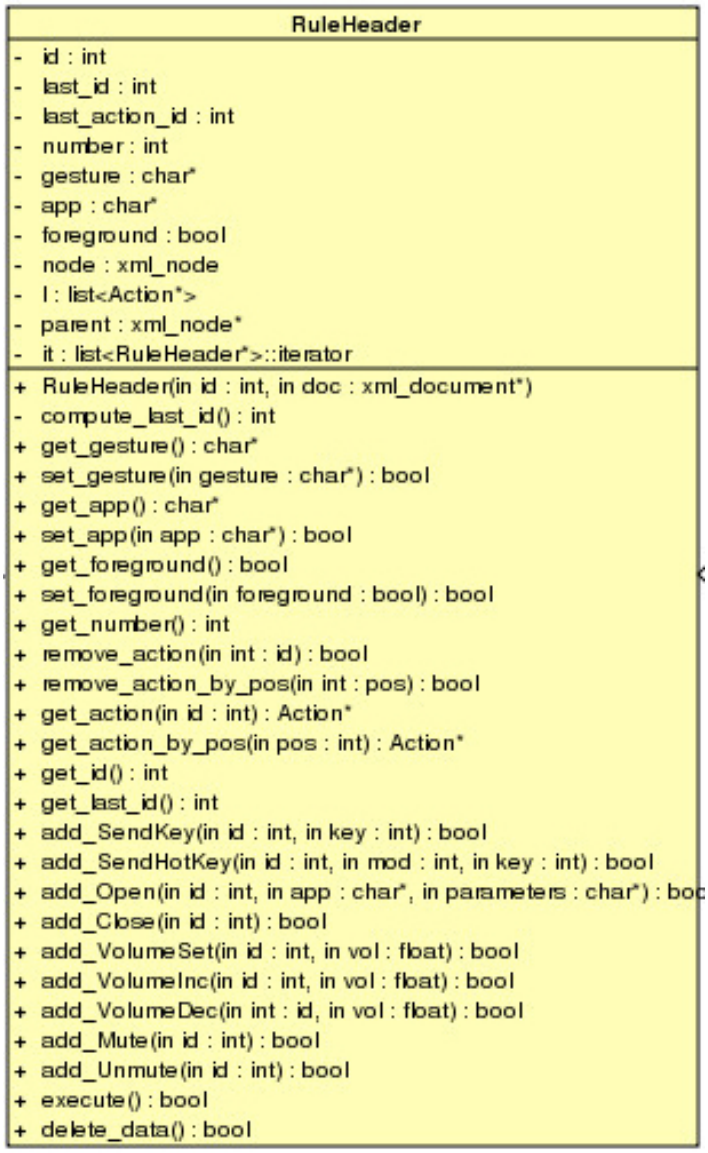

Figura 39: Diagrama de clase de RuleHeader.

En la Figura 39 vemos su modelo UML. A continuación se explicará cada unos de los miembros privados de los que consta RuleHeader. En ellos se almacena toda la información necesaria para la representación de la regla.

El primero miembro privado, y uno de los más importantes, es una variable de tipo entero y que tiene por nombre id. Esta variable contiene un valor entero con el número de identificación de la regla. Al igual que en las acciones, este número será único.

El siguiente miembro, `last_id`, es una variable de tipo entero. Esta variable se utilizará para administrar los identificadores de las acciones que están asociadas a la reglas. En ella se almacenará el último identificador asociado a una acción.

A continuación tenemos el miembro privado `number`. Es una variable de tipo entero y almacena el número de acciones que contiene la regla.

El siguiente miembro es `gesture` de tipo `char*` y almacena el nombre del gesto que se debe dar para que se active la regla.

Seguidamente se encuentran dos variables que se usan conjuntamente. La primera es app. Es una variable de tipo `char*` y almacena la aplicación que deberá tener el foco para que se valide la regla. La segunda es `foreground`. Esta variable de tipo booleano representa si debe darse la condición de la aplicación app con el foco o no. Si `foreground` es cierta deberá cumplirse, además del gesto, que la variable app es igual a la aplicación con el foco. En caso contrario, solo se comprobará la coincidencia en el gesto.

El miembro `node` es equivalente al que existe en la clase Action. Esta variable de tipo `xml_node` almacena el nodo XML que representa a la regla en el archivo XML.

Para el almacenamiento de las acciones asociadas a la regla se propusieron varias posibilidades. Una de ellas era la creación de un vector de Action donde se almacenara cada unas de las acciones. El problema de esta opción es que hay que controlar toda la lógica de las operaciones (desbordamiento, borrado, inserción). En alas de la simplificad y de no re-inventar la rueda, se tomó como opción la utilización de una lista enlazada STL.

Una vez elegida la acción se declaró el miembro privado `l` que era de tipo `list<RuleHeader*>`, que es una lista enlazada en la que cada elemento es un puntero a una instancia de la clase RuleHeader. Se tomó la decisión de almacenar punteros a objetos y no el objeto en si, porque facilita el posterior manejo del mismo.

El miembro parent es un puntero al nodo xml que apunta al sistema de reglas al que pertenece la regla.

La última de las variables que existen es `it`, la cual es un iterador sobre una lista de punteros a RuleHeader. Esta variable se utiliza para el acceso y modificación de la lista de acciones asociadas a la regla.

Por último, existe un único método que tiene una visibilidad privada. Éste se utiliza para resolver un pequeño problema que aparece cuando se eliminan y agregan acciones. Como se verá más adelante, cuando se añade una acción a una regla se le asigna como identificador el valor del miembro `last_id` aumentado en una unidad. A continuación se actualiza `last_id` en una unidad. Sin embargo, cuando se elimina una acción hay que calcular de nuevo cual es el id de mayor valor. Esta es la funcionalidad del método privado `compute_last_id()`. Simplemente recorre la lista de acciones asociadas a la regla y devuelve el id más grande.

A continuación, se explicarán cada uno de los métodos que componen la clase RuleHeader, a excepción de las operaciones sobre acciones, que se explicarán en los siguientes apartados.

El primer método público del que dispone la clase es obviamente el constructor. Tiene la siguiente cabecera:

```
RuleHeader(int id, pugi::xml_node* parent)
```

El primer parámetro representa el identificador que será generado por el RuleSystem al que pertenezca. El segundo es un puntero al nodo XML que representa el RuleSystem en el cual está contenida la regla.

La ejecución del constructor es muy similar al constructor de las acciones. Se pueden dar dos situaciones: o se está reconstruyendo un sistema de reglas desde un archivo XML o se está agregando una regla nueva al sistema.

En el primer caso, la regla ya existe en el fichero XML, por lo tanto hay que reconstruir la información de la regla. Se procede a leer cada uno de los atributos desde el XML y almacenarlos en el objeto. El siguiente paso es reconstruir las acciones que están asociadas a la regla. Para ello se recorre cada uno de los subnodos del nodo XML que representa la regla. Por cada uno de ellos se comprueba de que tipo son y se agrega esa acción al objeto RuleHeader.

En el segundo caso, se crea una regla donde sus parámetros estén vacíos. Como no existe un elemento XML que lo representa, se añade.

A continuación existen una serie de métodos que permiten acceder a las variables `gesture`, `app` y `foreground`. Paralelamente a los ejemplos anteriores estos métodos realizan las mismas operaciones. Los `get` devuelven el valor del miembro privado. Los set actualizan el nuevo valor tanto en el miembro como en el XML.

También existen los métodos `get_number()`, `get_id()` y `get_last_id()`. Estos devuelven el valor de las variables number, `id` y `last_id` respectivamente.

Antes de comenzar a tratar las operaciones sobre las acciones, hay dos métodos muy importantes dentro la clase RuleHeader.

El primero de ellos es `delete_data()`. Este método se encarga de eliminar toda la información relativa a la regla, desde las estructuras de datos hasta el nodo XML.

El segundo método y uno de los más importantes es `execute()`. Cuando una regla se valide, es decir, cumpla todas las condiciones de la misma, entonces se lanzará este método. A continuación se puede leer el código:

```
for(it = l.begin();it!=l.end();it++){
     (*it)->execute();
}
```

Cuando una regla se valida es que se ejecutarán cada una de las acciones que tiene asociada. Aquí podemos observar como se ha hecho uso del polimorfismo para la ejecución de las acciones. La variable `it` es un iterador sobre una lista de punteros a Action. En este bucle lo que estamos es recorriendo cada uno de

los elementos de la lista. Una vez hemos seleccionado uno, llamamos al método `execute()` del mismo.

Como se puede ver, en ningún momento se considera si las acciones son Open, Close, etc. Esto es así porque cada elemento de la lista es un puntero a Action pero en realidad a lo que apuntan es a un objeto de los tipos que heredan de Action. Por lo tanto, cuando se lanza el método `execute()` en realidad se está ejecutando el `execute()` sobreescrito de cada subclase. Gracias a este método se pueden agregar tantas acciones nuevas como se quieran, sin necesidad de modificar este código.

5.5 Operaciones sobre acciones

Una de las partes importantes del sistema de reglas, no es solo la ejecución, si no toda la administración de la información contenida en él. El usuario debe tener libertad para poder modificar el contenido de las reglas. Para ello se le proporciona una serie de operaciones que puede realizar sobre las acciones.

Las tres operaciones básicas que se implementan son:

- Inserción
- Modificación
- Borrado

Estas operaciones también estarán disponibles sobre las reglas cuando abordemos la gestión del sistema de reglas.

5.5.1 Inserción de una acción

La primera operación en la que se recala cuando se están diseñando las acciones, es la inserción de las mismas. Dado que no existe un único tipo de acción, se deberán crear tantos métodos para agregar una acción como subclases de Action. Cada una de estas operaciones deberá adecuarse a la acción que agregará a la regla. Por lo tanto, algunas tendrán parámetros adicionales que son necesarios, como por ejemplo la acción SendKey, y otras no, como Close.

Todos los métodos para agregar acciones al sistema recibirán al menos un parámetro. Este es el identificador de la regla y su nombre será id. Por lo tanto, todas aquellas acciones que no necesiten parámetros adicionales tendrán únicamente el identificador. Las demás tendrán tantos parámetros adicionales como sea necesario.

A continuación se detalla una lista con las cabeceras:

- `add_SendKey(int id, int key)`
- `add_SendHotKey(int id, int mod, int key)`
- `add_Open(int id,char* app, char* parameters)`
- `add_Close(int id)`
- `add_Maximize(int id)`

- add_Minimize(int id)
- add_ShowDesktop(int id)
- add_VolumeSet(int id, float vol)
- add_VolumeInc(int id, float vol)
- add_VolumeDec(int id, float vol)
- add_VolumeMute(int id)
- add_VolumeUnmute(int id)

Como se puede observar todos los métodos tiene como primer parámetro el identificador de la acción. Todas ellas tienen una ejecución casi igual. Como ejemplo veremos la implementación de la función add_SendKey.

```
bool RuleHeader::add_SendKey(int id, int key){
     SendKey* s = new SendKey(id,&this->node);
     s->set_key(key);
     l.push_back(s);
     this->last_id = compute_last_id();
     this->number++;
     return true;
}
```

Lo primero que se realiza es la creación de un objeto nuevo del tipo correspondiente a la acción. Como lo que se va a almacenar en la lista es un puntero a una acción, se crea un puntero del subtipo de acción correspondiente. Una vez creado el objeto, este no tendrán ningún parámetro establecido. Para ello se utilizarán los setter adecuados. A continuación se inserta el puntero en la lista l. Para ello se utiliza la función `push_back`, que emplaza el nuevo elemento al final de la misma. Por último, se actualizan las variables `last_id` y `number`.

Resumiendo, la inserción de nuevos elementos tiene cuatro pasos:

- Creación del objeto
- Asignación de los parámetros del objeto (si tiene)
- Inserción en la lista de acciones de la regla
- Actualización de las variables `last_id` y `number`.

Si se quiere ampliar el sistema con nuevas acciones, simplemente se tiene que crear la clase que herede de la clase Action. A continuación se añade un nuevo método en RuleHeader que se denomine `add_NameAction(int id, parameters...)`, para seguir con el formato establecido.

5.5.2 Modificación de una acción

El siguiente paso es la modificación. Para la modificación de una acción en concreto no se ha creado funciones que se han especificas de cada tipo. La solución propuesta es el diseño de una función de vuelva un puntero a la acción que se desea modificar. A partir de él se accederá a los métodos específicos de cada unas las subclases.

Para proporcionar la máxima flexibilidad posible, se han diseñado dos métodos para la recuperación de acciones:

- `get_action(int id)`
- `get_action_by_pos(int pos)`

El método `get_action` devuelve un puntero a un objeto de tipo Action el cual tiene como identificador `id`. El segundo método devuelve un puntero a un objeto de tipo Action que esté situado en la posición pos dentro de la lista.

Estas dos funciones devuelven un puntero a un objeto de tipo Action. Hay que tener en cuenta que el puntero que devuelvan al ser a Action no puede acceder a los nuevos métodos creados en las clases que heredan. Para poder acceder hay que realizar un casting de puntero a Action a puntero a la clase que deseemos. Aquí se muestra como realizarlo:

```
Action* a = rule->get_action(id);
if(a->get_type() == VOLUMESET_TYPE){
     VolumeSet* s = dynamic_cast<VolumeSet*> (a);
     s->set_vol(0.5);
}
```

La función `get_action` se llama sobre `rule`, que es un objeto de tipo RuleHeader, y nos devuelve un objeto de tipo `Action*`. Para poder acceder al método `set_vol`, por ejemplo, hay que convertir ese puntero a un puntero a VolumeSet. Realizaremos esta operación con el operador `dynamic_cast`.

5.5.3 Borrado de una acción

Las funciones de borrado también ofrecen dos formas de selección de la regla a borrar: acceder por identificador o acceder por posición. Por lo tanto las dos funciones serán:

- `remove_action(int id)`
- `remove_action_by_pos(int pos)`

La primera de ellas borrará la acción que tenga como identificador el valor id. Para ello comprobará que el id es mayor que 0 y a continuación recorrerá toda la lista hasta que lo encuentre. Una vez localizado realizará las siguientes operaciones:

```
(*it)->delete_data();
delete (*it);
l.erase(it);
this->number--;
this->last_id = this->compute_last_id();
```

La primera operación se encarga de eliminar toda la información relacionada con la acción. A continuación se libera la memoria del objeto creado. Después se borra el puntero de la lista y se actualizan los nuevos valores de las variables `number` y `last_id`.

En el segundo caso se realiza una búsqueda hasta alcanzar la posición deseada. Una vez determinado el punto, se realiza la misma operación comentada en el párrafo anterior.

5.6 Implementación de un sistema de reglas

Se empezó describiendo las acciones como unidad básica del sistema de reglas. A continuación se describió uno de los elementos más importantes, las reglas. Una regla actúa como un contenedor de acciones. Asimismo, el siguiente elemento a describir, y el último, tiene la forma de un contenedor de reglas.

La clase RuleSystem representa un sistema de reglas en su conjunto. Desde él se debe permitir al usuario administrar todas las opciones del mismo. A continuación se describirá en detalle cada uno de los elementos que la componen.

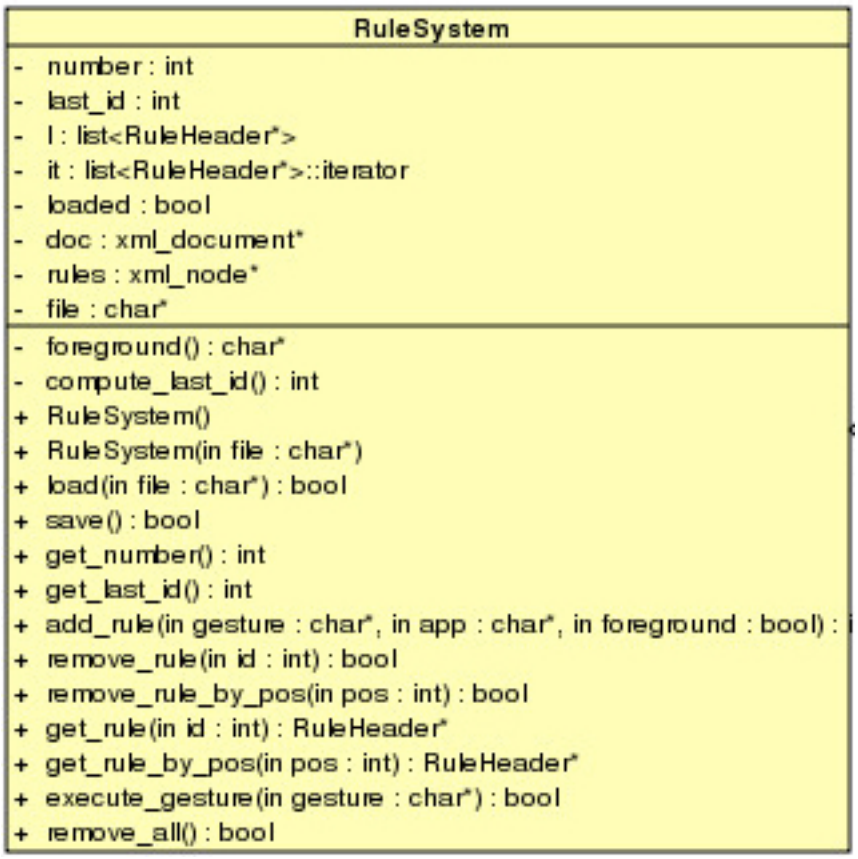

Figura 40: Diagrama de clase de RuleSystem.

En la Figura 40 vemos su modelo UML. El miembro privado `number` almacena el número de reglas que compone el sistema de reglas. La variable `last_id` representa el último id asignado a una regla.

Paralelamente a como está implementado RuleHeader, RuleSystem almacena las reglas asociadas a el dentro de una lista de STL. La implementación es muy parecida, cada elemento de la misma es un puntero a RuleHeader. Al igual que en el caso anterior, it es una iterador para la búsqueda dentro de la lista l.

En el miembro privado `loaded` se almacena un valor booleano que indica si hay un fichero XML cargado.

En la variable doc se almacena un puntero a un `xml_document`. Este tipo representa en pugixml un documento xml. Por otra parte, el miembro rules apunta directamente al primer nodo del XML.

En la variable `file` se almacena la ruta del archivo que contiene el XML.

Dentro de la parte privada encontramos dos métodos. Uno de ellos, compute_last_id, realiza la misma operación que el método del mismo nombre de la clase RuleHeader. En este caso recorre cada una de las reglas y encuentra la que tiene el máximo id.

Por otra parte `foreground()` utiliza una llamada a la API de windows para saber cual es la aplicación que tiene el foco en este momento. De esta forma se puede saber que aplicación tiene el foco necesaria para la búsqueda de un gesto.

Analizando ya la parte pública de la clase, el primer elemento que necesita consideración es el constructor. En este caso se ha optado por dotar a la clase de dos versiones del mismo. En el primero de ellos, simplemente se crea la clase. En el segundo, se pasa por parámetro el archivo desde el cual se cargarán las reglas. Para ello, este segundo constructor realiza lo mismo que el primero más una llamada al método load.

Como su propio nombre indica, el método load se encarga de cargar un archivo de reglas en formato XML. Recibirá como parámetro la dirección del archivo que se desea abrir. El proceso de cargar un archivo de reglas se divide en dos pasos.

En el primero de los pasos, se comprueba si ya exista algún sistema de reglas cargado con anterioridad. Si fuera cierto, se elimina cada uno de los elementos que componen el sistema de reglas antiguo.

La segunda parte, y que siempre se ejecutará, es la carga del fichero. Para ello se utiliza pugixml para parsear el archivo. Si todo ha funcionado correctamente, entonces se reconstruye el modelo de reglas del archivo con las clases correspondientes. Se recorre cada una de las reglas y se crea un objeto RuleHeader por cada una de ellas. A continuación se agrega el puntero al objeto a la lista l.

El método `save` se encarga de escribir toda la información modificada en el archivo XML. Dicha función simplemente llama a un método de la clase `xml_document` perteneciente a pugixml.

Al igual que en la clase RuleHeader, existen dos miembros públicos que devuelven el valor de `number` y `last_id` respectivamente.

5.7 Operaciones sobre reglas

Para el sistema de reglas, la unidad de almacenamiento serán las reglas. Para ello se han proporcionado, al igual que lo hacia RuleHeader con las acciones, una serie de operaciones que se pueden aplicar a las reglas. A las operaciones de inserción, modificación y borrado se han añadido otras como la ejecución de un gesto.

5.7.1 Ejecución de un gesto

De entre todas las operaciones que se pueden realizar, la que tendrá un mayor uso cuando se implante el sistema será la de ejecutar un gesto. Esta es la operación sobre la que se ha diseñado toda la interacción entre el sistema y los dispositivos de entrada.

Como se ha explicado durante el diseño del sistema, la interacción se realizará a través de un código que identificará el gesto unívocamente. Por lo tanto, la función `execute_gesture` simplemente recibirá como parámetro el gesto. La cabecera quedaría como:

```
bool execute_gesture(char* gesture)
```

A continuación se escribe un pequeño pseudocódigo para mostrar la idea básica de la implementación de la función:

```
for x in listofrules do
     if gesture equals x.gesture then
          if foreground is true then
               if app equals x.app then
                    x.execute
                    return 1
               else
                    return 0
               endif
          else
               x.execute
               return 1
          endif
     endif
endfor
```

Se recorre cada una de las reglas comprobando si cumplen las condiciones. Para ello se comprueba que cumplan cada una de las condiciones especificadas en la regla. Cuando se encuentra la regla, se ejecuta. Como se puede comprobar la primera regla que se valide será la que se ejecute.

5.7.2 Inserción de una regla nueva

La operación de inserción de reglas en el sistema, es similar a como se realiza la inserción de acciones. A diferencia del caso anterior, en este solo existe un tipo de regla, por lo tanto solo hay un método llamado `add_rule`.

Para añadir una regla nueva, simplemente se le debe indicar cuál es el gesto, la aplicación y el valor de `foreground`. Una vez se tiene esta información se crea un objeto nuevo de clase RuleHeader, se actualizan sus miembros privados a través de los getter y setter correspondientes. A continuación se inserta el puntero al RuleHeader en la lista l.

Con todo esto, ya se ha agregado una regla nueva al sistema. Para poder identificarla el miembro `add_rule` devuelve el identificador de la misma. De esta forma es fácilmente manejable después de su creación.

5.7.3 Modificación de una regla

Paralelamente a como se realiza la modificación de las acciones. Las reglas se pueden modificar a través de realizará la misma operación. Por lo tanto se proporcionarán dos operaciones que devolverán un puntero a la regla que se busca. Las cabeceras de ambas operaciones son:

```
RuleHeader* get_rule(int id)
RuleHeader* get_rule_by_pos(int pos)
```

En la primera se recorren todas la lista hasta que se encuentra la regla con el identificador almacenado en id. En la segunda, se devuelve el elemento pos-esimo de la lista.

5.7.4 Borrado de un regla

El borrado de reglas funciona al igual que en el caso de las acciones pero se añadirá una operación más para hacer el sistema más versátil.

Se dispondrán de tres operaciones:

- `remove_rule(int id)`
- `remove_rule_by_pos(int pos)`
- `remove_all()`

La primera operación borrará la regla que tenga el identificador id. El borrado de una regla es muy sencillo, simplemente se llama al método `delete_data()` de la regla, se elimina de la lista de reglas, se disminuye en una unidad `number`, se recalcula `last_id` y por último se libera la memoria ocupada por el objeto RuleHeader.

En la segunda operación la búsqueda se hace basándose en la posición del objeto en la lista. Por lo tanto se accede a la posición pos dentro de la lista. A continuación se realiza las mismas operaciones que en la operación anterior.

Por último tenemos la operación `remove_all()` que elimina todas las reglas contenidas en el sistema, dejándolo vacío.

5.8 Estructura cliente/servidor

Uno de los requisitos que se estableció en el análisis del problema es que el sistema debería permitir que diferentes dispositivos se conectaran a un único sistema de reglas. Para cumplir este requisito se ha optado por desarrollar una arquitectura cliente/servidor. De esta forma, cada dispositivo tendrá un cliente que se encargará de enviar la información necesaria al servidor. Por otra parte el servidor deberá administrar cada una de los gestos que recibirá.

Para la comunicación entre los dispositivos basándose en una estructura cliente/servidor se ha optado por montar un servidor TCP. Cada uno de los clientes leerá el gesto a ejecutar y mediante un socket TCP enviará la cadena de texto a un servidor conocido. Por parte del servidor se desarrollará un servidor TCP que lea de un puerto determinado las cadenas recibidas. A continuación se ejecutará el gesto.

La elección de montar un sistema cliente/servidor sobre TCP tiene una serie de ventajas. Las conexiones TCP comprueban que los paquetes enviados han sido recibidos por el receptor. De esta forma nos aseguramos que cualquier gesto realizado por el usuario se recibe en la parte del servidor. Otra ventaja es que el servidor y el cliente no tienen porque estar en el mismo ordenador. Se puede tener varios clientes en distintas máquinas que envíen, a través de una conexión a internet, el gesto realizado por el usuario. El servidor podrá recibir gestos de cualquier máquina conectada a internet.

Un inconveniente de este diseño es que la máquina o máquinas que estén involucradas deben tener una tarjeta de red.

Para la creación de los clientes y del servidor TCP se utilizará la plataforma .NET con el lenguaje C++.

5.9 Aplicación real del sistema

Se ha descrito en detalle cada una de las opciones que nos proporciona el sistema.

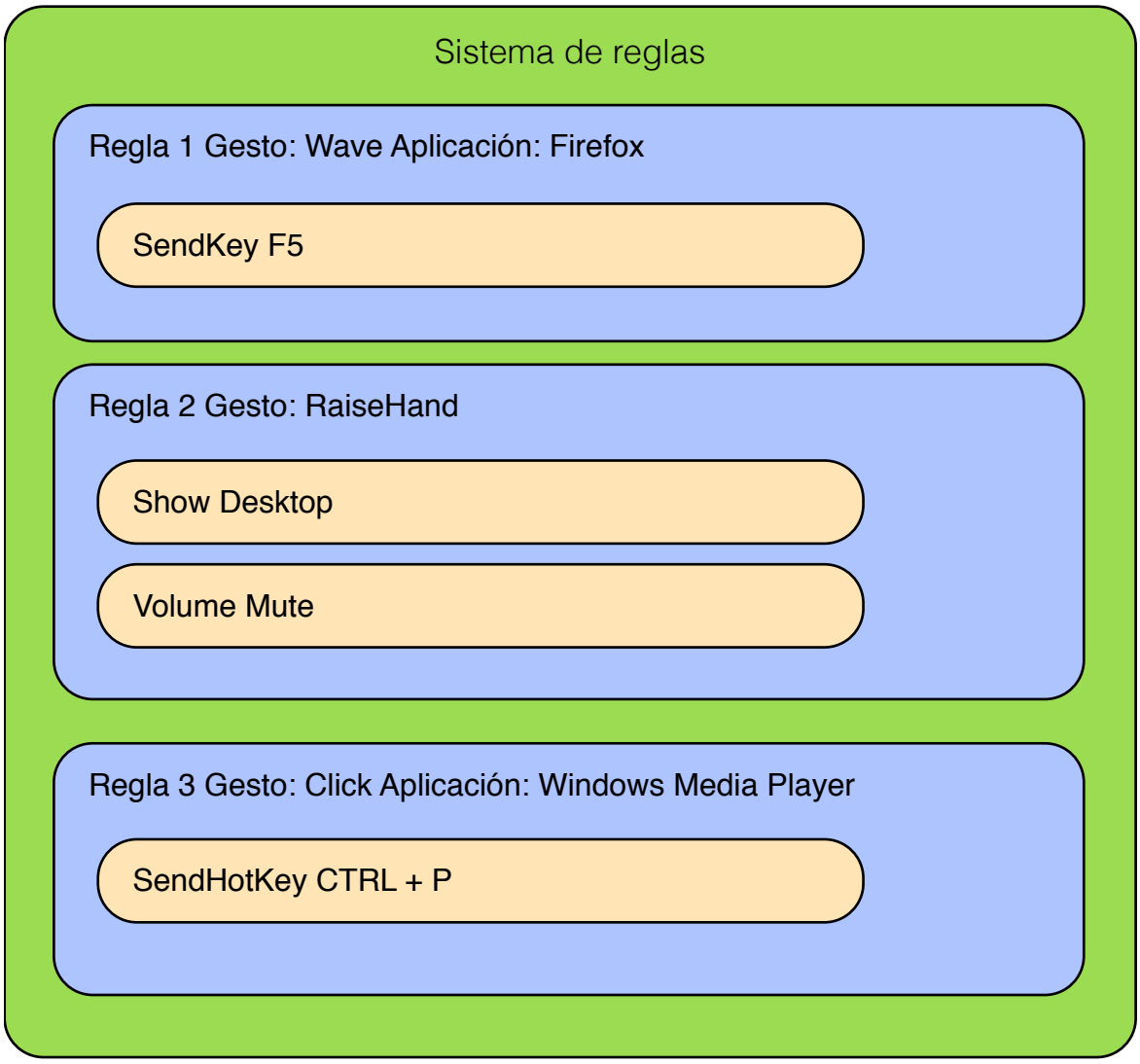

Figura 41: Ejemplo real de un base de datos de reglas.

En la Figura 41 se puede ver un ejemplo real de un sistema de reglas. En él se encuentran contenidas tres reglas.

La primera de ellas se validará cuando el usuario realice el gesto Wave y además la aplicación que tenga el foco sea el Firefox. En este momento se ejecutarán las acciones contenidas en ella. En este caso existe únicamente una acción que enviará la pulsación de la tecla F5 a la aplicación Firefox. Esto causará que la página web mostrada por la aplicación se recargue.
La segunda se ejecutará cuando el usuario realice el gesto RaiseHand. En este momento se realizarán las dos acciones asociadas a la regla número 2. Se mostrará el escritorio y a continuación se silenciará el sonido del sistema operativo.
Por último está regla número 3 que se validará cuando el usuario realice el gesto Click y la aplicación con el foco del SO sea el Windows Media Player. A continuación se enviará el atajo de teclado Ctrl + P que equivale a pulsar el botón Play/Pause.

Existen dos dispositivos de entrada que fueron explicados ampliamente, Microsoft Kinect y Arduino. Aunque son dispositivos diferentes, se demostrará que son fácilmente usables con el sistema de reglas. Para cada uno de ellos se

diseñará un sistema que reconozca uno o varios gestos y se lo comunique al sistema.

Aparte de ejecutar gestos, el sistema tiene un conjunto de operaciones sobre reglas y acciones. Para ello se ha diseñado una interfaz de usuario que permite al usuario realizar visualmente modificaciones sobre archivos de reglas. Gracias a ella, se podrá configurar todo un sistema y posteriormente utilizarlo con cualquier dispositivo que se desee.

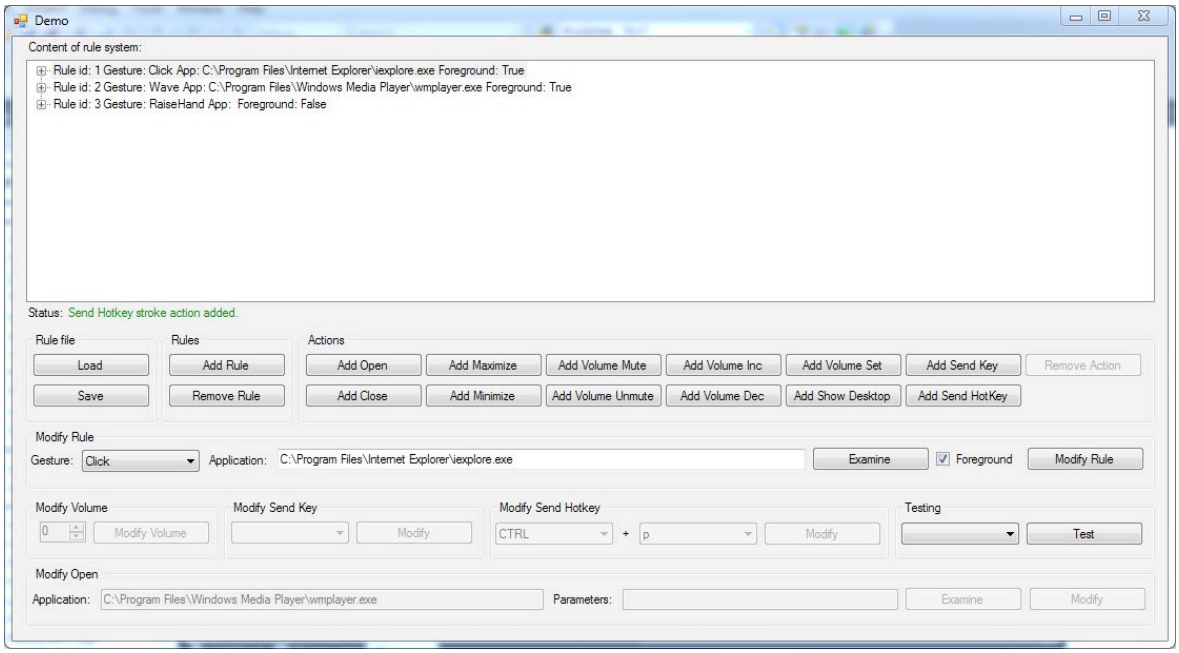

Figura 42: Interfaz de usuario para el manejo de reglas y acciones.

En la Figura 42 se muestra una captura de pantalla de la interfaz de usuario que se ha diseñado para mantener la base de datos de reglas. En ella se encuentran todos los controles necesarios para administrar todo el sistema de reglas. También permite probar las reglas antes de ejecutarlas con el sistema o cargar varios archivos de reglas.

Capítulo 6

Resultados

Después de analizar el problema que se nos presentó y de diseñar una solución tenemos una implementación que cumple todos los requisitos deseados. Desde un principio se ha buscado un sistema que ante todo fuera flexible a la hora de ser compatible los dispositivos de entrada. Como se puede ver en los anexos B y C, así ha sido. Gracias a esto no simplemente tenemos un sistema que es compatible con distintos dispositivos, si no que podrá ser adaptado para utilizarse con dispositivos que aparezcan en el futuro.

En el apartado interno del sistema toda la información está contenida en un archivo XML desde el cual se genera un DOM. Gracias a ésta técnica solamente se necesitarán realizar operaciones de entrada/salida de ficheros cuando se cargue o guarde el mismo. Para otras operaciones, se realizará todo el proceso en memoria.

Hay que tener en cuenta que estos sistemas deben ser lo más naturales posible, es decir, deben proporcionar un tiempo de respuesta rápido para que la interacción del usuario con el sistema sea natural. Parte de este tiempo de respuesta depende de la API del dispositivo de entrada. La otra parte es dependiente de nuestro sistema. En casos generales nuestro sistema de reglas contendrá del orden de 5 reglas y un máximo de 20. En estos casos el número de elementos no supondrá un lastre para el sistema.

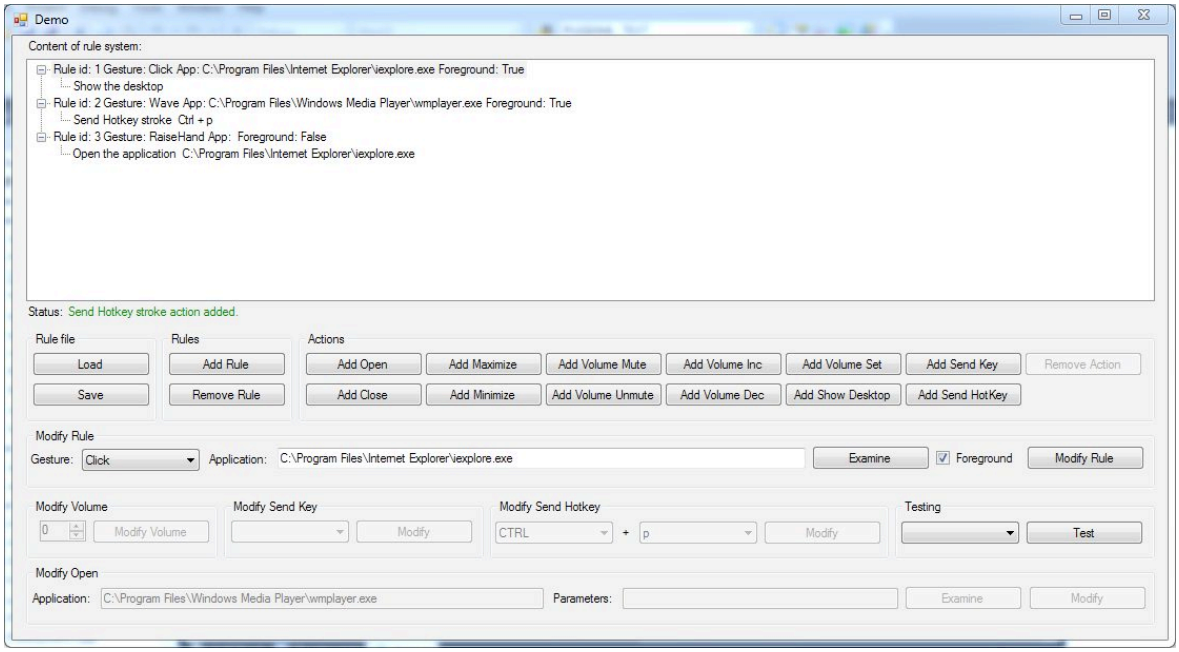

Figura 43: Interfaz de usuario para la administración de reglas.

Además de la flexibilidad que proporciona las reglas cabe mencionar la gran variedad de acciones que se pueden realizar. La interacción con el sistema de ventanas de Windows 7 es muy grande, pudiendo controlar las ventajas al antojo

del usuario. La posibilidad de generar pulsaciones de tecla y atajos nos permite un control más flexible de la aplicación.

Como resultado tenemos una herramienta (Figura 43) que proporciona todas las opciones posibles de administración del sistema. Gracias a ella el usuario tendrá la posibilidad diseñar visualmente el comportamiento deseado de nuestro sistema. La modificación, borrado e inserción de reglas y acciones se realiza de una forma muy simple.

Por último se mostrará un ejemplo de ejecución real del sistema.

Figura 44: Ejemplo de una base de datos.

En la Figura 44 vemos el ejemplo de base de datos que vamos a utilizar para mostrar una ejecución del sistema.

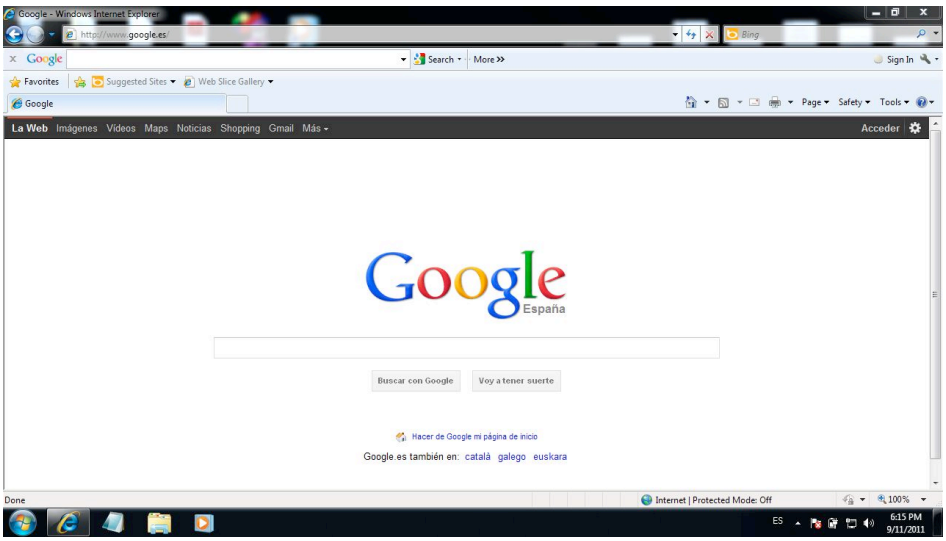

Figura 45: Windows 7 con IE abierto antes de que se ejecute el gesto Wave

En la Figura 45 se muestra una captura de pantalla del estado del sistema operativo antes de ejecutar el gesto. En este caso tenemos Windows 7 ejecutando Internet Explorer. A continuación se realizará el gesto Wave, por lo tanto se validará la regla 1 de la Figura 44.

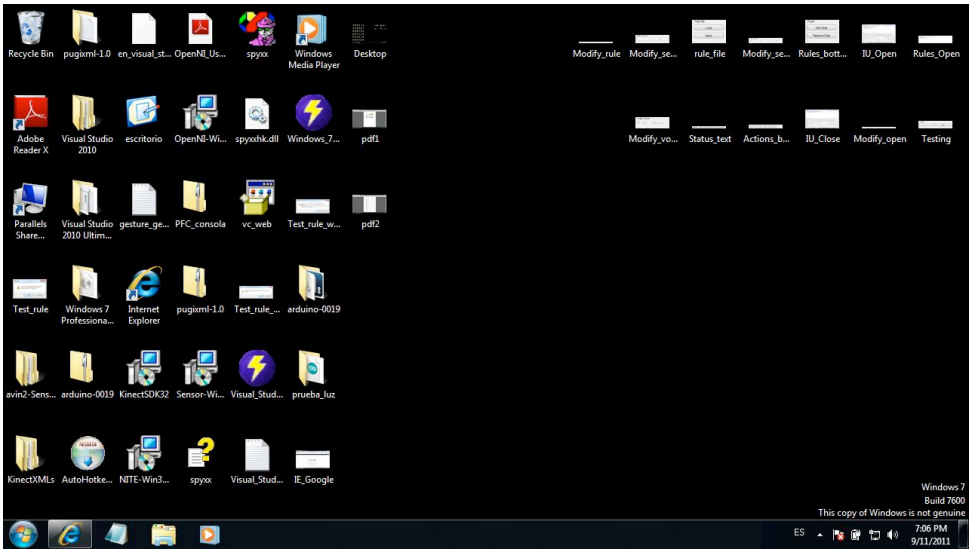

Figura 46: Estado de Windows 7 después de realizar el gesto Wave

Al validarse la regla número 1 se ejecuta la acción Show Desktop. Como resultado tenemos el estado mostrado en la Figura 46.

Para finalizar mostraremos una ejecución de la regla número 2.

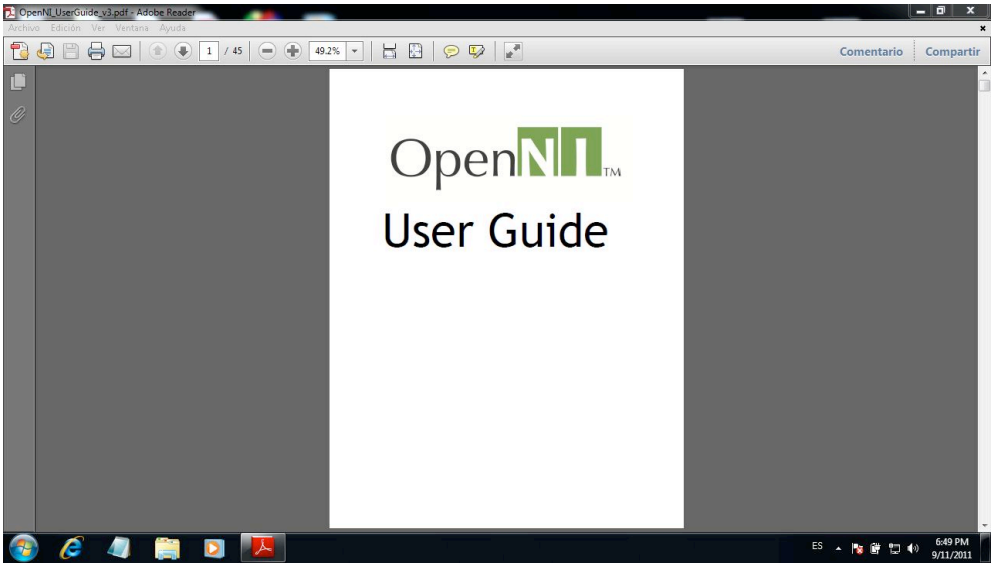

Figura 47: Estado de Windows 7 antes de que se realice el gesto Click

En la Figura 47 vemos un caso en el cual tenemos como foco la aplicación Adobe Reader mostrando un documento pdf. A continuación se realiza el gesto Click y por lo tanto se valida la regla número 2.

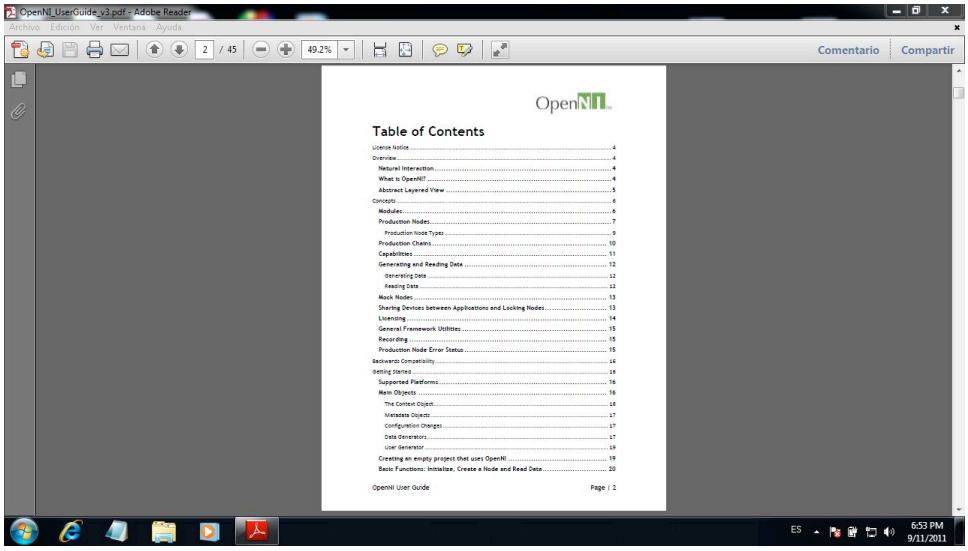

Figura 48: Estado de Adobe Reader después de la ejecución de la regla 2

En la Figura 48 podemos ver como al ejecutarse la regla número 2 se ha enviando la pulsación de la tecla flecha derecha. Con este atajo de teclado se pasa a la página siguiente del documento pdf.

Capítulo 7

Conclusiones

El proyecto ha consistido en el diseño y construcción de una solución a uno de los problemas que se plantean hoy en día en el mundo de las interfaces de usuario. Éstas han evolucionado con el paso del tiempo y este proyecto investiga una de las posibles vías que se abre delante nuestra: las interfaces de usuario sin contacto. La duración del proyecto ha sido de alrededor de un semestre y se ha dividido en varias fases.

En la primera fase del proyecto se asentó las bases de lo que se pretendía desarrollar. El objetivo era encontrar un sistema que permitiese al usuario utilizar el ordenador sin necesidad de tener contacto físico con el mismo. Antes de empezar su análisis se realizó un estudio exhaustivo de la tecnología que iba a ser utilizada en el proyecto. El dispositivo que se ha estudiado por utilizar es el Microsoft Kinect. Éste es un dispositivo originalmente diseñado para ser utilizado en la videoconsola Microsoft Xbox 360. Gracias a empresas y grupos de aficionados se han desarrollado una serie de APIs no oficiales debido a la falta de una oficial. De entre todas ellas se ha optado por utilizar la API OpenNI. Ésta proporciona una serie de funcionalidades que dan muchas opciones a la hora de utilizarla con Kinect. Adicionalmente a Kinect, que es un sistema cerrado, se ha investigado sobre Arduino que es un sistema muy abierto. Arduino es un controlador que permite la conexión de múltiples sensores como sensores de luz, sonido, temperatura, etc.

Después de seleccionar los dispositivos de entrada que se iban a utilizar en el proyecto, se empezó a analizar y a diseñar el sistema de reglas. En este proceso se encontraron diversos problemas y cuestiones las cuales introdujeron ciertos límites al sistema. Una de las cuestiones a tratar era saber hasta qué límite deberíamos incluir diferentes acciones para el usuario. La mayoría de las acciones son fácilmente implementables gracias a al API de Windows 7. Sin embargo las acciones de envío de tecla y atajos de teclado no se idearon desde el principio. La primera idea fue intentar comunicarse con cada una de las subventanas que conforman la ventana principal. De esta forma se podría controlar todos y cada uno de los elementos de la aplicación. Desgraciadamente la API de Windows complica mucho la situación, solamente deja acceder como mucho a dos subniveles de controles de una ventana. Después se tuvo la idea de enviar atajos de teclado directamente a la ventana principal. Otro de los requisitos era que se pudiera ampliar el sistema para añadir nuevas acciones fácilmente. Para ello se tuvo que utilizar técnicas de ingeniería del software como el polimorfismo. Otra decisión de diseño que se tuvo que tomar era cómo almacenar la información del usuario. Como ya se explica en la memoria, se eligió XML por su versatilidad y por ser suficientemente potente como para almacenar la información necesaria.

Como trabajos futuros, hay abiertas varias opciones dentro del sistema. En este momento el sistema funciona únicamente sobre Windows 7/Vista. Una de las opciones interesantes que se barajan, es la posibilidad de que el sistema sea

multiplataforma. Otra de las opciones que se barajan es la posibilidad de que existen diferentes sistemas de reglas dentro del mismo fichero XML. Cada sistema de reglas tendría un conjunto de reglas y acciones. De esta forma se podría cambiar entre sistemas de reglas sin necesidad de acceder a fichero diferentes. Otra opción sería montar el sistema bajo un servidor TCP. Los dispositivos enviarían el gesto a ejecutar mediante internet a un puerto en concreto y el servidor lo ejecutaría. De esta forma, no solo evitaremos tener que crear un sistema de reglas por dispositivo, si no que se pueden enviar los gestos desde otros ordenadores. Ésta opción daría una gran versatilidad.

Finalmente, mencionar que el sistema ha ido creciendo hasta convertirse en uno muy versátil, característica que se buscaba desde un principio. En el plano educativo, decir que este proyecto utiliza varias áreas de la informática como: Sistemas operativos, Ingeniera del Software, interfaces de usuario, programación de controladores, etc. Debido a ellos se han ampliado mis conocimientos en cada una de estas áreas.

A. Manual de la aplicación de edición de reglas

A continuación se explicará cada una de las opciones que se proporciona en la interfaz de usuario. Como se ha comentado antes esta debe aportar todas las opciones que están disponibles en el sistema. El programa está escrito en C++ utilizando la plataforma .NET.

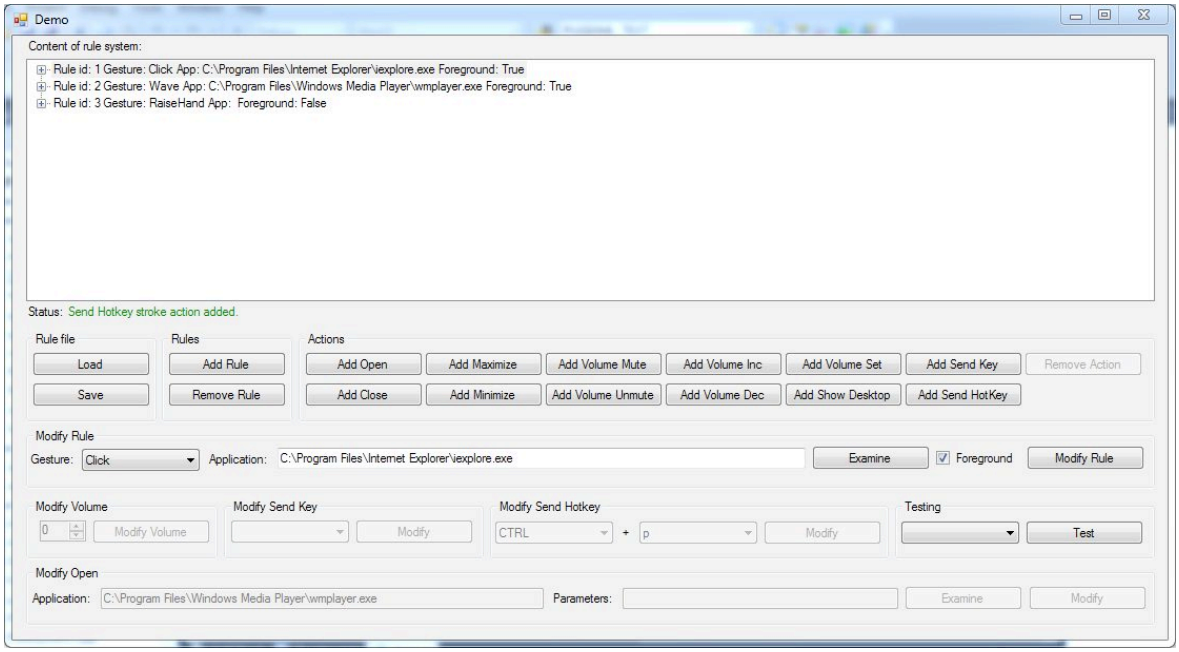

Figura 49: Visión general del programa para administrar reglas y acciones

En al Figura 49 se puede observar la ventana del programa que administrará todo un sistema de reglas y acciones. La interfaz está dividida principalmente en dos partes fácilmente diferenciables. En la parte superior se encuentra un elemento que permite la visualización del árbol de reglas y acciones. En la segunda parte existe una serie de controles que permiten al usuario realizar todas las operaciones que desee sobre el sistema. Las que modifica reglas y acciones harán que la parte superior se actualice.

Veamos en profundidad la parte superior de la interfaz de usuario. Aquí encontramos un elemento que es el denominado, TreeView. Este control permite representar árboles. Vamos a representar las reglas como nodos padre y las acciones asociadas a la regla como subnodos.

La Figura 49 muestra una base de datos con tres reglas. Cada una de ellas muestra:

- Identificador
- Gesto asociado
- Aplicación que debe tener el foco
- Variable foreground

Si una regla contiene acciones asociadas a la misma, a la izquierda de esta aparecerá un símbolo +. Pinchando sobre él, aparecerán todas las acciones que le pertenecen. Si una regla no tiene el símbolo + a su continuación, entonces la regla no tiene acciones asociadas.

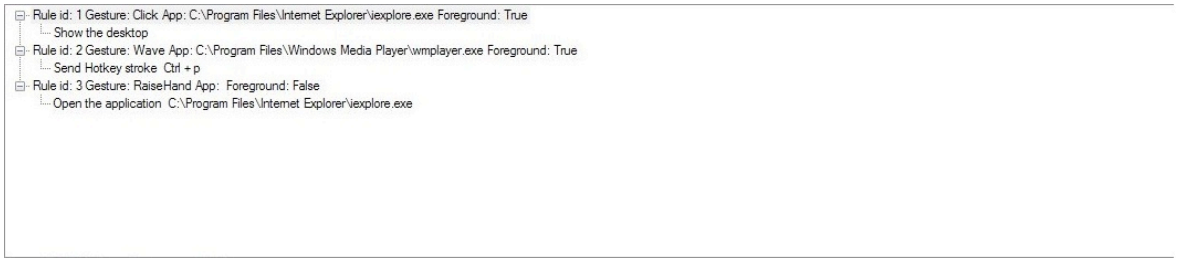

Figura 50: Visualización de las reglas desplegadas.

En la Figura 50 se puede observar como las reglas desplegadas. En su interior podemos hallar las acciones asociadas a las reglas. Cada acción viene acompañada por su información. Por ejemplo, la primera regla contiene una acción de tipo ShowDesktop la cual no contiene ninguna información añadida. Sin embargo, en la segunda regla existe una acción de tipo SendHotKey la cual muestra el atajo de teclado que va a ser enviado.

Los controles disponibles en la parte inferior de la ventana se activarán o desactivarán dependiendo del elemento seleccionado en la parte superior.

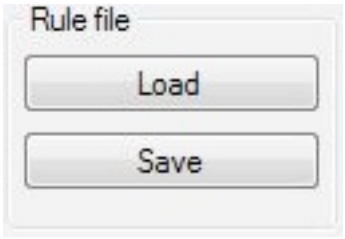

Figura 51: Operaciones sobre archivos de reglas.

La Figura 51 muestra dos operaciones sobre los archivos de reglas. Gracias al primero de ellos, se cargará un archivo de reglas. Cuando lo pulsemos aparecerá el cuadro de diálogo estándar para seleccionar un archivo. Una vez elegido el mismo será cargado y el árbol de reglas y acciones actualizado. Pulsando el botón Save se realizará un volcado de todas las modificaciones realizadas en el sistema al archivo XML. Si realizamos cualquier operación y no pulsamos este botón, ningún cambio se actualizará en el XML.

Figura 52: Añadir y borrar reglas.

La regla es el elemento más importante dentro del sistema de reglas. A partir de él se pueden modelar todas y cada una de las situaciones que deseemos. Con el botón Add Rule, que se muestra en la Figura 52, se podrá añadir una regla nueva al sistema. Ésta estará inicialmente vacía. Se podrán modificar más adelante con otras opciones.
Por otra parte está el botón Remove Rule. Como su nombre indica permite eliminar una regla del sistema. Como consecuencia de borrar una regla también lo hará las acciones asociadas a la misma.
Todas estas operaciones se podrán ejecutar únicamente si se tiene una regla seleccionada.

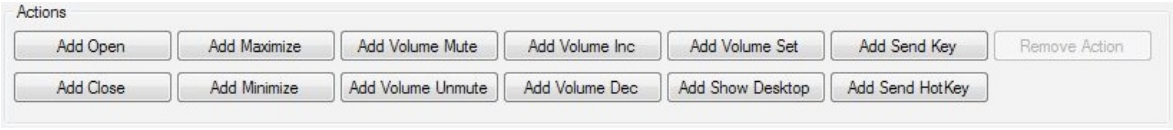

Figura 53: Operaciones relativas a las acciones

Una vez se ha añadido una regla, el siguiente paso es administrar las acciones relativas a esa regla. En la Figura 53 aparecen las posibilidades disponibles. Se ha diseñado un botón para añadir cada tipo de acción. Simplemente teniendo seleccionada la regla a añadir y pulsando uno de los botones, la acción se agregará. Al igual que pasaba en el caso de las reglas, las acciones se agregarán con los parámetros a nulo. Posteriormente se modificarán cada uno de ellos como se desee.
En la Figura 53 la opción de Remove Action está desactiva, porque lo que tenemos seleccionado ahora mismo es una regla. Seleccionando una acción y pulsando el botón Remove Action ésta ser borrará de la regla.

Figura 54: Modificación de una regla previamente creada.

Como se ha explicado anteriormente, todas las reglas nuevas que se agreguen lo harán con los parámetros a nulo. En la Figura 54 se muestra la parte de la interfaz de usuario que permite modificar los parámetros de una regla. Para modificar una regla en concreto deberemos seleccionarla en la parte superior de la interfaz. Cuando la tengamos seleccionada aparecerá toda la información correspondiente a la misma como se muestra en la Figura 54. En la lista gesture podremos modificar el gesto correspondiente a la regla de entre todos los disponibles. Por otra parte pulsando el botón Examine se abrirá una ventana para seleccionar la aplicación que debe tener el foco. Por último un checkbox nos permitirá seleccionar si queremos que la aplicación tenga el foco o no. Una vez realizado los cambios pertinentes pulsaremos el botón Modify Rule. Si no se pulsa, ningún cambio se guardará.

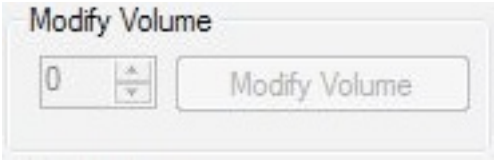

Figura 55: Modificación de una acción de tipo volumen.

Una vez hemos modificada la regla, podemos modificar las acciones que se han añadido con los valores predeterminados. En la Figura 55 podemos ver los controles que existen para modificar las acciones relacionadas con el volumen del sistema. Para poder acceder a estas opciones hay que seleccionar una acción que sea VolumeInc, VolumeDec o VolumeSet. Cuando se seleccione aparecerá el volumen de la acción en la Figura 55. Los valores están limitados entre 0 y 100. Pulsando el botón Modify Volume se actualizarán los cambios.

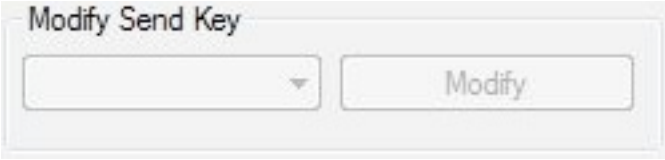

Figura 56: Modificación de una acción de tipo SendKey.

Para las acciones de tipo SendKey tenemos el recuadro mostrado en la Figura 56. Gracias a él podremos modificar el único parámetro que posee esta acción. En el ListBox aparecerán todas las posibles teclas que se pueden enviar como pulsaciones de teclado. Cuando hayamos seleccionado la deseada, pulsaremos el botón Modify.

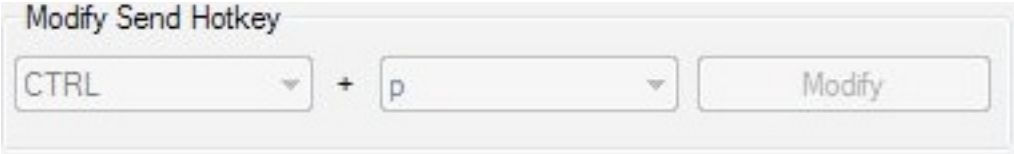

Figura 57: Modificación de una acción SendHotKey.

Paralelamente a como se realiza la modificación en las acciones SendKey, SendHotKey funciona de forma parecida (Figura 57). Al ser un envío de una atajo de teclado, este estará formado por dos teclas. La primera de ellas será la tecla que se mantendrá pulsada durante la ejecución del atajo. Dentro de la lista podremos encontrar las teclas más comunes que se encuentran en los atajos de teclado, como son la tecla control, shift o alt.

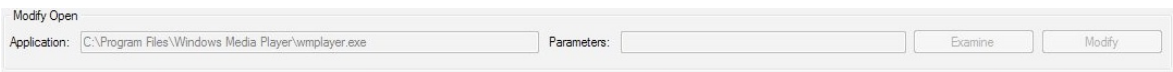

Figura 58 Modificación de una acción de tipo Open.

La última de las modificaciones corresponde la acción Open. Esta acción abrirá una aplicación pasándole los parámetros correspondientes. Por lo tanto en la Figura 58 podemos observar dos recuadros de texto. En el primero de ellos

seleccionaremos la aplicación a lanzar y en el segundo escribiremos los parámetros si los hubiera.

Hasta aquí hemos descrito todas las operaciones disponibles para la administración de las reglas y acciones. Pero esta interfaz de usuario proporciona un valor añadido al mostrar información y opciones adicionales.

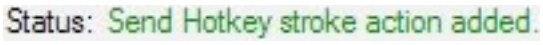

Figura 59: Texto que indica la última acción realizada.

En la Figura 59 se puede ver el texto Status que está situado justo debajo del cuadro de visualización de las reglas. Gracias a él podemos identificar si se ha ocurrido algún error en las operaciones lanzadas o por el contrario no existe ningún problema. En la Figura 59 aparece un ejemplo de lo que se muestra cuando se añada una acción de tipo SendHotKey.

Otra opción muy interesante que se proporciona al usuario es la de poder probar las reglas antes de utilizarlas con el dispositivo de entrada.

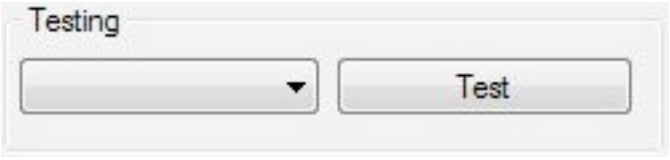

Figura 60: Testeo de las reglas.

En la Figura 60 podemos observar una imagen del testeo de reglas. Para ello se proporciona una lista en la cual aparecen los identificadores de todas las reglas presentes en el sistema. Se selecciona la regla que se desee comprobar y a continuación se pulsa el botón Test. Debido a que existe la variable foreground, existen dos casos posibles cuando se pulsa el botón Test.

- Primer caso. Foreground está a falso.

En este caso la variable foreground está a false lo que indica que la ejecución de la regla no depende del programa que tenga el foco del sistema operativo. Por lo tanto cuando pulsemos el botón Test aparecerá una ventana como la siguiente.

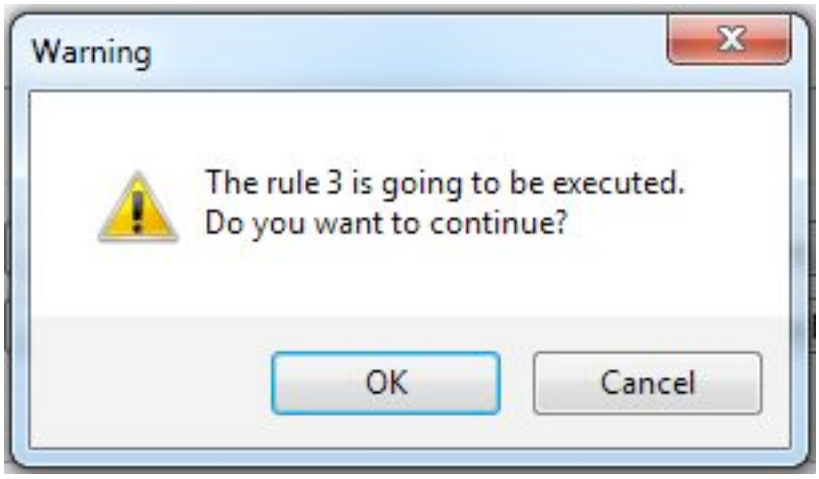

Figura 61: Aviso de ejecución de una regla.

La ventana de la Figura 61 nos avisa de que una regla, en este caso la 3, se va ejecutar y si deseamos continuar. Pulsando Cancel, se cerrará la ventana y no ocurrirá nada. Pulsando OK se ejecutará la regla.

- Segundo caso. Foreground está a verdadero.

En este caso las acciones se ejecutarán únicamente y exclusivamente si el programa indicado en la regla coincide con el programa que tiene el foco.
Puede darse el caso que las acciones asociadas dependan del programa que tiene el foco. Por ejemplo, el envío de un atajo de teclado al reproductor windows media player. Cuando pulsemos el botón Test nos aparecerá la siguiente ventana.

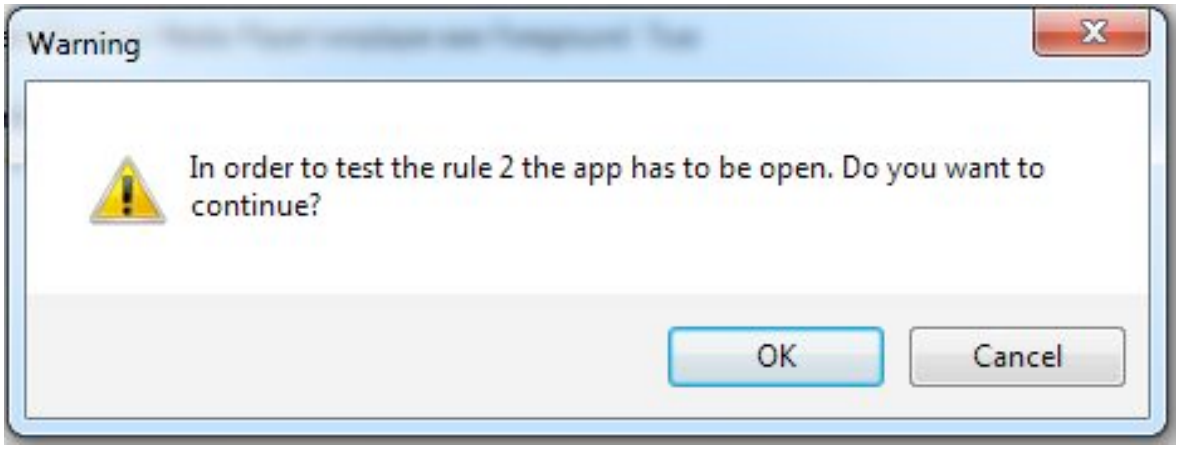

Figura 62: Aviso de apertura del programa de la regla.

En la Figura 62 vemos una ventana que nos avisa que la regla que se va ejecutar, la dos en este caso, necesita que la aplicación se abra. Si pulsamos OK se lanzará la aplicación y aparecerá otra ventana avisándonos de los siguientes pasos.

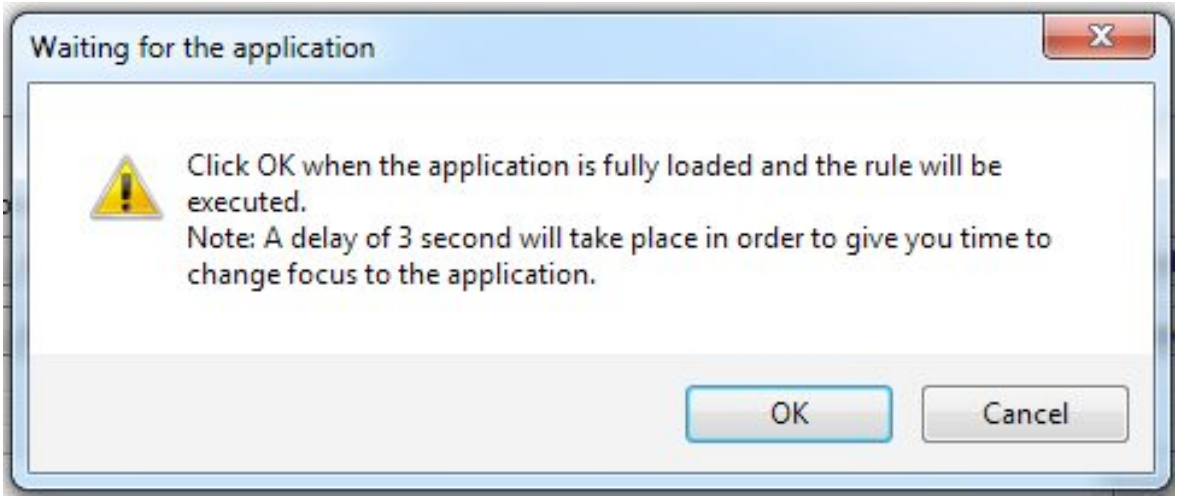

Figura 63: Aviso de ejecución de una regla sobre un programa.

Después de indicar que se lance la ejecución aparecerá una ventana como la de la figura 63. En ella se le pide al usuario que pulse OK cuando la aplicación este cargada. A continuación se producirá una pausa de tres segundos para que al usuario le de tiempo a cambiar el foco a la aplicación deseada. Esto se realiza así porque hay aplicaciones que tardan mucho en carga y si lanzáramos la regla justo al abrir la aplicación ésta no lo recibiría correctamente.

B. Ejemplo de implementación de un cliente con Microsoft Kinect

En este anexo se va a mostrar un ejemplo de como utilizar Microsoft Kinect y nuestro sistema para desarrollar una interfaz si contacto.

Como ya se explicó, existen varias API para Microsoft Kinect que nos dan acceso al dispositivo. En este caso vamos a utilizar OpenNI. La elección se ha tomado porque proporciona una serie de funciones a alto nivel que nos facilitan mucho el trabajo.

Se va a diseñar un programa que realice las siguientes operaciones. Cargará un archivo de reglas, empezará a buscar gestos del usuario, cuando reconozca uno se lo pasará al sistema de reglas y continuará buscando otros.

Para empezar a trabajar con OpenNI y nuestros sistema de reglas hay que incluir la librerías necesarias.

```
#include "stdafx.h"
#include <iostream>
#include <XnOpenNI.h>
#include <XnCodecIDs.h>
#include <XnCppWrapper.h>
#include "RuleSystem.h"
```

Las 4 librerías especificas son las siguientes:

- XnOpenNi.h : Incluye todas las funciones para el control del kinect.
- XnCodecIDs.h : Proporciona los codecs necesarios para conectar con el kinect.
- XnCppWrapper.h : OpenNI está escrita en C pero gracias a este wrapper podemos utilizarla con c++.
- RuleSystem.h : Nuestro sistema de reglas.

También se definirán una serie de constantes con el nombre de los gestos, para que sean fácilmente reconocibles.

```
#define GESTURE_CLICK "Click"
#define GESTURE_WAVE "Wave"
#define GESTURE_RAISE "RaiseHand"
```

Una vez se han incluido todas las librerías, se crean los objetos globales necesarios.

```
using namespace std;
xn::GestureGenerator g_GestureGenerator;
```

```
RuleSystem r;
```

El segundo es un generador de gestos necesario en OpenNI. El objeto r es nuestro sistema de reglas.

El siguiente paso es la creación de una callback para el reconocimiento de gestos. Ésta callback se llama cada vez que se reconozca un gesto mediante el GestureGenerator que se le pasa como parámetro. Siempre tendrá la misma cabecera.

```
void XN_CALLBACK_TYPE Gesture_Recognized(
      xn::GestureGenerator& generator,
      const XnChar* strGesture,
      const XnPoint3D* pIDPosition,
      const XnPoint3D* pEndPosition,
      void* pCookie)
{
      static int i = 0;
      cout<< "Gesture "<<i<<": " << strGesture << endl;
      r.execute_gesture((char*)strGesture);
}
```

Figura 64: Callback que se ejecuta al reconocer un gesto.

Dentro de la callback de la Figura 64 se imprime el nombre del gesto reconocido. A continuación se pasará el gesto mediante el método execute_gesture. A continuación tendremos que definir una callback para cuando se está procesando el gesto, es decir, mientras el usuario lo realiza.

```
void XN_CALLBACK_TYPE Gesture_Process(
            xn::GestureGenerator& generator,
            const XnChar* strGesture,
            const XnPoint3D* pPosition,
            XnFloat fProgress,
            void* pCookie)
{}
```

En nuestro caso, esta callback no contendrá código alguno porque no deseamos que realice ninguna acción mientras el usuario ejecuta el gesto.

Por último está el código principal, donde escribiremos toda la lógica del programa.

```
int _tmain(int argc, _TCHAR* argv[])
{
      xn::Context context;
      nRetVal = context.Init();
      if(!r.load_file("fichero.xml"))
            cout <<"Problem loading the XML." <<endl;
      // Create the gesture generator
      nRetVal = g_GestureGenerator.Create(context);
      // Register to callbacks
      XnCallbackHandle h1;
      g_GestureGenerator.RegisterGestureCallbacks
            (Gesture_Recognized, Gesture_Process, NULL, h1);
      // Start generating
```

```
	nRetVal = context.StartGeneratingAll();
	nRetVal = g_GestureGenerator.AddGesture(GESTURE_CLICK,NULL);
	nRetVal = g_GestureGenerator.AddGesture(GESTURE_WAVE,NULL);
	nRetVal = g_GestureGenerator.AddGesture(GESTURE_RAISE,NULL);
	while (TRUE)
	{
		// Update to next frame
		nRetVal = context.WaitAndUpdateAll();
	}
	// Clean up
	context.Shutdown();
	return 0;
}
```

Primero se creará un contexto y se inicializará. A continuación se cargará el archivo de reglas mediante la orden load_file sobre el objeto r. Después se creará el generador de gesto dentro del contexto. Se registrarán las callbacks que deseamos que sean llamadas al reconocer un gesto. A continuación se empezará a generar datos provenientes de la kinect. Se añadirán los gestos que deseamos reconocer, en nuestro caso, click, wave y raise. Entraremos en un bucle while donde se va actualizando los frames del kinect. Cuando acabemos limpiaremos todo utilizando el método shutdown.

Ejecutando este programa conseguimos que el usuario pueda realizar un gesto delante del Microsoft Kinect, este lo reconozco y envíe la información a nuestros sistema. Éste se encargará de realizar todas las operaciones asociadas.

C. Ejemplo de implementación de un cliente con Arduino

En este apartado diseñaremos un ejemplo que utilice el sistema Arduino como dispositivo de entrada. Los elementos utilizados son:

- Placa Arduino
- Danger Shield v1.0

Como ya se explicó en los dispositivos de entrada, Arduino es un controlador hardware que proporciona una serie de conexiones así como un lenguaje de programación. Por otra parte, la placa Danger Shield v1.0 irá conectada encima de la placa arduino. Ésta proporciona muchos sensores pero en este ejemplo utilizaremos el sensor de luz. El objetivo es detectar el paso de una mano por encima del sensor.

El sistema constará de dos partes bien diferenciadas. La primera de ellas es el código que estará ubicado en la placa Arduino. La otra parte es un programa que correrá sobre el ordenador donde se desean ejecutar las acciones.

Comenzaremos explicando la parte del Arduino. Éste está provisto de una memoria interna donde almacena el programa que tiene que ejecutar. En el momento en el que la placa recibe corriente eléctrica el programa se pone en ejecución. Para la comunicación la placa se utiliza un cable USB aunque después los datos se envían sobre un puerto de serie virtual.

A continuación se explicará el código que se ha utilizado. Se ha obviado la configuración de la placa Danger Shield, para mas información visitar su página.

```
//our sensor value
int value = 0;
int threshold = 0;
int nsamples = 10;
int b = 0;

void setup()
{
  ds_init();
  unsigned int aux = 0;
  for(int j = 0;j<nsamples;j++){
    aux = aux + analogRead(LIGHT_SENSOR_PIN);
  }
  aux = aux / nsamples;
  threshold = aux*0.9;
}


void loop()
{
```

```
  //get a reading from our analog pin
  value = analogRead(LIGHT_SENSOR_PIN);
  if(value < threshold && b == 0){
    digitalWrite(LED1_PIN, HIGH);
    b = 1;
  }

  if(value > threshold && b == 1){
    Serial.write("Hand\0");
    digitalWrite(LED1_PIN, LOW);
    b=0;
  }
}
```

El programa define cuatro variables globales que se utilizarán en el sistema:

- value. Almacenará el valor que se lee del sensor de luz.
- threshold. Valor a partir del cual se considera que hay una mano encima del sensor.
- nsamples. Número de muestras que se extraen al principio para determinar la cantidad de luz en el ambiente.
- b. Valor que utilizaremos como booleano para saber si hay una mano encima del sensor o no.

La función setup() empieza con la llamada a la función ds_init(). Ésta se encarga de configurar la placa Danger Shield, función que no vamos a entrar en detalle. A continuación se lee del sensor de luz tantas veces como indique la variable nsamples y se va acumulando las lecturas en la variable aux. Después se saca la media de esos valores y se multiplica 0.9. El resultado es el threshold que vamos a utilizar.

Por último está la función loop(). Lo que buscamos es que cuando el usuario pase la mano se envíe por el puerto serie un cadena de texto. El código realiza lo siguiente:

Se lee un valor del sensor. Ahora hay que detectar dos situaciones.

Si el valor del sensor es más pequeño que nuestro límite y ademas b, que significa que hay una mano encima, está a cero entonces ponemos b a 1. De esta forma indicamos que hay una mano encima en este momento.

En el siguiente caso es cuando se retira la mano. Por lo tanto se comprueba que el valor leído es ahora mayor que el límite y además que b está a 1, es decir, viene de un estado donde estaba la mano encima. En ese momento escribimos el gesto, en este caso Hand, por el puerto serie y ponemos b a 0.

Si cambian las condiciones de luz, simplemente tenemos que resetear la placa y se volverá a ejecutar setup. En esta función volveremos a leer los valores actuales de las condiciones de luz y se calculará otro umbral.

Ya hemos diseñado la parte del Arduino, de esta forma cuando se le conecte corriente al dispositivo empezará a leer del sensor de luz y a enviar los datos

pertinentes por el puerto serie. Para poder atender a este gesto hay que crear un programa en el otro lado que lea del puerto serie. A continuación se describe ese código:

```
#include "stdafx.h"
#include "SerialClass.h"
#include <iostream>
#include "RuleSystem.h"

#define BUFSIZE 10

int _tmain(int argc, _TCHAR* argv[])
{
	Serial s("COM3");
	if(s.IsConnected())
		std::cout<<"CONECTADO"<<std::endl;
	RuleSystem r;
	r.load_file("fichero.xml");
	char* buffer = new char[BUFSIZE];
	while(1){

		int x = s.ReadData(buffer, BUFSIZE-1);
		if( x != -1){
			std::cout << x << "-> " << buffer << std::endl;
			//Ejecutamos la regla
			r.execute_gesture(buffer);
		}
	}

	return 0;
}
```

Se ha incluido una librería llamada SerialClass [16] que está diseñada para usarse con Arduino. Básicamente el programa se conecta al serie puerto que hemos configurado también el Arduino. A continuación en un bucle lee del serie puerto y se lo pasa al sistema de reglas. Éste se encarga de ejecutar la regla.

Como se puede observar el programa que se ejecuta en el ordenador se puede utilizar con cualquier dispositivo que utilice Arduino. Solamente habría que adaptar el programa de Arduino para que utilizara los sensores correspondientes.

D. Códigos de teclas virtuales

A continuación se muestra la tabla de correspondencia entre códigos de tecla virtual y la tecla correspondiente. De esta forma se puede consultar cual es el código especifico si se desea implementar envíos de pulsaciones de teclado y atajos.

Symbolic constant equivalent	Hexadecimal value	Mouse or keyboard
VK_LBUTTON	01	Left mouse button
VK_RBUTTON	02	Right mouse button
VK_CANCEL	03	Control-break processing
VK_MBUTTON	04	Middle mouse button
	0507	Undefined
VK_BACK	08	BACKSPACE key
VK_TAB	09	TAB key
	0A0B	Undefined
VK_CLEAR	0C	CLEAR key
VK_RETURN	0D	ENTER key
	0E0F	Undefined
VK_SHIFT	10	SHIFT key
VK_CONTROL	11	CTRL key
VK_MENU	12	ALT key
VK_PAUSE	13	PAUSE key
VK_CAPITAL	14	CAPS LOCK key
	1519	Reserved for Kanji systems
	1A	Undefined
VK_ESCAPE	1B	ESC key
	1C1F	Reserved for Kanji systems
VK_SPACE	20	SPACEBAR
VK_PRIOR	21	PAGE UP key
VK_NEXT	22	PAGE DOWN key
VK_END	23	END key
VK_HOME	24	HOME key
VK_LEFT	25	LEFT ARROW key
VK_UP	26	UP ARROW key
VK_RIGHT	27	RIGHT ARROW key

VK_DOWN	28	DOWN ARROW key
VK_SELECT	29	SELECT key
	2A	Specific to equipment manufacturer
VK_EXECUTE	2B	EXECUTE key
VK_SNAPSHOT	2C	PRINT SCREEN key
VK_INSERT	2D	INS key
VK_DELETE	2E	DEL key
VK_HELP	2F	HELP key
	3A40	Undefined
VK_LWIN	5B	Left Windows key
VK_RWIN	5C	Right Windows key
VK_APPS	5D	Applications key
	5E5F	Undefined
VK_NUMPAD0	60	Numeric keypad 0 key
VK_NUMPAD1	61	Numeric keypad 1 key
VK_NUMPAD2	62	Numeric keypad 2 key
VK_NUMPAD3	63	Numeric keypad 3 key
VK_NUMPAD4	64	Numeric keypad 4 key
VK_NUMPAD5	65	Numeric keypad 5 key
VK_NUMPAD6	66	Numeric keypad 6 key
VK_NUMPAD7	67	Numeric keypad 7 key
VK_NUMPAD8	68	Numeric keypad 8 key
VK_NUMPAD9	69	Numeric keypad 9 key
VK_MULTIPLY	6A	Multiply key
VK_ADD	6B	Add key
VK_SEPARATOR	6C	Separator key
VK_SUBTRACT	6D	Subtract key
VK_DECIMAL	6E	Decimal key
VK_DIVIDE	6F	Divide key
VK_F1	70	F1 key
VK_F2	71	F2 key
VK_F3	72	F3 key
VK_F4	73	F4 key
VK_F5	74	F5 key
VK_F6	75	F6 key
VK_F7	76	F7 key
VK_F8	77	F8 key
VK_F9	78	F9 key
VK_F10	79	F10 key

VK_F11	7A	F11 key
VK_F12	7B	F12 key
VK_F13	7C	F13 key
VK_F14	7D	F14 key
VK_F15	7E	F15 key
VK_F16	7F	F16 key
VK_F17	80H	F17 key
VK_F18	81H	F18 key
VK_F19	82H	F19 key
VK_F20	83H	F20 key
VK_F21	84H	F21 key
VK_F22	85H	F22 key
VK_F23	86H	F23 key
VK_F24	87H	F24 key
	888F	Unassigned
VK_NUMLOCK	90	NUM LOCK key
VK_SCROLL	91	SCROLL LOCK key
VK_LSHIFT	0xA0	Left SHIFT
VK_RSHIFT	0xA1	Right SHIFT
VK_LCONTROL	0xA2	Left CTRL
VK_RCONTROL	0xA3	Right CTRL
VK_LMENU	0xA4	Left ALT
VK_RMENU	0xA5	Right ALT
	BA-C0	Specific to original manufacturer;
equipment reserved.		
	C1-DA	Unassigned
	DB-E2	Specific to original manufacturer;
equipment reserved.		
	E3 – E4	Specific to original manufacturer
equipment		
	E5	Unassigned
	E6	Specific to original manufacturer
equipment		
	E8	Unassigned
	E9-F5	Specific to original manufacturer
equipment		
VK_ATTN	F6	ATTN key
VK_CRSEL	F7	CRSEL key
VK_EXSEL	F8	EXSEL key

VK_EREOF	F9	Erase EOF key
VK_PLAY	FA	PLAY key
VK_ZOOM	FB	ZOOM key
VK_NONAME	FC	Reserved for future use
VK_PA1	FD	PA1 key
VK_OEM_CLEAR	FE	CLEAR key
VK_KEYLOCK	F22	Key used to lock device

E. Pugixml

En este anexo se explicará un ejemplo de uso de Pugixml dentro de un proyecto en C++.

Primero se añadirán el código fuente y la cabecera al proyecto. Una vez se han añadido los dos archivos ya se puede usar Pugixml. Para ello tendremos que incluir la librería mediante la cabecera `#include "pugixml.hpp"`.

Pugixml cuando lee un archivo o un buffer xml crea una estructura con forma de árbol en memoria. Esto es denominado como DOM, Document object model.
En Pugixml existen muchos tipos de nodos pero los más usados son

- node_document: Representa la raíz del árbol. A través de él se puede acceder a todo el árbol de nodos.
- node_element: Es el nodo más común y representa un elemento XML en el árbol. Desde él se puede acceder a su nombre, atributos y nodos hijos.

A continuación vamos a describir cómo se utiliza pugixml para tareas básicas como cargar un archivo XML, leer sus nodos, leer atributos, modificarlos y guardar un XML.

Para cargar un XML primero hay que crear un objeto de tipo `xml_document` en el cual se almacenará toda el árbol de información. Para ello basta con realizar las siguientes operaciones:

```
pugi::xml_document doc;

pugi::xml_parse_result result = doc.load_file("tree.xml");
```

Después de estas dos líneas de código, si todo ha ido bien, tendremos cargado todo el árbol en el objeto `doc`. En el objeto `result` tenemos toda la información de como ha ido la lectura de nuestro archivo tree.xml.

Aunque un objeto de tipo `xml_document` contiene un documento, este hereda de la clase `xml_node`. Por lo tanto, en nuestro caso, doc puede ser utilizado como si fuera un nodo.
Para acceder al primer nodo del documento se realiza la siguiente operación:

```
pugi::xml_node n = doc.first_child();
```

Una vez se tiene el primer nodo, deseamos acceder a información sobre él. Gracias al modelo DOM es muy sencillo realizar este tipo de operaciones. Por ejemplo para acceder a un atributo en concreto usaremos la siguiente función:

```
pugi::xml_attribute a = n.attribute("nombredelatributo");
```

Ahora la variable a, que es de tipo `pugi::xml_attribute`, contiene toda la información respecto al atributo. Desde ella podremos acceder, por ejemplo, a su nombre y valor:

```
std::cout << a.name() << " : " << a.value() <<endl;
```

El método `value()` devuelve el valor como una cadena de texto. Si el valordel nodo tiene otro tipo, existen una serie de operaciones que permiten una lectura correcta. Son las siguientes:

- `as_int()`
- `as_uint()`
- `as_double()`
- `as_float()`
- `as_bool()`

Por ejemplo, para leer el atributo a como un entero e imprimir su valor escribiríamos la siguiente línea:

```
std::cout << a.name() << " : " << a.as_int() <<endl;
```

Demos por hecho que hemos finalizado con el atributo a y con el nodo n, y nos gustaría pasar al siguiente nodo después del que contiene n actualmente. Esta es una operación muy importante cuando deseamos recorrer la estructura entera. Para ello utilizaremos la función `next_sibling()` de la siguiente forma:

```
n = n.next_sibling();
```

Y finalmente otro método que es importante para recorrer la estructura del árbol es `last_child()`. Este método nos devuelve el último nodo de ese nivel dentro del árbol.

Si se desea almacenar en el archivo XML todas las modificaciones que se han realizado, entonces se utilizará la siguiente línea de código:

```
doc.save_file("output.xml");
```

Bibliografía

[1] Windows 7. Dirección web: http://windows.microsoft.com/en-US/windows7/products/home

[2] Windows Vista. Dirección web: http://windows.microsoft.com/es-ES/windows-vista/products/home

[3] Microsoft Kinect. Dirección web: http://www.xbox.com/es-es/kinect

[4] Arduino. Dirección web: http://arduino.cc/

[5] Jennifer Lai, Clare-Marie Karat and Nicole Yankelovich. ***Conversational speech interfaces and technologies***. in The Human Computer Interaction Handbook, CRC Press, New York, p. 382

[6] Andrew T. Duchowski (2007). ***Eye tracking methodology.*** Springer, London, p 52-53.

[7] Tobii. Direccion web: http://www.tobii.com

[8] Elliptic Labs. Dirección web: http://www.ellipticlabs.com/

[9] EyeToy. Dirección web: http://es.playstation.com

[10] OpenKinect. Dirección web: http://openkinect.org/wiki/Main_Page

[11] OpenNI. Dirección web: http://www.openni.org/

[12] Microsoft Kinect SDK. Dirección web:http://research.microsoft.com/en-us/um/redmond/projects/kinectsdk/

[13] Windows API. Dirección web: http://msdn.microsoft.com/en-us/library/cc433218%28VS.85%29.aspx

[14] Pugixml. Dirección web: http://pugixml.org/

[15] IshellDispatch4. Dirección web: http://msdn.microsoft.com/en-us/library/bb774122(v=vs.85).aspx

[16] Serial Arduino. Dirección web: http://www.arduino.cc/en/Tutorial/SoftwareSerial