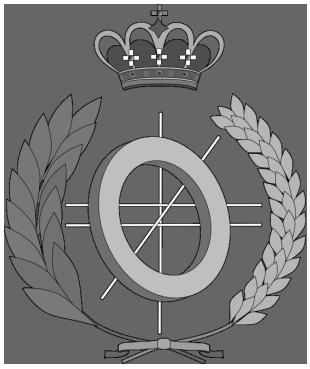

UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INFORMÁTICA

INGENIERÍA

EN INFORMÁTICA

PROYECTO FIN DE CARRERA

SimProc: Un simulador gráfico de procesos secuenciales automatizados

Óscar Gómez García

Marzo, 2006

Comentario:Siempre que le pongas algo nuevo puedes pensar en elaborar algunas notas para repasar luego durante el desarrollo y al cabo del tiempo. Algunos de estos documentos te ayudarán para elaborar la documentación del proyecto y los apéndices.

Índice general

Índice de figuras

Índice de listados

Capítulo 1

Motivación

Comentario:Importante investigar las sinergias entre la programación de sistemas industriales (Informática Industrial) y la ingeniería del software. ESTE ES UN PROYECTO DE INF. INDUSTRIAL QUE EXPLOTA LAS HERRAMIENTAS MÁS ACTUALES Y AVANZADAS DE INGENIERÍA DE SOFTWARE.

Esbozo de idea: La programación de sistemas industriales se hace con herramientas como Redes de Petri, Grafcets y diagramas de contactos. Tales herramientas no son sino mecanismos de representación que intentan simplificar la creación de algoritmos aplicados, en este caso, a problemas de índole industrial. La construcción de algoritmos y programas es una tarea compleja y propensa a errores que puede facilitarse por medio de herramientas que permiten tanto la creación como la depuración de los algoritmos, lo que permitirá la eliminación de errores en las etapas más tempranas del desarrollo de un sistema.

1.1. La informática industrial

La informática industrial es una rama de la informática orientada a la colaboración en el desarrollo de la industria de dos formas principales:

1. Ayudando en el control de procesos (e incluso realizándolo).

2. Ayudando en la creación de procesos.

En cuanto al control, esta parte de la informática industrial supone *mejorar* un proceso que ya ha sido automatizado.Las tareas que habitualmente desarrollan operadores humanos pueden llevarse a cabo por programas informáticos con las consiguientes ventajas:

- Mayor fiabilidad.
- Mejor tiempo de respuesta.
- Abaratamiento de costes.

Sin embargo la informática también puede colaborar con la industria ofreciendo herramientas que ayuden a automatizar procesos que realizados de forma manual serían inviables por razones prácticas o económicas (refinado de crudo, envasado de alimentos etc. . .). El desarrollo de procesos industriales es una labor similar al desarrollo de algoritmos y en este trabajo se explorarán las similitudes y se verá como una herramienta puede acelerar y facilitar esta tarea.

1.2. La ingeniería del software

Hasta 1968 los programas habían ido creciendo en tamaño y complejidad por lo que se hacía prácticamente imposible su desarrollo por parte de grupos pequeños[1]. Además, la aplicación de la informática a sistemas críticos tales como aviónica o defensa hizo crecer aún más las necesidades y por tanto la dificultad de desarrollo con la siguiente multiplicación de los defectos. En 1968, la OTAN acuñó el término “ingeniería del software” durante una conferencia en la que se discutió la necesidad de aplicar métodos sistemáticos a la programación de sistemas. La sistematización de tareas permitió abandonar, hasta cierto punto, la programación de forma artesanal para pasar a formas más disciplinadas y en suma, más cercanas a la ingeniería.

[1] En aquella época era frecuente el desarrollo de programas completos incluso por parte de una sola persona.

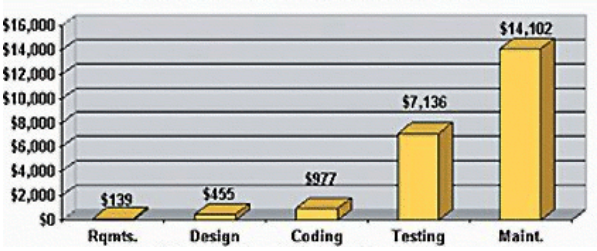

Figura 1.1: Crecimiento del coste en función del tiempo

1.3. Aspectos fundamentales de la ingeniería del software

Uno de los temas más abordados desde la ingeniería del software es la reducción de los defectos y a ser posible en las etapas más tempranas del desarrollo. En la figura 1.1 se muestra como el coste de los defectos se dispara cuanto más tarde se descubren. Esta médida, comprobada de forma empírica en multitud de estudios, ha afianzado aún más el interés por descubrir los defectos en los programas lo antes posible (la imagen pertenece a un estudio de Boehm publicado en la revista *IEEE Computer*([Boe91]).

Otra técnica sugerida desde la ingeniería del software se dirige a facilitar el desarrollo de aplicaciones. Algunas tareas como el desarrollo de interfaces o la programación de sistemas orientados a objetos pueden acelerarse y facilitarse mediante herramientas especialmente concebidas como por ejemplo Visual Basic, uno de los ejemplos más populares. En suma, todas estas tácticas se engloban bajo el término RAD (Rapid Application Development) y los principios fundamentales de la misma se desarrollaron por James Martin en un libro escrito en 1981 bajo el auspicio de IBM ([Mar81]).

Por último, la ingeniería del software ha impulsado el desarrollo de aplicaciones orientadas a cubrir todo el ciclo de vida de la aplicación, desde las etapas de diseño hasta las de mantenimiento. Estas herramientas, denominadas CASE, ofrecen al programador mecanismos de automatización de tareas e incluso sugerencias sobre posibles errores conceptuales. De esta forma, la cantidad de defectos se puede mantener en un nivel bajo a la vez que la complejidad se mantiene en niveles abordables. A modo de ejemplo, en la figura 1.2 se muestra una imagen de Rational Rose, una popular herramienta CASE asociada a la notación UML, que permite diseñar y generar software de forma visual y que incluye mecanismos de detección de errores.

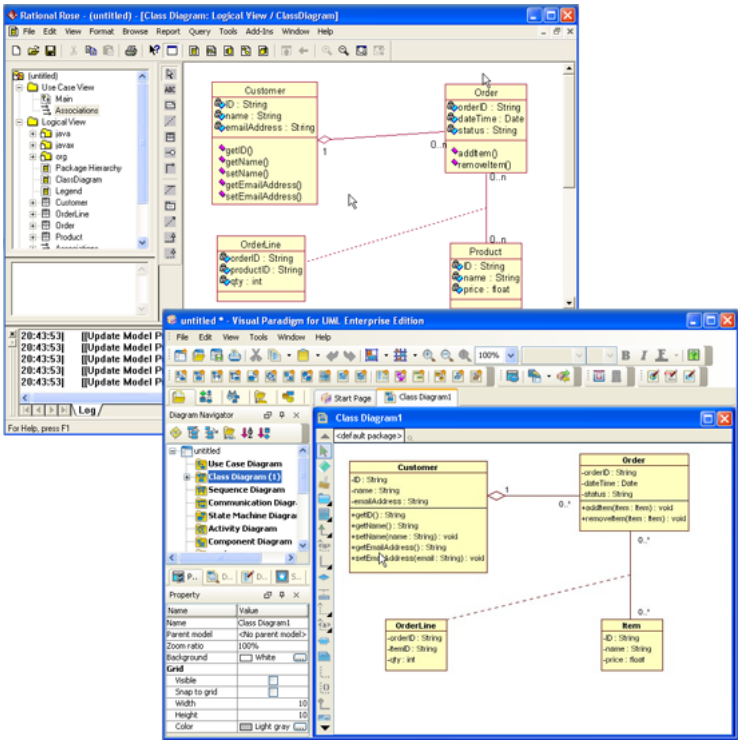

Figura 1.2: Rational Rose

1.4. Sinergias entre la informática industrial y la ingeniería del software

Comentario:Esta sección puede ser la clave de este capítulo y por tanto quizá haya que mirarlo cuidadosamente.

En las secciones anteriores se han examinado una serie de premisas importantes.

1. Por un lado, en la informática industrial se diseñan procesos automatizados que sustituyen a procedimientos artesanos. Estos procedimientos no dejan de ser algoritmos en los cuales los elementos ejecutores del sistema son elementos eléctricos, electrónicos y electromecánicos.

2. El desarrollo de algoritmos es una tarea compleja y muy propensa a errores. Estos errores pueden ser muy caros de subsanar e incluso ser críticos para la vida.

3. Existen enfoques y técnicas encaminados a reducir la complejidad y disminuir los errores en otras ramas de la informática.

En base a esto se pueden establecer paralelismos entre el desarrollo clásico de programas y el desarrollo de sistemas automatizados y desarrollar una *ingeniería del software de automatización*. En esta clase de ingeniería del software se aplicarán los conceptos y técnicas de la ingeniería del software clásica de forma que se mejore la calidad y fiabilidad de los sistemas automatizados.

En este trabajo se pretenden desarrollar los siguientes conceptos de forma práctica:

RAD: Construcción de una herramienta que permita desarrollar de forma rápida y sencilla prototipos de sistemas.

CASE: La herramienta permitirá cubrir el ciclo de vida de un sistema y permitir una fácil implantación del prototipo mediante kits de construcción de Fischer-Technik.

Detección de errores en etapas tempranas: El diseño y posterior simulación permitirá reducir el número de errores en los procesos antes de su implantación.

Capítulo 2

Objetivos

Los objetivos que este proyecto pretende cubrir son:

- Describir en XML los elementos que componen un sistema automatizado y construir un vocabulario de uso común para procesos. Este proyecto se ceñirá solamente al ámbito de control de procesos discretos.
- Desarrollar un modelo de objetos en C++ que repliquen en un sistema software el funcionamiento de los elementos de automatización.
- Desarrollar una aplicación multiplataforma que permita modelar, representar y simular procesos de automatización discretos en un entorno gráfico y que permita la interacción por parte del usuario. Esta aplicación servirá como apoyo a la docencia de la asignatura “Automatización Industrial” perteneciente al 4º curso de la titulación de Ingeniería Informática. Esta aplicación deberá poder comunicarse en un futuro con kits de construcción de sistemas automatizados. Además, esta herramienta buscará la consecución de las ventajas que ofrece la simulación por sofware frente a otras alternativas [Nee87].
  - Ayudar en el análisis, comprensión, diseño y control de sistemas.
  - Ahorro de costes al corregir y detectar errores antes de poner un sistema en producción.
  - La experimentación en entornos reales puede ser arriesgada y poco práctica.

- Descubrimiento de interrelaciones y efectos laterales que podrían haber permanecido invisibles hasta el momento de la construcción.
- Los modelos estrictamente matemáticos pueden ser difíciles de comprender para los no iniciados

2.1. Método y fases de trabajo

1. Investigación bibliográfica y del estado del arte: Antes de empezar el desarrollo de la herramienta se pretende acotar el alcance de la misma partiendo del estudio de las herramientas de simulación existentes y su adecuación a la práctica docente y educativa asi como del estudio de sistemas de automatización básicos [Mor95].

2. Análisis de las herramientas que proporciona XML y sus tecnologías relacionadas y como dichas tecnologías pueden ayudar a construir un lenguaje que describa sistemas de control de procesos discretos.

3. Utilizando C++ se procederá al modelado de los elementos, descritos en la fase anterior.Estos elementos escritos en C++ permitirán:

   - Crear un modelo de objetos en tiempo de ejecución que se correspondan con los objetos descritos en XML. Esto permitirá manipular dinámicamente los objetos y sistemas que han sido descritos estáticamente.
   - Guardar en un documento XML los objetos que componen un sistema automatizado que se esté simulando permitiendo así obtener una descripción a partir de un modelo en funcionamiento.

4. Construir un programa que permita de forma visual e intuitiva construir y simular modelos de procesos automatizados.

2.2. Medios que se pretenden utilizar

A continuación se describen las herramientas y tecnologías que se usarán así como las razones que han justificado su elección.

XML como formato de descripción de procesos. El uso de XML se justifica debido a lo extendido de su uso, a su estandarización y aceptación por herramientas de todo tipo y a su inteligibilidad para las personas [Hun05a].

Xerces. Xerces es un framework desarrollado como parte del proyecto Apache y que implementa los últimos estándares del World Wide Web Consortium [Fou03].Xerces proporciona todos los elementos necesarios para el procesado de XML.

C++ como lenguaje de desarrollo básico. La razón de utilizar C++ frente a Java u otros lenguajes actuales se debe a la buena adaptación de C++ (como sucesor/ampliación de C) a la programación de sistemas y a su relativamente bajo nivel [Str98b].

UML. Es el lenguaje de modelado más extendido en la actualidad [BJR98]. Permite documentar con claridad la estructura y el comportamiento de programas tanto a nivel interno (diseño e implementación) como a nivel externo (comportamiento).

GCC. Compilador disponible en multitud de plataformas [Fou06b] . Esta disponibilidad facilitará la portabilidad de la herramienta construida.

Dev-CPP. Es un entorno de desarrollo integrado para Windows que emplea GCC e incorpora herramientas para la depuración y que puede sustituir a herramientas comerciales sin excesivos esfuerzos de adaptación [LBL96].

WxWidgets. Es un conjunto de clases C++ orientadas al desarrollo de aplicaciones multiplataforma que utilicen un interfaz gráfico de usuario. WxWidgets también es software libre disponible en diversas fuentes WWW [Sma92].

Sistema operativo. El sistema operativo sobre el que se realizará el desarrollo será Windows. Dados los medios descritos anteriormente el proceso de adaptación de la aplicación a un sistema operativo Linux será relativamente sencillo.

Hardware de desarrollo. La máquina a utilizar será cualquiera basada en la plataforma Intel 80x86 debido a lo amplio de su disponibilidad.

Gestión de la configuración. Se usará un repositorio perteneciente a Berlios, que ofrece repositorios CVS con mantenimiento gratuito a los desarrolladores europeos de software.

Pruebas del software. Para las pruebas se utilizarán distintas herramientas dada la necesidad de abordar las pruebas del software desde distintas perspectivas. En concreto se usarán GCOV, GPROF y UNIT++.

- GCOV permitirá ejecutar pruebas de cobertura sobre el código y la comprobación de las distintas ramas de ejecución [Fou06a].
- GPROF se utilizará para las pruebas de rendimiento y análisis de las partes críticas del código.
- UNIT++ permitirá crear pruebas de unidad [Dra05]. Cabe destacar que las pruebas de unidad son una de las herramientas más utiles para la detección de errores [BA04].

Capítulo 3

Estado del arte

Comentario:En todo este capítulo faltan poner referencias a las webs de los programas, así como un par de libros para profundizar en redes de petri y afds

3.1. La representación de procesos

Los mecanismos de representación de procesos son muy variados y con características e intereses muy diversos. En esta sección se abordará el estudio de los más conocidos y se sentarán las bases para el uso de uno u otro en la herramienta a desarrollar.

Comentario:He cambiado el título de esta sección, me parecía más apropiado reencaminar de otra forma este trozo.

3.1.1. Diagramas de escalera

Los diagramas de escalera son una representación gráfica de las distintas etapas que atraviesa el flujo de control a lo largo de un proceso. Dicha representación utiliza símbolos tales

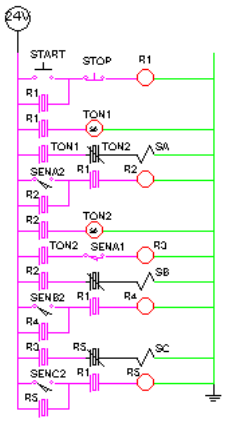

Figura 3.1: Un diagrama de escalera

como contactos o relés y los valores que pueden tomar dichos símbolo son variables lógicas que suelen tomar valores sencillos como “verdadero” o falso o números naturales.

Deben su nombre a su parecido con el esquema de una escalera, tal como se muestra en la figura 3.1.

Los diagramas de escalera son herramientas especialmente útiles para la representación de procesos discretos que tienen poco o ningún paralelismo. Su expresividad es muy limitada pero goza de la ventaja de que diversos controladores comerciales son programables directamente mediante este sistema.

Otra características reseñable de los diagramas de contactos es su gran popularidad, ya que al basarse en los esquemás clásicos de electricidad, permitió en su momento una fácil adaptación desde tecnologías más antiguas hacia los modernos autómatas programables.

3.1.2. Grafcet

GRAFCET significa GRÁFico de Control de Etapas de Transicion y es un sistema de representación de procesos secuenciales. Son más parecidos a los diagramas de flujo que los diagramas de escalera y son especialmente útiles para remarcar la secuencialidad de las etapas por la que pasa un proceso. El paso de una etapa a otra está controlada por transiciones y a dife-

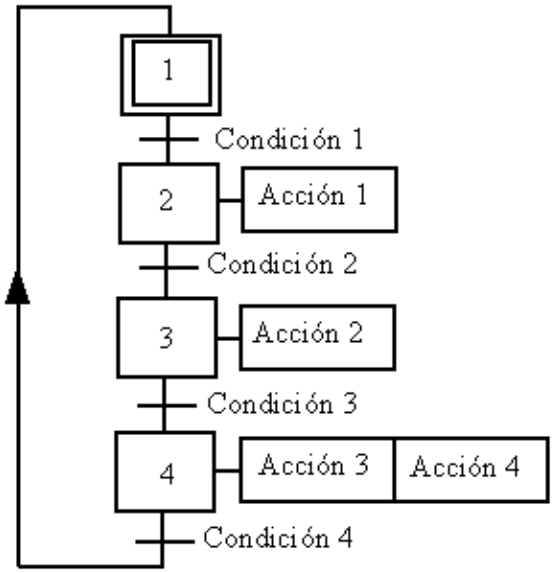

Figura 3.2: Un ejemplo de Grafcet

rencia de los diagramas de escalera, son más expresivos para mostrar el paralelismo de etapas.

En la figura 3.2 se puede ver un GRAFCET con 4 etapas. La etapa 1, con un borde doble, es la etapa inicial. El flujo de control no pasará a la etapa 2 hasta que no se cumpla la condición 1. Una vez que la condición 1 sea cierta, la etapa activa pasará a ser la etapa 2 y se ejecutará la acción 1.

Los GRAFCET son herramientas muy útiles para la representación de procesos discretos y dirigidos por eventos, ya que recalcan la secuencialidad de las etapas así como el orden de las etapas y la condicionalidad del paso de una etapa a otra.

3.1.3. Redes de Petri

Las redes de Petri son unos diagramas de representación de procesos orientados a la representación de procesos discretos pero *en los que desea mostrar o recalcar el paralelismo entre acciones*. Creados por Carl A. Petri en 1962 consisten, a grandes rasgos, en un grafo dirigido

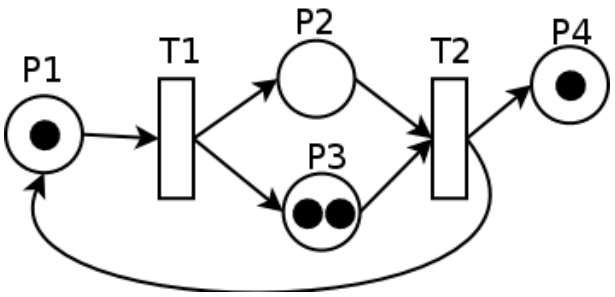

Figura 3.3: Red de Petri

en el cual hay nodos lugar, nodos transición y arcos que conectan lugares con transiciones. Se puede ver un ejemplo de red de Petri en la figura 3.3.

Se debe destacar el hecho de que las redes de Petri son diagramas *no-deterministas*, es decir, el estado de un proceso puede ser uno, cero o muchos de los “estados” representados en el diagrama. Debido a su no-determinismo, las redes de Petri se utilizan especialmente en el ámbito de la representación de procesos concurrentes.

Aunque existen todo un mecanismo formal matemático asociado a las redes de Petri, los principales aspectos a conocer son los siguientes:

- Cuando un lugar conduce a una transición, se denomina “lugar de entrada”.
- Cuando una transición conduce a un lugar, se denomina “lugar de salida”.
- Se puede partir de cualquier combinación de fichas en el lugar que se desee.
- Las etapas de transición envían las fichas a las transiciones.
- Las etapas de salida envían las fichas de las transiciones hacia los lugares.

Si se establece la similitud de la transferencia de fichas hacia distintos lugares con la transferencia del control entre etapas, se puede representar de forma sencilla el paralelismo entre etapas.

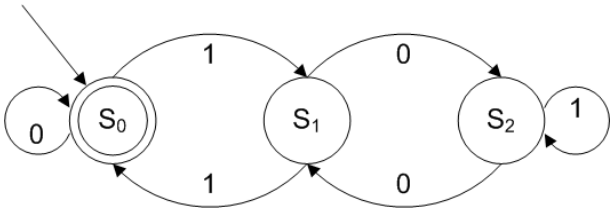

Figura 3.4: Un autómata determinista

3.1.4. Autómatas finitos

Los autómatas finitos son una representación de máquinas de estados gobernadas por una entrada o secuencia de entradas. Los autómatas finitos pueden ser deterministas y no deterministas. Tal división corresponde a la posibilidad de conocer o no de forma determinista en qué estado se encuentra un proceso dado. Aunque los autómatas determinstan son más parecidos al mundo real, los no deterministas tienen mayor poder expresivo y pueden ser útiles en situaciones en las que los autómatas deterministas no pueden representar ciertos aspectos de un sistema.

Un autómata finito es un conjunto de estados y transiciones. Dado un estado A se cambiará a un estado B si y solo si existe una entrada a y una transición que diga que se puede pasar de ese estado A a B. Un ejemplo más concreto se puede ver en la figura 3.4. El estado inicial es S0 (marcado con una flecha) y se permanecerá en él mientras se reciban ceros a la entrada. Cuando se reciba un 1 se producirá un cambio de estado y el nuevo estado actual será S1. Si estando en S1 se recibe un 1 se cambia al estado S0, el estado final (representado con un doble círculo). Si estando en S1 se recibe un 0 se cambia a un nuevo estado S2 donde las transiciones fluctuarán de forma similar.

Los autómatas finitos concentran la representación en el cambio de estados más que en la secuencialidad de un proceso concreto lo que puede ser más útil que los diagramas de escalera o GRAFCET en ciertas ocasiones. Además, los autómatas no deterministas pueden utilizarse también para representar el paralelismo, aunque las redes de Petri son más apropiadas en este sentido.

3.1.5. Programación de PLCs

Comentario:En este apartado se pueden poner muchas y muy distintas cosas. ¿Cuales abordo?

3.2. La simulación de procesos discretos

Comentario:¿Qué pongo en este apartado. ¿Comparar procesos discretos y continuos? ¿Hablar de las características específicas que tiene el simular procesos discretos?

3.3. Herramientas de simulación

Comentario:Muy importante: que lo centres en las herramientas de simulación de sistemas secuenciales “DISCRETE EVENTS DRIVEN SIMULATION”. Simulación de sistemas dirigidos por eventos.

En la actualidad existen muchas herramientas de simulación en el mercado con objetivos muy diversos, desde la simulación de pequeños a muy grandes proyectos, desde los que se concentran en sectores concretos de la industria a los que cubren todo el abanico de componentes eléctricos y electrónicos.

Todas las comentadas en esta sección permiten en general la simulación de cualquier tipo de proceso, ya sea continuo o discreto, secuencial o paralelo. En el estudio realizado se ha examinado la capacidad de la herramienta o las facilidades que ofrece para simular procesos discretos y dirigidos por eventos.

Este análisis permitirá enfocar el trabajo desarrollado en este proyecto: la creación de una herramienta *open source* que permita realizar experimentos de simulación de sistemas que van

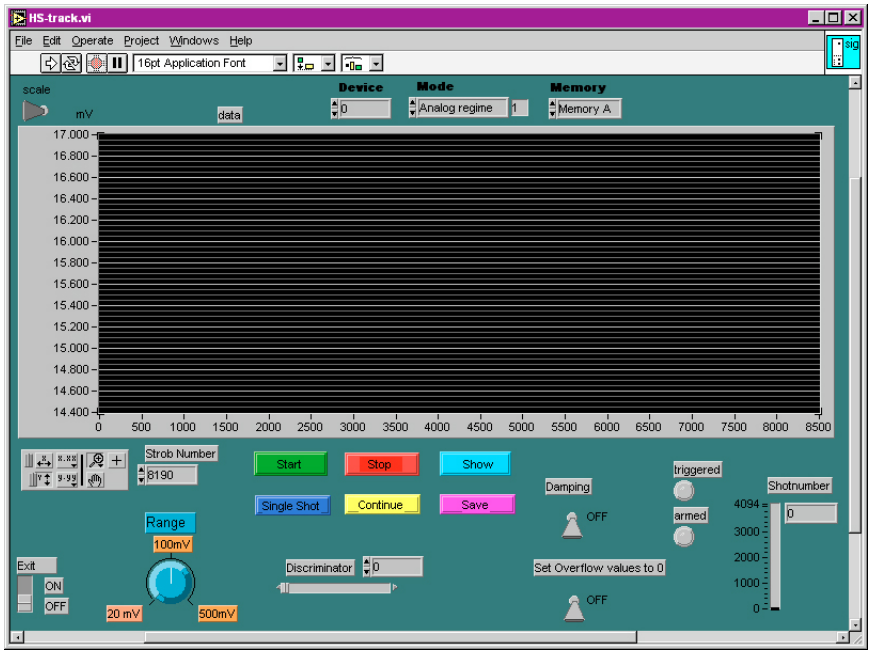

Figura 3.5: El laboratorio virtual de LabView

a ser implantados mediante kits didácticos de construcción de maquetas. Tales experimentos se ciñen a los objetivos a desarrollar dentro de la asignatura “Automatización Industrial” del cuarto curso de Ingeniería en Informática.

3.3.1. LabView

LabView es “un entorno gráfico para la adquisición de señales, análisis de medidas y presentación de datos.”, que incluye un lenguaje de programación propio y diseñado de forma que sea fácil la conexión del programa a multitud de dispositivos.

LabView es una de las herramientas de referencia en el mundo de la instrumentación por las capacidades que presenta y las herramientas que ofrece. Su lenguaje de programación propietario ha creado una gran comunidad de interés alrededor del mismo y ha construido módulos y bibliotecas de código para las funciones más diversas.

En suma, LabView se caracteriza por los siguientes aspectos.

- Precio (aunque existe una version limitada para estudiantes por 79$).

- Muy cómoda de usar. Los componentes se programan de forma visual y el programa convierte los diseños a programas escritos en un lenguaje interno.
- Genera programas ejecutables a partir de los diseños internos.
- Permite simular toda clase de sistemas.

3.3.2. Simulink

Es una extensión a MATLAB. MATLAB es un programa orientado al mundo matemático con grandes capacidades para la programación de calculos matriciales de forma sencilla. SimuLink es, según sus fabricantes “un simulador de sistemas dinámicos multidominio con entorno gráfico.”. También hay extensiones a SimuLink especializadas en el modelado de sistemas de otros dominios específicos (como SimMechanics o SimPowerSystems).

Esta herramienta se programa de forma interactiva mediante bloques gráficos y por ser una extensión a MATLAB permite también el acceso a todas las funciones matemáticas ofrecidas por el mismo.

Otra ventaja añadida reside en el hecho de la generación de código en C manejable por compiladores de ANSI C con lo que el modelo de tales sistemas puede trasladarse de manera cómoda a programas de gestion de control de procesos.

Al igual que LabView, también permite la simulación de toda clase de sistemas.

3.3.3. Simplorer

Según sus diseñadores Simplorer es “un software de simulación de sistemas multi-dominio”. Su filosofía de funcionamiento es muy parecida a la del resto de aplicaciones de simulación. Proporciona un laboratorio de diseño de sistemas orientado a la electromecánica e incluye VHDL como parte de su motor de simulación.

Otra característica interesante de Simplorer es el hecho de incluir un simulador específico de máquinas de estado. Tal simulador está dirigido por los eventos y está especialmente orientado

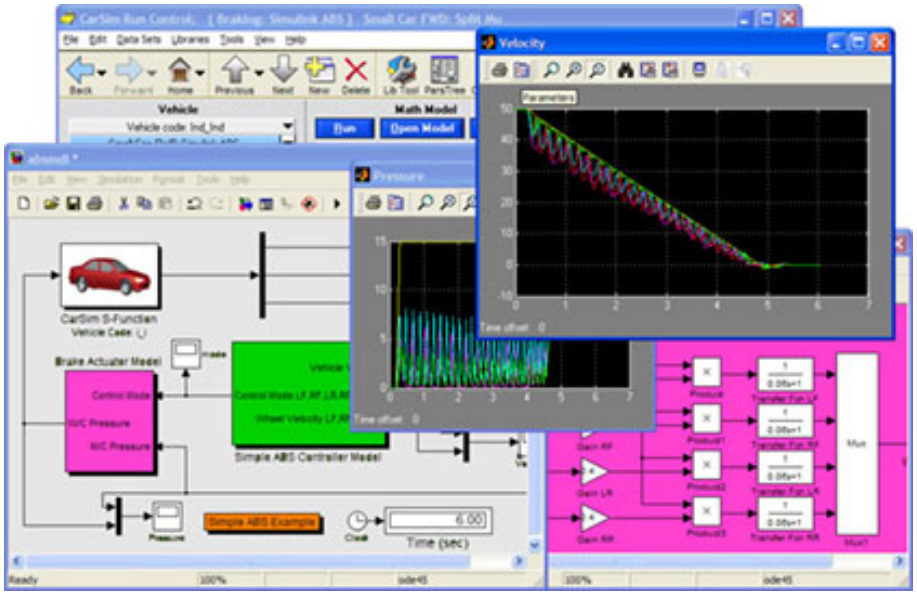

Figura 3.6: Simulink

a facilitar la simulación de procesos discretos por lo que muchas de las características de este programa supondrán un modelo de las características deseables en SimProc.

Sus características se pueden resumir en:

- También es una herramienta cara, aunque de nuevo existe una versión limitada para uso académico.
- Incluye herramientas específicas para procesos discretos.
- Incorpora herramientas para manejar el paralelismo en la ejecución de procesos, tarea ardua de realizar y depurar.
- El simulador de máquinas VHDL se considera uno de los más avanzados y útiles.

3.3.4. GTools

Comentario:Analizar la herramienta de la universidad de Vigo: Ya está analizada, es MUY BUENA.Ninguna

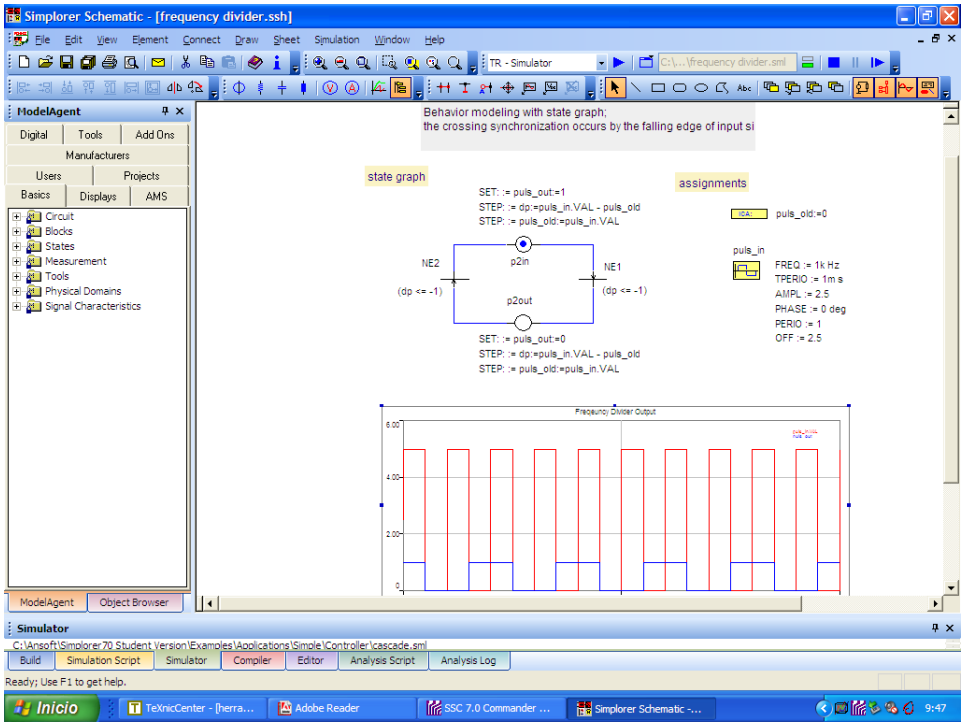

Figura 3.7: Simplorer simulando un divisor de frecuencias

herramienta usa XML ni lenguajes de descripción de procesos.

3.3.5. Kits de Fischer-Technik

Comentario:¿Pongo este apartado?

3.3.6. Resumen

LabView, Simulink y Simplorer son herramientas de muy alto nivel, con capacidades de interconexión a dispositivos, con grandes posibilidades para el análisis matemático de los sistemas, y con interfaces de usuario potentes que permiten la simulación de casi cualquier sistema imaginable. Por otra parte, estas herramientas son de código cerrado y de complejidad alta o muy alta lo que eleva la curva de aprendizaje mucho más de lo deseado para personas que se inician en el mundo de la automatización. Otra característica a destacar es su estrecho vínculo

a plataformas Windows, así como la gran voracidad de recursos que muestran. Por último se destacará la opacidad de sus formatos de almacenamiento interno, que impiden de facto, la portabilidad de sistemas desarrollados en una herramienta hacia cualquiera de las otras, ignorando la creciente estandarización en los formatos de almacenamiento basados en vocabularios XML.

Una nota a destacar en todas las aplicaciones es que no se permite la simulación interactiva, es decir, una vez construidos los elementos, el motor de ejecución de la aplicación muestra el comportamiento de los elementos sin permitir que el usuario interactúe con ellos. Así, por ejemplo, dado un elemento como un final de carrera, dicho elemento solo puede ser activado por alguno de los otros elementos del diseño, y nunca por el usuario, con lo que se impide, por ejemplo, simular un error.

3.4. Los orígenes de XML

3.4.1. Los formatos de almacenamiento clásicos

En los primeros sistemas informáticos, las capacidades de almacenamiento eran muy limitadas, por lo que ni siquiera se concebía la idea de almacenar nada que no fuera lo que conocemos como texto plano. Además, las limitadas capacidades de cómputo, así como el escaso espacio tanto en discos como en memorias, hacía totalmente inviable el uso de nada que no fueran caracteres. A medida que los microprocesadores mejoraban y aumentaba el espacio disponible en disco como en memoria, empezaron a aparecer programas que no utilizaban el texto como su formato de almacenamiento sino datos binarios, en lugar de los archivos de texto tan populares hasta ese momento. Por ejemplo podemos citar:

- Los programas de tratamiento de texto que empezaron a incorporar la posibilidad de incorporar imágenes.
- Los programas de proceso gráfico que empezaron a hacerse visibles al mercado doméstico.

- Entornos de desarrollo integrado que permitían acercar el desarrollo de software al gran público.

Comentario:¿Son los espaciados de la clase book?: No, están alterados para adecuarse a la norma.

El uso de archivos binarios supone que el almacenamiento de información ya no se hace en texto ASCII sino en forma de una secuencia de bits comprensible solamente para el programa que lo creó. El uso de archivos binarios permitió que dichos archivos ocuparan menos espacio (por ejemplo, incluyendo un archivo de imagen tal cual, en lugar mediante codificación Base64) y permitía que los programas trabajaran más deprisa, ya que en general, las codificaciones binarias son más eficientes.

Se puede decir a grandes rasgos que todos estos programas empezaron lo que supuso una revolución para dos sectores muy claramente diferenciados:

- Por un lado, los usuarios vieron como las posibilidades se multiplicaban y se empezó a demandar más y más. Véase el caso de los procesadores de texto que permiten no ya la inclusión de gráficos, sino también de audio e incluso de vídeo.
- Por otro lado, a la misma velocidad que las posibilidades del usuario crecían, crecían los problemas para desarrolladores de aplicaciones que intentaban que los formatos de almacenamiento de sus programas fueran comprensibles para otros programas, o incluso entre distintas versiones del mismo programa.

3.4.2. Los primeros lenguajes de marcas

A pesar de que los formatos de almacenamiento binarios gozaban de ventajas tales como su reducido tamaño o la eficiencia de su procesamiento, rápidamente se observó que los ficheros de texto poseían otra cualidad muy interesante: eran legibles por parte de todos los programas con relativa facilidad e incluso, en ciertos casos, comprensible por las personas.

Esta cualidad fue la que llevó al desarrollo de un mecanismo que permitiera almacenar la información en forma de texto, pero a la vez permitiendo la existencia de datos binarios. Este

mecanismo de almacenamiento quedó plasmado en SGML, un lenguaje que definía un formato de almacenamiento basado fundamentalmente en texto, pero que incluía la definición de marcas que daban "información sobre la información" –o metainformación–.

SGML significa Standard Generalized Markup Language (Lenguaje de Marcas Generalizado Standard) y fue un serio intento para definir un formato universal para el marcado de información. SGML adquirió mucha popularidad entre los desarrolladores de sistemas de gestión documentales aunque no fue utilizado por los fabricantes de software debido a dos grandes problemas:

1. Era un lenguaje muy potente, pero como tal, también suponía una gran complejidad y suponía abandonar todo el conocimiento adquirido hasta entonces en cuanto al almacenamiento y empezar a pensar desde una nueva perspectiva.

2. Debido a dicha complejidad, el soporte de herramientas para su uso era muy escaso y en ocasiones limitado por lo que en el momento de su aparición SGML prácticamente generaba más problemas de los que resolvía.

Con el paso del tiempo los formatos binarios ganaron más y más popularidad hasta la aparición de la WWW. Los documentos almacenados en páginas Web fueron concebidos para ser "interpretados" por un programa específico denominado "navegador" o "browser". Estos documentos utilizaban HTML, una versión muy simplificada de SGML, y que estaba orientado al formato casi en exclusiva. El objetivo era incluir dentro de los documentos HTML marcas tales como el tamaño o el color del texto que el navegador tenía que mostrar.

El uso de un conjunto de marcas más reducido dio lugar a varias consecuencias interesantes para los desarrolladores:

- Era relativamente fácil crear un programa para la creación de páginas Web, ya que el conjunto de marcas era pequeño y bien conocido.

- Las páginas Web podían ser modificadas por cualquier programa, ya que el análisis del texto de las marcas es sencillo.

- La simplificación del lenguaje de marcas facilitó la creación de navegadores.
- Un creador de documentos puede confiar en que su documento se verá de igual forma en todos los programas de navegación sin necesidad de código adicional.

Hoy día, muchos procesadores de texto permiten almacenar los resultados en ficheros de tipo HTML, lo que muestra la amplia difusión de dicho lenguaje de marcas.

3.4.3. La aparición de XML

El nuevo lenguaje de marcas se popularizó muy deprisa debido a su facilidad de uso, así como al auge de las herramientas que lo utilizaban. Desde simples editores de texto freeware, hasta paquetes profesionales de diseño, todo el mercado de herramientas de procesamiento de texto adoptó HTML como opción a la hora de almacenar documentos.

Sin embargo, a lo largo de la discusión expuesta en este capítulo se ha hecho notable que toda la argumentación sobre los lenguajes de marcas ha terminado centrada en las aplicaciones de procesamiento de textos. No es casualidad, ya que HTML es un lenguaje que define marcas orientadas a la presentación. Para aclarar más esto se propone el ejemplo del listado 3.1, que muestra un supuesto de pedido a un almacén de repuestos automovilísticos.

Listado 3.1: Un ejemplo de fichero con HTML

```
1  <html>
2      <head>
3          <title>
4              Almacen
5          </title>
6      </head>
7      <body>
8          <h1>
9              Pedido de repuestos
10         </h1>
11         <ol>
```

```
12              <li>
13                  100 neumaticos R-35
14              </li>
15              <li>
16                  50 filtros de aire AF-91
17              </li>
18          </ol>
19      </body>
20  </html>
```

Si abrimos este fichero en un navegador cualquiera observamos lo siguiente:

Se puede abrir este ejemplo con cualquier otro navegador y se obtendría el mismo resultado (o al menos uno muy parecido). Sin embargo no hay nada que indique que esto es un pedido, ni los elementos que se pide reponer, ni de qué tipo son, ni nada por el estilo, y es que se debe recalcar que HTML es un lenguaje orientado a la presentación.

Los elementos de marcado como solamente indican que una cadena de caracteres es un ítem de una lista pero ningún programa que analizara el HTML de la Ilustración 1 podría obtener ninguna información.

Existen mecanismos que permiten añadir información sobre como se deben presentar las marcas en los navegadores pero en ningún caso hay ninguna forma de separar el contenido de la presentación, que es en suma el problema principal de HTML.

Así, XML surge como solución a este problema al proponer un enfoque diferente a los propuestos por SGML y HTML.

- XML es notablemente más sencillo que SGML por lo que el desarrollo de herramientas que lo usan se simplifica en gran medida.
- XML no está enfocado a la presentación, sino a contener información sobre la información marcada. Es decir, a servir de metainformación. La presentación de la información marcada mediante XML se hará mediante distintos lenguajes orientados a la presentación de archivos XML.

Este último punto es el que hace más atractivo a XML como formato de intercambio universal. XML permite marcar un documento con información que describa el contenido del archivo, sin indicar como se mostrará y dejando la presentación para otros lenguajes. Esta separación de lenguajes es la que permite separar dos intereses encontrados como son la presentación y el marcado del contenido.

Listado 3.2: Un ejemplo de fichero con XML

```
1  <pedido>
2      <suministrador>Almacen</suministrador>
3      <necesidades>
4          <elemento>
5              <nombre>Neumatico</nombre>
6              <cantidad>50</cantidad>
7          </elemento>
8          <elemento>
9              <nombre>Filtro de aire</nombre>
10             <cantidad>100</cantidad>
11         </elemento>
12     </necesidades>
```

```
13 </pedido>
```

En la el listado 3.2 se muestra un ejemplo de cómo se puede realizar el mismo documento anterior en XML aportando una cantidad de información mayor tanto al programa que necesite interpretar dicho fichero como a una persona que lo leyera.

Cabe destacar que XML no es un lenguaje en sí mismo, sino un estándar que permite la definición de lenguajes propios mediante los criterios definidos en la especificación de XML. Dicha especificación es un documento público y libremente accesible, desarrollado por el World Wide Web Consortium (o W3C) por lo que cualquier desarrollador o usuario puede implementar herramientas o documentos que sigan dicho estándar.

Existen multitud de tecnologías asociadas a XML encaminadas a definir estándares o a resolver problemas. Se citan a continuación las que están mas avanzadas:

- XML 1.0 y 1.1 son las recomendaciones del W3C para construir documentos con marcas creadas por el usuario.
- Las Definiciones de Tipo de Documento o DTD y los XML Schemas dictan las reglas a seguir para escribir las gramáticas a las cuales se deben ceñir los documentos XML. Se discutirán las diferencias entre ellos más adelante.
- Aparte de reglas para escribir documentos, el W3C publica normas para la construcción de bibliotecas que vayan a leer o escribir documentos XML. Estas normas indican los nombres de las clases, los métodos, los interfaces y el comportamiento que deben tener las herramientas. Se hace una discusión más detallada en la página 2 ("Implementaciones de APIs para XML")
- XPath documenta las reglas a seguir para referirse a partes concretas de un documento.
- CSS es un lenguaje heredado de HTML que aplicado a XML permite escribir normas para que los documentos sean visibles en navegadores.
- XSLT se usa para transformar un documento XML en otro documento XML que podría ceñirse a una gramática DTD distinta.

- XSL:FO está enfocada a convertir documentos XML en documentos imprimibles.
- En XML hay una característica muy popular en los lenguajes de programación denominada “espacios de nombres” y que permiten que marcas de uso común puedan repetirse en distintos documentos indicando que pertenecen a espacios distintos.

3.4.4. Definiciones de tipo de documento

Como ya se ha explicado XML no es ningún lenguaje sino un conjunto de especificaciones que permiten construir lenguajes nuevos, también llamados *vocabularios XML*: Para poder construir estos vocabularios es necesario que existan mecanismos para determinar sus reglas de escritura y sintaxis y las definiciones de tipo de documento, o DTDs, son una de las posibilidades:

Las DTDs son gramáticas escritas en notación BNF: La notación BNF consiste en un conjunto de reglas para un lenguaje donde en cada una de ellas se indica como un elemento se puede transformar en otra posible secuencia de elementos: A modo de ejemplo, veamos el listado 3.3

Listado 3.3: Una definición DTD

```
1 <!ELEMENT lista_de_personas (persona*)>
2 <!ELEMENT persona (nombre, fechanacimiento?,
3     genero?, numeroseguridadsocial?)>
4 <!ELEMENT nombre (#PCDATA) >
5 <!ELEMENT fechanacimiento (#PCDATA) >
6 <!ELEMENT genero (#PCDATA) >
7 <!ELEMENT numeroseguridadsocial (#PCDATA)>
```

El primer elemento, denominado *elemento raíz* indica que una lista de personas consiste de de 0 o más personas (en la sintaxis el asterisco es el cuantificador “0 o más”). Una persona, a su vez es una secuencia de un nombre, 0 ó 1 fechas de nacimiento, 0 ó 1 géneros y 0 ó 1 números de seguridad social. Finalmente, un nombre es un conjunto cualquiera de caracteres.

Utilizando estas reglas se puede determinar si un lenguaje XML se ajusta a esta norma o no, por medio de herramientas automáticas de verificación de sintaxis. En general, se puede decir

que un documento XML está *bien formado* si respeta las reglas de la escritura de ficheros XML (no anidar mal los elementos, cerrar las etiquetas siempre, etc...), pero el uso de una gramática permite dictaminar si un fichero XML es *válido*, lo que supone un nivel adicional de garantía la hora de procesar un documento.

Las DTD son muy sencillas de utilizar al basarse en las gramáticas BNF, muy utilizadas en el ámbito de los lenguajes informáticos, pero adolecen de muchos defectos [Hun05a].

Las DTD no son ampliables: Si se usa una DTD, todas las reglas deben estar en esa DTD, por lo que se complica la ampliación desde el exterior.

Solo se puede usar una DTD por documento: Si en el ejemplo anterior se deseara utilizar una DTD reglamentada por la Seguridad Social que dictaminara las normas sobre números válidos sería necesario *copiar y pegar* tales normas, con lo que se corre peligro en caso de cambios futuros, al tener que copiar y pegar de nuevo las reglas nuevas.

Puede haber problemas si se necesitan espacios de nombres: Si se necesita usar varias veces un elemento cuyo nombre sea `código`, el espacio de nombres usado se tiene que explicitar *todas y cada una de las veces*.

Escaso controls sobre los tipos de datos: No se puede indicar si un elemento contiene números enteros, en coma flotante o fecha. Todos los elementos se derivan del tipo de datos *cadena*.

No hay herencia: No se pueden heredar reglas en definiciones más especializadas. La única posibilidad es cortar y pegar.

Una DTD interna puede sustituir a una DTD externa: Con lo que en realidad, una DTD podría no servir de nada si un fichero XML incluye su DTD propia, con lo que, irónicamente, una DTD no puede garantizar *nunca* que vaya a ser cumplida.

No tiene soporte DOM: No se puede manipular desde un programa externo, pues no hay interfaces para su programación.

No usa la sintaxis XML: Una cuestión más deontológica para los puristas de XML pues *las reglas gramáticales para todo XML no están en XML*.

3.4.5. XML Schemas

La definición oficial de un XML Schema es "documento cuyo propósito es definir y describir una clase de XML mediante el uso de componentes para restringir y documentar el significado, uso e interrelaciones de sus partes constituyentes: tipos de datos, elementos y sus contenidos, atributos y valores" [TDb]. Los esquemas XML intentan superar las limitaciones de las definiciones de tipo de documento en la forma siguiente:

Sintaxis XML: los esquemas es escriben utilizando la sintaxis XML.

Tipos de datos: el soporte para tipos de datos es muy superior, permitiendo indicar que el valor de un elemento puede ser de tipos tan concretos como enteros no negativos, tipos fecha, tipos cadena e incluso enumeraciones de valores.

Soporte de herramientas: al basarse en la sintaxis XML, se pueden aprovechar las herramientas ya existentes para manipular esquemas mediante los mecanismos habituales de programación.

Soportan herencia: se utiliza una sintaxis orientada a objetos que permite definir propiedades que pueden ser heredadas en elementos hijo, siguiendo las típicas jerarquías de herencia en programación.

Se puede encontrar en la bibliografía una discusión mucho más profunda de las ventajas de los esquemas sobre las DTDs ([TDa], [Hun05a]).

Por sus múltiples ventajas, los esquemas XML serán las reglas utilizadas en este proyecto para definir la sintaxis de un vocabulario XML que permita describir la estructura de sistemas automatizados. A modo de ejemplo, se puede ver en el listado 3.4 un ejemplo de esquema.

Listado 3.4: Ejemplo de esquema XML

```
1 <xs:simpleType name="tiponombre">
2         <xs:restriction base="xs:string">
3             <xs:pattern value="[a-zA-Z]([a-zA-Z0-9]{0,29})"/>
4             <xs:whiteSpace value="collapse"/>
5         </xs:restriction>
```

```
6     </xs:simpleType>
7
8     <xs:complexType name="tipoelemento">
9         <xs:sequence>
10            <xs:element name="nombreelemento" type="tiponombre"/>
11        </xs:sequence>
12        <xs:attribute name="x" type="xs:integer" use="required"/>
13        <xs:attribute name="y" type="xs:integer" use="required"/>
14    </xs:complexType>
```

En el ejemplo, se puede ver que se define un tipo llamado `tiponombre`. Este tipo indica que los nombres son secuencia alfanuméricas de hasta 30 símbolos que deben empezar por una letra. En el siguiente segmento, se construye un tipo complejo a partir de tipos simples. El tipo se llama `tipoelemento` y tiene un nombre que sigue las reglas anteriores y dos atributos `x` e `y` de uso obligatorio y que adoptan valores enteros.

Schemas y XDR

En general, los esquemas XML han sido adoptados oficialmente por todas las compañías de software lo que ha producido innumerables ventajas en cuanto a la compatibilidad de las herramientas y los documentos producidos por ellas. Sin embargo, la compañía Microsoft ha comenzado a utilizar recientemente una variación de los esquemas XML denominada XML Data Reduced o XDR que se basa en la modificación de la sintaxis para algunos elementos. Esta pequeña variación supone sin embargo la incompatibilidad a muchos niveles y el impacto de este movimiento no se ha valorado aún por parte de la comunidad.

XDR también es considerado una versión “recortada” de los esquemas XML por lo que desde algunos sectores se ha apuntado que podría ser útil como acercamiento didáctico a los esquemas, que presentan una gran complejidad para el no iniciado.

3.5. Implementaciones de APIs para XML

A la hora de manipular documentos XML el W3C deseaba mantener todas las recomendaciones lo más aisladas posible de cualquier lenguaje de programación o compañía concreta por lo que decidió construir un estándar denominado DOM (Modelo de Objeto Documento).

El modelo de objeto documento describe una forma clara de manipular ficheros XML de acuerdo a una serie de reglas bien definidas y que respetan los principios de la orientación a objetos. Así, un documento pasa a ser considerado un objeto con una serie de propiedades y métodos y para el cual se definen una serie de interfaces que permiten su manipulación.

Por ejemplo, en las recomendaciones del W3C se indica que debe existir un objeto que representa un elemento nodo y que se llamará DOMNode. Además dicho objeto debe tener una serie de métodos como getChildNode() y que este método devuelve un DOMNode que será el nodo hijo del elemento XML con el que se esté tratando. Este elemento DOMNode podría implementarse en C++, Java, Visual Basic o C#, pero lo verdaderamente importante es que el W3C publica todos los interfaces y cualquier desarrollador puede construir herramientas que implemente el estándar DOM con lo que se consigue un doble objetivo:

- Por un lado, todos los desarrolladores pueden aprender fácilmente a manejar nuevas bibliotecas que manipulen ficheros XML ya que conocen los mecanismos de funcionamientos, las clases y los métodos que ofrecen.
- Los creadores de bibliotecas pueden centrarse en construir aplicaciones eficientes al disponer de un análisis y un diseño claros.

A continuación se describen con más detenimiento las principales APIs del W3C

3.5.1. El API DOM

DOM es un Interfaz

3.5.2. El API SAX

SAX es un Interfaz

3.5.3. Oracle XML-Dev Kit

El kit de Oracle

3.5.4. Xerces

Xerces es una herramienta XML

3.6. Bibliotecas para la creación de interfaces gráficas

3.6.1. GTK+

GTK+ son las iniciales de Gimp ToolKit. GTK+ es una biblioteca para la construcción de interfaces gráficas de usuario creada ex-profesor para el desarrollo de GIMP, el popular programa de diseño gráfico. Hoy en día, GTK+ ha crecido a la vez que se ha convertido en el fundamento principal de GNOME, el sistema de escritorio para GNU/Linux.

GTK+ en realidad no es una sola librería sino un conjunto de ellas, las cuales se detallan a continuación:

GTK+: Proporciona los principales controles así como la funcionalidad de alto nivel.

GDK: A grandes rasgos proporciona la conectividad con el sistema de ventana subyacente..

GLib: Contiene diversas funciones de apoyo usadas por las capas anteriores.

Pango: Gestiona el posicionamiento de los textos. No es obligatorio utilizarla aunque las características que proporciona hacen que sea muy popular.

ATK: Proporciona características que facilitan la programación de herramientas con mayor accesibilidad.

GTK+ es una biblioteca muy popular, con abundante documentación, ampliamente soportada y con un buen diseño. Sin embargo, cuando se usa sobre sistemas operativos Windows el aspecto general de las aplicaciones es excesivamente diferente. Cabe destacar que el deseo original de los desarrolladores era imitar el aspecto de aplicaciones basadas en la biblioteca Motif, cuyo aspecto es también muy diferente. Por otra parte, GTK+ está escrita en C, que carece de la potencia de C++ para programar aplicaciones basadas en objetos. Por ello, el uso de GTK+ se hace menos interesante para un proyecto de estas características[1].

3.6.2. QT

La biblioteca QT desarrollada por la compañía noruega TrollTech es probablemente el mayor competidor de GTK en el mundo del desarrollo. Su lenguaje de desarrollo nativo es C++ y su madurez y potencia suponen una alternativa a tener en cuenta en la programación de GUI's.

Por otra parte, QT siempre ha estado restringida a un tipo de licencia bastante restrictivo que ha supuesto un pequeño obstáculo para una mayor popularización. Dicha licencia no se ajustaba a lo estrictamente marcado por las directrices del software libre por lo que su expansión en este ámbito se vió frenado. Además, el soporte oficial para QT en el mundo Windows se da solo para los productos de la gama de Visual Studio, cuyo precio se hace prohibitivo para usuarios del mundo académico[2].

[1] Cabe destacar que hoy en día, con el uso de temas para GTK y mediante el port para C++, estas desventajas han dejado de existir. Sin embargo, en el momento de arranque de este proyecto, estos inconvenientes seguían vigentes.

[2] Sin embargo, al terminar este proyecto, se pueden conseguir versiones gratuitas de Visual Studio, que aunque con ciertas limitaciones, cumplen su cometido.

3.6.3. FLTK

3.6.4. WxWidgets

Capítulo 4

Desarrollo

4.1. Análisis de las necesidades del usuario

4.2. Especificación de requisitos software

Hablar de las características que tiene que tener el programa.

Hablar de qué elementos hacen falta,de las entradas y salidas que tienen que tener, de que hacen falta vectores, de que se desea crear un lenguaje de descripción nuevo, etc...

4.3. Diseño

4.3.1. Introducción

En este apartado se define el diseño del sistema en términos de paquetes, clases, interrelaciones y comportamiento. Se utilizarán diagramas UML para mostrar aspectos específicos a destacar en el diseño global del sistema como el diseño de la interfaz de usuario o el diseño estructural.

Existen algunos detalles importantes que un desarrollador debería conocer a la hora de entender, modificar o ampliar el diseño sugerido:

- La arquitectura global de la aplicación está basada en el patrón Modelo-Vista-Control (MVC). Este patrón muestra la solución más clásica y eficaz para el desarrollo de una aplicación interactiva separando claramente los componentes más elementales: Los datos que maneja la aplicación (el "Modelo"), los mecanismos que implementan la lógica de control (el "Controlador") y el código de la interfaz de usuario (la "Vista").

- El Modelo no está desarrollado mediante un lenguaje de programación clásico, sino que consiste en la definición de un vocabulario XML que declara los componentes básicos necesarios para la automatización de un proceso, como por ejemplo relés, motores e interruptores.

- El vocabulario del Modelo se especifica en un documento XML que contiene una definición escrita en XML Schema y donde se define la sintaxis que debe tener un modelo de sistema automatizado. La comprensión de la sintaxis del Schema puede resultar difícil para un desarrollador que se aproxime a ella por primera vez por lo que se recomienda leer la especificación del World Wide Web Consortium para XML Schemas que se documenta en INSERTAR——REFERENCIA—-AQUI——

- El Controlador ha sido escrito en C++ estándar de forma que existe una correspondencia estricta entre los elementos definidos en el esquema y las clases que implementan un sistema. Así para cada elemento atómico del esquema existe una clase que lo implementa.

- Mantener la correspondencia entre una implementación C++ y un documento XML que la represente es una tarea de gran complejidad. Por ello se ha utilizado Xerces, un framework escrito en C++ que implementa fielmente las recomendaciones del W3C para XML. Xerces posee implementaciones en C++ y Java, ademas de disponer de proyectos de implementación en lenguajes alternativos.En ¡¡INSERTAR REFERENCIA TUTORIAL WXWIDGETS AQUI!! se ilustra el uso de Xerces en un programa.

- La Vista se ha implementado utilizando WxWidgets, un conjunto de clases multiplataforma para la escritura de interfaces de usuario altamente portable y con un aspecto dife-

renciado para los distintos sistemas de escritorio. La utilización de WxWidgets también presenta numerosas dificultades para un no iniciado por lo que su uso se describe con mayor amplitud en INSERTAR AQUI REFERENCIA TUTORIAL WXWIDGETS!!

- La programación del análisis de ficheros XML se puede hacer desde cero o utilizando las implementaciones de las recomendaciones del W3C. Estas recomendaciones especifican dos grandes enfoques de la programación para XML, a saber: DOM y SAX. En este proyecto se ha utilizado la implementación DOM de Xerces escrita en C+++

El diseño del software se ha centrado en la consecución de una serie de objetivos:

- Corrección: se pretende que el diseño permita construir un programa que satisfaga los requisitos especificados.
- Inteligibilidad: el diseño no solamente es fácil de transformar en una implementación software sino que mediante el uso de patrones debería ser fácilmente comprensible para distintas personas que pretenden modificar o ampliar el sistema.
- Modular: se ha intentado separar en capas los distintos aspectos concernientes a la implementación de forma que las modificaciones supongan un impacto mínimo, o al menos pequeño, en los distintos módulos.
- Extensible: el uso de estándares bien conocidos así como el aislamiento del código en distintas capas debe facilitar la ampliación del programa.

4.3.2. Diseño estructural

En la figura 4.1 se muestra el diagrama de la arquitectura global de la aplicación

La arquitectura MVC permite un desacoplamiento entre los distintos objetos que componen el sistema, permitiendo minimizar el impacto de los cambios entre capas del software. Además, supone una arquitectura bien conocida, fácil de entender, con riesgos calculados y que facilita la implementación en un lenguaje dado, la portabilidad y las modificaciones.

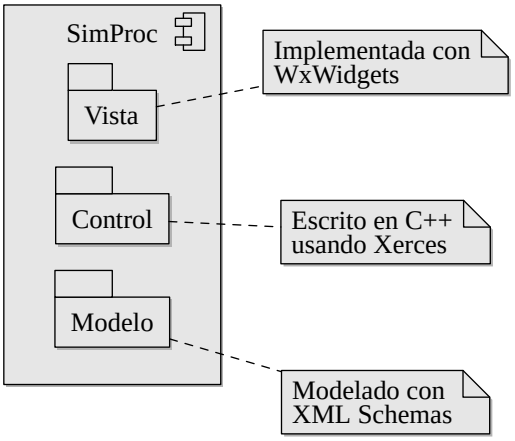

Figura 4.1: Arquitectura de la aplicación

Además, la elección de una arquitectura MVC no solo fomenta el desarrollo modular del programa sino que también expone claramente dicha modularidad, facilitando así la comprensión del programa por parte de futuros desarrolladores. La estructura de dichos módulos se explicará en subsiguientes párrafos.

El software se ha desarrollado como una aplicación de escritorio monoproceso, pensada para ejecutarse en un entorno gráfico. No se debe confundir la arquitectura MVC con las clásicas aplicaciones Web de 3 capas donde existen 3 procesos distintos que se ejecutan en tres nodos distintos a saber: El navegador Web que muestra la página HTML, el servidor que atiende las peticiones HTML y de datos y el servidor de bases de datos que almacena y entrega los datos bajo petición del servidor Web. El programa también utiliza 3 capas, una de presentación, una de lógica de control y una de almacenamiento pero su propósito principal es la modularidad, y no la ejecución separada de los distintos procesos.

A continuación se muestran los componentes principales de cada capa:

- Capa de Interfaz de usuario.

  Barra de herramientas: Este componente mostrará de forma visual el conjunto de posibles elementos a utilizar en pantalla (pulsadores, relés...).

  Ventana de registro de acciones: El programa mostrará en pantalla las acciones realizadas hasta el momento con un doble fin:

- Facilitar la depuración.
- Informar al usuario de las acciones realizadas hasta el momento.

Ventana de proyecto: En este control se mostrará de forma gráfica los elementos de automatización usados por el usuario. También mostrará la simulación del sistema que se esté automatizando.

- Capa de control.

  Parser: Trabaja con los ficheros XML en dos sentidos:

  Leyendo: El párser analiza los ficheros XML y construye dinámicamente objetos que correspondan a los descritos en el fichero.

  Escribiendo: A partir de una serie de objetos que implementan un sistema automatizado el párser construye un fichero XML que describe dicho sistema.

  Elemento: Es el componente básico que se utilizará internamente dentro del programa. Este control marca el comportamiento y propiedades básicas de todos los demás elementos concretos que vayan a formar parte de un sistema automatizado.

- Capa de modelo.

  Fichero XML Schema: Marca las reglas de sintaxis a las cuales se tienen que ceñir los ficheros de proyecto.

  Ficheros de proyecto: Describen sistemas automatizados en términos de componentes individuales y de relaciones entre ellos.

Debido a la simplicidad de su arquitectura el despliegue de la misma no debe suponer ningún problema, al haberse pensado la misma para construir un único programa ejecutable dependiente de algunas bibliotecas y que se ubicará mediante un instalador en el directorio seleccionado por el usuario. En el manual de uso se describirá más ampliamente el proceso de despliegue. Las distintas capas de esta arquitectura han presentado ciertos problemas a la hora de su desarrollo ya que algunos problemas no se conocían en las etapas iniciales, como por ejemplo la integración entre los distintos tipos de cadena de caracteres que ofrecen las distintas bibliotecas usadas al codificar.

Una de las consideraciones básicas de diseño ha sido estructurar la aplicación de forma que sea relativamente sencillo introducir nuevos elementos que permitan ampliar el comportamiento. En las secciones 4.3.3 (pág 48 y ss.) se detallan las decisiones que se han tomado para llevar a cabo dicho objetivo.

Las tareas realizadas por cada capa son las siguientes:

1. La capa de interfaz de usuario debe mantener una referencia al sistema que se está diseñando. Cada nuevo sistema añadido será un subsistema del sistema principal, de forma que la aplicación debe poder permitir trabajar de forma aislada sobre los distintos subsistemas en ventanas distintas. Asimismo permitirá la exportación de subsistemas aislados con el fin de reutilizar los diseños posteriormente.

   Se desea también que el interfaz ofrezca cuadros de diálogo que permitan modificar las propiedades por defecto de los elementos y que impida, hasta cierto punto, que el usuario tome decisiones incorrectas que le llevarían a diseños fallidos.

2. La capa de control actúa como intermediaria entre la vista y el modelo. Contiene el modelo de objetos del programa y se basa fundamentalmente en el objeto *Elemento*. Su tarea más importante reside en controlar el ciclo de programa del sistema automatizado a simular para lo cual se implementa un modelo de eventos que es regulado en esta capa.

   El código correspondiente al control se ha implementado como un fichero de biblioteca aparte con el fin de poder paralelizar el desarrollo y forzar al máximo la separación entre capas. En la sección 4.3.4 en la página 53 se detalla el diseño.

3. La capa de modelo no se basa en un lenguaje de programación sino en uno de descripción como ya se ha indicado anteriormente. Aunque es cierto que Xerces, en su implementación C++, controla dicho modelo, se destaca nuevamente que la descripción de los proyecto a automatizar se realiza enteramente en XML, lo que permite convertir a la aplicación en un “motor de simulación”.

En la figura 4.3.2 se muestra una visión global más detallada de la arquitectura.

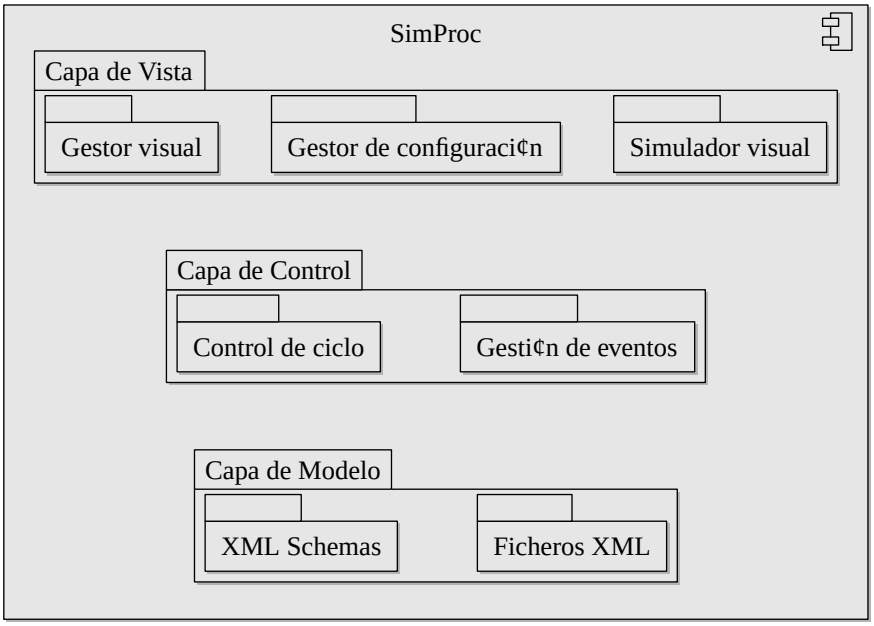

Figura 4.2: Visión detallada de la arquitectura de SimProc.

4.3.3. Diseño de la capa de modelo

Consideraciones generales

Debido a que esta capa se implementará mediante un lenguaje de descripción es importante conocer algunos puntos sobre los XML Schemas.

Tipos de datos: Los esquemas XML permiten definir elementos a partir de una serie de tipos de datos elementales y construir tipos nuevos a partir de los originales,tanto por extensión como por derivación, de forma muy similar a los lenguajes de programación tradicionales. En concreto estos tipos de datos se nombran mediante la etiqueta `<xs:simpleType name=nombre de tipo>` y las restricciones aplicadas se basarán en tipos elementales como `<string>` para los nombres de los elementos e `<integer>`[1] para las magnitudes.

[1] Se ha optado por no utilizar tipos de datos numéricos que incluyan valores decimales.

Elementos definidos: Cabe destacar que todo proceso de diseño es, en general, independiente del lenguaje de programación sobre el que se vaya a implementar dicho modelo. En el caso del diseño del modelo se han tenido en cuenta los siguientes aspectos:

Componentes elementales: Las partes básicas de un sistema automatizado. En SimProc se han utilizado diversos elementos.

- Dispositivos de entrada.
  - Pulsador.
  - Reed.
  - Final de carrera.
  - Fotosensor.
- Dispositivos de salida.
  - Zumbador.
  - Motor.
  - Lámpara.
  - Electroimán.
  - Relé.
- Dispositivos de control.
  - Temporizador.
  - Contador.
  - Biestable.
  - Comparador.
- Sistema. No es un dispositivo en sí mismo. Existe para poder permitir la composición recursiva de sistemas.

Se puede observar que uno de los elementos utilizados es un sistema en sí mismo. Dicha decisión de diseño refleja el patrón de diseño *Composite*. En los párrafos siguientes se da una breve descripción del mismo aunque este patrón ha sido descrito con más detalle en diversa bibliografía.

En la figura 4.3 se puede ver una descripción gráfica de dicho patrón. El uso de este patrón de diseño ofrece diversas ventajas como *"permitir no tener que distinguir*

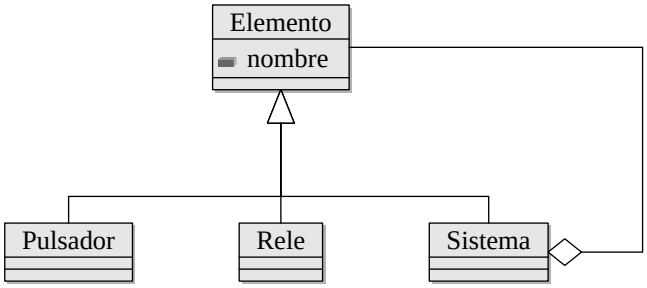

Figura 4.3: El patrón Composite

entre objetos primitivos y compuestos, simplificar el código de la clase cliente y facilitar el añadir nuevos tipos de componentes" [GHJ02].

Relaciones entre los elementos: En el modelo simplemente se indica la forma correcta de describir una relación *aunque no se impide el uso de relaciones incorrectas*. Es decir, el diseño solo va a indicar la forma correcta de construir relaciones y será la capa de control la que velará por que se impidan relaciones estructurales incorrectas, como por ejemplo conectar la salida de una lámpara a la entrada de un pulsador.

Agregación de elementos: Para facilitar la construcción y representación de sistemas se permite la posibilidad de construir agrupaciones de elementos en forma de vectores. Así, un usuario que necesite construcciones repetitivas necesitará construir los componentes una sola vez. Sobre los vectores se imponen algunas restricciones:

Los elementos tienen que ser del mismo tipo. No es posible construir un vector de elementos genéricos ni se permite polimorfismo de ningún tipo.

Las características de los elementos han de ser las mismas. No se permi te modificar por ejemplo el voltaje de entrada o de salida de solo uno de los elementos.

La posición de los elementos no puede ser arbitraria Se asume que todos los componentes están situados en una representación plana y con igual distancia unos de otros a partir de una posición de inicio dada.

Denominación de elementos: Los elementos pueden tener un nombre que puede constar de hasta 30 símbolos alfanúmericos. No se permiten signos de puntuación, ni especiales.

El número de símbolos no es especialmente importante y de hecho el diseño permite la modificación rápida y sencilla del número de símbolos.

Es importante reseñar que no es obligatorio el uso de nombres en los elementos y que la aplicación permite dejar en blanco los mismos, asignando si es necesario cadenas en blanco cuando sea necesario.

Diseño de esquemas XML

Comentario:Definición de elementos y propiedades. Me pondré con ello. Diagrama de conexión-activación entre elementos (objetos). No entiendo muy bien esto. Vista de elementos en la simulación. Aún no he dibujado los iconos, he pensado usar los símbolos en B/N de la NEMA. Al igual que ocurre con los lenguajes de programación tradicionales existen multitud de documentos dedicados a describir la sintaxis pero no todos ellos explican como utilizar las características de un lenguaje para construir programas. De la misma forma, existen referencias bibliográficas que describen en detalle las características de los esquemas XML pero solo algunas de ellas ofrecen indicaciones de valor para construir buenos esquemas ([Hun05b]). Algunas direcciones Web ofrecen también información de utilidad tanto en la sintaxis de los esquemas como para el buen uso de los mismos ([Ken]).

Para diseñar el modelo de datos se han dado los siguientes pasos:

1. Identificar las distintas magnitudes con las que el programa va a trabajar. En concreto, SimProc reconoce el uso de las siguientes magnitudes:

   - Voltajes.
   - Colores.Utilizado para definir las características de lámparas y en especial la sensibilidad a la que responderán los fotosensores.
   - Velocidad angular. Aplicada a motores, los únicos valores que tendrá corresponderán a las posibilidades más simples, a saber: Giro en sentido horario, giro en sentido antihorario y parada.

- Conexión. Tan solo indica si un elemento está activado o no.
- Magnetismo. Al igual que la conexión, solo se utilizará para generar o detectar magnitudes magnéticas, sin medir en ningún caso la intensidad de campo.
- Frecuencia. Indica la frecuencia a la cual pueden emitir sonidos algunos elementos avisadores.
- Tiempo.

El hecho de separar magnitudes cuando tal vez podrían estar representadas por tipos lógicos en torno a los valores "verdadero" y "falso" responde a la necesidad de poder evitar en algún momento que el usuario conecte elementos imposibles de enlazar en la vida real. Como se ha indicado anteriormente, aunque la capa de modelo solo defina la forma de las distintas construcciones de datos, la capa de control evitará posibles errores de construcción pero para ello necesita cierto soporte por parte del modelo de datos.

2. Definición de los elementos que se van a utilizar.

3. Identicación de características comunes a todos los elementos y que se pueden factorizar en un elemento común. Esto permitirá disponer de una cierta jerarquía de herencia ya en el modelo de datos [2].

4. Definición de los elementos en términos de entradas y salidas. La programación de su comportamiento se realizará mas adelante. En este punto solo interesa conocer el nombre de las entradas y salidas, así como su número y la magnitud que utilizan las mismas.

5. Definición de la estructura de las relaciones entre los elementos. Se ha optado por una definición recursiva sencilla que permite una sintaxis limpia y a la vez una gran potencia. En términos generales una relación se da entre un elemento conocido con el nombre de *elemento primario* que ofrece una salida y un elemento llamado *elemento secundario* a cuya entrada se conecta la salida del primario.

6. Definición de la estructura de agrupaciones de elementos. Se permitirá el uso de vectores de un solo tipo de elemento. Este vector se representará en el plano dibujando los elemen-

[2] Los esquemas XML ofrecen soporte a la herencia

tos a partir de una posición (x, y) determinada y con una diferencia $(\Delta x, \Delta y)$ entre los elementos del vector.

Pregunta para Jesús:¿Se debería mostrar tal vez en este punto un poco del esquema y un ejemplo de un fichero de proyecto o sería muy aparatoso?. Se debe mostrar todo lo que aclare conceptos. Comentario:Poner un trozo del esquema y explicarlo.

4.3.4. Diseño de la capa de control

Consideraciones generales

En la capa de control, se replica la estructura representada en la figura 4.3 (pág 50). Utilizando C++ se construye una jerarquía de objetos en C++ que refleja la obtenida en la capa del modelo aunque se dejan para más adelante los detalles sobre aspectos técnicos.

En esta capa también se establece la correspondencia entre datos almacenados en XML y objetos de programa. La transición de una capa a otra requiere operaciones muy distintas:

Transición de XML a objetos: para poder dar este paso es necesario que en la capa de control haya un párser que pueda leer sistemas en XML, obtener los elementos que hay descritos, sus propiedades y las relaciones entre elementos y a partir de esta información construir objetos que se puedan manipular. El párser a utilizar para este proceso será el ofrecido por la biblioteca Xerces.

Antes de poder proceder a construir objetos, se comprobará que la estructura del fichero es sintácticamente correcta, para lo cual se comprobará el sistema XML con la gramática establecida en el diseño de la capa de datos. Una vez que se determina que un fichero es correcto, se leerá el fichero nodo a nodo y se irán construyendo los distintos objetos.

Transición de objetos a XML: Consiste en el volcado de sistemas a ficheros XML. Este volcado es conceptualmente muy sencillo, dado que el diseño de la aplicación va a considerar

todo sistema como *un conjunto*[3]. Es razonable adoptar esta decisión dado que ningún sistema automatizado puede forzar el orden entre los elementos ni siquiera en términos de ejecución ya que el orden de actividades puede variar e incluso ser concurrente.

El procesamiento de XML

Tanto la lectura como la creación de XML son tareas de las cuales se pueden extraer conclusiones importantes a la hora de diseñar programas que las realizen. Una de tales conclusiones es el claro paralelismo que se establece entre la estructura de XML y la estructura del diseño propuesto en las capas de datos y de control. El uso del patrón Composite permite abstraer la creación de XML del tipo de elemento con el cual se esté trabajando. Así, dado un elemento cualquiera, se deberá poder invocar a un método llamado `getNodoXML()` que devuelve una representación textual. La programación de dicho método será distinto en el caso de elementos contenedores de otros elementos, en cuyo caso la llamada a este método consistirá en la llamada al mismo método *de todos y cada uno de los elementos contenidos*, de forma tal que la representación de un contenedor será la concatenación de las representaciones textuales de sus elemenos.

Aparte de esto, gran parte de las tareas de procesamiento será muy similares en los distintos elementos aunque no iguales, con lo que se impide la herencia. Para poder solventar esta situación se pueden usar diseños basados en el patrón *Template Method* [GHJ02].

El patrón *Template Method*, descrito en la figura 4.4 se implementa en SimProc mediante:

1. Un método común a todos los elementos llamado `getNodoXML()` y que todo elemento representable debe implementar.
2. Un método llamado `obtenerDatosComunes()` que permite factorizar el código común a todos los elementos, tal como el volcado del nombre del elemento o su posición.
3. Una clase llamada `XMLSerializable` de la cual puede heredar todo objeto que desee volcar su estado a XML.

[3]En el sentido matemático del término, es decir, sin relaciones de orden entre los elementos que pertenecen a él.

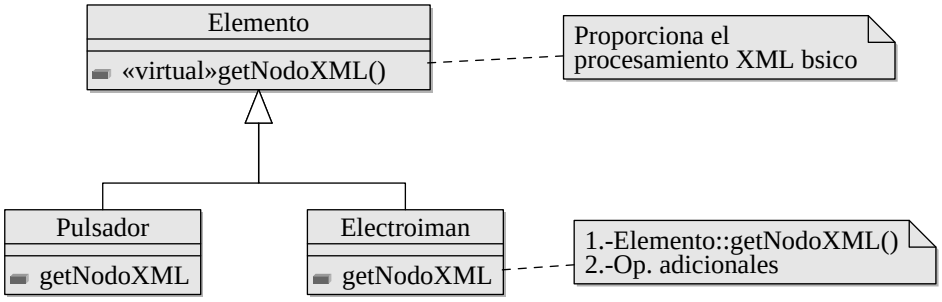

Figura 4.4: Método plantilla

4. Métodos de `XMLSerializable` que proporcionan operaciones primitivas que los distintos objetos concretos pueden invocar para crear su propia representación XML.

De esta forma, las distintas responsabilidades quedan repartidas entre las distintas clases y se permite una mayor reutilización de código con las consiguientes ventajas añadidas: reducción de tiempos de desarrollo, de depuración y mejores factores de reutilización.

Como alternativa, se podría haber incluido el código de procesamiento de XML en las mismas clases concretas, manteniendo en una misma clase todas las partes de programa involucradas. Sin embargo, se habrían violado principios básicos de diseño al mezclar código no relacionado en una misma clase. Esta separación proporciona una separación de intereses bastante beneficiosa y evita el problema de “la clase Blob” [BMM98].

El ciclo de ejecución

Uno de los objetivos de SimProc es poder simular el comportamiento de sistemas. Sin embargo los modelos de ejecución de los autómatas son muy diferentes de los de los sistemas programables basados en microprocesador [Pie93]. Para comprender mejor el diseño realizado es interesante comparar ambos modelos, incluso aunque la aplicación no simule sistemas basados en autómatas.

Si se desea diseñar una aplicación se puede optar por dos grandes tipos de diseño:

Flujo de ejecución secuencial: El flujo secuencial simplifica el diseño de programas. Este flujo de ejecución se utiliza cuando la secuencia de pasos es clara y los flujos de eventos se pueden modelar en torno a una o unas pocas secuencias.

Flujo de ejecución con paralelismo: El flujo paralelo permite la ejecución concurrente de varios flujos de programa que pueden variar con el tiempo o incluso con distintas ejecuciones permitiendo incluso la creación y destrucción de procesos en tiempo de ejecución. En este caso el tratamiento de eventos es más real aunque el diseño de programas se complica.

En el caso de los sistemas gobernados por autómatas programables el modelo de ejecución es ligeramente distinto y se puede resumir en los siguientes pasos:

1. Se graba una copia del estado de las variables de entrada.
2. Se ejecuta el programa almacenado en memoria utilizando como entradas las almacenadas en la copia de memoria. Es decir, *no se tienen en cuenta los cambios en las entradas durante la ejecución.*
3. Se hace una copia en las variables de la salida de los resultados que el autómata ha calculado para su estado.
4. Se vuelve al paso inicial.

Es evidente, que si se desea que el autómata responda a cambios rápidos en las variables de entrada, este ciclo de programa debe realizarse con mucha rapidez, si se desea evitar que el PCL vaya “por detrás” de las entradas.

Si examinamos los modelos de ejecución en microprocesadores se observa que en el primero de los casos el diseño del programa hace que la simulación de sistemas se implemente por medio de un bucle que recorre continuamente todos los elementos para comprobar su estado. En caso de cambio de estado se toma nota de la variable modificada y la simulación continúa el bucle anotando los cambios de estado y modificando estos cambios de estado en los elementos cuando el flujo de ejecución les alcanza.

En el siguiente listado se muestra un diseño simplificado del algoritmo de simulación.

```
1  while (true)
2  {
3      for (int numObjeto=0;
4      numObjeto < totalObjetos;
5      numObjeto++)
6      {
7      /*  Se comprueba si el estado del objeto actual
8          se ve afectado por alguna de las variaciones
9          en los objetos anteriores */
10         estado[numObjeto] = comprobarEstado (numObjeto, estado);
11
12         /*      Se pasa el nuevo vector de estados al objeto
13             para modificar el estado del mismo
14             en caso de que fuera necesario */
15         modificarEstado (numObjeto, estado);
16     }
17 }
```

El motor de simulación

Teniendo en cuenta los datos expuestos en la sección anterior y para lograr una mayor fidelidad en la simulación el diseño del motor se basará en un gestor de eventos que permita flujos de ejecución paralelos. Esta gestión de eventos puede variar mucho de unos lenguajes a otros e incluso variar dentro del mismo lenguaje si se cambia de plataforma. Para evitar esta clase de problemas y mantener a la vez un diseño sencillo de comprender y potente a la vez se utilizará un control de eventos basado en el patrón Observer ([GHJ02]) que implemente la gestión de eventos mediante un sistema de suscripción.

El patrón Observer permitirá disponer de objetos que se comunican los eventos y a la vez permitir una simulación lo más fiel a la realidad sin complicar el diseño demasiado. En la figura 4.5 se muestra la estructura general tal y como se describe en la bibliografía.

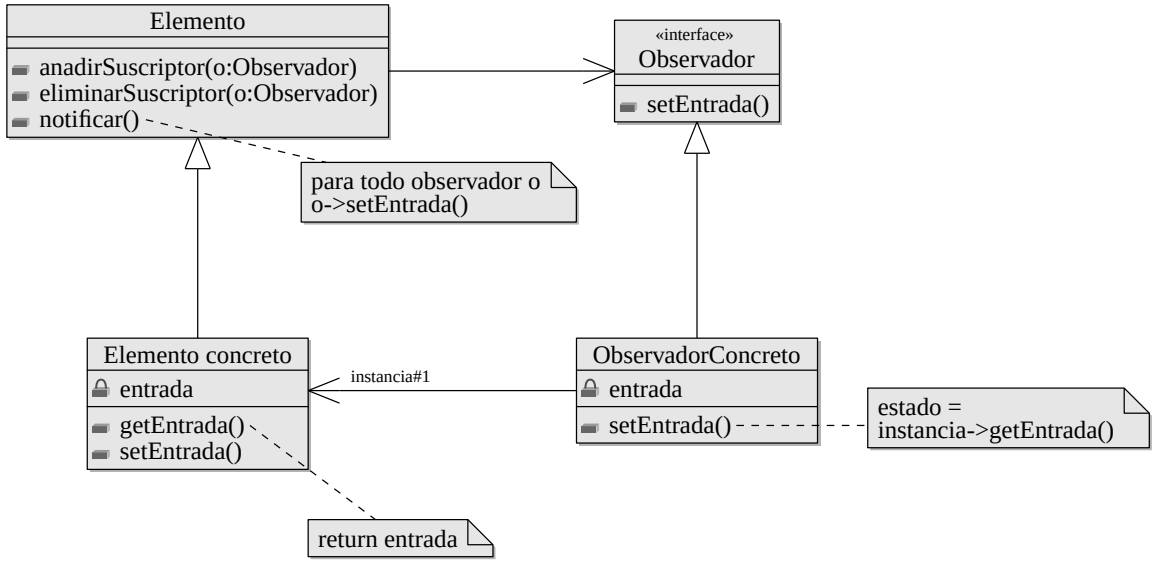

Figura 4.5: El patrón Observer

Para facilitar la comprensión del diseño se puede estudiar un caso concreto: un elemento cualquiera (p.ej. un fotosensor) debe disponer de los dos métodos de gestión de suscriptores (denominados `anadirSuscriptor` y `eliminarSuscriptor`. Cuando se quiere conectar un elemento distinto (como un electroimán) a la salida del elemento primero, se dice que el segundo elemento *se suscribe* a la salida del primero, lo que se hará llamando al método `anadirSuscriptor` del primero. Cada vez que la salida del primer elemento cambie, se modificará la entrada del segundo en base al valor de dicha salida. Si se permite que varios elementos se suscriban a uno dado, se obtiene una gestión de eventos sencilla, potente y fiel a la realidad.

La estructura general de este patrón Observer tiene en cuenta que los elementos y los observadores podrían ser clases distintas. El diseño de SimProc utilizará como observadores de los elementos a los propios elementos para así poder maximizar la recursividad de los sistemas y simplificar el diseño (y no incurrir así en el antipatrón “Poltergeists” [BMM98]).

Comentario:En algún punto tiene que haber una breve explicación de UML y referencias para ampliar información.¿En qué apartado lo pongo? ¿En apéndice o antes?

4.3.5. Diseño de la capa de vista

El diseño de interfaces de usuario es una labor de gran complejidad que solo se puede abordar mediante la descomposición. Esta capa de la aplicación tiene de la misma forma muchos detalles que considerar y que se examinarán por separados en los apartados siguientes.

Comentario:Quizá las explicaciones sean claras para ing. de software pero harían falta explicaciones más intuitivas de lo que se pretende en cada paso. (Me pondré con ello). Para los prototipos de la capa de vista se puede emplear algo que solo genere los gráficos, como Glade. Me pondré con ello.

Estructura general

El interfaz gráfico de usuario se divide en cuatro grandes componentes representados en la figura 4.6:

1. Ventana global de programa.
2. Barra de herramientas.
3. Ventana de simulación.
4. Registro de acciones.

En él se puede observar que la ventana de la aplicación es una composición de elementos visuales más pequeños, lo que permitirá dividir el desarrollo en partes más comprensibles así como separar los intereses. La ventana de simulación es la encargada de representar gráficamente la simulación de un sistema teniendo en cuenta sus cambios de estado y las variaciones en las entradas y salidas de los elementos.

Existe una pequeña parte del interfaz gráfico destinado a mostrar tanto las acciones que toma el usuario con respecto al interfaz como los eventos que ocurren a lo largo de una simulación. El objetivo de dicha ventana es ayudar a la depuración de sistemas con fallos de diseño.

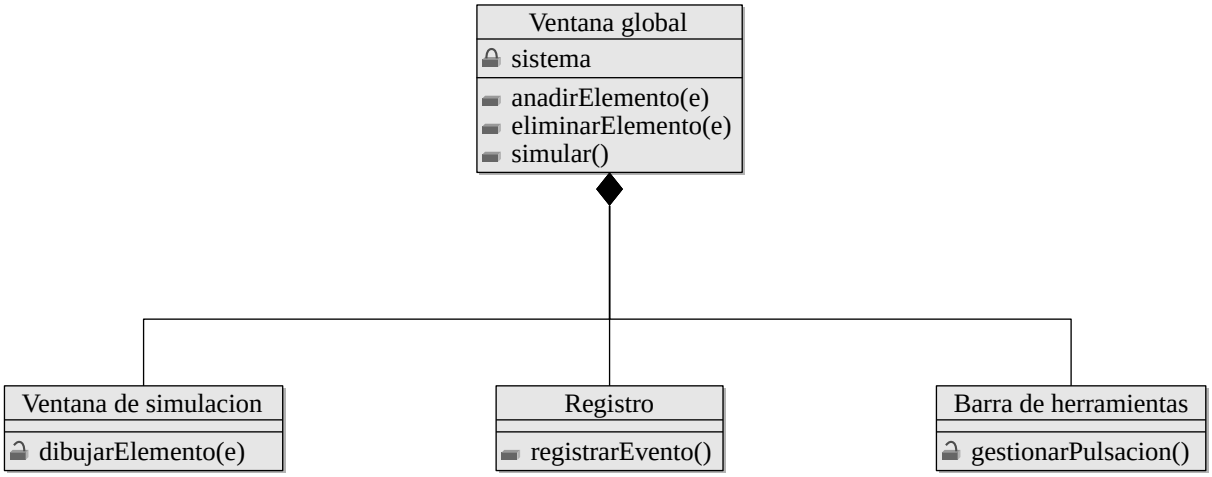

Figura 4.6: Diagrama UML de la Vista

La barra de herramientas mostrará los distintos componentes disponibles para la construcción de proyectos de automatización.

Configuración de los elementos

La aplicación debe poder permitir modificar la configuración tanto de los nuevos elementos que se añadan al sistema como modificar los elementos existentes.Para ello, existirán cuadros de diálogo que permitan realizar estas acciones. Es frecuente que se incurran en faltas de consistencia en el diseño de esta clase de ventanas, donde las tareas están muy relacionadas y sin embargo el diseño de los controles es muy diferentes, proporcionando la sensación de diferencia pese a la similitud de las tareas. Estas faltas de consistencia pueden producir confusión e incluso errores por parte del usuario([Shn98]).

En aras de ofrecer una mayor homogeneidad en el diseño de las ventanas de la aplicación y a la vez simplificar el diseño, se ha optado por construir una jerarquía de cuadros de diálogo donde el comportamiento común (y con ello el aspecto visual común) se incluya en una clase base separada y el resto de cuadros de diálogo hereden este código y esta organización visual. Esta decisión permitirá mantener la coherencia y mantener el diseño sencillo así como facilitar las tareas de implementación.

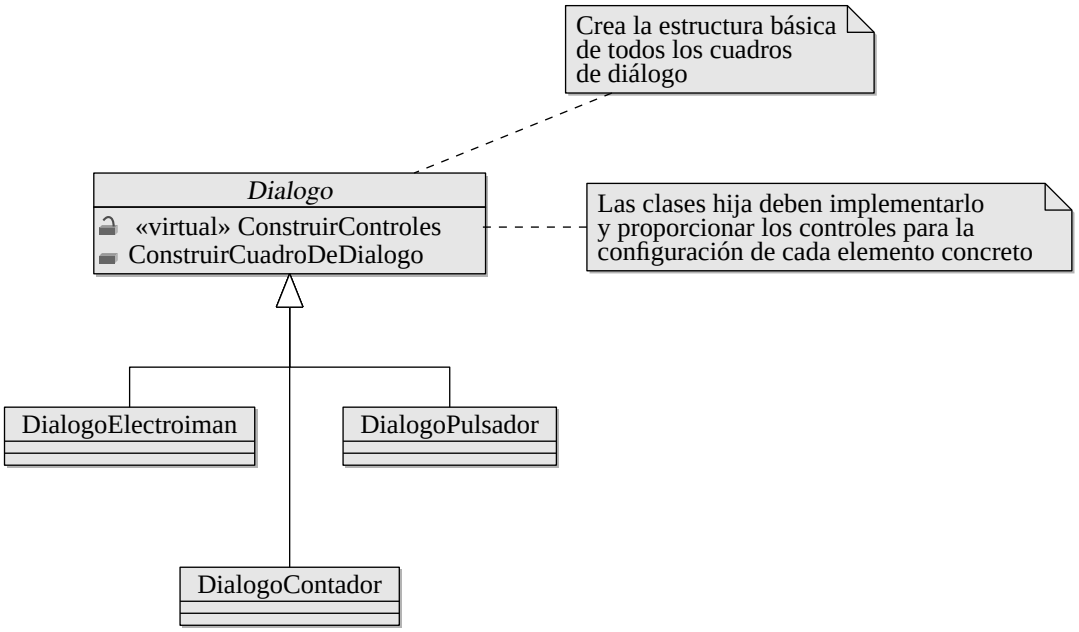

Figura 4.7: Arquitectura de los cuadros de diálogo de configuración de elementos

En la figura 4.7 se puede ver el diseño de los cuadros de diálogo que permiten configurar los elementos.

El aspecto principal a comentar en el diagrama es la existencia de un método llamado `ConstruirCuadroDeDialogo()` en la clase padre Diálogo. Este método proporciona el comportamiento y estructura visual comunes a todos los cuadros de diálogo. Por otra parte, se proporciona un método virtual llamado `ConstruirControles()` que todas las clases hija Diálogo deben implementar.

Así, la implementación de un cuadro de diálogo consistirá de dos tareas básicas:

- Por una parte se debe construir la interfaz del cuadro de diálogo hijo que sea particular para un elemento. La mayor parte de controles y comportamiento se pueden obtener a partir de una llamada a `ConstruirCuadroDeDialogo()`.
- Por otra parte se deberán añadir métodos que permitan transferir la información de los elementos concretos hacia los cuadros de diálogo y viceversa. Estos detalles se estudiarán más a fondo en las secciones dedicadas a la implementación.

Simulación gráfica

Explicar como los elementos gráficos siguen siendo Observadores de los elementos que siguen la estructura de la sección 4.3.4.

4.4. Implementación

4.4.1. Implementación del esquema

En un esquema XML se pueden definir los tipos de datos que tendrán las diferentes etiquetas de un documento XML. Para poder construir un lenguaje que permita describir un proyecto de automatización es de especial importancia definir claramente la sintaxis que tal lenguaje ha de tener de cara a evitar ambigüedades, facilitar la escritura de documentos y maximizar la coherencia de lenguaje.

El estudio de la sintaxis del lenguaje creado se realizará desde sus tres perspectivas fundamentales:

- Los tipos de datos manejados por los elementos, sus valores máximos, mínimos y las reglas de escritura.
- La forma de describir los distintos elementos que pueden pertenecer a un sistema.
- Las interrelaciones que se dan entre estos elementos para poder construir un sistema.

Tipos de datos

Los tipos de datos elementales que van a poder utilizarse dentro de una descripción de sistema serán los siguientes:

Nombres: Las reglas definidas para construir los nombres de los elementos son muy sencillas:

- Los nombres pueden constar de hasta 30 símbolos alfanuméricos.

- Los nombres deben empezar por una letra.
- Dentro del nombre están permitidas las mayúsculas y las minúsculas.
- No se permite ningún otro símbolo especial o gráfico (barras, signos de puntuación, símbolos gráficos...)

Voltajes utilizados: Muchos de los elementos utilizan voltajes a sus entradas y/o salidas. En aras de mantener la coherencia dentro de todo un proyecto y dado que los requerimientos de la aplicación en cuanto al uso de valores reales no es muy estricto se ha optado por mantener todos los valores de esta magnitud en torno a $+9V$ y $-9V$.

Colores: Las reglas para la escritura de colores son las mismas que en HTML. Los colores se escriben como una secuencia de números hexadecimales que representan respectivamente la intensidad de rojo, de verde y de azul, pudiendo ser cada una de valores comprendidos entre 0 (00 en hexadecimal) y 255 (FF).

Movimiento de motores: Solo se utiliza en casos específicos, en concreto para representar el tipo de acción que un motor desempeña. Los valores elegidos han sido.

"cw": Para representar que un motor gira en sentido de las agujas del reloj (**C**lock**W**ise).

"ccw": Para representar que un motor gira en sentido contrario a las agujas del reloj (**C**ounter**C**lock**W**ise).

"stop": Para indicar que el motor se detiene.

Conexion: Algunos elementos no devuelven salidas proporcionales a la entrada o incluso sus entradas no son continuas. Estos elementos todo-nada manejan estas situaciones mediante los valores "conectado" y "no-conectado". Mediante estos dos valores un documento XML puede representar las posibilidades a utilizar en sus entradas o salidas.

Magnetismo: La intensidad de campo magnético es una variable continua. En este proyecto sin embargo se ha optado por simplificar el comportamiento y reducir los valores a las posibilidades de que exista o no exista magnetismo (representado por los valores "on" y "off" respectivamente[4].

[4]Intencionadamente se han elegido valores distintos con respecto a las conexiones para destacar la diferencia entre magnitudes.

Frecuencias: Esta magnitud solo es usada por los zumbadores y aunque la aplicación no emite sonidos diferentes, se permite la posibilidad de describir que en un sistema existen diferentes sonidos.

Posiciones: Todos los elementos tienen una posición en un plano bidimensional representada por dos atributos x e y de valores enteros. Aunque la construcción de esquemas permite la posibilidad de restringir los valores a enteros exclusivamente positivos, lo que sería congruente con la realidad, se ha decidido permitir la posibilidad de que existan valores negativos para poder representar posiciones relativas, si esto fuera necesario.

Estructuras elementales

Una regla común marcada para todos los elementos es que todos deben tener un primer elemento <*nombreelemento*> para indicar el nombre. Las reglas para el contenido de esta etiqueta son las comentadas en la página 62.

Pregunta: ¿Como describir la estructura de la implementación?. Se me ocurren dos posibilidades, la que se muestra en el pulsador, en forma de tabla y la que se muestra en el rele, con una descripción textual.

Las reglas de escritura para los diversos elementos estudiados a lo largo de todo el proyecto son las siguientes:

Pulsadores: En la tabla siguiente se puede ver la ordenación de los elementos de un pulsador.

Relés: El relé tiene cuatro etiquetas adicionales, que *deben* ir en el orden aquí descrito y que se describen a continuación:

1. <*entradareposo*> . Esta etiqueta indica la entrada necesaria para llevar el relé a su estado de reposo. El valor se ha definido de tipo Voltaje.
2. <*salidareposo*> . Define la salida que produce el relé cuando a la entrada se recibe el valor determinado por <*entradareposo*>.

Etiqueta	Descripción	Definición de tipo
<*entradareposo*>	Entrada inicial del pulsador	Conexión
<*entradaactivacion*>	Entrada para la cual el pulsador se activa	Conexión
<*salidareposo*>	Salida cuando el pulsador está en reposo	Voltaje
<*salidaactivacion*>	Salida del pulsador cuando recibe en la entrada el valor de activación	Voltaje

Cuadro 4.1: Elementos de un pulsador

3. <*entradaactivacion*> . La entrada necesaria para llevar el relé a su estado de activación.
4. <*salidaactivacion*>. La salida que se produce cuando a la entrada se recibe el valor <*entradaactivacion*>.

Sensores reed: .

Temporizadores: .

Contadores: .

Fotosensores: .

Lámparas: .

Electroimanes: .

Zumbadores: .

Motores: .

Biestables: .

Relaciones entre estructuras

Aqui se verian los vectores, las relaciones y la composicion recursiva.

4.4.2. Implementación del control

4.4.3. Implementación de la vista

Como ya se ha comentado antes, la parte gráfica de la aplicación se ha desarrollado utilizando WxWidgets. WxWidgets es un framework para el desarrollo de interfaces gráficos de usuario desarrollado con vistas a la máxima portabilidad entre sistemas operativos. Debido a esta característica de diseño, no ha sido necesario evaluar demasiados aspectos específicos de cada plataforma aunque sí ha sido necesario estudiar la adaptación de este framework a SimProc.

Implementación visual de elementos de automatización

En secciones anteriores se ha comentado la implementación interna de la lógica de control de elementos tales como pulsadores o lámparas. Sin embargo, se necesita también considerar diversas alternativas a la hora de enlazar la vista de la aplicación con la capa de control.

Un objeto Elemento desarrollado en la capa de control tiene algunas responsabilidades adicionales cuando trabaja en la capa de vista. Resulta lógico entonces desarrollar un nuevo tipo de control que encapsule de alguna forma el comportamiento implementado en el control de los elementos. Esta encapsulación se ha denominado `ElementoVisual` en el desarrollo del interfaz gráfico pero la forma exacta de realizar la encapsulación requiere estudiar algunas posibilidades muy distintas unas de otras.

Alternativa 1: Herencia simple

En esta situación se construiría un objeto en la capa Vista que heredaría todas las propiedades del principal objeto de la capa de control, el objeto `Elemento`. Este `ElementoVisual` en la capa vista adquiriría así automáticamente todas las propiedades genéricas que existen para todos los elementos tal y como se muestra en la figura 4.8 y además añadiría las responsabilidades propias de los elementos en la parte gráfica (tales como dibujar los objetos, configurarlos de forma gráfica, etc. . .).

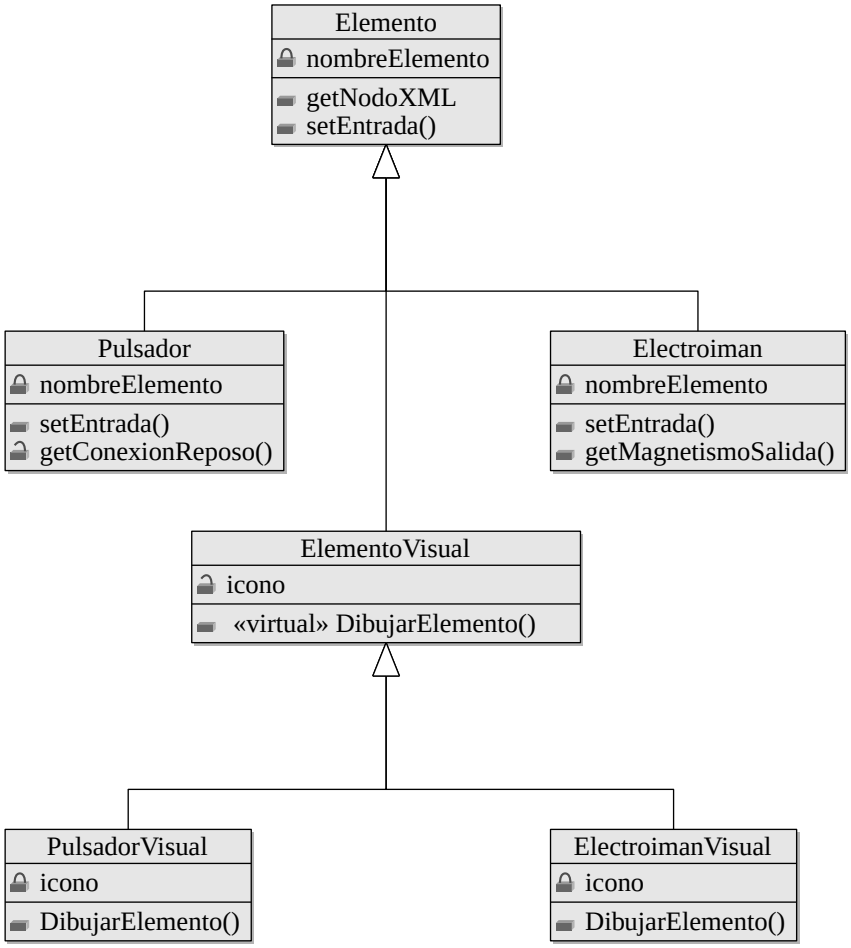

Figura 4.8: Enlace entre capas mediante herencia simple

Ventajas: Se facilitarían muchas tareas de implementación ya que todo elemento visual sería un elemento y se podría tratar a todos los elementos con homogeneidad.

Inconvenientes: Se haría necesario reimplementar la funcionalidad de la lógica de control de *todos y cada uno de los elementos* repitiendo tareas de programación de forma innecesaria.

Alternativa 2: Herencia múltiple

El objeto `ElementoVisual` a implementar proporciona las responsabilidades propias de los elementos visuales y fuerza a los elementos concretos (pulsadores gráficos, electroimanes

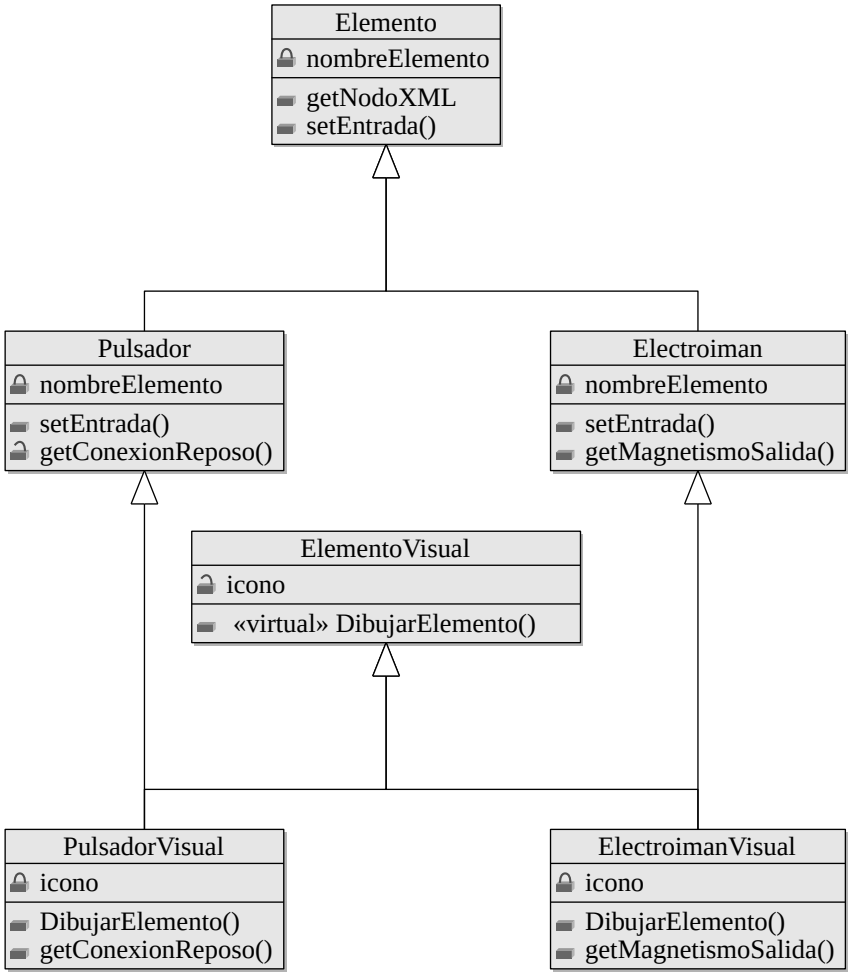

Figura 4.9: Enlace entre las capas mediante herencia múltiple

gráficos, etc...) a heredar tanto de sus respectivos componentes (pulsadores o electroimanes) como del `ElementoVisual genérico`. El resultado de esto se muestra en la figura 4.9.

Ventajas: No es necesario reimplementar *ninguna funcionalidad*. Las responsabilidades estarían bien repartidas ya que el código responsable de la parte gráfica estaría en la clase `ElementoVisual` y el código para la parte de control estaría en `Elemento.`

Inconvenientes: El tratamiento de los distintos elementos no sería homogéneo del todo aún. Además, las posibilidades de cometer errores se multiplicarían ya que sería necesario realizar continuas conversiones entre punteros en función de la capa en la que estuviéramos situados.

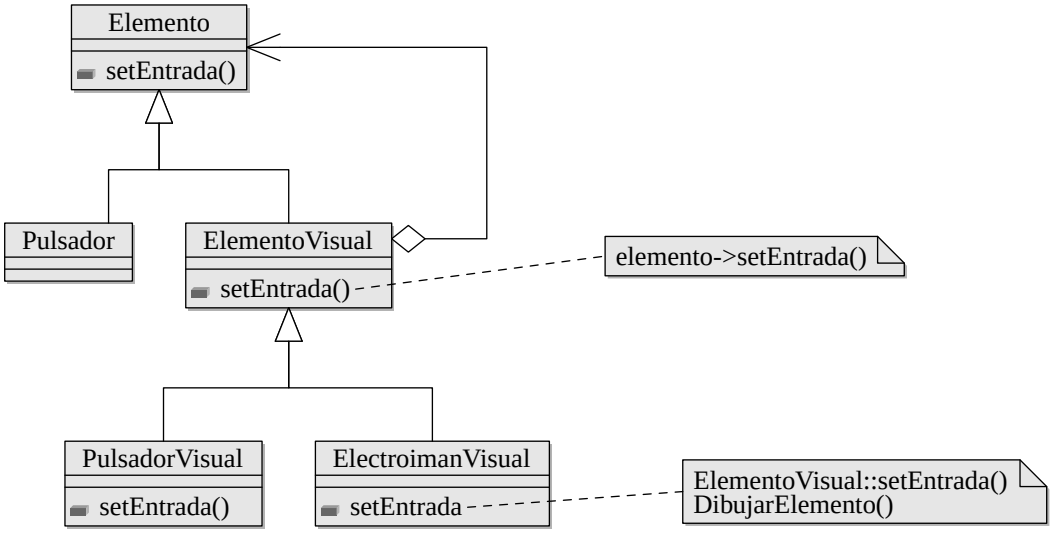

Figura 4.10: Enlace entre capas mediante un Decorator

Alternativa 3: Patrón Decorator

La estructura del enlace entre la vista y el control se podría implementar mediante un patrón Decorator tal y como se muestra en la figura 4.10. En este caso todo elemento visual es un elemento por naturaleza propia, lo que refleja el modelo del mundo real.Este esquema de implementación se aproxima bastante, aunque resulta insuficiente ya que vincula demasiado estrechamente los elementos en su nivel visual y en su nivel de capa de control.

Ventajas: La implementación es muy sencilla de realizar y el código es muy compacto.

Inconvenientes: Aunque no se muestra en la figura el esquema de implementación se complicaría al tener que incluir en todo momento copias del dispositivo de contexto donde se va a dibujar un elemento. Este dispositivo de contexto es un detalle de implementación que el patrón no tiene en cuenta.

Solución: El patrón Bridge

Aunque no se tuvo en cuenta inicialmente, parece claro que los elementos gráficos o visuales van a reflejar de forma clara la misma jerarquía de objetos diseñada en la capa de control.

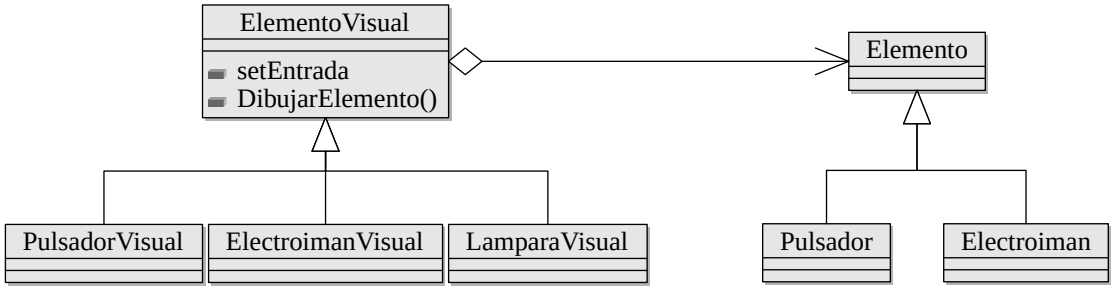

Figura 4.11: Enlace de capas mediante un patrón Bridge

Aunque un patrón Decorator envolvería los objetos de la capa de control sin ningún problema, interesa separar los aspectos relativos al código de dibujado del los relativos al código de control. El patrón Bridge proporciona esta separación de intereses en el código de forma efectiva y su estructura se muestra en la figura 4.11.

En el listado se puede ver la declaración de la clase ElementoVisual de la cual heredan todos los controles gráficos que la aplicación muestra en pantalla. Se puede observar como existe en todo momento una referencia a una implementación de algún objeto de la capa de control que es quien realmente procesa las entradas y no el interfaz visual ofrecido por estos objetos.

```
1  class ElementoVisual : public wxObject
2  {
3          public:
4                  ElementoVisual();
5                  ~ElementoVisual();
6
7                  wxBitmap GetBitmap() { return bitmap;}
8                  void             DibujarElemento (wxClientDC& dc);
9                  void             DibujarRelaciones (wxPaintDC& dc);
10
11                 virtual void     Configurar (Elemento* elemento);
12                 virtual void     Configurar();
13                 virtual Dialogo* FabricarDialogo() = 0;
14
15                 void SetXY( unsigned int x, unsigned int y);
```

```
16
17                  void SetNombre (const char *nombre);
18                  void SetNombre (wxString nombre);
19
20                  Elemento* GetElemento();
21
22                  virtual int setEntrada
23                    (Senal* entrada, int posicion=POS_ENTRADA);
24                  Senal* getSalida (int posicion=POS_SALIDA);
25          protected:
26                  Elemento*       elemento;
27  }
```

Como se puede ver, la implementación de las entradas en los elementos visuales remite a la implementación real que ofrecen los elementos concretos programados en la capa de control. La única diferencia es que existe un objeto gestor de entradas que captura todas las entradas con el fin de avisar a otros ventanas (como la ventana de depuración) de dicho cambio en la entrada.

```
1   int ElementoVisual::setEntrada (Senal* entrada, int posicion)
2   {
3           int codigo_de_error;
4           if ( gestor !=NULL )
5           {
6                   gestor->setEntrada (entrada, posicion);
7           }
8           codigo_de_error = elemento->setEntrada (entrada, posicion);
9           return error;
10  }
```

4.4.4. El parpadeo

El dibujado de elementos gráficos en pantalla puede resultar una tarea ardua a la hora de resolver ciertos problemas tales como el parpadeo. El parpadeo de los gráficos es un fenómeno que se produce cuando el sistema operativo necesita redibujar varios elementos en pantalla y no todos ellos se pueden redibujar a la vez.

Una antigua solución pasaba por la programación de rutinas de servicio de interrupción que esperaban el aviso hardware de repintado de pantalla para dibujar en ese momento de forma que no se produjeran tales parpadeos. Esta solución es totalmente innecesaria en las bibliotecas gráficas modernas e incluso en WxWidgets, donde se ofrece un objeto alternativo en el cual se pueden realizar todos los dibujos y que automáticamente esperará tales eventos para poder eliminar el parpadeo en gráficos.

Esta solución ahorra innumerables problemas a la hora de pulir el aspecto final de la aplicación de una forma extremadamente sencilla y efectiva con el simple uso de objetos instancia de la clase `wxBufferedDC` en lugar de objetos de la clase `wxPaintDC`. Existe más ayuda sobre los detalles internos de estas clases en la documentación de la biblioteca.

Por otra parte, la programación de los eventos de dibujado de la aplicación se tienen que estudiar con detenimiento. En toda clase que tenga una interfaz de ventana[5], el núcleo de la biblioteca proporcionará dos funciones de gestión de eventos muy parecidas, una llamada `OnDraw` y otra llamada `OnPaint`. La diferencia entre `OnDraw` y `OnPaint` es pequeña pero importante, ya que si reimplementamos el método `OnDraw` en una ventana en la cual haya barras de desplazamiento, WxWidgets habrá llamado automáticamente a una función llamada `DoPrepareDC` que mueve el origen de coordenadas en funcion del scroll. `OnPaint` no hace el recalculado de coordenadas así que si se reimplementa habrá que recordar llamar a `DoPrepareDC` en la primera línea de la función `OnPaint` para mover el origen de coordenadas a la posición correcta. De no hacerlo así, el redibujado se mezclará con el repintado y el resultado gráfico en pantalla no coincidirá con lo deseado.

[5]En general todas las clases que hereden de WxWindow.

4.5. Casos de estudio

4.5.1. Formato de las pruebas

4.5.2. Casos de prueba

4.5.3. Las herramientas xUnit para la prueba de software

Las pruebas de unidad son un tipo de pruebas que en los últimos tiempos han alcanzado una gran popularidad en el desarrollo de programas. Tradicionalmente el ciclo de vida de los programas ha venido caracterizándose por las fases enumeradas y secuenciadas en la

Capítulo 5

Resultados

Capítulo 6

Propuestas

Nada todavía

Apéndice A

Referencia rápida de WxWidgets

A.1. Introducción

Comentario:Se ha hecho un estilo un poco más compacto para los apéndices, como se indicaba en la revisión anterior. Se ha reducido un poco el espacio entre líneas y el espacio entre párrafos. Quizá se pueda compactar aún más, pero esperar próxima revisión. También se ha modificado el tipo de letra de los listados (antes era el mismo del texto y ahora es Courier) y se les ha añadido números de línea y referencias

Un programa en WxWidgets solo se puede lanzar desde dentro de una clase especial por lo que para empezar se debe crear una clase que herede de esta clase llamada wxApp y que por ejemplo se llamará Aplicación. Dentro de esta clase se pueden implementar todos los métodos que se deseen pero sobre todo se debe implementar el método OnInit.

Normalmente dentro de OnInit se creará la ventana principal del programa pero sobre todo es obligatorio terminar devolviendo un valor bool que indique a wxWidgets si debe comenzar o no el bucle de procesado de eventos, por lo que a no ser que se desee algo muy especial se devolverá `true` la mayor parte de las veces.

En este punto sería posible preguntarse donde está la función `main` o `WinMain` o el punto de entrada al programa. WxWidgets lo añade automáticamente, ya que gestiona el punto de

Listado A.1: Fichero aplicacion.h

```
1  // Fichero de cabecera aplicacion.h
2  #include "wx/wx.h"
3
4  class Aplicacion : public wxApp
5  {
6   public:
7     virtual bool OnInit();
8  };
9
10 DECLARE_APP (Aplicacion)
```

Listado A.2: Fichero aplicacion.cpp

```
1  //Fichero aplicacion.cpp
2
3  #include "aplicacion.h"
4  #include "ventana.h"
5
6  IMPLEMENT_APP (Aplicacion)
7
8  bool Aplicacion::OnInit()
9  {
10   Ventana *v=new Ventana();
11   v->Show(true);
12         return (true);
13 }
```

entrada a través de macros. Lo único que necesita saber la biblioteca es el nombre de la clase que tiene que asignar a un objeto interno denominado wxTheApp (que normalmente no es necesario manipular). Para indicar dicho objeto se usa la macro `DECLARE_APP (nombre de clase)`. Cabe destacar que este mecanismo de inicialización es necesario para que WxWidgets pueda garantizar la portabilidad de la aplicación.

Para que la función pueda enlazar el método OnInit al punto de entrada al programa se debe utilizar la macro `IMPLEMENT_APP (nombre de la clase)`.En el listado A.1 y A.2 se muestra un ejemplo de aplicación muy sencilla y en el que no se incluye ninguna implementación adicional a la incluida por WxWidgets.

Listado A.3: Fichero ventana.h

```
1 // Fichero ventana.h
2 #include "wx/wx.h"
3
4 class Ventana : public wxFrame
5 {
6  public:
7   Ventana();
8 };
```

Como se puede ver, lo único que hace el método `OnInit` es crear un objeto de la clase Ventana (que se comentará en seguida) y llamar a su método Show, devolviendo true al final y comenzando así el bucle de eventos.

A.2. Creación de ventanas

Por sí mismo, todo lo anterior no produce ningún resultado de utilidad o de especial interés. Una biblioteca para construcción de interfaces de usuario debe proporcionar mucho más. Para empezar, se verá como crear una simple ventana. Existen muchas clases de ventana pero la más elemental es wxFrame, que contiene el comportamiento más básico de cualquier ventana. Hay que destacar que en wxWidgets se permite un control total sobre el comportamiento de TODOS los controles por lo que la complejidad de la programación con esta biblioteca también es muy grande. Sin embargo una vez superada la curva de aprendizaje se dispone del control total sobre el más mínimo detalle sumado a las ventajas de la programación multiplataforma.

Primero se creará una ventana sin ningún elemento adicional. En el listado A.3 se muestra la declaración de la clase y en A.4 se muestra la implementación de la misma.

Como se puede ver, lo único que se hace es llamar al constructor de la clase padre. Se pueden pasar muchos parámetros (consultar la ayuda de WxWidgets para ampliar) pero con los tres primeros es suficiente. El primer parámetro pide el identificador de ventana de la ventana “padre” de esta ventana. Si se desea que ésta sea la ventana principal pasamos NULL. El segundo parámetro permite indicar el identificador de esta ventana. Usando la constante predefinida

Listado A.4: Fichero ventana.cpp

```
1 //Fichero ventana.cpp
2
3 #include "ventana.h"
4
5 Ventana::Ventana() :
6  wxFrame (NULL, wxID_ANY, "Ventana")
7 {
8 }
```

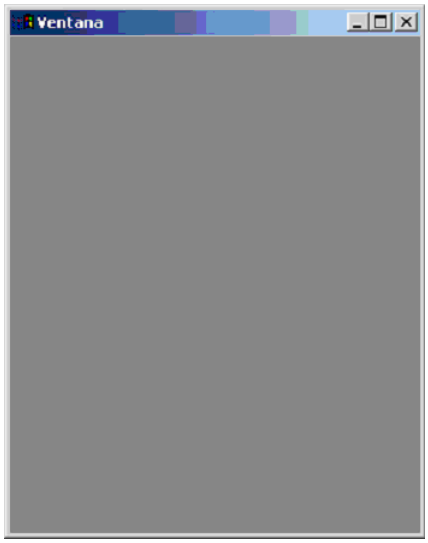

Figura A.1: Ventana mínima de WxWidgets

`wxID_ANY` se permite que la biblioteca asigne cualquiera que esté libre. El tercer parámetro es el título que debe aparecer en la ventana.

Aparte de eso, no se hace nada más. En la clase principal aplicación se creó un objeto ventana y se llamó al método `Show()` que mostrará esta ventana principal. Al ejecutar el programa se obtiene una ventana como la de la Figura A.1.

Esta ventana se puede mover, cerrar, cambiar de tamaño etc... sin haber programado nada. La clase `wxFrame` proporciona el comportamiento básico por defecto para una ventana estándar.

Listado A.5: Método gestor de eventos

```
1 void OnPulsaTecla (wxKeyEvent &evento);
```

A.3. Eventos

Una ventana vacía como la construída en la sección anterior no sirve de mucho. Una aplicación normal constará de botones, cuadros de texto, menús, etc... Y aquí entra los eventos, otro de los conceptos fundamentales en WxWidgets. En todas las bibliotecas orientadas a la programación gráfica existe el concepto de funciones que se disparan cuando se produce un evento y que se denominan funciones de retro-llamada o funciones callback. En WxWidgets se deben declarar cuales son las funciones que se disparan cuando se produzca un cierto evento y conectar esas funciones a eventos concretos. Una ventana o control que desee responder a eventos deberá incluir la macro `DECLARE_EVENT_TABLE` en su fichero de declaración (en este caso ventana.h) y macros que conecten los eventos (identificados por un número que declararemos más adelante) a dichas funciones. Esta conexión se suele hacer en el fichero de implementación (ventana.cpp) y usando las macros `BEGIN_EVENT_TABLE()` y `END_EVENT_TABLE()`.

Por ejemplo, si se desea que la ventana reaccione a la pulsación de la tecla C y que su reacción consista en cerrar la ventana es necesario usar una gestión del evento “pulsación de tecla”. Para ello, hay que crear una función callback en la clase Ventana como la del listado A.5

La clase wxKeyEvent almacena un evento de pulsación de tecla. No necesitamos manejarla ya que el bucle de eventos nos entregará dicho evento de pulsación en la aplicación. La implementación del gestor de eventos mostrada en A.6 comprueba si se ha pulsado la C

Listado A.6: Implementación de ejemplo de un gestor de eventos

```
1 void Ventana::OnPulsaTecla (wxKeyEvent &evento)
2 {
3  //Se comprueba si el ócdigo ASCII de la tecla pulsada es C
4  if (evento->m_keyCode == 67)
5  {
6   Close();
7  }
8 }
```

Y se conecta el evento de pulsación de teclado a esta función como se muestra en A.7.

Listado A.7: Tabla de eventos de una clase

```
1  BEGIN_EVENT_TABLE(Ventana, wxFrame)
2   EVT_KEY_DOWN (Ventana::OnPulsaTecla)
3  END_EVENT_TABLE()
```

Listado A.8: Fichero de declaración de una clase con eventos

```
1  #include "wx/wx.h"
2
3  class Ventana : public wxFrame
4  {
5   public:
6    Ventana();
7          void OnPulsaTecla (wxKeyEvent &evento);
8   private:
9    DECLARE_EVENT_TABLE ()
10 };
```

Por lo que el fichero ventana.h queda como se muestra en el fichero A.8 y el fichero ventana.cpp queda como se muestra en A.9.

Listado A.9: Implementación de una clase que usa eventos

```
1  #include "ventana.h"
2
3  Ventana::Ventana() :
4   wxFrame (NULL, wxID_ANY, "Ventana")
5  {
6  }
7
8
9  void Ventana::OnPulsaTecla (wxKeyEvent &evento)
10 {
11  //Se comprueba si el ócdigo ASCII de la tecla pulsada es C
12  if (evento.m_keyCode == 67)
13  {
14   this->Close();
15  }
16 }
17
18 BEGIN_EVENT_TABLE(Ventana, wxFrame)
19  EVT_KEY_DOWN (Ventana::OnPulsaTecla)
20 END_EVENT_TABLE()
```

A.4. Sizers

Los sizer son “ajustadores de tamaño”. Son elementos corrientes en la programación gráfica y sobre todo en la multiplataforma, existen en GTK, en QT, en Java y en otros toolkits menos populares como FLTK, etc... La idea de un sizer es almacenar controles gráficos ajustando el tamaño de lo que contienen. En principio podría desearse que los botones, paneles o menús tuvieran un tamaño fijo pero esto no es buena idea debido a lo siguiente:

- Las variaciones de resolución de unos sistemas a otros.
- Es interesante disponer de la posibilidad de poder reajustar automáticamente el tamaño de los controles si alguien cambia el tamaño de la ventana

Un sizer se limita a almacenar controles y calcular automáticamente el mejor tamaño, manteniendo la proporcionalidad. El algoritmo de cálculo de tamaño de un control se basa en si hay controles hijo y en el tamaño total de la ventana (ya que puede haber otros sizers conteniendo más controles) y dicho algoritmo es bastante complejo. No se necesitan saber los detalles pero se puede acudir a la documentación de WxWidgets si se desea más información. Para mayor comodidad se puede pensar simplemente que los sizer son “contenedores invisibles” y que ajustan el tamaño automáticamente.

Existen varios tipos de sizer: Contenedores que ajustan los controles horizontalmente, verticalmente, en rejilla, etc... Al utilizar los distintos sizer se debe de olvidar el concepto de que un botón mide x píxeles y otro botón mide y píxeles. Al usar contenedores como los sizer solo se necesita pensar en el “tamaño relativo”.

Se puede suponer por ejemplo una ventana en la que haya dos botones y un control de texto. Cuando se pulse uno de los botones se imprimirá un mensaje en el control de texto y al pulsar el otro se imprime un mensaje distinto. Si el control de texto tiene que ir en la mitad inferior y el par de botones la mitad superior podría pensarse en apilar los controles. Esto se puede hacer mediante un wxBoxSizer, que se limita a poner los controles uno detrás de otro en horizontal (pasando el parámetro `wxHORIZONTAL`) o poniendo uno debajo de otro (usando `wxVERTICAL`)

Listado A.10: Un ejemplo del uso de sizers.

```
1  //Fichero ventana.cpp
2
3  Ventana::Ventana() :
4   wxFrame (NULL, wxID_ANY, "Ventana")
5  {
6   contenedor = new wxBoxSizer(wxVERTICAL);
7   boton1     = new wxButton (this, wxID_BOTON1, "Mensaje 1");
8   boton2     = new wxButton (this, wxID_BOTON2, "Mensaje 2");
9   texto      = new wxTextCtrl(
10                 this, wxID_ANY, "Aqui va el texto",
11                 wxDefaultPosition, wxDefaultSize,
12                 wxTE_MULTILINE);
13
14  contenedor->Add (boton1);
15  contenedor->Add (boton2);
16  contenedor->Add (texto);
17  this->SetSizer (contenedor);
18 }
```

Para conseguir esto, a grandes rasgos solo hay que crear el contenedor, crear los tres controles, añadir los controles al contenedor y hacer que el contenedor sea el contenedor principal de la ventana. Así que en el fichero ventana.cpp la implementación del constructor queda como es muestra en el listado A.10:

Y el fichero ventana.h, mostrado en el listado A.11 incluye ahora algunas cosas:

Al compilar se obtiene algo semejante a la Figura A.2.

Se observan varias cosas

- Aunque se pulsen los botones no pasa nada.
- Los controles están demasiado pegados.
- La apariencia de los controles no es la que esperábamos.
- Hay que asignar una identificación numérica a los controles.

En cuanto al primer problema la respuesta es simple. Hay que gestionar los eventos de pulsación y conectarlos con funciones que impriman cosas en el control. El segundo y el tercer

Listado A.11: Declaración de componentes.

```
1  class Ventana: public wxFrame
2  {
3    public:
4        ...
5    private:
6                  wxBoxSizer* contenedor;
7                  wxButton*       boton1;
8                  wxButton*       boton2;
9                  wxTextCtrl* texto;
10         DECLARE_EVENT_TABLE()
11 };
12
13 enum
14 {
15    wxID_BOTON1=7000, wxID_BOTON2=7001
16
17 };
```

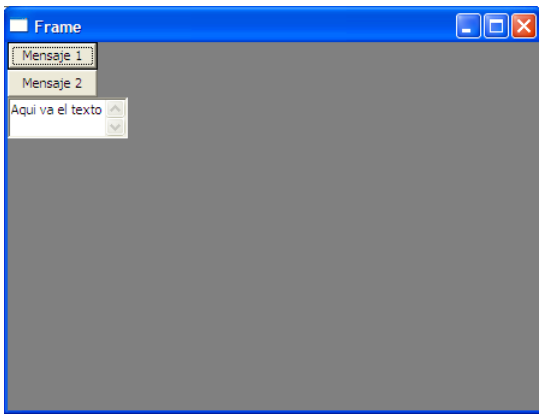

Figura A.2: Un ejemplo básico con controles

Listado A.12: Implementación de un evento.

```
1 void Ventana::OnPulsacionBoton1(wxCommandEvent& evento)
2 {
3  texto->AppendText ("Se ha pulsado el boton 1\n");
4 }
```

Listado A.13: Eventos de la ventana.

```
1 BEGIN_EVENT_TABLE(Ventana, wxFrame)
2  EVT_KEY_UP (Ventana::OnPulsaTecla)
3  EVT_BUTTON (wxID_BOTON1, Ventana::OnPulsacionBoton1)
4 END_EVENT_TABLE()
```

problema tienen que ver con los contenedores y su gestión Para resolver el primer problema simplemente se crea una función como la del fichero de implementación del listado A.12

Y se conecta el evento de pulsación de botón con la función mostrada en el fichero del listado A.13

De forma que el fichero ventana.h queda como en A.14 (se muestra solo el código nuevo)

Y ventana.cpp se muestra ahora en el listado A.15:

Con esto, al pulsar el primer boton se añade un mensaje control de texto.

El siguiente problema es algo más difícil de resolver y tiene que ver con como conseguir que los controles tengan la apariencia deseada. Por ejemplo se puede pensar en hacer que los botones estuvieran centrados horizontalmente. En la documentación se ofrece la posibilidad por la que se puede añadir el flag wxALIGN_CENTER al método Add del contenedor, que los centrará. En el listado A.16 se puede ver como usarlo.

Y se obtiene la imagen de la Figura A.3.

La explicación a esto es que el contenedor está poniendo controles *unos encima de otros* por lo que al centrar se dedica a centrar *verticalmente*. La solución al problema pasa por las jerarquías de contenedores. Los botones irán en su propio wxBoxSizer en horizontal, y dichos contenedores se pondran unos encima de otros. Los contenedores de los botones alinearán en horizontal y esos “alineadores horizontales” deben ir “alineados verticalmente” unos encima de otros.

Listado A.14: Métodos añadidos a la ventana.

```
1  #include "wx/wx.h"
2
3  class Ventana : public wxFrame
4  {
5   public:
6    Ventana();
7          void OnPulsaTecla (wxKeyEvent &evento);
8          void OnPulsacionBoton1(wxCommandEvent& evento);
9          void OnPulsacionBoton2(wxCommandEvent& evento);
10  private:
11   wxBoxSizer* contenedor;
12   wxButton* boton1;
13   wxButton* boton2;
14   wxTextCtrl* texto;
15   DECLARE_EVENT_TABLE ()
16 };
17
18 enum {
19  wxID_BOTON1=1,
20  wxID_BOTON2=2
21 }
22 }; //Fin de la clase
```

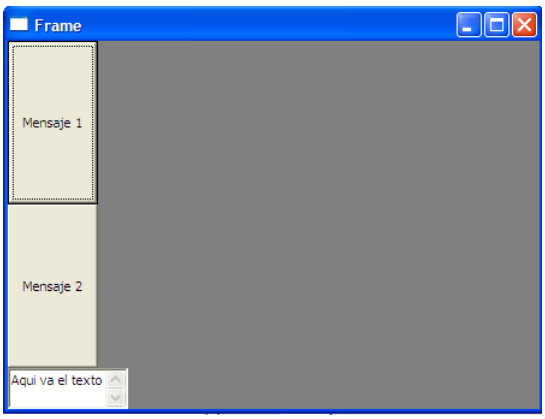

Figura A.3: Ventana con los controles centrados

Listado A.15: Nueva implementación de la ventana.

```
1  #include "ventana.h"
2
3  Ventana::Ventana() :
4   wxFrame (NULL, wxID_ANY, "Ventana")
5  {
6   /* Se crea el contenedor */
7   contenedor= new wxBoxSizer(wxVERTICAL);
8
9   /* Se crean los controles*/
10  boton1 = new wxButton (this, wxID_BOTON1, "Mensaje 1");
11  boton2 = new wxButton (this, wxID_BOTON2, "Mensaje 2");
12  texto = new wxTextCtrl(this, wxID_ANY, "Aqui va el texto",
13  wxDefaultPosition, wxDefaultSize, wxTE_MULTILINE);
14
15  /* Se insertan los controles*/
16  contenedor->Add (boton1);
17  contenedor->Add (boton2);
18  contenedor->Add (texto);
19
20  /* Y se establece el contenedor*/
21  this->SetSizer (contenedor);
22 }
23
24 void Ventana::OnPulsaTecla (wxKeyEvent &evento)
25 {
26  //Se comprueba si el ócdigo ASCII de la tecla pulsada es C
27  if (evento.m_keyCode == 67)
28  {
29   this->Close();
30  }
31 }
32
33 void Ventana::OnPulsacionBoton1(wxCommandEvent& evento)
34 {
35  texto->AppendText ("Se ha pulsado el boton 1\n");
36 }
37
38 BEGIN_EVENT_TABLE(Ventana, wxFrame)
39  EVT_KEY_UP (Ventana::OnPulsaTecla)
40  EVT_BUTTON (wxID_BOTON1, Ventana::OnPulsacionBoton1)
41 END_EVENT_TABLE()
```

Listado A.16: Centrado de los componentes

```
1 contenedor->Add (boton1, wxALIGN_CENTER);
2 contenedor->Add (boton2, wxALIGN_CENTER);
```

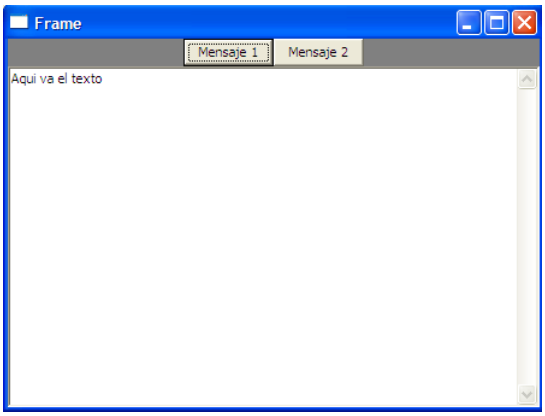

Figura A.4: Ventana con los controles centrados

Así, habrá un contenedor para los botones que irá en la parte superior y otro para el control de texto que irá en la parte inferior. Cuando se añadan los botones al contenedor se podrá especificar el flag wxALIGN_CENTER que centrará el botón horizontal y verticalmente dentro del contenedor, y si se usa el flag wxALL a la vez que un tamaño de borde se puede especificar ese borde de distancia entre el botón y los controles que estén en todas partes (se podría poner distancia solo por la izquierda por ejemplo con wxLEFT).

Otro concepto importante que se da al añadir un control a un contenedor es la proporción. Si especificamos una proporción distinta de 0 al añadir un control estamos permitiendo que ese control ocupe más espacio relativo dentro de un contenedor. Si por ejemplo se añaden 3 botones y uno de ellos se añade con proporcion 1, obtendrá más espacio para “agrandarse” que los demás.

A continuación la nueva implementación de ventana.cpp.

En la Figura A.4 se puede ver que ahora se obtiene el resultado deseado. La programación con contenedores es en realidad mucho más potente que el clásico diseño de controles de tamaño fijo, con la salvedad de que se recomienda el diseño en papel de los diferentes formularios y ventanas de que consta la aplicación.

Listado A.17: Componentes con proporciones.

```
#include "ventana.h"

Ventana::Ventana() :
 wxFrame (NULL, wxID_ANY, "Ventana")
{
 /* Se crea el contenedor */
 contenedor= new wxBoxSizer(wxVERTICAL);

 /* Se crean los contenedores de los botones*/
 contenedorBotones = new wxBoxSizer (wxVERTICAL);
 /* Se crean los controles*/
 boton1 = new wxButton (this, wxID_BOTON1, "Mensaje 1");
 boton2 = new wxButton (this, wxID_BOTON2, "Mensaje 2");
 texto  = new wxTextCtrl(this, wxID_ANY, "Aqui va el texto",
  wxDefaultPosition, wxSize (180,60), wxTE_MULTILINE);

 /*Se insertan los controles con 10 pixeles de borde por todos
 los lados y alineados horizontal y verticalmente.*/
 contenedorBotones->Add(boton1,0, wxALL | wxALIGN_CENTER, 10);
 contenedorBotones->Add(boton2,0, wxALL | wxALIGN_CENTER, 10);

 /* Se insertan en el contenedor principal la botonera y
 el control de texto. El control de texto puede expandirse
 hacia los lados (ya que la gestion vertical la realiza
 su contenedor) y áadems ocupa mayor proporcion que
 los botones */
 contenedor->Add (contenedorBotones,0, wxALIGN_CENTER);
 contenedor->Add (texto, 1, wxEXPAND);

 /* Y se establece el contenedor*/
 this->SetSizer (contenedor);

 /* El contenedor se ensancha hasta ocupar
 elá rea de la ventana */
 contenedor->Fit (this);

 /* Y la ventana regira su tamano minimo por el de
  este contenedor */
 contenedor->SetSizeHints (this);
}
```

A.5. Cuadros de diálogo

WxWidgets dispone de los cuadros de diálogo clásicos en todas las aplicaciones así como de los controles y clases necesarios para construir cuadros de diálogo propios. Para construir cuadros de diálogo propios se necesita

1. Diseñar el cuadro de diálogo, utilizando los conceptos sobre controles y contenedores comentados anteriormente.
2. Diseñar la estructura de la información que el cuadro de diálogo intercambiará con la aplicación.
3. Codificar la conexión entre el cuadro de diálogo y la información mostrada.
4. Codificar la conexión entre el cuadro de dialogo y la aplicación.

Para llevar a cabo el punto 1 solo se necesita diseñar la estructura de controles contenedores y controles activos. Para evitar pérdidas de tiempo en la codificación de los controles del diálogo se pueden utilizar herramientas visuales que generan el código de la aplicación. Se debe tener en cuenta que estas herramientas están todavía en fases muy tempranas del desarrollo y con frecuencia el código generado no responde a lo construido en pantalla.

La estructura de la información (punto 2) a intercambiar entre el cuadro de diálogo y la aplicación suele contener como mínimo la información que el usuario establecerá mediante los controles. Además puede ser necesario derivar información adicional, por lo que la clase Diálogo construida puede contener miembros adicionales. Se recomienda utilizar las convenciones clásicas de la orientación a objetos y utilizar métodos *getter* y *setter* que permitirán a la aplicación obtener los datos deseados del cuadro de diálogo resolviendo así el punto 4.

El punto 3 reviste especial importancia por lo que se examinará más de cerca.

A.5.1. Creación de cuadros de diálogo

En WxWidgets la creación de componentes visuales sigue una convención. Por un lado, puede existir un único constructor que recibe todos los parámetros y que arranca el control y por otro lado puede existir un único constructor por defecto y dos métodos *Create* e *Init* que

crean la parte visual e inicializan las variables respectivamente. Por tanto, al crear una clase que vaya a crear un cuadro de diálogo se deben tener en cuenta los siguientes puntos:

- Es obligatorio heredar de la clase wxDialog.
- Es recomendable proporcionar dos constructores: Un constructor por defecto y sin parámetros y otro con todos los parámetros necesarios (consultar la clase wxDialog en la ayuda de WxWidgets). El único constructor realmente necesario es el que utiliza parámetros.
- Si se proporciona un constructor por defecto sin parámetros se deben proporcionar dos métodos:
  1. Método Init(). Inicializará los campos de la clase.
  2. Método Create(). Construye el diálogo en sí y en él se pondrá la llamada a un método que construya los controles para este diálogo concreto.
- Se recomienda poner el código de creación de controles en un método aislado con algún nombre significativo como *CreacionControles o* CrearControles.

La clase wxDialog, de la cual deben heredar los cuadros de diálogo implementados por el usuario, tiene dos métodos de especial importancia.

- TransferDataFromWindow: Mueve la información de los controles a los campos de la clase Diálogo.
- TransferDataToWindow: Mueve la información de los campos de la clase a los controles.

Así, para pasar información hacia y desde el cuadro de diálogo se tienen dos posibilidades. Una es implementar estos métodos y otra utilizar validadores. Los validadores son clases que comprueban si el contenido de un control es válido de acuerdo a los criterios del programador. Si lo son, transfieren el contenido del control a la variable indicada. Además, existen validadores estándar tales como comprobar si un cuadro de texto contiene solo texto o comprobar si contiene solo números. Existen libros de programación de este framework donde se recomienda el segundo método ([SHC05]).

Listado A.18: Uso de RTTI.

```
1
2 if ( typeid( *forma ) == typeid ( Triangulo ) )
3 {
4         //Hacer algo especifico para el átringulo
5 }
6 else if (typeid( *forma ) == typeid ( Circulo ))
7 {
8         //Hacer algo especifico para el circulo
9 }
```

A.6. RTTI

El acrónimo RTTI corresponde en castellano al concepto Identificación de Tipos en Tiempo de Ejecución (**R**un-**T**ime **T**ype **I**dentification). La identificación de tipos es una característica de C++ que permite a un programa averiguar el tipo que tiene una variable o puntero dados para, en función del tipo, tomar un camino de ejecución u otro. En general los autores desaconsejan el uso de esta característica, ya que conduce a código no estructurado como el mostrado en A.18:

En general, se recomienda el uso de funciones virtuales frente a RTII ([Str98a]) tanto por razones de diseño como por razones de portabilidad[1].

En WxWidgets no se hace uso de RTTI aunque el diseño del sistema necesita saber en ocasiones características de las clases. Por ello, WxWidgets utiliza una serie de macros para ayudar a la biblioteca a conocer dichas características.

DECLARE_CLASS (Nombre de clase) : Se utiliza para hacer saber al sistema que cierta clase pertenece a la jerarquía de WxWidgets. Se debe usar en el fichero de declaración de una clase que herede de algún componente WxWidgets (por ejemplo un cuadro de diálogo personalizado que hereda de wxDialog) .

IMPLEMENT_CLASS (Nombre de la clase, Nombre de la clase base) : Se utiliza en el fichero de implementación de una clase personalizada.

[1] Existen compiladores que no dan soporte a esta característica y aunque la den, no todos los depuradores la soportan

A.7. Problemas de interés para el mantenedor

Comentario:Esto se ha corregido y no es un capítulo aparte

A continuación se dan algunas pistas para resolver problemas que pueden aparecer y que son difíciles (cuando no imposibles) de resolver solo mediante la documentación.

A.7.1. Problemas con WxWidgets

1. Al ejecutar la aplicación los controles no salen donde se desea y/o incluso aparecen controles superpuestos.

   Todo control creado en memoria con `new()` tiene que haber sido añadido a algún contenedor con `contenedor->add()`. Si no se hace así, la biblioteca asignará una posición absoluta por defecto y la colocación de objetos no será la deseada.

2. La compilación del programa es demasiado lenta.

   Si se utiliza Dev-C++ puede que esté activada una opción por defecto que hace que el entorno averigüe de forma exacta que ficheros dependen de otros. Al hacer su trabajo demasiado bien, puede hacer que ficheros de SimProc (que cambian contínuamente) queden relacionados con ficheros de la biblioteca WxWidgets. Dado que los ficheros de biblioteca no cambian nunca, el compilador examina dependencias de forma innecesaria, ralentizando el proceso de compilación. Para evitarlo en el menú Herramientas hay un submenú "Opciones" donde se puede activar una opción etiquetada con el letrero "Usar generador de dependencias rápido pero imperfecto". También existe la posibilidad de suministrar un Makefile construido a medida.

3. Al pulsar un botón el programa parece bloquearse.

   Si en el programa se construyen diálogos pero luego no se llama a su método `Show` la aplicación está esperando que el usuario interactúe con un diálogo que no se está mostrando en pantalla. Se debe revisar la programación de los cuadros de diálogo.

4. Al compilar una aplicación se observa un mensaje de error con el texto *"invalid use of struct wxSpinCtrl"*.

En teoría basta incluir el fichero de cabecera wx.h para utilizar cualquier control. En la práctica esto no parece suceder siempre por lo que se hace necesario incluir el fichero de cabecera de ese control (en este caso `wxSpinCtrl.h`)

5. Las cadenas de búsqueda de ficheros en el cuadro de diálogo para la apertura de ficheros no se muestran correctamente.

   En los cuadros de dialogo FileDialog no se pueden dejar espacios en las cadenas. P.ej se debe sustituir “Ficheros xml | *.xml | ...” por “Ficheros xml|*.xml|...”

6. El compilador no compila el método de inicialización de la aplicación WxWidgets.

   Probablemente se está confundiendo onInit (mal) en vez de OnInit (bien) para arrancar wxApp. WxWidgets utiliza la notación clásica de programación de aplicaciones Windows con C++ mientras que SimProc utiliza una codificación de nombres más cercana al estilo Java.

A.7.2. Problemas con Xerces

1. El parser da un error de análisis en una línea que se sabe que está bien.

   Xerces no parece dar ningún fallo en la mayoría de ficheros de ejemplo. Si se da un error de análisis puede ser debido a una de las siguientes causas:

   a) Ha habido un error al escribir un fichero XML pero el error está más arriba.

   En este sentido, Xerces es un parser como otro cualquiera y respeta las reglas de sintaxis de forma estricta intentando recuperarse de errores. Los errores que señala se deben a símbolos que no esperaba en un cierto momento. Con toda probabilidad el error debe haberse producido en una línea anterior.

   b) Se ha usado la acentuación.

   Para mejorar la compatibilidad al máximo, todo el programa se ha construido sobre la hipótesis de que ningún símbolo de los ficheros XML va a tener tildes. Una tilde en una palabra de un comentario producirá un error de análisis.

   c) Realmente hay un error en la biblioteca.

Listado A.19: Activacion del espacio de nombres.

```
1  #ifdef XERCES_CPP_NAMESPACE_USE
2    XERCES_CPP_NAMESPACE_USE
3  #endif
```

Se recomienda consultar los foros de Xerces. Puede ser necesario actualizar a una versión superior.

2. Al compilar un fichero que usa Xerces se obtiene un mensaje de error como el siguiente:
   `elemento.h ISO C++ forbids declaration of 'DOMElement' with no type .`

   Por defecto, el programa no está usando el espacio de nombres definido por Xerces. Debe forzarse el uso del mismo con el código siguiente

3. Al compilar un fichero que usa Xerces se obtiene un mensaje de error como el siguiente:
   `BaseRefVectorOfC unknown`.

   En este fichero de Xerces se define como se van a utilizar las plantillas en C++. En el compilador usado en este proyecto no hay soporte para el uso no estándar de plantillas que Xerces hace, por lo que se debe activar un cierto parámetro para forzar el uso correcto. Esta activación se hace en concreto con la línea `#define XERCES_TMPLSINC`

4. El programa falla al llamar a un método `appendChild`

   Si al llamar al método `appendChild` se para el programa es porque se está tratando de coger un nodo que pertenece a un documento A y se está intentando insertar en otro documento B directamente, lo cual es incorrecto. La forma de realizar dicha operación con Xerces es la siguiente: el documento B debe llamar al método `importNode` con el nodo de A que quiera importar y se obtendrá el nodo listo para insertar en B con `appendChild`.

   Esto se debe a que el comportamiento que el estándar XML del W3C asocia a estos métodos no es todo lo intuitivo que debiera ser y en pocas palabras se puede decir que `importNode`, fabrica una copia y "desasocia" esa copia del documento A. No se puede insertar en un árbol un nodo XML asociado a otro árbol.

A.7.3. Otros problemas

Otros datos que se deben tener en cuenta al modificar o ampliar SimProc son los siguientes:

1. Puede haber errores inesperados si se usa true (palabra reservada de C++) en vez de la macro TRUE de WxWidgets.

2. Si aparece el mensaje “La aplicacion ha solicitado la terminacion” es un fallo que Xerces ha encontrado. Por ejemplo hay que crear la etiqueta etqElemento en el constructor de un elemento de sistema. Tambien se debe pasar las señales correctas a anadirElementoXML.

3. Se debe construir un parser ANTES DE EMPEZAR a trabajar con cualquier elemento ya que es el parser el que inicializa el subsistema XML que necesitan los elementos.

4. Ninguno de los directorios del proyecto debe contener espacios como por ejemplo “C:Mis documentos” ya que el entorno o el compilador producen errores.

5. Si aparecen muchos errores del tipo *“linker error: undefined reference to XML...”* y la mayor parte de ellos son símbolos que se refieren a funciones de Xerces, es porque el fichero `xerces.lib` no está enlazado o el path hacia el fichero no es correcto. También influye el orden de enlazado de los ficheros, es necesario enlazar primero la libreria de XML creada en el fichero `libreriaXML.a` y luego `xerces.lib`.

6. En la biblioteca para la manipulación de elementos con XML no se deben confundir los métodos. En concreto se debe tener cuidado con confundir PrimeraEntrada con PrimeraEntradaUtil y PrimeraSalida con PrimeraSalidaUtil. Los primeros métodos hacen referencia a la entrada de habilitación y los segundos a las entradas de señales propiamente dichas.

7. Si se ha añadido código para ampliar o mejorar las pruebas de unidad se debe tener cuidado y comprobar que quizá no se haya recompilado la libreria y sí se han recompilado los test.

A.8. Tareas pendientes

Mejorar la bibliografía

Escribir el estudio de los toolkits gráficos Escribir introducción UML descrita.

[CLM03]

Bibliografía

[BA04] Kent Beck and Cynthia Andres. *Extreme Programming Explained*. Addison-Wesley Professional, 2004.

[BJR98] Grady Booch, Ivar Jacobson, and James Rumbaugh. *Designing the user interface*. Pearson, 1998.

[BMM98] William Brown, Raphael Malveau, and Hays McCormick. *Antipatterns*. Wiley, 1998.

[Boe91] Barry Boehm. Top 10 software defects. *IEEE Computer*, 1991.

[CLM03] Bernardo Cascales, Pascual Lucas, and José Manuel Mira. *El libro de Latex*. Prentice Hall, 2003.

[Dra05] Claus Draeby. Unit++ for unit testing, extreme programming and test-driven development. http://unitpp.sourceforge.net, 2005.

[Fou03] The Apache Software Foundation. Xerces, a framework for xml processing. http://xerces.apache.org, 2003.

[Fou06a] The Free Software Foundation. Gcov, a coverage analysis tool. http://gcc.gnu.org, 2006.

[Fou06b] The Free Software Foundation. The gnu compiler collection for software development. http://gcc.gnu.org, 2006.

[GHJ02] Erich Gamma, Richard Helm, and Ralph Johnson. *Patrones de diseño*. Addison-Wesley, 2002.

[Hun05a] David Hunter. *XML*. Inforbooks, 2005.

[Hun05b] David Hunter. *XML. Curso de iniciación*. Inforbooks, 2005.

[Ken] John F. Kennedy. The W3C Schools. http://www.w3schools.org.

[LBL96] Collin Laplace, Mike Berg, and Hongli Lai. Bloodshed dev-cpp. http://www.bloodshed.net/devcpp.html, 1996.

[Mar81] James Martin. *Rapid Application Development*. Prentice Hall, 1981.

[Mor95] Brian Morriss. *Automated manufacturing systems*. McGraw-Hill, 1995.

[Nee87] Frances Neelamkavil. *Computer Simulation and Modelling*. John Wiley and Sons, 1987.

[Pie93] Ramón Piedrafita. *Ingeniería de la automatización industrial*. Ra-Ma, 1993.

[SHC05] Julian Smart, Kevin Hock, and Stefan Csomor. *Cross-Platform GUI programming with WxWidgets*. Prentice Hall PTR, 2005.

[Shn98] Ben Shneiderman. *Designing the user interface*. Pearson, 1998.

[Sma92] Julian Smart. Wxwidgets, a multiplatform gui framework. http://www.wxwidgets.org, 1992.

[Str98a] Bjarne Stroustroup. *El lenguaje de programación C++*. Addison Wesley, 1998.

[Str98b] Bjarne Stroustrup. *El lenguaje de programación C++*. Addison-Wesley, 1998.

[TDa] Henry Thompson and Beech David. Xml schema: Part 0. http://www.w3c.org.

[TDb] Henry Thompson and Beech David. Xml schema part 1: Structures. http://www.w3c.org.