ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN
UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Proyecto Fin de Carrera

"Implementación de un editor gráfico de circuitos eléctricos con Qt"

AUTOR: León Alberto Martínez Muñoz
DIRECTOR: Juan Carlos Sánchez Aarnoutse

Junio / 2011

Autor	León Alberto Martínez Muñoz
E-mail del Autor	leonlematico@gmail.com
Director(es)	Juan Carlos Sánchez Aarnoutse
E-mail del Director	juanc.sanchez@upct.es
Codirector(es)	
Título del PFC	"Implementación de un editor gráfico de circuitos eléctricos con Qt"
Descriptores	

Resúmen

Desarrollo de un Editor de Esquemas Eléctricos bajo el lenguaje de programación Qt que permite diseñar circuitos eléctricos para su posterior interpretación con un simulador de redes PLC.

El Editor de Esquemas Eléctricos permita dibujar sobre un área de diseño un esquema eléctrico. El usuario dispone de cualquier componente eléctrico que pueda ser encontrado en una vivienda.

Permite modificar los parámetros de cada componente utilizando los diferentes menús que ofrece la aplicación y exportar el diseño a un archivo de texto.

Titulación	Ingeniero Técnico en Telecomunicaciones, esp. Telemática
Intensificación	
Departamento	Tecnologías de la Información y las Comunicaciones
Fecha de Presentación	Junio-2011

Índice

Lista de figuras

Capítulo 1. Introducción

Hace ya algunas décadas se propuso emplear las líneas de cableado eléctrico existentes para la transmisión de información. Sin embargo, dicha tecnología, denominada PLC (*Power Line Communication*), quedó relegada a ciertas aplicaciones industriales debido a que el medio de propagación presenta numerosos inconvenientes para la transmisión de señales a partir de ciertas frecuencias.

En esta última década y posiblemente en estos últimos cinco años, la aplicación de nuevas técnicas ha posibilitado que la tecnología PLC vuelva a estar en auge para la creación de redes de área local, compitiendo en aplicación y prestaciones con las redes inalámbricas.

Evidentemente, el medio de transmisión (el cableado de la vivienda), no ha sufrido ningún tipo de mejora ni se ha sustituido en los edificios antiguos, por lo tanto el medio sigue siendo inadecuado para la transmisión de ciertas frecuencias.

Considerando todo eso, un simulador para estudiar las redes PLCs tendría que tener en cuenta todos los parámetros que afectan a la transmisión: sección y longitud del cable, parámetros de conductividad, ruido, empalmes y bifurcaciones, equipos eléctricos conectados, su ciclo de funcionamiento, el factor de ruido introducido por éstos, etc.

La labor de introducir todos los componentes y elementos del esquema eléctrico de una vivienda básica en un editor de texto para los archivos de configuración del simulador resulta laboriosa, tediosa y puede suponer una fuente de errores considerable por lo que una herramienta muy útil sería un editor gráfico que permitiera seleccionar los elementos de un esquema eléctrico para la creación de éste y la posterior modificación de los parámetros de cada uno de los elementos.

Los programas de edición de esquemas eléctricos actuales permiten crear esquemas con todo tipo de detalles, sin embargo no están pensados para operar con los ciertos parámetros necesarios para un simulador de PLC, ni permiten la exportación de dicha información en archivos entendibles por un simulador.

Por ello, en el presente proyecto fin de carrera correspondiente a los estudios de la titulación Ingeniera Técnica de Telecomunicaciones especialidad en Telemática se ha desarrollado un Editor de Esquemas Eléctricos, que permite diseñar circuitos eléctricos para su posterior interpretación con un simulador de redes PLC.

Entre las diferentes alternativas existentes a la hora de elegir el lenguaje de programación se ha optado por desarrollar el editor íntegramente con librerías Qt. Aunque es un lenguaje poco común, es utilizado para desarrollar interfaces gráficas de usuario y también para el desarrollo de programas sin interfaz gráfica como herramientas de la consola y servidores. Qt utiliza el lenguaje de programación C++ de forma nativa aunque puede ser utilizado en varios otros lenguajes de programación a través de bindings.

Como referencia, cabe mencionar que Qt es utilizada principalmente en KDE, Google Earth, Skype, Qt Extended, Adobe Photoshop Album, VirtualBox , etc.

Otra gran ventaja es que Qt funciona en todas las principales plataformas y tiene un amplio apoyo en la comunidad internacional de programadores.

El Editor de Esquemas Eléctricos desarrollado en este trabajo (denominado con el acrónimo EDEE) permite dibujar sobre un área de diseño cualquier componente eléctrico que existe dentro de una vivienda, representando así el esquema deseado por el usuario.

El EDEE permite a su vez:

- Editar las propiedades de cada componente eléctrico.
- Exportar el diseño realizado por el usuario a un archivo de texto.
- Guardar y Cargar el diseño realizado por el usuario.

Un esquema eléctrico es una representación gráfica de una instalación eléctrica o de parte de ella, en la que queda perfectamente definido cada uno de los componentes de la instalación y la interconexión entre ellos. En un esquema eléctrico, cada componente tiene un símbolo único que lo identifica. Cada componente del esquema posee ciertas propiedades que lo caracterizan.

Los componentes que se pueden encontrar son: Enchufes, Interruptores, Bombillas, Diferenciales, Magnetotérmicos, Cables, Empalmes, ICP (Interruptor de Control de Potencia) y PLC (*Power Line Communications*).

1.1. Objetivos

Una vez realizada la introducción, la justificación y una breve descripción de la aplicación a desarrollar se presentan los objetivos iniciales de este proyecto final de carrera que se enmarcan en tres aspectos diferenciados.

1. Crear una aplicación consistente en un editor de esquemas eléctricos basado en componentes.
2. Cada componente tendrá una serie de propiedades que podrán ser editadas mediante los correspondientes menús.
3. La aplicación permitirá la exportación de la información contenida en el esquema a un archivo de texto que identifique los principales componentes, sus propiedades y su conexionado.

Capítulo 2. ¿Qué es Qt?

2.1. Introducción a Qt.

Qt es un *framework* para el desarrollo de aplicaciones multiplataforma creado por la compañía Trolltech y que actualmente es propiedad de Nokia. Trolltech fue la compañía que desarrollo Qt en 1992. En 2008 Nokia compró esta compañía y pasó a ser responsable del proyecto Qt.

La función más conocida de Qt es la creación de interfaces de usuario, sin embargo no se limita a esto, ya que también provee varias clases para facilitar ciertas tareas de programación como: el manejo de sockets, soporte para programación multihilo, comunicación con bases de datos, manejo de cadenas de caracteres, y también para el desarrollo de programas sin interfaz gráfica como herramientas de la consola y servidores.

Qt utiliza C++ de manera nativa, lo cual otorga muchas ventajas para ciertos programadores, de hecho se puede decir que casi no se percatan que están trabajando en este lenguaje. Adicionalmente puede ser utilizado en varios otros lenguajes de programación a través de bindings como Python mediante PyQt, Java mediante QtJambi, o C# mediante Qyoto. Al ser un *framework* nunca se está saliendo del lenguaje C++, así que las sentencias para el preprocesador siguen siendo validas, se puede compilar en cualquier entorno, es decir se puede seguir utilizando sobre Linux, MAC, Windows, y ahora también móviles.

El API de la biblioteca cuenta con métodos para acceder a bases de datos mediante SQL, así como uso de XML, gestión de hilos, soporte de red, una API multiplataforma unificada para la manipulación de archivos y una multitud de otros para el manejo de ficheros, además de estructuras de datos tradicionales.

Qt es utilizada principalmente en Autodesk, KDE, Google Earth, la Agencia Espacial Europea, Skype, Qt Extended, Adobe Photoshop Album, VirtualBox, VLC media player, Samsung, Philips, Panasonic y Opie.

Uno de los ejemplos mencionados anteriormente y muy famoso es KDE, este escritorio de Linux también usa Qt, así como todas las aplicaciones que corren sobre éste. En 1998 algunos representantes de KDE junto con Trolltech llegaron a un acuerdo de donde salió la licencia QPL con la cual se dice que los desarrolladores de KDE siempre tendrán una versión libre de Qt.

Una característica muy importante que maneja Qt es el paradigma señales y *slots*, también visto como eventos. Esto es un mecanismo a través del cual los objetos de una aplicación se comunican. Este mecanismo es una de las características distintivas de Qt.

Una señal es una notificación que un objeto emite cuándo cambia su estado de manera que podría interesarle a otros objetos. Un *slot* es una función que se ejecuta cuándo una señal se emite. Los *widgets* que nos proporciona Qt poseen señales y *slots* predefinidos, pero también es posible crear un *widget* personalizado con el fin de que el programador pueda añadir sus propias señales y *slots*.

En este ejemplo se aprecia la conexión de una señal con un *slot*. El objeto A tiene una señal llamada envió, y el objeto B tiene un *slot* llamado recepción:

connect(A, **SIGNAL**(envio), B, **SLOT**(recepcion));

Otras características de las señales y los *slots* son:

- Una señal puede conectarse a varios *slots*.
- Muchas señales pueden conectarse a un mismo *slot*.
- Una señal se puede conectar a otra señal.
- Las señales y los *slots* pueden enviar y recibir parámetros.
- Las señales pueden ser borradas.

Ya que C++ no trabaja con el paradigma de señales y *slots*, Qt genera unos archivos llamados MOC para suplir esta ausencia.

Para la compilación de código se utiliza *QMake*. Vale la pena aclarar que *QMake* es una herramienta para generar los MOC, y *MakeFiles* correspondientes para cada sistema operativo. *QMake* sigue unas reglas dadas en un archivo de configuración, un ".pro", el cual no es difícil de generar, de hecho *QTCreator*, y *QDevelop* lo hacen.

Qt ofrece una suite de aplicaciones para facilitar y agilizar las tareas de desarrollo, las aplicaciones que componen esta suite son:

- *Qt Assistant*: Herramienta para visualizar la documentación oficial de Qt.
- *Qt Designer*: Herramienta para crear interfaces de usuario.
- *Qt Linguist*: Herramienta para la traducción de aplicaciones.
- *Qt Creator*: es un IDE para el lenguaje C++, pero especialmente diseñado para Qt, integra las primeras dos herramientas mencionadas. Para la realización de este proyecto fin de carrera se ha utilizado Qt Creator.

Qt Creator es un IDE (*Integrated Development Environment*, entorno de desarrollo integrado) creado por Trolltech para el desarrollo de aplicaciones con las bibliotecas Qt. Los sistemas operativos que soporta en forma oficial son:

- GNU/Linux 2.6.x, para versiones de 32 y 64 bits con Qt 4.x instalado. Además hay una versión para Linux con gcc 3.3.
- Mac OS X 10.4 o superior, requiriendo Qt 4.x
- Windows XP, Vista y 7, requiriendo el compilador MinGW y Qt 4.4.3 para MinGW.

Al ser un IDE, contiene un editor de texto que le permite:

- Escribir código y formato.
- Anticipar lo que se va a escribir y completar el código.
- Visualización en línea de errores y mensajes de advertencia.
- Navegar semánticamente por las clases, funciones y símbolos.
- Recibir ayuda sensible de las clases, funciones y símbolos.

Qt Assistant es una ayuda que ofrece Qt para la búsqueda de documentación. Se ofrece como una ventana estándar en la que aparecen una barra de menú y una barra de tareas. La ventana principal está dividida en tres partes. A la izquierda aparecen dos menús, uno superior y otro inferior. El menú superior muestra las librerías y los ejemplos. En el menú

inferior se encuentra el nombre de la librería que se está buscando. En la parte derecha aparece toda la información de la librería en cuestión.

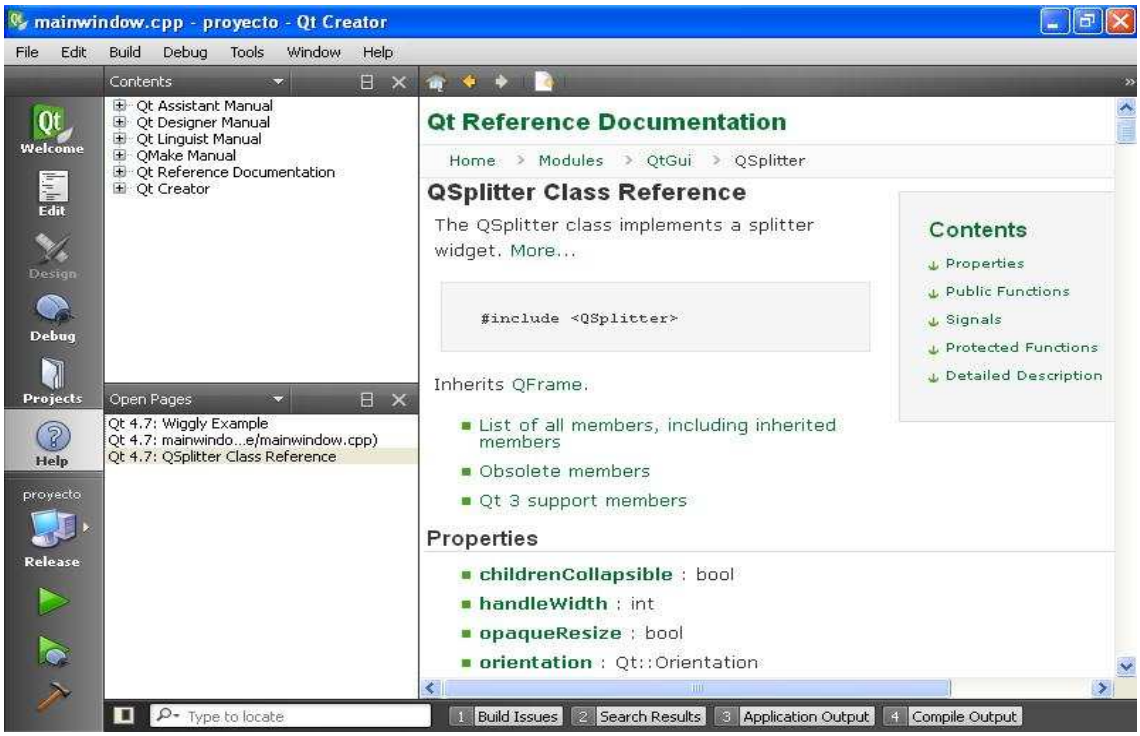


Figura 2.1 Pantalla principal Qt Assistant.

Qt Designer es una herramienta para el diseño y la creación de interfaces gráficas de usuario (GUI). Con esta herramienta se puede diseñar arrastrando los elementos, utilizar diferentes estilos de manera sencilla, personalizar un elemento, definir los cambios de estado entre transiciones, y los cambios de estado que se generan por las acciones de los usuarios. El código de programación se integra sin problemas con *Qt Designer*. Los *widgets* pueden unirse entre sí mediante el mecanismo explicado anteriormente de señales y *slot*. Las propiedades de cada elemento se pueden cambiar con *Qt Designer*. Para trabajar con esta herramienta se muestra en la parte izquierda una paleta con botones, *widgets*, *ítems*, contenedores y *display*. En su parte central aparece una sección que permite arrastrar los componentes de la paleta y colocarlos. En la izquierda un menú sirve para configurar cada *ítem*, *widget*, contenedor etc.

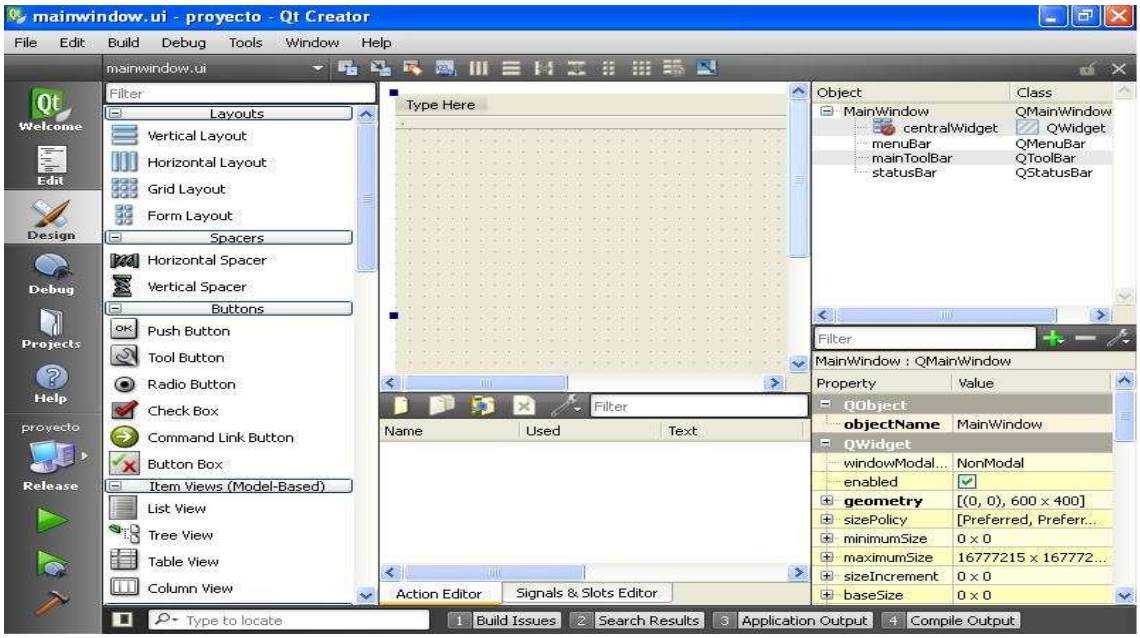


Figura 2.2 Pantalla principal Qt Designer.

Qt Creator puede generar un paquete de instalación cuando un dispositivo móvil se conecta a un PC, al conectarlo se crea un código y lo ejecuta.

2.2. Qt 4.7.0

Recientemente Qt ha sacado a la luz la versión 4.7.0. Esta versión ofrece muchas mejoras con respecto a las versiones anteriores de la serie de Qt 4.

La característica clave de las versiones Qt 4.7 y *Qt Creator* es *Qt Quick*. Una tecnología de Interfaz de Usuario (UI) de alto nivel que permite a los desarrolladores y diseñadores de interfaces de usuario trabajar juntos para crear aplicaciones ligeras, animadas e interfaces de usuario táctil. Incluye:

- QML es un lenguaje declarativo orientado en JavaScript que utiliza las caracteristicas de Qt *Meta-Object* para permitir a los diseñadores y desarrolladores crear aplicaciones animadas de usuario.
- Qt declarativo es una biblioteca C + + que permite un nuevo enfoque de programación declarativa.
- *Qt Creator* mejora la edición interactiva de interfaces de usuario a través de arrastrar y soltar. El editor de texto es compatible con la sintaxis QML, proporciona auto-completar, búsqueda de errores, ayuda en las operaciones de búsqueda y ofrece una vista previa de la interfaz de usuario.

2.2.1. Portador de gestión de redes

El nuevo módulo de *QtNetwork* permite a la aplicación identificar si el sistema está en línea y la cantidad de interfaces que existen conectadas, así como iniciar y detener las interfaces.

2.2.2. Mejoras en la función QtWebKit

La clase *QGraphicsWebView* mejora el rendimiento de desplazamiento y el zoom. Se puede crear animaciones con el metodo zoom-in o efectos de alejamiento. Qt 4.7 añade soportes de aceleración lo que mejora el rendimiento de las animaciones CSS y transiciones.

2.2.3. Otras mejoras

En Mac OS X, se utiliza diferentes implementaciones para los *widgets* (llamados "alien *widgets*"), que mejora la capacidad de respuesta de las aplicaciones que tienen interfaces de usuarios.

Se introduce la clase *QStaticText*, que puede ser utilizada para mejorar el rendimiento de la representación del texto.

La clase *QPainter* tiene una nueva representación de mapa de píxeles (QPainter:: drawPixmapFragments), que mejora el rendimiento del renderizado de las aplicaciones que necesitan de mapa de pixels.

Una versión actualizada del motor de JavaScriptCore para el módulo *QtScript*, que mejora el rendimiento de JavaScript en ejecución.

Bibliografía

http://es.wikipedia.org/wiki/Qt_%28biblioteca%29
http://en.wikipedia.org/wiki/Qt_Creator
http://www.zonaqt.com/foro
http://qt.nokia.com/products/
libro C++ GUI Programming with Qt 4, Second Edition
http://doc.qt.nokia.com/4.7-snapshot/qt4-7-intro.html
http://developer.qt.nokia.com/wiki/Qt_Quick_Spanish

Capítulo 3. Esquemas eléctricos y Elementos

Un esquema eléctrico es una representación gráfica de una instalación eléctrica o de parte de ella, en la que queda perfectamente definido cada uno de los componentes de la instalación y la interconexión entre ellos. Constituye las interconexiones que permiten la circulación de la energía eléctrica entre los diferentes tableros, interruptores y enchufes de una vivienda; asimismo permiten interconectar la energía externa al domicilio con las conexiones internas.

En el diseño de los circuitos eléctricos se emplean interruptores o magnetotérmicos que deben cortar la circulación de la corriente sobre uno de los conductores de la red de distribución. Este diseño es por razones de seguridad, dado que si una persona accede al tomacorriente con el interruptor abierto está vinculado con la energía, que normalmente es la que da origen a accidentes eléctricos por contacto directo, este tipo de protectores se denominan interruptores diferenciales.

3.1. Elementos típicos en un esquema eléctrico

La siguiente es una relación básica de elementos gráficos que se suelen encontrar en un esquema eléctrico.

3.1.1. Leyendas

En un esquema, los componentes se identifican mediante un descriptor o referencia. A menudo el valor del componente se pone en el esquemático al lado del símbolo. Las leyendas (como referencia y valor) no deben ser cruzadas o invadidas por cables ya que esto hace que no se entiendan dichas leyendas.

3.1.2. Símbolos

Los estándares o normas en los esquemáticos varían de un país a otro y han cambiado con el tiempo. Lo importante es que cada dispositivo se represente mediante un único símbolo a lo largo de todo el esquema, y que quede claramente definido mediante la referencia en el esquema.

3.1.3. Cableado y conexiones

El cableado se representa con líneas rectas, colocándose generalmente las líneas de alimentación en la parte superior e inferior del dibujo y todos los dispositivos, y sus interconexiones, entre ambas líneas. Las uniones entre cables suelen indicarse mediante círculos (empalmes), u otros gráficos, para diferenciarlas de los simples cruces sin conexión eléctrica.

Para diseñar y plantear un circuito completo de electricidad debe tomarse en cuenta el tipo de corriente que se necesitará, por lo general para viviendas es monofásica; los circuitos monofásicos que alimentan bocas de salida para iluminación y enchufes se utilizan básicamente en el interior de los edificios y viviendas; de amplio uso para los circuitos de iluminación, pueden conectarse a ellos artefactos de iluminación, de ventilación, combinaciones de ellos, u otras cargas unitarias. Es importante que estos circuitos tengan

instaladas cajas o tableros de protección; para el caso de establecer un circuito, debe considerarse una corriente menor a 16 Amperios, es necesario que la protección sea en ambos polos, ello se consigue con interruptores del tipo bi-polar en el módulo de contadores; a su vez el número de salidas por circuito debe ser como máximo catorce. Las conexiones al exterior deben estar protegidas contra la intemperie o en su defecto emplear protectores plásticos con tapa a presión para colocar en ellos dados de salida.

3.2. ¿Que es un PLC?

Power Line Communications, también conocido por sus siglas PLC, es un término inglés que puede traducirse por *comunicaciones mediante cable eléctrico* y que se refiere a diferentes tecnologías que utilizan las líneas de energía eléctrica convencionales para transmitir señales de radio para propósitos de comunicación. La tecnología PLC aprovecha la red eléctrica para convertirla en una línea digital de alta velocidad de transmisión de datos, permitiendo, entre otras cosas, el acceso a Internet mediante banda ancha.

3.2.1. Control de hogar (banda estrecha)

La tecnología PLC puede usar el cableado eléctrico doméstico como medio de transmisión de señales.

Típicamente, los dispositivos para control de hogar funcionan mediante la modulación de una onda portadora cuya frecuencia oscila entre los 20 y 200 kHz inyectada en el cableado doméstico de energía eléctrica desde el transmisor. Esta onda portadora es modulada por señales digitales. Cada receptor del sistema de control tiene una dirección única y es gobernado individualmente por las señales enviadas por el transmisor. Estos dispositivos pueden ser enchufados en la toma eléctrica convencional o cableada en forma permanente en su lugar de conexión. Ya que la señal portadora puede propagarse en los hogares o apartamentos vecinos al mismo sistema de distribución, estos sistemas tienen una "dirección doméstica" que designa al propietario. Esto, por supuesto es válido cuando las viviendas vecinas poseen sistemas de este tipo; situación muy común en las zonas residenciales de Japón.

3.2.2. Cableado de redes caseras (banda ancha)

Las tecnologías *HomePlug* y *HomePlug AV*, son los dos estándares más populares empleados en el hogar, sin necesidad de instalar cableado adicional

La tecnología PLC también puede usarse en la interconexión en red de computadoras caseras y dispositivos periféricos, incluidos aquellos que necesitan conexiones en red, aunque al presente no existen estándares para este tipo de aplicación. Las normas o estándares existentes han sido desarrolladas por diferentes empresas dentro del marco definido por las organizaciones estadounidenses *HomePlug Powerline Alliance* y la *Universal Powerline Association*. Los problemas de los PLC en redes domésticas suelen venir dados por la potencia contratada en una casa (inferior a la del ámbito empresarial); dado que si la red eléctrica no tiene una instalación concienzudamente correcta, podía darse el caso de interferencias o picos de tensión que acabarían afectando a los aparatos eléctricos conectados a dicha red. Así mismo, aunque los fabricantes aseguran que el consumo de un PLC es mínimo o nulo por trabajar en un circuito cerrado, la sola

conversión de los datos provenientes del par de cobre de la línea telefónica a un tipo determinado de electricidad conlleva un consumo energético. Así mismo el paso del chorro de información de estos aparatos genera un tránsito energético, si bien difícil de cuantificar por el tipo de conexión a la línea eléctrica, está ahí y eso es innegable.

3.2.3. Acceso a Internet (Banda ancha sobre líneas eléctricas)

La Banda ancha sobre líneas eléctricas (abreviada BPL por su denominación en inglés *Broadband over Power Lines*) representa el uso de tecnologías PLC que proporcionan acceso de banda ancha a Internet a través de líneas de energía ordinarias. En este caso, una computadora (o cualquier otro dispositivo) necesitaría solo conectarse a un *módem* BPL enchufado en cualquier toma de energía en una edificación equipada para tener acceso de alta velocidad a Internet.

A primera vista, la tecnología BPL parece ofrecer ventajas con respecto a las conexiones inalámbricas ya que utiliza medios guiados, al igual que la banda ancha basadas en cable coaxial o en DSL: la amplia infraestructura disponible permitiría que la gente en lugares remotos tenga acceso a Internet con una inversión de equipo relativamente pequeña para la compañía de electricidad. También, tal disponibilidad ubicua haría mucho más fácil para otros dispositivos electrónicos, tal como televisiones o sistemas de sonido, el poderse conectar a la red.

Las características físicas y de capilaridad de la red eléctrica y las altas prestaciones de los estándares por parte del IEEE, posicionan a las tecnologías basadas en *power line* como una excelente alternativa, siempre que se disponga de redes privadas de cable sobre las que inyectar las señales.

El ancho de banda que un sistema BPL se caracteriza por su estabilidad.

Los módems PLC transmiten en las gamas de media y alta frecuencia (señal portadora de 1,6 a 30 MHz). La velocidad asimétrica en el módem va generalmente desde 256 kbit/s a 2,7 Mbit/s. En el repetidor situado en el cuarto de medidores (cuando se trata del suministro en un edificio) la velocidad es hasta 45 Mbit/s y se puede conectar con 256 módems PLC. En las estaciones de voltaje medio, la velocidad desde los centros de control de red (*head end*) hacia Internet es de hasta 134 Mbit/s. Para conectarse con Internet, las empresas de electricidad pueden utilizar un *backbone* (espina dorsal) de fibra óptica o enlaces licenciados. Estandar HomePlug AV(2005)

La especificación HomePlug 1.0 fue lanzado en junio de 2001, que fue seguido por HomePlug AV en 2005 y HomePlug Green PHY en 2010. HomePlug AV2 está prevista para el primer trimestre de 2011.

Las diferencias en los sistemas de distribución de energía eléctrica en América y Europa afectan la puesta en práctica de la tecnología BPL. En el caso de Norteamérica, relativamente pocos hogares están conectados con cada transformador de distribución, mientras que en la práctica europea puede haber centenares de hogares conectados con

cada subestación. Puesto que las señales de BPL no se propagan a través de los transformadores de distribución eléctrica, solo se necesita equipos adicionales en el caso norteamericano. Sin embargo, ya que la anchura de banda es limitada, esto puede aumentar la velocidad a la cual cada casa puede conectarse, debido a los pocos usuarios que comparten la misma línea.

El sistema tiene un número de problemas complejos, siendo el primero que las líneas de energía intrínsecamente constituyen ambientes muy ruidosos. Cada vez que un dispositivo se enciende o apaga, introduce voltajes transitorios en la línea. Los dispositivos ahorradores de energía introducen a menudo armónicos ruidosos en la línea. El sistema se debe diseñar para ocuparse de estas interrupciones naturales de las señales y de trabajar con ellas.

Las tecnologías de banda ancha sobre líneas eléctricas se han desarrollado más rápidamente en Europa que en Estados Unidos debido a una diferencia histórica en las filosofías de diseño de sistemas de energía. Casi todas las grandes redes eléctricas transmiten energía a altos voltajes para reducir las pérdidas de transmisión, después en el lado de los usuarios se usan transformadores reductores para disminuir el voltaje. Puesto que las señales de BPL no pueden pasar fácilmente a través de los transformadores (su alta inductancia los hace actuar como filtros de paso bajo, dejando pasar solo las señales de baja frecuencia y bloqueando las de alta) los repetidores se deben unir a los transformadores. En Estados Unidos, es común colocar un transformador pequeño en un poste para uso de una sola casa, mientras que en Europa, es más común para un transformador algo más grande servir a 10 o 100 viviendas. Para suministrar energía a los clientes, esta diferencia en diseño es pequeña, pero significa que suministrar el servicio BPL sobre la red de energía de una ciudad típica de los Estados Unidos requerirá más repetidores en esa misma promoción, que los necesarios en una ciudad europea comparable. Una alternativa posible es utilizar los sistemas BPL como redes de retorno para las comunicaciones inalámbricas, por ejemplo colocando puntos de acceso Wi-Fi o radio bases de telefonía celular en los postes de energía, permitiendo así que los usuarios finales dentro de cierta área se conecten con los equipos que ya poseen. En un futuro próximo, los BPL se pudieran utilizar también como redes de retorno para las redes de WiMAX.

El segundo problema principal de BPL tiene que ver con la intensidad de la señal junto con la frecuencia de operación. Se espera que el sistema utilice frecuencias en la banda de 10 a 30 MHz, que es utilizada por los radio aficionados, así como por emisoras radiales internacionales en onda corta y por diversos sistemas de comunicaciones (militar, aeronáutico, hospitales, emergencias, Protección Civil, etc.). Las líneas de energía carecen de blindaje y pueden actuar como antenas para las señales que transportan, y tienen el potencial de eliminar la utilidad de la banda de 10 a 30 MHz para los propósitos de las comunicaciones en onda corta.[1]

Los sistemas modernos de BPL usan la modulación OFDM, que permite minimizar la interferencia con los servicios de radio, por la remoción de las frecuencias específicas usadas. Un estudio del 2001 realizado en conjunto por la ARRL (American Radio Relay League) y HomePlug, demostró que los módems que usaban esta técnica “junto a la separación moderada de la antena de la estructura con la señal de HomePlug, en general hacía que las interferencias fueran apenas perceptibles”, y hubo interferencias sólo cuando la “antena estaba físicamente cerca de las líneas de energía”.[1]

Las transmisiones de datos a velocidades mucho más altas usan las frecuencias de microondas transmitidas mediante un mecanismo recientemente descubierto de propagación superficial de ondas, denominado *E-Line*, que se ha probado usando solamente una sola línea de energía. Estos sistemas han demostrado el potencial para las comunicaciones simétricas y de Full Duplex a velocidades mayores a 100 Mbps en cada dirección. Se han probado múltiples canales de Radio sobre el cable con señales de 2,4 y 5,3 Ghz, con grandes ventajas por la carencia de interferencias, que sin embargo en Wi-Fi y Wimax sin licencia, suponen una grave limitación, siendo al mismo tiempo una ventaja por la facilidad de implantación sin necesidad de permisos.[1]

Además, debido a que puede funcionar en la banda de 100 MHz a 10 GHz, esta tecnología puede evitar totalmente los problemas de interferencias asociados al uso de un espectro compartido, ofreciendo la mayor flexibilidad para la modulación y los protocolos hallados para cualquier otro tipo de sistemas de microondas.[1]

3.3. ELEMENTOS

A continuación se detalla una explicación breve de los elementos que se encontran en un esquema eléctrico:

Interruptor: Un interruptor eléctrico es un dispositivo utilizado para desviar o interrumpir el curso de una corriente eléctrica. En el mundo moderno las aplicaciones son innumerables, van desde un simple interruptor que apaga o enciende una bombillá, hasta un complicado selector de transferencia automático de múltiples capas controlado por computadora.

Su expresión más sencilla consiste en dos contactos de metal inoxidable y el actuante. Los contactos, normalmente separados, se unen para permitir que la corriente circule. El actuante es la parte móvil que en una de sus posiciones hace presión sobre los contactos para mantenerlos unidos.

Interruptor sencillo

Interruptor en un esquema electrico

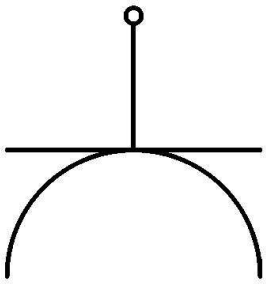
[1]http://es.wikipedia.org/wiki/Power_Line_Communications

Los interruptores se pueden clasificar como**:** Actuantes, Pulsadores, cantidad de polos, cantidad de vías, interruptor de doble vía y combinaciones.

A priori para el presente trabajo no tiene relevancia el tipo de interruptor, por lo tanto se considerará siempre el caso más sencillo: interruptor unipolar.

Enchufe: Un enchufe es un dispositivo formado por dos elementos, la clavija y la toma de corriente (o tomacorriente), que se conectan uno al otro para establecer una conexión eléctrica que permita el paso de la corriente.

Enchufe de tipo C, de origen europeo y muy usado internacionalmente.

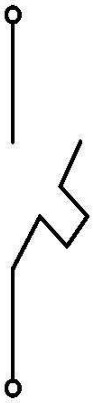


Enchufe en un esquema eléctrico.

Un *enchufe macho* o *clavija* es una pieza de material aislante de la que sobresalen varillas metálicas que se introducen en el enchufe hembra para establecer la conexión eléctrica. Por lo general se encuentra en el extremo de cable. Su función es establecer una conexión eléctrica con la toma de corriente que se pueda manipular con seguridad. Existen clavijas de distintos tipos y formas que varían según las necesidades y normas de cada producto o país.

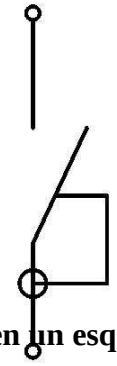
El *enchufe hembra*, *tomacorriente* o *toma de corriente* generalmente se sitúa en la pared, ya sea colocado de forma superficial (*enchufe de superficie*) o empotrado en la pared montado en una caja (*enchufe de cajillo* o *tomacorriente empotrado*), siendo éste el más común. Constan, como mínimo, de dos piezas metálicas que reciben a sus homóloga macho para permitir la circulación de la corriente eléctrica. Estas piezas metálicas quedan fijadas a la red eléctrica por tornillos o, actualmente con mayor frecuencia, por medio de unas pletinas plásticas que, al ser empujadas, permiten la entrada del hilo conductor y al dejar de ejercer presión sobre ellas, unas chapas apresan el hilo, impidiendo su salida.

Magnetotérmico: Un interruptor termomagnético, o disyuntor termomagnético, es un dispositivo capaz de interrumpir la corriente eléctrica de un circuito cuando ésta sobrepasa ciertos valores máximos de intensidad, debido al funcionamiento de varios aparatos eléctricos o de un cortocircuito. Su funcionamiento se basa en dos de los efectos producidos por la circulación de corriente eléctrica en un circuito: el magnético y el térmico (efecto Joule). El dispositivo consta, por tanto, de dos partes, un electroimán y una lámina bimetálica, conectadas en serie y por las que circula la corriente que va hacia la carga.



Magnetotérmico **Magnetotérmico en un esquema electrico**

Diferencial: Un interruptor diferencial exponencial, también llamado disyuntor por corriente diferencial o residual, es un dispositivo electromecánico que se coloca en las instalaciones eléctricas con el fin de proteger a las personas de las derivaciones causadas por faltas de aislamiento entre los conductores activos y tierra o masa de los aparatos.



Diferencial **Diferencial en un esquema eléctrico**

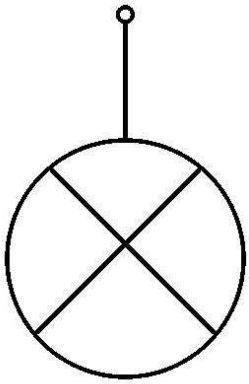
Gracias a sus dispositivos internos, tiene la capacidad de detectar la diferencia entre la corriente absorbida por un aparato consumidor y la de retorno. La intensidad que entra en el dispositivo debe ser igual a la intensidad de salida, cuando se produce una fuga de corriente estas intensidades no son iguales, es aquí donde el dispositivo desconecta el circuito para prevenir electrocuciones, actuando bajo la presunción de que la corriente de fuga circula a través de una persona que está conectada a tierra y que ha entrado en contacto con un componente eléctrico del circuito.

Aunque existen interruptores para distintas intensidades de actuación, el Reglamento Electrotécnico de Baja Tensión exige que en las instalaciones domésticas se instalan normalmente interruptores diferenciales que actúen con una corriente de fuga máxima de 30 mA y un tiempo de respuesta de 50 ms, lo cual garantiza una protección adecuada para las personas y cosas.

La norma UNE 21302 dice que se considera un interruptor diferencial de alta sensibilidad cuando el valor de ésta es igual o inferior a 30 miliamperios.

Bombilla: Una lámpara eléctrica es un dispositivo que produce luz mediante el calentamiento por efecto Joule de un filamento metálico, por fluorescencia de ciertos metales ante una descarga eléctrica o por otros sistemas. Pero en todos casos se produce luz mediante el paso de corriente eléctrica. En la actualidad se cuenta con tecnología para producir luz con eficiencias del 10 al 70%.

Luminaria



Luminaria en un esquema eléctrico.

Desde el desarrollo de la primera lámpara eléctrica incandescente a finales del siglo XIX se fueron sucediendo diversos e innovadores sistemas: *lámpara incandescente, lámpara compacta fluorescente, lámparas de haluro metálico, lámpara de neón, lámpara de descarga, lámpara de plasma, Lámpara de inducción, lámpara de vapor de sodio, lámparas de vapor de mercurio, lámpara de deuterio y lámpara led.*

ICP: El Interruptor de Control de Potencia (ICP) es un interruptor magnetotérmico automático que instala la compañía suministradora de energía eléctrica al inicio de la instalación eléctrica de cada vivienda que controla la potencia consumida por el cliente en cada momento, de tal forma que, cuando dicha potencia consumida supera la potencia contratada, entra en acción automáticamente cortando el suministro eléctrico y es necesario rearmarlo para reanudarlo.

Interruptor de Control de Potencia

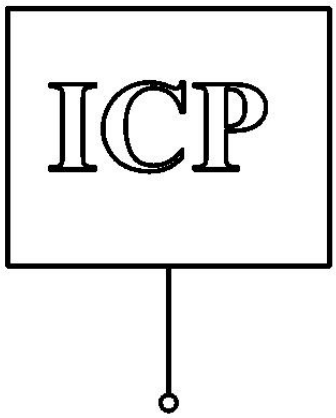


Imagen interruptor de Control de Potencia obtenida de la aplicación.

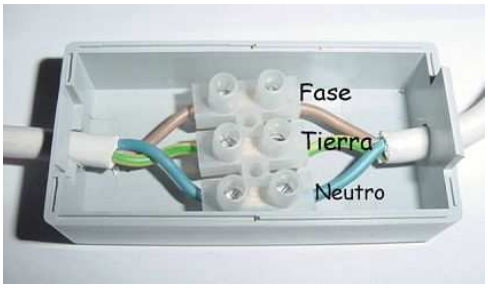
En España está regulado por la norma UNE 20317, que define la curva de disparo, es decir, el tiempo de disparo en función de la sobrecorriente.

Empalme: Un empalme eléctrico es la unión de dos o más cables de una instalación eléctrica o dentro de un aparato o equipo eléctrico o electrónico.

Aunque por rapidez y seguridad hoy en día es más normal unir cables mediante fichas de empalme y similares, los electricistas realizan empalmes habitualmente.

La realización de empalmes es un tema importante en la formación de los electricistas (y electrónicos) ya que un empalme inadecuado o mal realizado puede hacer *mal contacto* y hacer fallar la instalación. Si la corriente es alta y el empalme está flojo se calentará. El *chisporroteo* o el calor producido por un mal empalme es una causa común a muchos incendios en edificios. Antes de trabajar en la instalación eléctrica de un edificio o de un equipo eléctrico/electrónico se debe tener la formación técnica necesaria.

Las normativas de muchos países prohíben por seguridad el uso de empalmes en algunas situaciones. Es común la prohibición de realizar empalmes donde se puedan acumular gases inflamables.



Empalme

Empalme en un esquema eléctrico

Debe consultarse la normativa de cada país en caso de duda.

En España, por ejemplo, el *reglamento electrotécnico de baja tensión* prohíbe el uso de empalmes, tanto en el recorrido de los cables como en las cajas de empalme donde deben usarse regletas de conexión (y similares) adecuadas a la normativa UNE. Los empalmes sólo deben usarse de manera provisional.

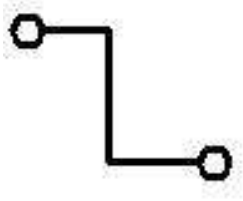
Cuando hay que unir cables coaxiales (datos, vídeo, antena, etc.) se deben emplear conectores en lugar de empalmes pues un empalme inapropiado puede modificar su impedancia y alterar la señal.

Una vez realizados los empalmes eléctricos se pueden soldar para conseguir un mejor contacto. Si existe el riesgo de cortocircuito con otros empalmes o cables se deben aislar mediante algún tipo de cinta aislante.

Cable: Se llama cable a un conductor (generalmente cobre) o conjunto de ellos generalmente recubierto de un material aislante o protector.

Los cables cuyo propósito es conducir electricidad se fabrican generalmente de cobre, debido a la excelente conductividad de este material, o de aluminio que aunque posee menor conductividad es más económico.

Cable eléctrico



Cable en un esquema eléctrico

Generalmente cuenta con aislamiento en el orden de 500 µm hasta los 5 cm; dicho aislamiento es plástico, su tipo y grosor dependerá de la aplicación que tenga el cable así como el grosor mismo del material conductor.

Las partes generales de un cable eléctrico son:

- Conductor: Elemento que conduce la corriente eléctrica y puede ser de diversos materiales metálicos. Puede estar formado por uno o varios hilos.

- Aislamiento: Recubrimiento que envuelve al conductor, para evitar la circulación de corriente eléctrica fuera del mismo.

- Capa de relleno: Material aislante que envuelve a los conductores para mantener la sección circular del conjunto.

- Cubierta: Está hecha de materiales que protejan mecánicamente al cable. Tiene como función proteger el aislamiento de los conductores de la acción de la temperatura, sol, lluvia, etc.

Clasificación de los conductores eléctricos (Cables)

Los cables eléctricos se pueden subdividir según:

Nivel de Tensión

- cables de muy baja tensión. (hasta 50 V)
- cables de baja tensión (hasta 1000 V)
- cables de media tensión (hasta 30 kV)
- cables de alta tensión (hasta 66 kV)
- cables de muy alta tensión (por encima de los 77 kV)

Componentes

- conductores (cobre, aluminio u metal)
- aislamientos (materiales plásticos, elastoméricos, papel impregnado en aceite viscoso o fluido
- protecciones (pantallas, armaduras y cubiertas).

Número de conductores

- Unipolar: un solo conductor.
- Bipolar: 2 conductores
- Tripolar:3 conductores
- Tetra polar: 4 conductores

Materiales empleados

- Cobre.
- Aluminio.
- Almelec (aleación de aluminio, magnesio y silicio).

Flexibilidad del conductor

- Conductor rígido.
- Conductor flexible.

Aislamiento del conductor

- Aislamiento termoplástico
    - PVC (policloruro de vinilo)
    - PE (polietileno)
    - PCP (policloropreno), neopreno o plástico
- Aislamiento termoestable
    - XLPE (polietileno reticulado)
    - EPR (etileno-propileno)
    - MICC Cable cobre-revestido Mineral-aislado

Cables de Baja, Media y Alta Tensión

Aplicaciones

- conexión de generadores.
- transformadores auxiliares.
- entrada a subestaciones.
- sifones (se trata de cables intercalados en una línea aérea).
- mallado de una red urbana.
- enlace entre dos subestaciones.

Partes constitutivas

- conductor.
- capa semiconductora interna.
- aislamiento.
- capa semiconductora externa.
- pantalla o cubierta metálica.
- armadura.
- cubierta exterior.

Parámetros eléctricos

- Resistencia óhmica.
- Inductancia y reactancia inductiva.
- Capacidad y reactancia capacitiva.
- Caída de tensión.
- Campo eléctrico.
- Pérdidas eléctricas.

Materiales aislantes

- Cables en papel impregnado:
    - Papel impregnado con mezcla no migrante.
    - Papel impregnado con aceite fluido.
- Cables con aislamientos poliméricos extrusionados:
    - Polietileno reticulado.(XLPE)
    - Goma etileno propileno (HEPR)
    - Polietileno termoplástico de alta densidad (HDPE).

Como se verá en el capítulo 4, para este PFC únicamente resultan de interés ciertos parámetros (Ruido, Tipo, Grosor y Longitud) por lo que inicialmente no se considerará un parámetro en los esquemas el grosor del aislante y el tipo de aislante. No obstante, el software creado permitiría añadir fácilmente nuevos parámetros a cada tipo de elemento.

Bibliografía:

http://es.wikipedia.org
http://www.elmercadodelavivienda.com/circuitos-electricos-en-viviendas.html

Capítulo 4. La aplicación

4.1. Introducción

En el presente capítulo se presenta la aplicación desarrollada en Qt para la edición gráfica de circuitos eléctricos. Dicha aplicación ofrece una interfaz gráfica que permite a los usuarios que lo deseen diseñar un esquema eléctrico para viviendas de forma sencilla y práctica. También permite exportar en un archivo de texto el diseño eléctrico esquematizado para su posterior utilización en otros programas.

La aplicación proporciona iconos de los elementos más comunes descritos en el capítulo anterior necesarios para diseñar un circuito eléctrico, tales como: Enchufes Interruptores, Cables, Luminarias, ICP, Magnetotérmicos, Diferenciales, Empalmes y dispositivos PLC. Cada elemento eléctrico tiene unas propiedades que pueden ser editadas por el usuario de manera sencilla Para facilitar la comprensión de los esquemas realizados, la aplicación permite cambiar el color de cada elemento (rojo, azul, verde y negro). Adicionalmente permite la inserción de texto y modificar sus parámetros como: cambiar la fuente, el color, el estilo y el tamaño. Por último, ofrece un área de dibujo en la que se insertan todos los ítems (elementos eléctricos y texto).

Para editar las propiedades de un elemento eléctrico se dispone de un menú desplegable, que aparece en la parte inferior de la aplicación al pulsar doble clic con el ratón sobre un ítem o pulsando botón derecho *propiedades* sobre el elemento seleccionado.

La aplicación presenta un aspecto agradable, cómodo y sencillo de manejar para el usuario. En la parte superior aparece una barra de menús desplegables que permite realizar diferentes acciones y editar las propiedades de los elementos. Debajo de este menú superior se muestra un panel móvil (menú gráfico) que contiene todas las acciones que se pueden ejecutar por parte del usuario.

En la parte izquierda se presenta un panel de selección que muestra los botones con los ítems descritos anteriormente: elementos eléctricos y el texto. Para insertar los ítems, la aplicación dispone de un área de dibujo en su parte central.

Gracias a que el código está escrito en Qt, que utiliza el lenguaje C++, la aplicación se adapta fácilmente a otros sistemas operativos ya que las instrucciones son válidas para el preprocesador. Simplemente basta con compilar el código en el sistema operativo que se desee utilizar y ésta funcionará perfectamente.

En conclusión, la aplicación es una interfaz gráfica sencilla, manejable y flexible. Ofreciendo al usuario todas las herramientas necesarias para el diseño esquemático de un circuito eléctrico. Además, gracias a su sencillez, su facilidad de manejo con el ratón y de las opciones de las que el usuario dispone, se hace muy cómodo realizar un diseño eléctrico.

4.2. Descripción.

La aplicación desarrollada sobre la edición de circuitos eléctricos con Qt presenta una pantalla rectangular dividida en varias secciones: una barra de menús en la parte superior, un menú gráfico, un panel en la parte izquierda que contiene todos los ítems y un área de dibujo en la parte central. Cada sección tiene una función específica que se detallará a continuación.

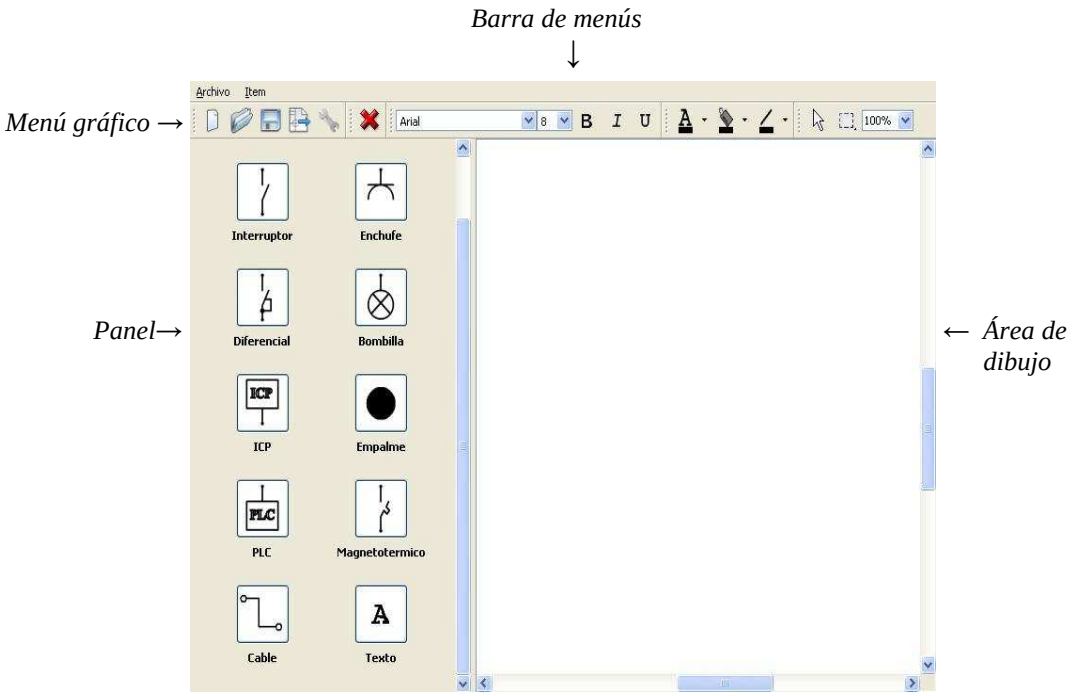


Figura 4.1 Pantalla principal de la aplicación.

4.2.1. Panel

Esta sección de la aplicación de diseños de circuitos eléctricos con Qt se puede definir como el panel encargado de sostener los botones con todos los ítems necesarios para el diseño de un circuito eléctrico. Los ítems se muestran en un recuadro con un dibujo y un texto que lo identifica. Además de los elementos eléctricos podemos encontrar un ítem para insertar texto. Esta sección, para mayor comodidad del usuario, puede ser ocultada arrastrándola hacia la izquierda con el ratón, de tal manera que el área de dibujo quede plenamente libre, ocupando toda la parte central de la aplicación. También dispone de una barra de desplazamiento en su parte derecha, ya que dependiendo de la resolución de la pantalla es posible que no se muestren todos los botones.

Los botones se muestran de la siguiente manera:

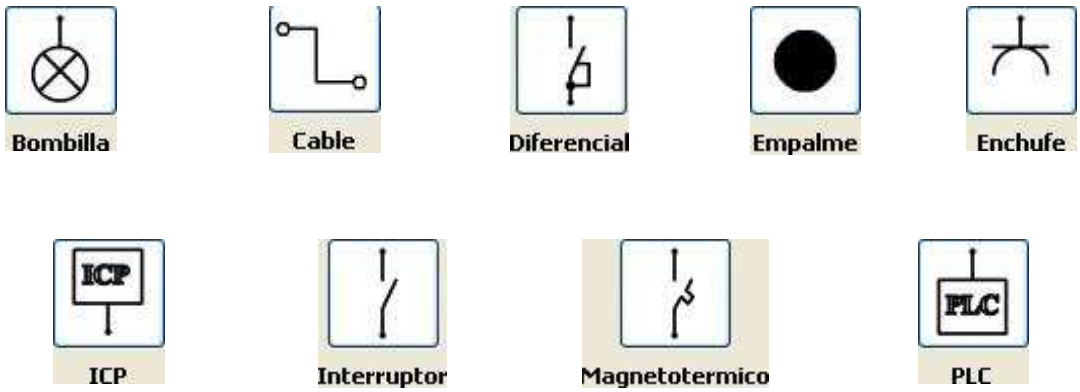


Figura 4.2 Botones que muestra los elementos eléctricos.

Para utilizar los botones que ofrece el panel, basta seleccionar con el puntero del ratón uno de ellos, desplazarse hasta el área de dibujo y pinchar sobre el lugar donde se quiera insertar el elemento. Inmediatamente la aplicación genera el dibujo correspondiente.

Para conseguir conectar dos elementos utilizamos el botón “cable”. Para realizar esta acción se debe pinchar sobre un elemento y arrastrar el cursor hacia el que se quiera unir, si la selección entre los dos elementos ha sido satisfactoria, entonces se dibujará una línea. En caso contrario no se dibujará nada.

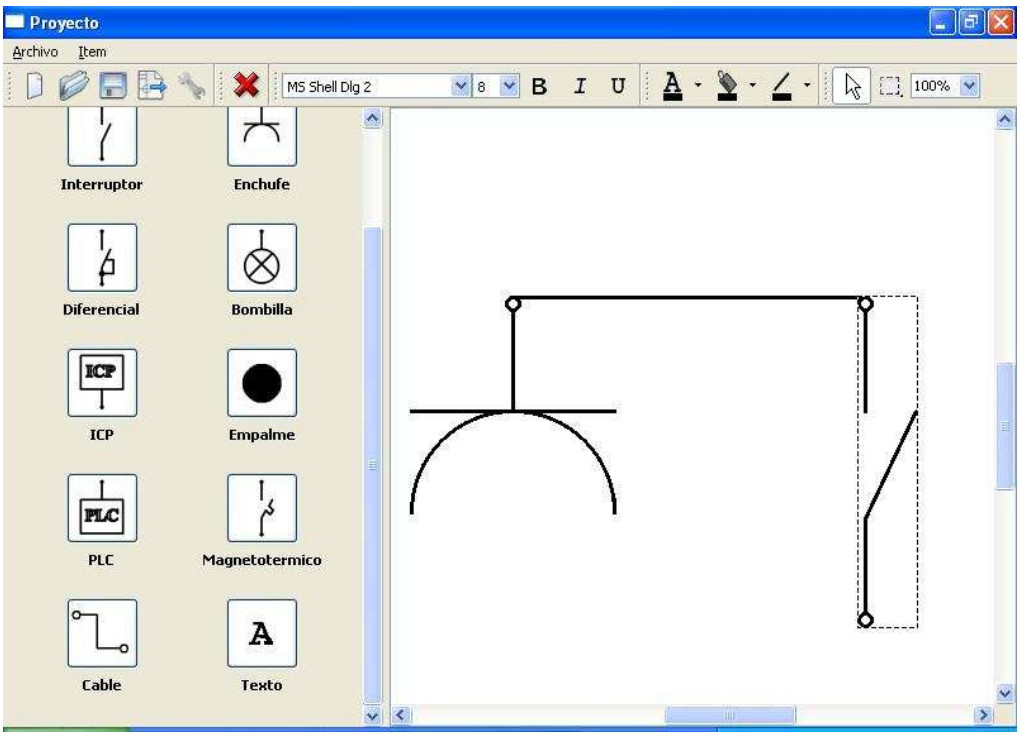


Figura 4.3 Ejemplo de conexión entre dos elementos.

Por último tenemos el botón texto: el único que no es un elemento eléctrico, no contiene parámetros que editar aunque si se puede modificar su formato. Se inserta de igual manera que los elementos eléctricos, pulsándolo y pinchando sobre el área de dibujo.

Un circuito puede convertirse en un diagrama complejo y por ello se necesita de alguna herramienta para poder identificar zonas o elementos eléctricos unos de otros. Tener este tipo de botón es un extra que permite identificar elementos eléctricos sobre el área de dibujo, identificar zonas del circuito o poner comentarios, de manera que facilite al usuario su diseño.

Figura 4.4 Botón texto.

4.2.2. Barra de menús.

La barra de menús, situada en la parte superior de la interfaz, ofrece al usuario las opciones y las acciones necesarias para el manejo del programa. Este menú se divide en dos submenús: *Archivo e Ítem.*

- **Archivo**

En él encontramos las funciones:

- Nuevo
- Abrir
- Guardar
- Guardar Como…
- Exportar
- Salir

-La función *Nuevo* se encarga de preparar la aplicación para empezar un nuevo proyecto. Para ello el proceso consiste en limpiar el área de dibujo, poner todas las variables internas a cero y reiniciar los contadores. Esta función puede enviar una advertencia si el usuario, por ejemplo, no ha guardado previamente su proyecto, en este caso la aplicación advertirá de esto con un mensaje que se compone de una frase y tres botones: “Este documento ha sufrido modificaciones ¿Desea guardar?”; *“OK” ”NO” ”CANCEL”*. Esta función contiene un atajo de teclado si el usuario lo desea, al pulsar “Ctrl.+N” el proceso se ejecuta automáticamente.

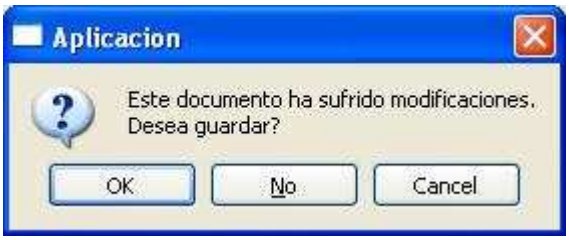


Figura 4.5 Advertencia para guardar un documento.

-La función *Abrir* muestra un cuadro de dialogo que permite al usuario buscar en el disco duro un archivo del tipo *".pfc"* que permite cargar un proyecto ya guardado. Esta extensión es un filtro para representar únicamente los archivos previamente guardados por la aplicación, sin embargo, también se comprueba la cabecera antes de cargar los datos. Esto sirve para evitar cargar datos que no reconozca la aplicación y provocar así inestabilidad en el programa.
Esta función obtiene la información para cargar todas las variables, contadores, ítems y parámetros necesarios para dejar el programa en perfecto funcionamiento y evitar errores futuros de ejecución. Si el proyecto no ha sido guardado se envía un aviso al usuario advirtiendo que el proyecto no ha sido guardado (figura 4.5). El atajo de teclado para esta función es "Ctrl.+A".

-Las funciones *Guardar/Guardar Como…*tienen las dos la misma finalidad, ofrecen la posibilidad de guardar el diseño del circuito eléctrico que el usuario esté realizando. Almacena todas las características de cada ítem como: posición, color, identificador, formato, sus conexiones y sus propiedades. La diferencia entre estas dos funciones es simple, al seleccionar la función *guardar como* aparece un cuadro de diálogo para seleccionar la ruta donde archivar la información.

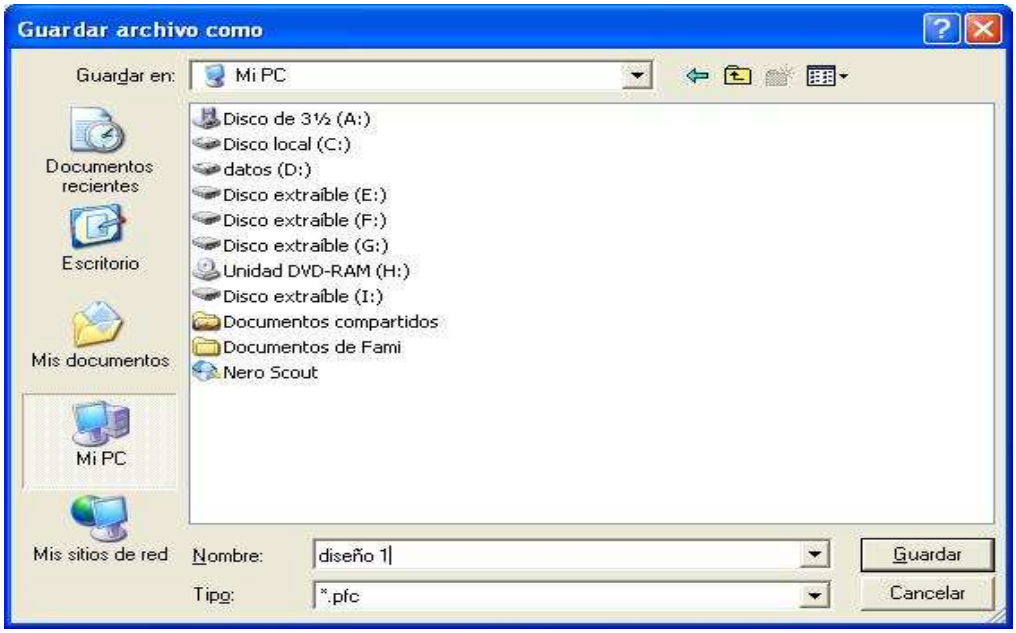


Figura 4.6 Cuadro de diálogo.

En cambio la función *guardar* comprueba la ruta del archivo previamente guardado. Si se guardó el proyecto anteriormente significa que se ha creado una ruta, la cual el programa

utiliza para guardar directamente la información, en caso de que no exista, se utiliza la función *guardar como.* La información se guarda en un archivo de texto, con un encabezado y una extensión del tipo "*.pfc*" para evitar futuros errores de carga. Solo la función *guardar* dispone de atajo de teclado "Ctrl.+G".

-La función *exportar* es una de las más importantes de la aplicación. Su tarea es crear un archivo de texto a partir del circuito diseñado, es decir, esta función traduce el esquema representado en el área de dibujo y las propiedades de cada elemento eléctrico en un archivo de texto. En un circuito eléctrico cada dispositivo está conectado a un conector, gracias a esta característica y que toda la información de los ítems se guarda en una tabla, la aplicación conoce en todo momento qué conexiones hay establecidas entre cables y elementos, por ello la forma que utiliza el programa para exportar el dibujo y las propiedades de cada elemento no es muy compleja. Simplemente recorre la tabla, obtiene los datos necesarios y los graba en el archivo. En el siguiente capítulo se detallará internamente el algoritmo utilizado para este proceso.

En ocasiones, puede ocurrir que algún dispositivo no esté conectado o que no existan elementos en el área de dibujo, para estas situaciones la aplicación devuelve un mensaje advirtiendo de esta anomalía (figura 4.7). El atajo de teclado para ejecutar la función exportar es "Ctrl+E".

Figura 4.7 Advertencia elemento sin conectar/sin elementos.

En el siguiente ejemplo se muestra la conexión de tres enchufes con un empalme, éste a su vez conectado a otro empalme, y una luminaria con un interruptor que conecta con este último. Se utiliza la función exportar.

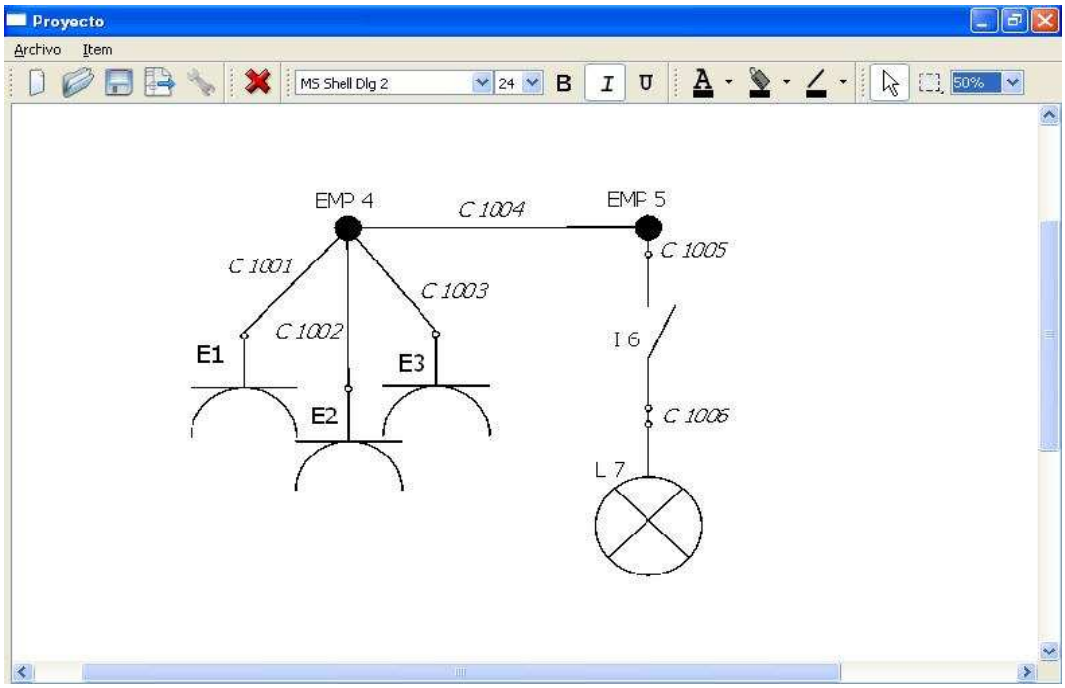


Figura 4.8 Circuito eléctrico sencillo.

A continuación se muestra el archivo de texto generado al exportar un diseño eléctrico:

Enchufe 1: Nombre 0, Potencia 0, Ruido 0, Elemento 0, Horario 0, Conector 1001;
Enchufe 2: Nombre 0, Potencia 0, Ruido 0, Elemento 0, Horario 0, Conector 1002;
Enchufe 3: Nombre 0, Potencia 0, Ruido 0, Elemento 0, Horario 0, Conector 1003;
Empalme 4: Nombre 0, Ruido 0, Tipo 0, Conector 1001, Conector 1002, Conector 1003, Conector 1004;
Empalme 5: Nombre 0, Ruido 0, Tipo 0, Conector 1004, Conector 1005;
Interruptor 6: Nombre 0, Ruido 0, Tipo 0, Conector 1005, Conector 1006;
Luminaria 7: Nombre 0, Potencia 0, Ruido 0, Tipo 0, Horario 0, Conector 1006;
Conector 1001: Nombre 0, Ruido 0, Tipo 0, Grosor 0, Longitud 0, Conexion con: Enchufe 1,Empalme 4;
Conector 1002: Nombre 0, Ruido 0, Tipo 0, Grosor 0, Longitud 0, Conexion con: Empalme 4,Enchufe 2;
Conector 1003: Nombre 0, Ruido 0, Tipo 0, Grosor 0, Longitud 0, Conexion con: Enchufe 3,Empalme 4;
Conector 1004: Nombre 0, Ruido 0, Tipo 0, Grosor 0, Longitud 0, Conexion con: Empalme 4,Empalme 5;
Conector 1005: Nombre 0, Ruido 0, Tipo 0, Grosor 0, Longitud 0, Conexion con: Empalme 5,Interruptor 6;
Conector 1006: Nombre 0, Ruido 0, Tipo 0, Grosor 0, Longitud 0, Conexion con: Luminarias 7,Interruptor 6;

Se puede observar que la aplicación almacena en un archivo de texto todas las conexiones entre los elementos y sus propiedades. Para ejecutar esta función el programa crea un archivo, lo abre y escribe toda esta información utilizando una línea por cada elemento.

Cada componente eléctrico dispone de unas propiedades que el usuario puede editar si lo desea, y que también son exportadas al archivo de texto como se puede justificar. Si una propiedad no ha sido editada, ésta aparece con el valor 0. Un dispositivo eléctrico es reconocido internamente por la aplicación con un identificador único y que podemos apreciar al lado del nombre del dispositivo: *Enchufe 1*, *Luminaria 7* y *Conector 1001*.

Hay que reseñar que el propósito de esta función es traducir toda la información de un diseño eléctrico en un archivo de texto que será de base para otras aplicaciones diseñadas para calcular y simular transmisiones con dispositivos PLC en una vivienda. La

importancia de las conexiones entre elementos y toda la información que cada dispositivo tenga guardado es relevante para llevar a cabo dicho cálculo. Aunque este proceso no será llevado a cabo por la aplicación presentada en este trabajo.

-La función *salir* sirve para abandonar el programa. La aplicación también dispone de un botón de cerrar en la esquina superior derecha que ofrece la misma funcionalidad. Al pulsar estos dos botones, si el usuario no ha guardado el proyecto aparecerá la advertencia anteriormente descrita (ver figura 4.5). Esta función cierra la aplicación y detiene todos los procesos en ejecución. Se puede ejecutar al pulsar en el teclado “Ctrl+S”.

- **Ítem**

Este menú contiene las funciones que hacen referencia a un ítem. Llamamos ítem a cualquier elemento eléctrico o texto que puede ser insertado en el área de dibujo. En este menú encontramos las siguientes funciones:

- Eliminar

- Propiedades

La función *eliminar* es la encargada de borrar del área de dibujo todo Ítem seleccionado por el usuario. Internamente todo lo relacionado con el ítem eliminado será borrado para dejar espacio, optimizar la eficiencia y conseguir mejores prestaciones de la aplicación. Como se ha explicado anteriormente para unir dos elementos se utiliza un cable, pero un cable no puede quedar conectado sólo de un extremo, necesita de otro elemento para poder unirse. Es por esto que al eliminar un elemento todas sus conexiones serán eliminadas automáticamente, de esta manera nunca se puede tener un cable conectado a un extremo y al otro no. Esta función también se ejecuta al pulsar la tecla “suprimir”.

La función *propiedades* ofrece al usuario la posibilidad de editar cada elemento eléctrico. Cada uno se compone de una serie de propiedades. Para editarlas el usuario tiene como herramienta esta función. Puede acceder a ella de tres maneras diferentes: hacer doble click sobre el elemento deseado, utilizar la opción propiedades de un menú contextual que surge al pulsar el botón derecho del ratón y por último con el atajo de teclado “Ctrl+X”.

Al pulsar esta función, asomará en la parte inferior de la aplicación un pequeño panel rectangular compuesto con:

- **Título:** Aparece el nombre del elemento seleccionado.

- **Propiedades:** Aquí se muestran las propiedades del elemento escogido, una etiqueta muestra el nombre de la propiedad y seguidamente una línea de edición de texto se encarga de obtener la información que el usuario edite.

- **Botones:** Se cuenta con tres botones: aceptar, atrás y cerrar. Aceptar guarda la información obtenida en las líneas de edición de texto, el botón *atrás* recupera los valores iniciales, y por último el botón *cerrar* se encarga de ocultar el menú.

Este pequeño panel contiene los mismos botones estándar que podemos encontrar en una ventana de un programa cualquiera. En su esquina superior derecha contiene un botón para

ocultar el panel. A su lado otro botón desacopla el panel de la aplicación de tal manera que se puede mover y cambiar la posición arrastrándolo con el ratón. De esta manera el usuario, si lo desea, puede organizar las secciones de la pantalla principal a su antojo.

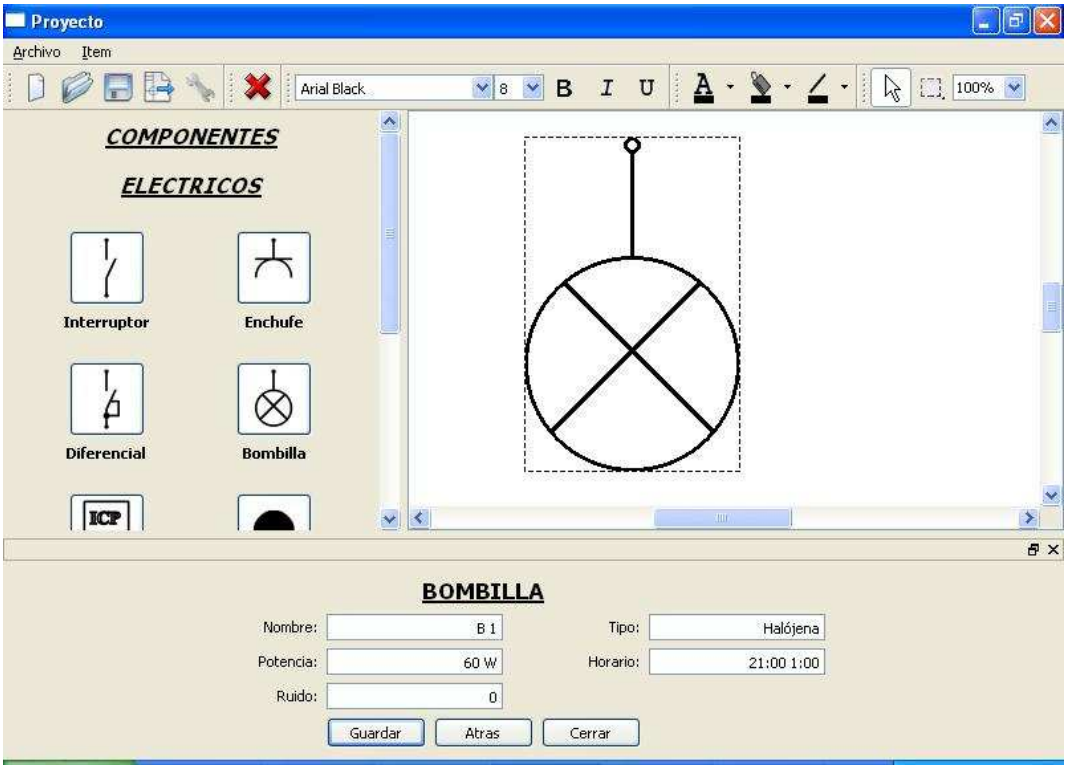


Figura 4.9 Panel para editar las propiedades de cada elemento eléctrico.

La figura 4.9 muestra como el panel inferior aparece en la pantalla principal de la aplicación. Como se puede evidenciar la sección derecha y el área de dibujo se adaptan a la nueva situación de manera transparente al usuario sin que este tenga que cambiar nada, gracias a esto la aplicación tiene un manejo cómodo y sencillo.

Los parámetros de cada elemento son:

Interruptor
Nombre
Ruido
Tipo

Enchufe
Nombre
Potencia
Ruido
Elemento
Horario

Diferencial
Nombre
Ruido
Tipo

Luminaria
Nombre
Potencia
Ruido
Tipo
Horario

ICP (Interruptor de control de potencia)
Nombre
Ruido
Tipo

Empalme
Nombre
Ruido
Tipo

PLC (Power Line Communications)
Nombre
Tipo
Horario

Magnetotérmico
Nombre
Ruido
Tipo

Cable
Nombre
Ruido
Tipo
Grosor
Longitud

Dada la finalidad perseguida en este trabajo (explicada en el capítulo 1) para este PFC únicamente resultan de interés los parámetros mostrados arriba, no obstante, el software creado permitiría añadir fácilmente nuevos parámetros a cada tipo de elemento.

4.2.3. Menú gráfico.

El menú gráfico se encuentra en la parte superior de la aplicación debajo de la barra de menús. Brinda al usuario un panel con iconos accesible directamente desde la parte principal. Está separado por bloques y cada bloque dispone de una serie de funciones y opciones. Esta sección consigue agilidad en las acciones y una vista agradable.

Los bloques tienen la propiedad de que pueden moverse o ser quitados de la aplicación. Se dividen en cinco partes: Archivo, Editar, Fuente, Color y Puntero.

- Bloque Archivo: Se compone de las funciones: Nuevo, Abrir, Guardar y Exportar. Estas funciones también se encuentran para el usuario en el *menú Archivo*. Su funcionalidad es exactamente la misma que la descrita en el apartado *barra de menús*. La función Nuevo se representa con la imagen de una hoja en blanco, la función abrir con una carpeta, guardar con el icono de un disquete y exportar se representa con el diseño de un circuito sobre una hoja y una flecha de color azul apuntando hacia la derecha.

Figura 4.10 Bloque Gráfico Archivo.

- Bloque Editar: Este bloque se compone de la función Eliminar y de la acción Propiedades. Muestra todo lo relacionado con acciones directas sobre un ítem. Para utilizar estas funciones solo hay que seleccionar un elemento y pulsar una de estas opciones. La función eliminar se representa con una equis de color rojo, y la función propiedades con el icono de una llave inglesa.

Figura 4.11 Bloque Gráfico Editar.

- Bloque Fuente: Permite al usuario cambiar el formato del texto. La fuente se obtiene del sistema operativo, es decir, solo se puede elegir la fuente predeterminada en el sistema operativo. Para seleccionar la fuente el usuario puede pinchar en el cuadro de edición y elegir la fuente deseada. El cuadro de edición de tamaño del texto permite modificar el tamaño de los caracteres. Se puede escribir cualquier número entre 8 y 64. El estilo de la fuente varia entre Cursiva, Negrita y Subrayado. Para aplicar estos formatos basta con pinchar sobre los botones especificados. El botón negrita se representa con una “B”, cursiva con una “I” y el subrayado se representa con una “U”.

Figura 4.12 Bloque Gráfico Fuente.

- Bloque Color: Este bloque permite al usuario cambiar el color del texto, o el de un elemento eléctrico. Los colores disponibles son negro, rojo, azul y verde. Para seleccionar un color el usuario dispone de una pestaña al lado de los iconos, esta pestaña extiende un panel con los colores disponibles. Para cambiar el color se dispone de tres botones, un icono con la letra “A” sirve para modificar el color del texto, el botón de un elemento eléctrico se representa

como una botella, y por último para cambiar el color de un cable el botón se representa con una línea.

Figura 4.13 Bloque Gráfico Color.

- Bloque Puntero: Este bloque se compone de tres botones. El usuario tiene la opción de seleccionar el botón puntero que permite seleccionar varios ítems a la vez con el botón agrupar o puede cambiar el zoom del área de dibujo. El botón puntero se representa con la imagen del típico puntero que se conoce actualmente y que se utiliza en todos los sistemas operativos. El botón agrupar tiene como función seleccionar a varios ítems. Se representa con un cuadrado con líneas discontinuas. Para utilizarlo se debe pulsar el botón izquierdo de ratón y arrastrar el ratón por el área de dibujo. De esta forma se creará un área rectangular que seleccionará todo los ítems que queden dentro de este rectángulo. Por último se dispone de una pestaña para cambiar la presentación de la vista del área de dibujo, el usuario dispone de seis opciones: 30%, 50%, 75%, 100%, 150% y 200%.

Figura 4.14 Bloque Gráfico Puntero.

4.2.4. Área de dibujo.

El área de dibujo es una superficie rectangular de dos dimensiones que permite la inserción de elementos gráficos. Contiene en su parte inferior y en el lateral derecho unas barras de desplazamiento para moverse tanto por el eje X como por el eje Y del área de dibujo. Gracias al zoom del que dispone la aplicación, la superficie puede modificar su dimensión, los objetos ser visibles a distintos tamaños, y de esta manera suministrar comodidad al usuario en el momento de crear diseños eléctricos. También permite arrastrar los objetos hacia cualquier posición de la superficie.

El área de dibujo captura todos los movimientos de ratón, de esta manera la aplicación conoce en todo momento en que posición está el ratón y qué elemento tiene por debajo. Para facilitar el manejo del programa se ha incorporado una ayuda gráfica, el ratón cambia de icono cuando está encima de un elemento.

Figura 4.15 Cambio del ratón cuando está sobre un ítem.

El texto tiene la peculiaridad de aparecer siempre por encima de los elementos eléctricos.

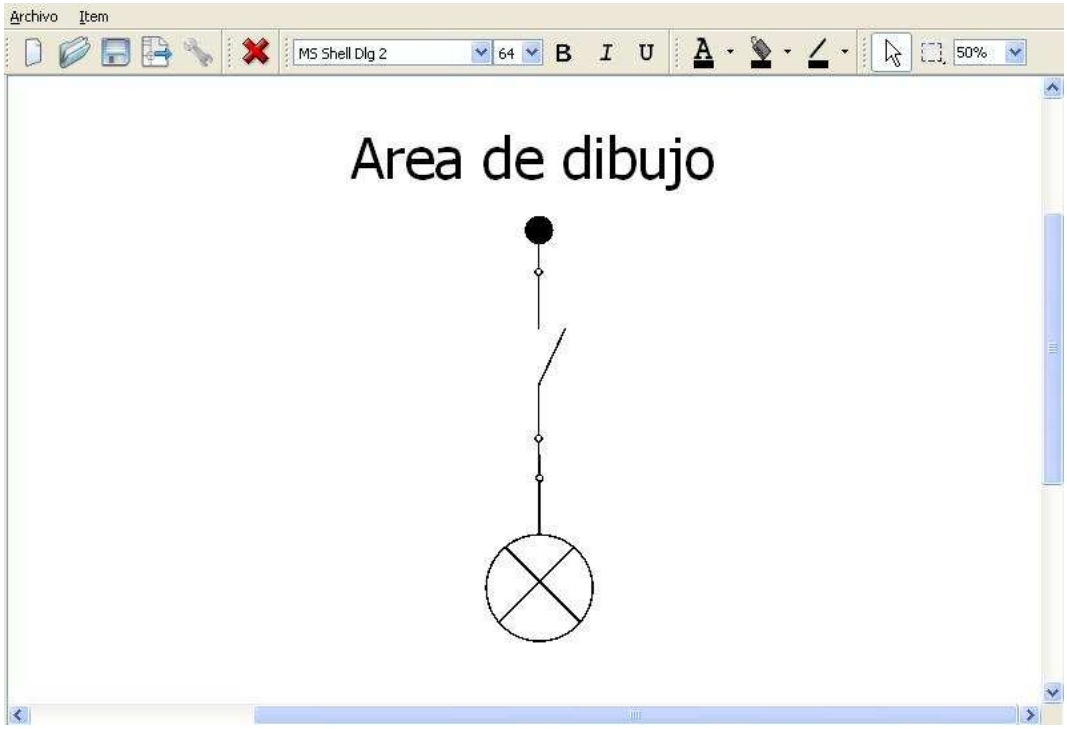


Figura 4.16 Área de dibujo.

Capítulo 5. El código e interioridades

5.1. Introducción

En el presente capitulo se describe el código de la aplicación. Se pretende explicar de manera detallada y ordenada los algoritmos utilizados para la creación de las diferentes clases, objetos y métodos.

Un algoritmo es una secuencia de pasos, instrucciones o reglas bien definidas, ordenadas y finitas que permite realizar una actividad mediante pasos sucesivos que conducen a la realización de una tarea. Los métodos utilizan los algoritmos para transformar un dato de entrada en un dato de salida que puede ser utilizado por otro método o por un objeto.

El código de la aplicación está estructurado en clases, es decir, se utiliza la orientación a objetos para organizar todo el código fuente. La orientación a objetos permite mejoras en la forma de diseño, desarrollo y mantenimiento del software. Con este modo de programación se logran una serie de ventajas:

- Fomenta la reutilización y extensión del código.
- Permite crear sistemas más complejos.
- Portabilidad en el código.
- Facilita el trabajo en equipo.
- Facilita el mantenimiento del software.

5.2. Definición Clase, Objeto y Método

Objeto es el concepto clave de la Programación Orientada a Objetos, la idea es similar a la del mundo real, un objeto puede ser una silla, una mesa. Los objetos tienen dos características: Un estado y un comportamiento. Un coche es un objeto, también tiene un estado: Cantidad de puertas, color, tamaño, etc. y un comportamiento: acelerar, frenar, subir cambio, bajar cambio, girar izq., girar der., etc. Se puede definir a un objeto como un conjunto de atributos (variables) y métodos relacionados. Un objeto mantiene su estado en una o más "variables", e implementa su comportamiento con "métodos". Un método es una función o subrutina asociada a un objeto.

Los objetos dentro de un programa tienen un comportamiento específico y todos sus componentes pueden comunicarse entre ellos.

Un objeto consta de:

• Tiempo de vida: La duración de un objeto en un programa siempre está limitada en el tiempo. La mayoría de los objetos sólo existen durante una parte de la ejecución del programa. Los objetos son creados mediante un mecanismo denominado instanciación (creación de un objeto), y cuando dejan de existir se dice que son destruidos.

• Estado: Todo objeto posee un estado, definido por sus atributos. Con él se definen las propiedades del objeto, y el estado en que se encuentra en un momento determinado de su existencia.

• Comportamiento: Todo objeto ha de presentar una interfaz, definida por sus métodos, para que el resto de objetos que componen los programas puedan interactuar con él.

Una Clase es una construcción que se utiliza como un modelo (o plantilla) para crear objetos de ese tipo. El modelo describe el estado y el comportamiento que todos los objetos de la clase que comparten. Un objeto de una determinada clase se denomina una instancia de la clase. La clase que contiene esa instancia se puede considerar como el tipo de ese objeto, por ejemplo, una instancia del objeto de la clase “Ítem” seria del tipo “Ítem”se define como la generalización de un objeto en particular. Es decir, una clase representa a una familia de objetos concretos.

Una clase encapsula el estado y el comportamiento del concepto que representa, el estado a través de marcadores de datos llamados atributos (variables), y el comportamiento a través de secciones de código reutilizables llamados métodos.

Un método es una subrutina o procedimiento que consiste generalmente en una serie de sentencias para llevar a cabo una acción, un juego de parámetros de entrada que regularan dicha acción y posiblemente, un valor de salida (valor de retorno) de algún tipo. A un método se le da un determinado nombre de tal manera que sea posible ejecutarlas en cualquier momento sin tenerlas que rescribir sino usando sólo su nombre.

La diferencia entre un procedimiento (conocido normalmente como *función*) y un método es que éste último, al estar asociado con un objeto o clase en particular, puede acceder y modificar los datos privados del objeto correspondiente de forma tal que sea consistente con el comportamiento deseado para el mismo. Un método se debe entender no como una secuencia de instrucciones sino como la forma en que el objeto es útil. Por lo tanto, podemos considerar al método como el pedido a un objeto para que realice una tarea determinada o como la vía para enviar un mensaje al objeto y que éste reaccione acorde a dicho mensaje.

5.3. Estructura Editor de Esquemas Eléctricos

Las clases que componen el diseño de la aplicación son: *Main, Mainwindow, Principal, Texto, Ítem, Cable, Conexiones, Cargar, Guardar, Propiedades, Datos y Exportar*.

La aplicación se presenta al usuario en una ventana principal, encargada de sostener los diferentes menús y los botones necesarios para el usuario. Esto es relativamente similar a lo que se puede encontrar en el diseño del código. Para la realización de este proyecto han sido necesarias bastantes líneas de código, por ello, y como anteriormente se ha indicado, el proyecto está diseñado estructuralmente.

En el siguiente diagrama UML se puede apreciar la organización estructural de las distintas clases para la elaboración del proyecto.

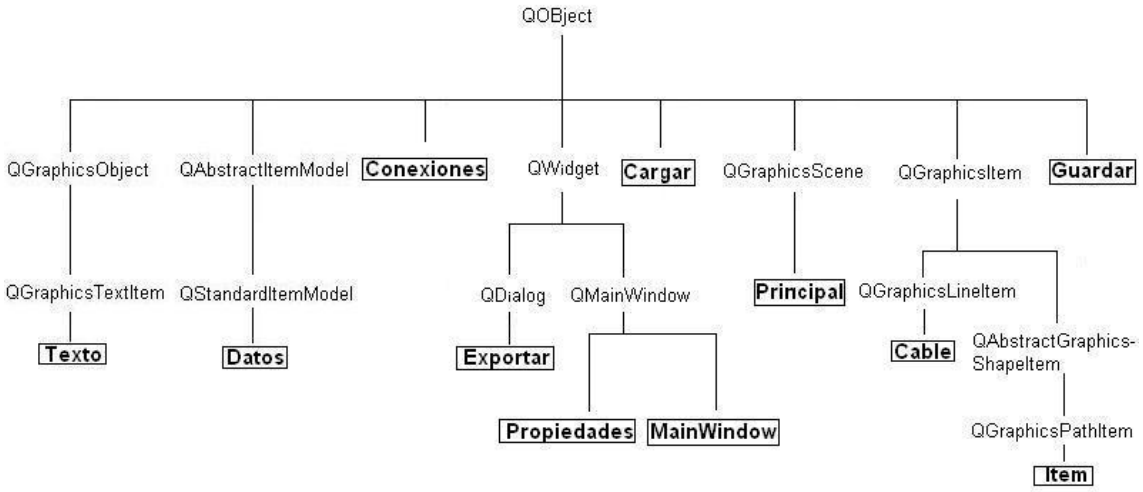


Figura 5.1 Esquema UML del EDEE.

Qt ofrece en su documentación todo tipo de clases. Una de las características es sencillamente la variedad en su librería, ofrece todo tipo de clases para la realización de interfaces gráficas, no obstante para llevar a cabo el código ha sido necesario programar específicamente diferentes tipos de clases. En la figura 5.1 podemos observar la relación entre clases, su herencia y las clases creadas concretamente (resaltadas) para llevar a cabo distintas métodos, algoritmos y encapsulación de código.

La clase *QObject* es la clase base de todos los objetos de Qt. La característica de esta clase es el mecanismo para la comunicación entre objetos conocido con el nombre de señales y *slots*. Los objetos que heredan de *QObject* son llamados objetos hijos y estos a su vez son padres de los objetos que descienden de sí mismos. La herencia es el mecanismo fundamental para implementar la reutilización y extensibilidad del código. Facilita la creación de clases a partir de otras ya existentes, obteniendo características (métodos y atributos) similares a los ya existentes.

Se nombra a cada clase de manera que detalle su función específica en el código. Gracias al sistema de herencia, las librerías Qt ofrecen métodos y variables que facilitan mucho la

tarea de desarrollar código complejo. Algunas clases como: *Conexiones, Guardar y Cargar*, cuelgan directamente de *QObject* ya que no ha sido necesario obtener métodos de clases más específicas, en cambio para las demás clases: *Texto, Datos, Exportar, Propiedades, Mainwindow, Principal, Cable* e *Ítem* se ha necesitado heredar de clases a niveles más bajos, ya que éstas ofrecían los métodos y variables más específicas para la implementación de éstas.

La clase *Main* es la encargada de implementar el método *main*, como se conoce es necesario en lenguajes como C o C++, es el método en el que comienza la ejecución del programa. En el caso del proyecto se encarga de crear un objeto del tipo *MainWindow* y devolver la salida de dicho objeto. Hay que reseñar que *MainWindow* se encarga de sostener prácticamente toda la aplicación.

En este esquema se puede ver de manera esquematizada la relación entre las diferentes clases que componen el EDEE.

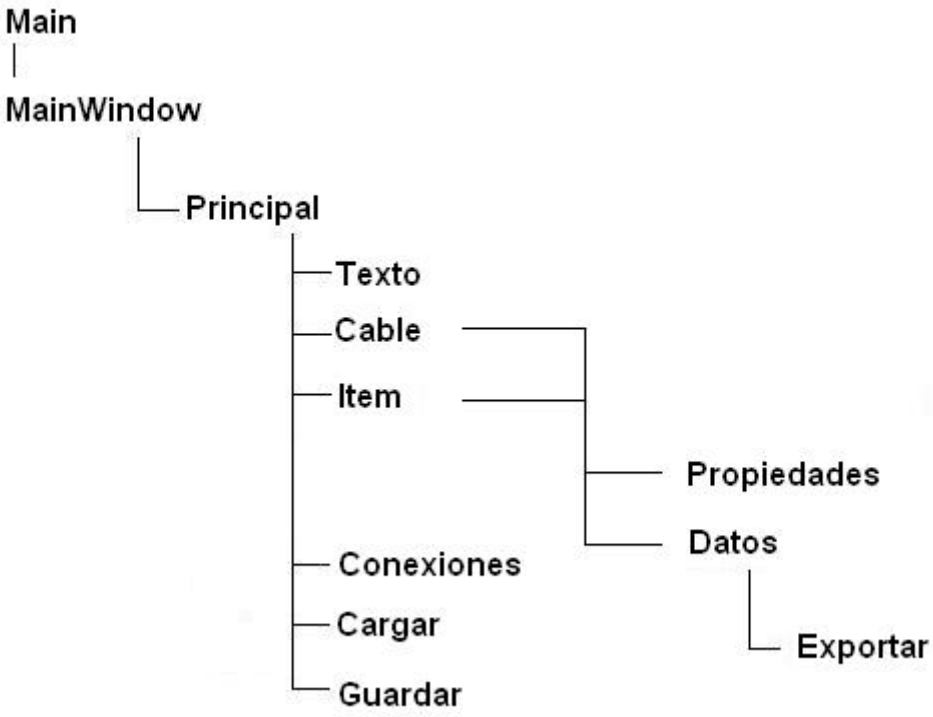


Figura 5.2 Jerarquía entre clases.

5.4. Clase *MainWindow*

La clase *MainWindow* es la base de toda la aplicación, ofrece soporte a las demás clases, se encarga de crear la ventana que sostiene los componentes y agrega funcionalidad para manejar objetos de otras clases. Crea y establece los *widgets* en el código.

Un *widget* en el contexto de la programación de aplicaciones visuales, también conocido como artilugio o control, es un componente gráfico, con el cual el usuario interactúa, como por ejemplo, una ventana, una barra de tareas o una caja de texto.

La clase *MainWindow* establece todos los botones e iconos que se muestran en la parte principal de la aplicación, crea el menú superior, el panel de la izquierda con sus elementos eléctricos y el área de diseño.

Conecta las señales de los objetos con *slots* creados a propósito para la aplicación o simplemente con *slots* que ya vienen predefinidos en las librerías de Qt. Estas conexiones sirven para intercambiar mensajes entre objetos, de esta manera se notifica a un objeto o método de cambio de estado ya que esto puede interesarle a otros objetos.

MainWindow actualiza el estado de la aplicación, al insertar cualquier elemento, mover un Ítem o seleccionar un grupo de estos, los objetos envían una notificación a otro objeto, gracias a esto el sistema está conectado entre sí, aunque el código se compone de doce clases diferentes, la aplicación conoce en todo momento en que estado se encuentra.

```
Public:
    MainWindow();

Protected:
    void closeEvent(QCloseEvent *event);

Private slots:
    void nuevoArchivo();
    void abrir();
    void clickBoton(int id);
    void deleteItem();
    void grupoPunteros(int id);
    void itemInsertado(Item *item,QString var1,QString var2,QString var3,QString
                       var4,QString var5);
    void textoInsertado(QGraphicsTextItem *item);
    void cableInsertado(Cable *cables,QString var1,QString var2,QString var3,
                       QString var4,QString var5);
    void cambiarFuente(const QFont &font);
    void sizeFuente(const QString &size);
    void zoom(const QString &scale);
    void colorTexto();
    void colorItem();
    void colorCable();
    void asignarColorTexto();
    void asignarColorItem();
```

```
    void asignarColorCable();
    void fuente();
    void crearDockWindows();
    void restar_contDockWidget(int id);
    void exportar_Diagrama();
    bool guardar();
    bool guardar_como();

Private:
   void crearAcciones();
   void crearPaleta();
   void crearMenus();
   void crearToolBars();
   bool modificado();
   bool guardarArchivo(const QString &nombreArchivo);
   QWidget *botonItem(const QString &text,
                      Item::TipoDiagrama tipo);
   QWidget *botonCable(const QString &texto);
   QMenu *createColorMenu(const char *slot, QColor defaultColor);
   QIcon crearColorIcono(QColor color);
   QIcon botonToolIconos(const QString &imagen, QColor color);
```

Figura 5.3 Métodos de la clase *MainWindow*.

A continuación se detallará los métodos que componen esta clase. Como se ha advertido anteriormente, *MainWindow* es la base que sostiene toda la aplicación. Por norma general todas las clases se componen de métodos públicos, protegidos, privados y *slots*.

5.4.1. Constructor

```
MainWindow::MainWindow()
 {
     crearAcciones();
     crearPaleta();
     crearMenus();
     scene = new Principal(itemMenu);
     scene->setSceneRect(QRectF(0, 0, 2000, 2000));
     connect(scene,                                  SIGNAL(itemInsertado(Item
*,QString,QString,QString,QString,QString )),
                  this,                               SLOT(itemInsertado(Item
*,QString,QString,QString,QString,QString)));
     connect(scene, SIGNAL(textoInsertado(QGraphicsTextItem *)),
               this, SLOT(textoInsertado(QGraphicsTextItem *)));
     connect(scene,                                 SIGNAL(cableInsertado(Cable
*,QString,QString,QString,QString,QString)),
                  this,                               SLOT(cableInsertado(Cable
*,QString,QString,QString,QString,QString)));
     connect(scene, SIGNAL(abrirPropiedades()),
                  this, SLOT (crearDockWindows()));
     crearToolBars();
     QSplitter *layout=new QSplitter();
     layout->addWidget(menu_izquierda);
     vista = new QGraphicsView(scene);
     layout->addWidget(vista);
```

```
    setCentralWidget(layout);
    setWindowTitle(tr("Editor de Esquemas Eléctricos"));
    setUnifiedTitleAndToolBarOnMac(true);
    for(int i=0;i<2;i++){
        pro[i]=NULL;
    }
    model=new Datos();
}
```

El constructor es un método miembro especial que sirve para inicializar un objeto de la clase al mismo tiempo que la declara. Llama a los métodos *crearAcciones(), crearPaleta(), crearMenus()* con el objetivo de crear los *widgets* y colocarlos antes de llamar al objeto "*scene*" que se encarga de crear el área de diseño.

Se conectan las señales *itemInsertado (), textoInsertado ()* y *cableInsertado ()* para indicar que un elemento ha sido insertado en el área de diseño y comunicar de esta manera al sistema. La señal *abrirPropiedades()* se conecta al *slot crearDockWindows()* para abrir las propiedades al hacer dos clicks en el área de diseño sobre un elemento eléctrico. La barra de herramientas se crea después del área de diseño ya que se conecta a señales propias de la clase *Principal*.

Se llama al método *crearToolBars()* que es el encargado de realizar el menú gráfico, se establece la ventana del EDEE poniendo la paleta a la izquierda y el resto de la ventana para el área de diseño.

Por último se crea la tabla de datos con el nombre *model*, encargada de guardar toda la información de cada uno de los elementos, además de otras opciones internas que utiliza el sistema para funcionar de manera eficiente.

5.4.2 Protected

En la herencia, las clases derivadas "heredan" los datos y los métodos miembro de las clases base, pudiendo las clases derivadas redefinir estos comportamientos (polimorfismo) y añadir comportamientos nuevos propios de las clases derivadas. El concepto de método protegido es referente a la herencia, es decir, todo método que no hereda de esta clase obtiene este tipo de dato como privado pero es público para aquellas clases que sí derivan de la clase que se ha definido el método como protegido.

En *MainWindow* se utiliza este método:

```
void closeEvent(QCloseEvent *event);
```

Cierra la aplicación al pinchar el usuario el botón "X" de la esquina superior derecha, este método envía al *slot close()* una señal indicando el suceso.

5.4.3 Private Slots

Los *slots* con casi idénticos al método miembro de C++. Pueden ser virtuales, se puede sobrecargar, pueden ser públicos, protegidos o privados, y pueden ser llamadas como otro método de C++ y sus parámetros pueden ser de cualquier tipo válido. La diferencia es que un *slot* se puede conectar a una señal, en cuyo caso se llama automáticamente cada vez que se emite la señal.

```
void nuevoArchivo();
```

Se llama a este *slot* si el usuario desea empezar un nuevo diseño eléctrico. Primero se comprueba si existen modificaciones (llama al método *modificado()*) en el diseño actual, si no es el caso reestablece todas las variables, limpia el área de diseño y restaura la tabla de datos.

```
void abrir();
```

El *slot abrir()* se invoca cuando el usuario pincha sobre la acción "abrir". Se comprueba antes de nada si existen variaciones en el diseño (método *modificado()*). Aparece un cuadro de dialogo que pide al usuario que elija un archivo. Si el usuario elige un archivo (es decir, el nombre del archivo no es una cadena vacía), se crea un objeto de la clase C*argar* para recuperar los datos del archivo.

```
void clickBoton(int id);
void grupoPunteros(int id);
```

Estos dos *slots* se ejecutan cuando el usuario pincha sobre un botón de la paleta (desea insertar un ítem) o pincha sobre el grupo de botones de selección. Cada botón tiene un id que lo diferencia de los demás, de esta manera el sistema reconoce que tipo de botón se pulsa. Si es uno de selección la aplicación cambiará a modo moverItem, en cambio si pulsa un botón de la paleta la aplicación cambia al modo insertarItem. Los cambios de estados afectan más a la clase principal que es la encargada de dar soporte al área de diseño.

```
void deleteItem();
```

Elimina cualquier ítem, cable o texto. Recorre en un bucle el área de diseño en busca de cualquier elemento seleccionado. Si es un texto se elimina directamente. Si es un cable, se actualiza las conexiones de los ítems a los que está conectado, se elimina la entrada en la tabla de datos y se borra del área de diseño el cable. Un ítem antes de ser eliminado llama a un método interno de la clase *Ítem* que se encarga de borrar todos los cables a los que está conectado el ítem en cuestión, después de este proceso se borra el ítem.

```
void itemInsertado(Item *item,QString var1,QString var2,QString var3,QString
var4,QString var5);
void textoInsertado(QGraphicsTextItem *item);
void cableInsertado(Cable *cables,QString var1,QString var2,QString var3,
QString var4,QString var5);
```

Estos *slots* se activan al insertar en el área de diseño cualquier ítem. En el caso de *itemInsertado(Item *item, QString var1, QString var2…..),* comprueba antes de almacenar

los datos que tipo de ítem(puede ser Enchufe, Interruptor, ICP etc….) es el insertado. La información se almacena en una tabla de datos (la clase *Datos* se encarga de este procedimiento), esta información se utiliza para las propiedades de cada elemento eléctrico. *TextoIsertado(QGraphicsTextItem *item)* simplemente se encarga de cambiar el estado de la aplicación a moverItem no guarda nada en la tabla. *CableInsertado(Cable *cables,QString var1,QString var2,…..)* invoca al método del objeto *Datos* encargado de guardar los datos en la tabla.

```
void cambiarFuente(const QFont &font);
void sizeFuente(const QString &size);
void fuente();
```

Cuando el usuario solicita el cambio de fuente en el texto, pinchando en los botones disponibles en el menú gráfico, se crea un objeto *QFont* y establece las propiedades para que coincida con el estado del widget.

```
void zoom(const QString &scale);
```

Zoom cambia la escala en el área de diseño. Este *slot* se activa al pulsar el usuario sobre el bloque de escalado en el menú gráfico.

```
void colorTexto();
void colorItem();
void colorCable();
void asignarColorTexto();
void asignarColorItem();
void asignarColorCable();
```

Estos *slots* manejan la solicitud por parte del usuario para cambiar el color de un ítem, cable y texto.

```
void crearDockWindows();
void restar_contDockWidget(int id);
```

El *slot crearDockWindows()* está conectado a la señal propiedades y al objeto *Scene*(encargado del área de diseño). Al ejecutarse este *slot,* se comprueba que no hay más de dos ítems seleccionados, en caso de ser así, se devuelve una advertencia al usuario indicando que debe seleccionar 1 o 2 ítems a la vez, para esto se utiliza un contador. Comprueba que tipo de ítem esta seleccionado crea un objeto de la clase *Propiedades*

Cuando se crea un menú, éste se conecta al *slot restar_contDockWidget(int id).* El usuario puede abrir un máximo de dos menús inferiores, si al tener uno abierto se desea abrir otro, este nuevo menú se colocará en la parte derecha, en caso de querer abrir un nuevo menú. Si al tener dos menús abiertos se intenta abrir uno nuevo, el menú que desaparece es el de la derecha para dejar espacio al nuevo. Para todo este proceso se utiliza un pequeño array de dos posiciones, en cada una se guarda el puntero que apunta a un menú abierto. En un bucle se comprobará qué posición del array está vacía y cuál no. En la vacía se colocará el puntero del nuevo menú, siempre por supuesto de manera ordenada. Por ejemplo:
Se nombra al array “pro” y tendría este aspecto.

Pro[vacia][vacia]. En memoria al inicializar este array queda de esta manera.

Al abrir un menú.

Pro[ocupada][vacia].

Se abre un segundo menú

Pro[ocupada][ocupada].

Si se elimina el menú de la izquierda.

Pro[vacia][ocupada].

Pro[ocupada][vacia]. En este caso se mueve la posición de la derecha a la izquierda, dejando el array de esta manera.

Con este algoritmo conseguimos una manera cómoda y sencilla para el usuario, ya que siempre va a tener dos menús como mucho abiertos y puede comparar resultados entre cables e ítems. Este *slot* contiene un algoritmo que proporciona la ordenación del array "pro" moviendo las posiciones anteriormente descritas. Este *slot* se ejecuta cuando el usuario pincha sobre el botón cerrar del menú inferior o también cuando pincha sobre la "x" del menú.

```
void exportar_Diagrama();
```

Conectado a la señal exportar, se ejecuta al pinchar el usuario sobre la acción exportar. Antes de atender la petición comprueba que haya elementos en el área de diseño o que estén todos conectados, después se abre un cuadro de diálogo para que el usuario pueda guardar el archivo de texto. Por último se crea un objeto de la clase *Exportar*, esta clase es la encargada de realizar todo el proceso de exportación, se explicará en el aparado de esta clase.

```
bool guardar();
bool guardar_como();
```

Estos dos *slots* tienen la misma funcionalidad. Su función es comprobar que el diseño actual no se ha modificado y llamar al método privado *guardarArchivo(const QString &nombreArchivo).*

5.4.4. Private

Un método privado solo es accesible a través de la propia clase, y no desde otra clase.

```
void crearAcciones();
```

Este método es invocado en el constructor. Su objetivo es crear todas las acciones relacionadas con el EDEE: Nuevo, Abrir, Guardar, Guardar Como, Salir, Eliminar,

Propiedades, Exportar. Acciones que se relacionan con el texto: Negrita, Cursiva y Subrayado. Una acción proporciona una operación que se puede insertar en los *widgets*. Las acciones se agregan al menú superior y a las barras de herramientas manteniéndolos sincronizados. Conecta las señales de cada acción a los *slots* correspondientes y agrega un icono que servirá de ayuda para el usuario y el EDEE tendrá un aspecto agradable. Por último inserta un atajo de teclado para cada acción.

```
void crearPaleta();
```

Establece en la parte de la izquierda de el EDEE el panel con los botones de todos los ítems, cable y texto. Invoca a los métodos *botonItem(const QString &text,Item::TipoDiagrama tipo)* y *botonCable(const QString &texto).*

```
void crearMenus();
```

Proporciona al EDEE una barra de menús en su parte superior. Las acciones creadas en la llamada al método *crearAcciones()* se añaden a este menú.

```
void crearToolBars();
```

CrearToolbars() establece el menú gráfico. Coloca los cinco bloques: Archivo, Editar, Fuente, Color y Puntero. El bloque Archivo y Editar agrega las acciones ya creadas anteriormente, en cambio los demás bloques tiene que crear sus botones, configurarse y conectarse a un *slot.*

```
bool modificado();
```

Devuelve ***true*** o ***false*** si el usuario ha modificado el diseño eléctrico desde el último guardado**.** Para este proceso se comprueba si la tabla de datos ha sufrido modificaciones (nuevos ítems o cables insertados). Existe la variable *modificado* que indica si hay un cambio en el diseño.

```
bool guardarArchivo(const QString &nombreArchivo);
```

Se recibe el nombre del archivo. Se almacena el nombre en una variable que indica la ruta del archivo actual y un objeto del tipo *Guardar* realiza todo el proceso para escribir los datos dentro del archivo y poder ser recuperados posteriormente.

```
QWidget *botonItem(const QString &text,Item::TipoDiagrama tipo);
QWidget *botonCable(const QString &texto);
```

Estos metodos crean un *widget* con sus iconos y una etiqueta que lo identifica. Los *widgets* se devuelven al método que los invocó, en este caso dentro del método *crearPaleta()* que hace las llamadas a estos métodos para ir creando los botones.

```
QMenu *createColorMenu(const char *slot, QColor defaultColor);
```

Se establece el botón de color en el menú de herramientas. Se crean los colores (Negro, Azul, Verde y Rojo) que el usuario puede seleccionar para cada ítem, cable o texto. El usuario al pulsar sobre un color se establece en el elemento seleccionado.

```
QIcon crearColorIcono(QColor color);
QIcon botonToolIconos(const QString &imagen, QColor color);
```

Dibujan el rectángulo donde se muestra el color.

5.5. Clase *Principal*

La clase *Principal* hereda de *QGraphicsScene* y es la encargada de crear el área de diseño. Proporciona una superficie en 2D para dibujar los Ítems, Cables y el Texto. Suministra funcionalidad que permite determinar de manera eficiente las posiciones de los elementos y cuales son visibles dentro de un espacio arbitrario de la escena. Con *QGraphicsView* se puede visualizar toda el área de diseño, ampliar o solo ver partes de ella.

Dentro de *Principal* con un clic de ratón puede darse cinco acciones diferentes: mover un ítem, insertar un ítem, insertar un texto, conectar un cable y por último seleccionar un grupo de ítems. Estas acciones dependen del estado del EDEE en ese momento, para ello se inicializa una variable con esos cinco posibles estados. Para modificar la variable se llama al método *setModo().*

Esta clase establece el color de los ítems, cables y del texto. El color y la fuente del texto se puede ajustar con los métodos *setLineaColor (), setTextoColor (), setItemColor () y setFont ().* Al insertar un elemento dentro del área de dibujo, se utiliza el método *setItemTipo()* para establecer que tipo de ítem se ha insertado.

Cuando el usuario desea cargar un fichero con un diseño guardado, esta clase contiene un método encargado de ejecutar todas las instrucciones para llevar a cabo dicho proceso *cargar_items().*

Cada ítem y Cable contiene un identificador que se llama id, con el cual el sistema reconoce en todo momento a cada ítem o cable, como si fuera un DNI. Este id se actualiza en esta clase al insertar un elemento. El texto no necesita tener un id ya que la información que contiene no es relevante para el funcionamiento del EDEE.

Lo más importante que puede gestionar esta clase son los eventos que suceden con el ratón. Al entrar el ratón en el área de diseño, o se posa encima de un elemento se captura dichos eventos para procesarlos. Todo esto ya viene implementado en las librerías de Qt.

```
Public:
    Principal(QMenu *itemMenu, QObject *parent = 0);
    enum Modo { InsertarItem, InsertarCable, InsertarTexto, MoverItem,Seleccionar };
    void setCableColor(const QColor &color);
    void setTextoColor(const QColor &color);
    void setItemColor(const QColor &color);
    void setFont(const QFont &font);
    bool elemento(int type);
    void cargar_items(QStringList list);
    int limpiar_escena();
    void valores_iniciales();

 Public Slots:
    void setModo(Modo actualizar);
    void setItemTipo(Item::TipoDiagrama type);
    void editorLostFocus(Texto *item);

Signals:
    void itemInsertado(Item *item,QString var1,QString var2,QString var3,
                QString var4,QString var5);
    void textoInsertado(QGraphicsTextItem *item);
    void cableInsertado(Cable *arrow,QString var1,QString var2,QString var3,
                 QString var4,QString var5);
    void abrirPropiedades();

 Protected:
    void mousePressEvent(QGraphicsSceneMouseEvent *mouseEvent);
    void mouseMoveEvent(QGraphicsSceneMouseEvent *mouseEvent);
    void mouseReleaseEvent(QGraphicsSceneMouseEvent *mouseEvent);
    void mouseDoubleClickEvent ( QGraphicsSceneMouseEvent *mouseEvent);
    int id_max(int id_actual,int id_archivo);
```

Figura 5.4 Métodos de la clase Principal

5.5.1 Constructor

```
Principal::Principal(QMenu *itemMenu, QObject *parent)
     : QGraphicsScene(parent)
 {
     myItemMenu = itemMenu;
     modoActual = MoverItem;
     myItemType = Item::Enchufe;
     cable = 0;
     re=0;
     textItem = 0;
     myItemColor =Qt::black;
     myTextoColor = Qt::black;
     myCableColor = Qt::black;
 }
```

En el constructor se recibe myItemMenu para utilizar el menú contextual cuando se crea un ítem o un cable. El modo por defecto es "modoActual:MoverItem" ya que es el estado predeterminado de *QGraphicsScene*. Se inicializan las variables: myItemType, re, cable y texItem. Y por último el color de cada elemento.

```
enum Modo { InsertarItem, InsertarCable, InsertarTexto, MoverItem,Seleccionar };
```

Modo es una enumeración. Una enumeración es un conjunto de constantes enteras. A la enumeración se le puede asignar un nombre, que se comportará como un nuevo tipo de dato que solo podrá contener los valores especificados en la enumeración. Se utiliza para cambiar el estado del EDEE.

5.5.2. Public

```
void setCableColor(const QColor &color);
void setTextoColor(const QColor &color);
void setItemColor(const QColor &color);
void setFont(const QFont &font);
```

Estos métodos reciben una variable que indica que tipo de color establecer en el ítem, cable o el texto. *SetFont(const QFont &font)* modifica el tipo de fuente en el área de dibujo.

```
bool elemento(int type);
```

Este método evita errores durante la ejecución del EDEE. Devuelve ***true*** o ***false*** indicando si el elemento seleccionado es un ítem, un cable o un texto, es decir, si se desea cambiar el color del texto, este método nos indica si el elemento seleccionado es un texto.

```
void cargar_items(QStringList list);
```

El objetivo de este método es cargar todos los datos que se recibe en la variable list(vector de información), en ella se encuentra toda la información para reestablecer los datos. Se ejecuta cuando el usuario desea cargar un archivo. Al entrar a este método se comprueba que tipo de elemento se inserta. Por ejemplo:

- Si es un Ítem: Se extrae el id, el color, la posición en el área de dibujo, las conexiones que tiene establecidas y por último se guarda la información de sus propiedades en la tabla de datos.

- Si es un Cable: Obtiene el id, el color a que dos ítems esta conectado y finaliza con la información de sus propiedades.

- Si es un texto: El texto cambia un poco con respecto al ítem y el cable. Extrae la frase del texto, su color, su posición, su tipo de fuente, el tamaño, si está en cursiva, negrita o subrayado.

```
int limpiar_escena();
void valores_iniciales();
```

Estos métodos se utilizan cuando el usuario desea empezar un nuevo proyecto sin cerrar el programa, lo que viene a ser la acción "Nuevo". *Limpiar_escena()* borra todos los elementos que hay en el área de diseño y *valores_iniciales()* restaura los id´s de los ítems y de los cables.

5.5.3. Public Slots

```
void setModo(Modo actualizar);
void setItemTipo(Item::TipoDiagrama type);
```

Estos *slots* los utiliza la clase *Principal* para actualizar su estado y conocer que tipo de ítem es seleccionado por el usuario. *SetModo()* actualiza el estado del EDEE. *SetItem()* guarda en una variable el tipo de ítem elegido por el usuario.

```
void editorLostFocus(Texto *item);
```

Este *slot* tiene como objetivo ofrecer la opción al usuario de una vez insertado un texto poder seleccionarlo.

5.5.4. Signals

```
void itemInsertado(Item *item,QString var1,QString var2,QString var3,
                   QString var4,QString var5);
void textoInsertado(QGraphicsTextItem *item);
void cableInsertado(Cable *arrow,QString var1,QString var2,QString var3,
                    QString var4,QString var5);
void abrirPropiedades();
```

Para simplificar la comunicación entre objetos, se debe tener alguna forma de conocer cuando el usuario pulsa un botón, selecciona un ítem de un combo o inserta texto. Por ello Qt proporciona un mecanismo basado en *Signals* y *Slots*. La idea es, cuando se produce un evento en un objeto, éste puede emitir una señal. Esa señal se puede conectar con un *slot*.

Las señales *itemInsertado(), textoInsertado() y cableInsertado()* tienen la misma misión. Se emiten siempre que un elemento es colocado en el área de dibujo. Puede ser de manera manual por el usuario o cuando se carga un archivo que se hace de manera automática. En el caso de *itemInsertado() y cableInsertado()* se envía la dirección del elemento y toda la información de sus propiedades. La clase *MainWindow* se encarga de conectar estas señales con sus *slots*, aunque estos tengan el mismo nombre, esto es simplemente para hacer más fácil el código.

La señal *abrirPropiedades()* se conecta al *slot crearDockWindows()* en la clase *ManiWindow,* pero es emitida en esta clase en el método *mouseDoubleClickEvent().*

5.5.5. Protected

```
void mousePressEvent(QGraphicsSceneMouseEvent *mouseEvent);
```

```
void mouseMoveEvent(QGraphicsSceneMouseEvent *mouseEvent);
void mouseReleaseEvent(QGraphicsSceneMouseEvent *mouseEvent);
void mouseDoubleClickEvent ( QGraphicsSceneMouseEvent *mouseEvent);
```

Estos métodos están reimplementados, esto significa que se han modificado para nuestro interés:

- *MousePressEvent()* captura la posición de ratón y al hacer clic dibuja en el área de dibujo el elemento seleccionado.

- *MouseMoveEvent()* este método se activa al pasar con el ratón por encima de un elemento, al pulsar el botón se puede mover un elemento.

- *MouseReleaseEvent()* se active al pulsar un botón de ratón y arrastrar simultáneamente. Este método dibuja el cable y lo conecta en los ítems. Además actualiza las conexiones de cada ítem.

- *MouseDoubleClickEvent()* al hacer doble clic sobre un elemento este método emite la señal *abrirPropiedades().*

```
int id_max(int id_actual,int id_archivo);
```

Devuelve un número entero, que es el máximo entre id_actual e id_archivo. Al cargar los datos de un archivo, se actualizan las variables encargadas de llevar la cuenta de cada id tanto de un ítem como el de un cable. Este método evita que al cargar un archivo puedan coincidir dos id´s iguales, esto provocaría inestabilidad en el programa y no funcionaría correctamente.

5.6. Clase *Ítem*

La clase *Ítem* representa una imagen en el área de diseño (clase *Principal*). Hereda de *QGraphicsPathItem* que proporciona propiedades para dibujar una figura y darle color.

Esta clase contiene tres identificadores diferentes pero muy importantes:

- Identificador "Type" que ofrece la librería Qt, este identificador sirve para diferenciar clases, también se utiliza en la clase cable y texto. De esta manera al utilizar un bucle for() o un bucle foreach() con este identificador se sabe a que tipo de clase se accede.

- La enumeración "TipoElemento" contiene los distintos tipos de ítems que puede utilizar el usuario: Enchufe, Interruptor, Pias, Luminarias, PLC, Empalmes, ICP y Magnetotérmicos. Cuando se crea un objeto de esta clase se inicializa con el tipo de elemento elegido.

- Identificador "id", como se ha explicado anteriormente es como un DNI, identifica a cada ítem y cable de manera única durante la ejecución del programa.

Se dispone de una lista donde se almacenan los cables que están conectados a un ítem, esto es necesario ya que un ítem debe conocer cuando cambia sus conexiones. Igualmente esta clase dispone de dos tipos de conexiones, es decir hay diferentes ítems unos tienen una conexión y otros dos, en ese caso se necesita de dos variables que indiquen que conexión tiene conectado un cable y cual no.

Public:

```
Item(TipoElemento tipoelemento, QMenu *contextMenu,int identificador,
    QColor color,bool posA, bool posB,QGraphicsItem *parent = 0, QGraphicsScene
*scene = 0);
enum { Type = UserType + 15 };
enum TipoElemento { Enchufe, Interruptor, Pias, Luminarias, PLC,
    Empalmes,ICP,Magnetotermicos };
void eliCable(Cable *f);
void eliCa();
TipoElemento tipoItem() const { return myDibujo; }
int getId() const{ return id; }
bool getConexionA() const { return conexionA; }
bool getConexionB() const { return conexionB; }
int getConexiones() const { return conexiones; }
void setConexiones(const int c) { conexiones=conexiones+c; }
void setConexionA(bool var) { conexionA=var; }
void setConexionB(bool var) { conexionB=var; }
void setColor(const QColor &color){ myColor = color; }
QColor getColor() { return myColor; }
QPixmap imagen(TipoElemento tipoelemento) const;
void addCable(Cable *f);
int type() const { return Type;}
```

Protected:

```
void contextMenuEvent(QGraphicsSceneContextMenuEvent *event);
QVariant itemChange(GraphicsItemChange change, const QVariant &value);
```

Figura 5.5 Métodos de la clase Ítem

5.6.1. Constructor

```
Item::Item(TipoElemento tipoelemento, QMenu *contextMenu,int
identificador,
           QColor color,bool posA, bool posB,QGraphicsItem *parent,
QGraphicsScene *scene)
     : QGraphicsPathItem(parent, scene)
 {
     myDibujo = tipoelemento;
     myContextMenu = contextMenu;
     myColor = color;
     id=identificador;
     conexiones=0;
     conexionA=posA;
     conexionB=posB;
  if(myDibujo==Interruptor){
      path.moveTo(0.0, 200.0);
      path.lineTo(0.0, 175.0);
      path.lineTo(40.0, 90.0);
      path.moveTo(0.0,10.0);
      path.lineTo(0.0, 90.0);
      path.moveTo(0.0,0.0);
```

```
        path.arcTo(-5.0, 0.0, 10.0, 10.0, 90.0, 360.0);
        path.moveTo(0.0,200.0);
        path.arcTo(-5.0, 250.0, 10.0, 10.0, 90.0, 360.0);
        setPen(QPen(myColor, 3, Qt::SolidLine, Qt::RoundCap,
                    Qt::RoundJoin));
        setPath(path);
        setFlag(QGraphicsItem::ItemIsMovable, true);
        setFlag(QGraphicsItem::ItemIsSelectable, true);
        setCursor(Qt::PointingHandCursor);

    }else if(myDibujo==Enchufe){
                    .
                    .
                    .
                    .
                    .



   }
}
```

En el constructor se crea la imagen del ítem seleccionado por el usuario. Se inicializan las variables, sus conexiones (pasA y posB) y con un "if" se comprueba que tipo de item hay que dibujar. Para que los ítems tengan la opción de moverse y seleccionarse se activa unos flags con: *setFlag(QGraphicsItem::ItemIsMovable)* y *setFlag(QGraphicsItem::ItemIsSelectable)*. Por último *setCursor(Qt::PointingHandCursor)* active un flag que cambia el icono del puntero cuando éste se posa encima de un ítem.

```
enum { Type = UserType + 15 };
enum TipoElemento{Enchufe, Interruptor, Pias, Luminarias,
                  PLC,Empalmes,ICP,Magnetotermicos };
```

Type identifica esta clase con respecto a otras clases. TipoElemento enumera los diferentes tipos de ítems que se pueden seleccionar.

5.6.2. Public

```
void eliCable(Cable *f);
void eliCa();
```

Estos métodos se utilizan para quitar los cables que tiene conectados cada ítem ya que un ítem tiene una lista de cables conectados a el. Este método es público ya que se accede desde *MainWindow*.

```
TipoElemento tipoItem() const { return myDibujo; }
```

Este método retorna el tipo de ítem seleccionado como por ejemplo: Enchufe, Interruptor, PLC, ICP etc…

```
int getId() const{ return id; }
```

Devuelve el id que identifica a cada ítem durante la ejecución de la aplicación. Este id se utiliza en la tabla de datos.

```
bool getConexionA() const { return conexionA; }
bool getConexionB() const { return conexionB; }
int getConexiones() const { return conexiones; }
void setConexiones(const int c) { conexiones=conexiones+c; }
void setConexionA(bool var) { conexionA=var; }
void setConexionB(bool var) { conexionB=var; }
```

Los métodos *getConexionA()* y *getConexionB()* devuelven un booleano, es decir, si hay conexión en el conector A o B se devuelve ***true***, sino se devuelve ***false***. De esta manera se conoce cual de las dos conexiones esta libre (en caso de que el ítem admita dos conexiones). El método *getConexiones()* indica el número de conexiones que tiene establecido el ítem. Los métodos que tienen delante un set establecen los valores.

```
void setColor(const QColor &color){ myColor = color; }
QColor getColor() { return myColor; }
```

Establece el color del ítem, el método *getColor()* devuelve una variable del tipo Color.

```
QPixmap imagen(TipoElemento tipoelemento) const;
```

Este método devuelve un objeto *QPixmap*. Esto se utiliza para representar una imagen en un bloque. El *pixmap* lleva la imagen de cada ítem y este se coloca en los botones de la paleta.

```
void addCable(Cable *f);
int type() const { return Type;}
```

Como se ha explicado anteriormente cada ítem tiene una lista donde se guarda cada conexión establecida, pues *addCable()* es el método encargado de añadir cada conexión a esa lista. El otro método es la encargado de devolver el tipo para cada clase en este caso siempre devuelve del tipo ítem.

5.6.3. Protected

```
void contextMenuEvent(QGraphicsSceneContextMenuEvent *event);
```

Muestra el menú contextual, al hacer clic con el botón derecho al seleccionar un elemento. El menú contextual se utiliza para abrir las propiedades de cada ítem o cable y para eliminarlo del área de diseño.

```
QVariant itemChange(GraphicsItemChange change, const QVariant &value);
```

Método reimplementado. Actúa siempre que ocurre algún cambio en un ítem cuando esta en el área de diseño.

5.7. Clase *Cable*

La clase *Cable* es un elemento gráfico que conecta dos objetos ítems. Es necesario utilizar métodos virtuales que son reimplementadas en esta clase para detectar por ejemplo las colisiones entre dos ítems y las selecciones. Esta clase hereda de *QGraphicsLineItem*. Proporciona una línea que se puede añadir al área de diseño. Cuando la línea se conecta con dos ítems esta se puede mover a la misma vez que lo hacen los ítems.

El color de la línea se puede establecer con *setColor()*. La forma es con *boundingRect ()*, que es un método reimplementado de *QGraphicsLineItem* y para ayudarnos con las colisiones se tiene *shape()* también reimplementada. Se utiliza el método *UpdatePosition* para recalcular la posición de la línea. Item1 e Item2 son los ítems a los que está conectado el cable. Además de contener los ítems a los que esta conectado, tiene pos1 y pos2 que indica la conexión en cada ítem.

Esta clase dispone de dos identificadores:

- Identificador “Type” que ofrece la librería Qt, este identificador sirve para diferenciar clases, también se utiliza en la clase ítem y texto. De esta manera al utilizar un bucle for() o un foreach() con estos identificadores podemos saber a que tipo de clase estamos accediendo.

- Identificador “id”, este identificador es el DNI, identifica a cada ítem y cable de manera única. Este id no lo utiliza la clase texto.

```
Public:
    Cable(Item *item1, Item *item2,QMenu *contextMenu,int id,int pos1,int pos2,
          QGraphicsItem *parent = 0, QGraphicsScene *scene = 0);
    enum { Type = UserType + 4 ;}
    int type() const { return Type; }
    int getId() const { return id; }
    QRectF boundingRect() const;
    QPainterPath shape() const;
    void setColor(const QColor &color) { myColor = color; }
    QColor getColor() { return myColor; }
    Item *startItem() const { return myItem1; }
    Item *endItem() const { return myItem2; }
    int getPosicion1()const { return posItem1;  }
    int getPosicion2()const { return posItem2;  }

 Public Slots:
    void updatePosition();

 Protected:
    void paint(QPainter *painter, const QStyleOptionGraphicsItem *option,
          QWidget *widget = 0);
    void contextMenuEvent(QGraphicsSceneContextMenuEvent *event);
    void mousePressEvent(QGraphicsSceneMouseEvent *mouseEvent);
```

Figura 5.6 Métodos de la clase Cable

5.7.1 Constructor

```
Cable::Cable(Item      *item1,Item      *item2,QMenu      *contextMenu,int
identificador,int pos1,int pos2,
          QGraphicsItem *parent, QGraphicsScene *scene)
     : QGraphicsLineItem(parent, scene)
 {
     myItem1 = item1;
     myItem2 = item2;
     id=identificador;
     myMenuContextual = contextMenu;
     setFlag(QGraphicsItem::ItemIsSelectable, true);
     myColor = Qt::black;
     setPen(QPen(myColor,      3,      Qt::SolidLine,      Qt::RoundCap,
Qt::RoundJoin));
     posItem1=pos1;
     posItem2=pos2;
     setCursor(Qt::PointingHandCursor);
 }
```

Como se ha comentado anteriormente, se inicializa el cable con los dos ítems a los que está conectado, se le inserta el id, se le activa el *flag* para ser seleccionable, se actualizan las variables de sus posiciones y se le añade el color.

5.7.2. Public

```
enum { Type = UserType + 4 };
int type() const { return Type; }
```

Se inicializa Type, este es el identificador que es utilizado para diferenciar esta clase con las demás. El método int *Type()* devuelve el valor en forma de entero.

```
int getId() const { return id; }
```

Devuelve el identificador que diferencia a cada ítem y cable.

```
QRectF boundingRect() const;
QPainterPath shape() const;
```

Son métodos reimplementados encargados de dibujar de una manera la línea. Se utilizan para comprobar las colisiones y las selecciones con el ratón.

```
void setColor(const QColor &color) { myColor = color; }
QColor getColor() { return myColor; }
```

Establece y obtiene el color de la línea.

```
Item *startItem() const { return myItem1; }
Item *endItem() const { return myItem2; }
```

Cuando se instancia un objeto de *Cable*, se inicializa con los dos ítems a los que está conectado, por ello, cuando son llamadas devuelven esta información. Item1 es el primer ítem conectado e ítem2 es el segundo.

```
int getPosicion1()const { return posItem1;  }
int getPosicion2()const { return posItem2;  }
```

Al conectar un cable a un ítem, el cable puede estar en el enganche superior o en el enganche inferior. Estas dos posiciones son diferenciadas con un número entero comprendido entre 0 y 260. La parte superior del ítem es el 0 y la inferior es el 260 esto es por el sistema de coordenadas que utiliza Qt con los ítems.

5.7.3. Public Slots

```
void updatePosition();
```

Este método lo ofrece Qt, lo que significa que esta reimplementado, actualiza la posición del cable en los dos ítems.

5.7.4. Protected

```
void paint(QPainter *painter, const QStyleOptionGraphicsItem *option,
        QWidget *widget = 0);
```

Este método que pertenece a *QGraphicsItem* dibuja el contenido de un elemento en las coordenadas. Este método se reimplementa para proporcionar implementación en el dibujo. Actualiza en tiempo real y constantemente el dibujo del cable, es decir al mover los ítems el cable también se mueve, por lo tanto necesita redibujarse. Utiliza las variables de posición de cada ítem (posItem1 y posItem2) para saber donde dibujar en cada momento.

```
void contextMenuEvent(QGraphicsSceneContextMenuEvent *event);
void mousePressEvent(QGraphicsSceneMouseEvent *mouseEvent);
```

Al igual que en la clase ítem, actúa automáticamente al seleccionar botón derecho ratón sobre un cable y al seleccionar un cable.

5.8. Clase *Texto*

Texto contiene las señales o propiedades que contiene un objeto de este tipo. Se activan los flags seleccionable y movible ya que por defecto están desactivados. Esta clase solo contiene el identificador "Type".

Public:
Texto(QGraphicsItem *parent = 0, QGraphicsScene *scene = 0);
enum { Type = UserType + 3 };
int type() const { return Type; }

Signals:
void lostFocus(Texto *item);
void selectedChange(QGraphicsItem *item);

Protected:
QVariant itemChange(GraphicsItemChange change, const QVariant &value);
void focusOutEvent(QFocusEvent *event);
void mouseDoubleClickEvent(QGraphicsSceneMouseEvent *event);

Figura 5.7 Métodos de la clase Texto

5.8.1. Constructor

```
Texto::Texto(QGraphicsItem *parent, QGraphicsScene *scene)
    : QGraphicsTextItem(parent, scene)
{
    setFlag(QGraphicsItem::ItemIsMovable);
    setFlag(QGraphicsItem::ItemIsSelectable);
    setCursor(Qt::PointingHandCursor);
}
```

El constructor de esta clase activa los flags para mover, seleccionar y cambiar el puntero cuando se posa en encima de un elemento de esta clase. Los métodos que contiene esta clase son o tienen la misma funcionalidad que las que ya hemos visto. Simplemente indicar las dos señales *lostFocus() y selectedChange()* se emiten ante cualquier cambio en el texto o selección.

5.9. Clase *Conexiones*

Esta clase se ha realizado para dar soporte extra a las conexiones de cada ítem ya que es un tanto complejo. Al mismo tiempo se estructura el código y se ordena, ya que aunque las líneas de código pueden ir insertadas en la clase *MainWindow* se ha preferido crear esta clase para hacer más fácil su comprensión y no cargar de código *MainWindow*. Interactúa con la clase *Principal* y con *MainWindow* ya que son estas clases las que crean un objeto de *Conexiones* para utilizar sus métodos.

```
Public:
  Conexiones();
  int conexion_item1(Item *item1,QPointF puntoA)const;
  int conexion_item2(Item *item2,QPointF puntoB)const;
  void eliminar_conexion(Cable *cables)const;
```

Figura 5.8 Métodos de la clase Conexiones

Los métodos *conexión_item1() y conexión_item2()* son utilizadas por la clase *Principal* al crear un cable en el área de dibujo y las dos realizan el mismo proceso. Se les pasa el item1 e item2 respectivamente junto con el punto exacto donde apunta el ratón al conectar un cable. Se comprueban los ocho ítems diferentes (Enchufe, Interruptor, Pias, Luminarias, PLC, Empalmes, ICP y Magnetotérmicos) y dependiendo de las conexiones que disponga y de si ya tiene alguna conexión ocupada o libre, devolverá un valor diferente. Si la conexión es en la parte superior se devuelve un cero, si es en la parte inferior se devuelve un 260. Estos valores son porque el sistema de coordenadas de Qt empieza en cero en la parte superior y aumenta al bajar (imaginamos un eje Y). Para comprobar las conexiones se llama al método *getConexionA()* y *getConexionB()*. En la siguiente imagen se puede observar un ejemplo visual.

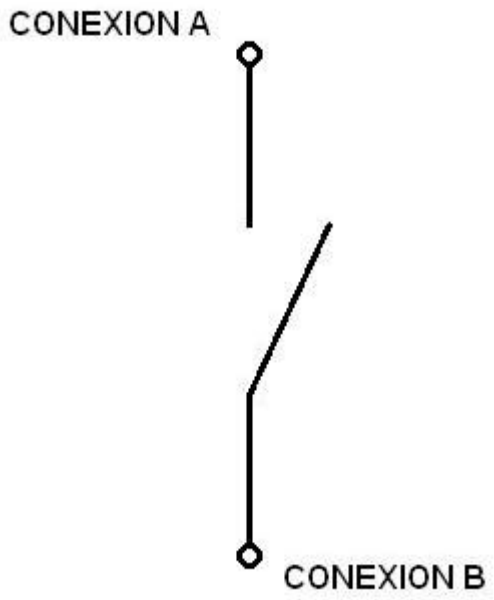


Figura 5.9 Interruptor, se muestra las conexiones que tiene.

Para finalizar, el método eliminar *conexión()* es invocada por la clase *MainWindow* dentro de *deleteItem()*. Al eliminar un cable este método es notificado para que actualice las variables de los ítems.

5.10. Clase *Guardar*

La clase *Guardar*, es la encargada de insertar en un fichero de texto la información que contiene un diseño eléctrico. Este archivo puede ser abierto en cualquier momento por el usuario y utilizando un objeto de *Cargar* se realiza el proceso inverso. El usuario al invocar la acción guardar, la clase *MainWindow* llama al método privado *guardarArchivo()*, este método crea un objeto del tipo *Guardar*, llama al método *generar_archivo()* y de esta manera se accede al algoritmo que se encarga de realizar el proceso de guardado.

Esta clase se compone de cinco métodos con el objetivo de realizar el proceso lo más eficiente posible.

Public:
Guardar(Datos *model_aux, QString nom);
bool generar_archivo(Principal *scene)const;

Protected:
bool guardar_item(Item *item)const;
bool guardar_cable(Cable *cables)const;
bool guardar_texto(Texto *txt)const;
QString tipo_item(Item *item)const;

Figura 5.10 Métodos de la clase Guardar

5.10.1. Constructor

```
Guardar::Guardar(Datos *model_aux, QString nom)
 {
    nombre=nom;
    model=model_aux->getModel();
 }
```

El objeto instanciado de esta clase recibe en el constructor la dirección de la tabla de datos y el nombre del fichero donde se precisa guardar la información. La tabla de datos contiene las propiedades de cada ítem y cable. Se necesita de la dirección de la tabla de datos para poder acceder a toda la información de cada cable o ítem.

5.10.2. Públic

```
bool generar_archivo(Principal *scene)const;
```

Este método tiene como objetivo construir el archivo utilizando la clase *QFile* de Qt. Una vez abierto y que no existen problemas en la apertura, se inserta una cabecera para evitar cargar archivos que no tengan esta línea: **“dat.est.proyecto.pfc”**, lo que puede provocar comportamientos inesperados en el sistema**.** A continuación se lleva a cabo un barrido en el área de diseño, es decir, se utiliza un bucle parta obtener todos los elementos (ítems, cables y texto) que están insertados en el área de diseño. Se invoca a los métodos:

guardar_item(), guardar_texto(), guardar_cable() cada vez que se obtiene un elemento. Este método devuelve ***true*** si todo ha ido correctamente.

5.10.3. Protected

```
bool guardar_item(Item *item)const;
bool guardar_cable(Cable *cables)const;
bool guardar_texto(Texto *txt)const;
```

Los tres métodos siguen el mismo sistema para guarda la información, difiere un poco pero no es muy significativo. Para cada línea de información se separa con el carácter "&". Los tres métodos al finalizar cierran el archivo y devuelven ***true***. En caso de error devuelven ***false***.

El método *guardar_item()* primero guarda el nombre del ítem en cuestión, el segundo bloque es el id, a continuación el color, las posiciones, sus conexiones(conexión A y conexión B) , el número de conexiones totales, y por último toda la información de la tabla de datos.

Continuamos con el siguiente método *guardar_cable()*. Primero guarda el nombre "Cable" para diferenciar que es un cable, seguidamente el id, después el color, la información de la tabla y para finalizar guarda los id´s de los ítems a los que esta conectado.

Guardar_texto() es el método relativamente más simple. No necesita guardar la información de la tabla. Empieza guardando "Texto", seguido de la palabra o frase que haya escrito el usuario, el color, la posición y todo lo relacionado con su formato: cursiva, negrita subrayado, tipo de fuente y tamaño.

```
dat.est.proyecto.pfc
Texto&hola*&#000000&1080&1148&20&MS Sans Serif&true&true&false
Enchufe&1&#ff0000&994&950&true&false&1&5345345&0&0&0&0&1001&
Pias&2&#00ff00&1200&966&true&false&1&0&0&0&&&1002&
Empalme&3&#0000ff&1108&932&true&false&2&0&0&0&&&1002&1001
Cable&1001&#000000&0&0&0&0&0&0&0&1&3
Cable&1002&#000000&0&0&0&0&0&0&0&3&2
```

Figura 5.11 Ejemplo de archivo.

En la figura 5.11 se puede apreciar como se estructura la información en el fichero de texto.

```
QString tipo_item(Item *item)const;
```

Este método tiene como objetivo descargar un poco y organizar el código ya que puede llegar a ser muy lioso y engorroso, por ello se construye este método. Su objetivo es simplemente devolver el nombre del ítem seleccionado, por eso devuelve un *QString*.

5.11. Clase *Cargar*

Esta clase es la encargada de recuperar del archivo anteriormente guardado la información. Contiene una serie de métodos con los que puede recuperar cada elemento como ítems, cables y texto. Además un método se encarga de comprobar si la cabecera es la correcta para recuperar la información. Esta clase es invocada por la clase *MainWindow* cuando el usuario hace clic en abrir, se crea un objeto y se llama al método *extraer_archivo()*.

Se abre el archivo solo en modo lectura, si se logra abrir el archivo se usa un *QTextStream* que convierte automáticamente los datos de 8 bits en un *QString* Unicode, además de que soporta varias codificaciones.

Aunque esta clase no se encarga directamente de cargar cada elemento, si que es la que invoca a los métodos externas para dicho fin.

Public:
```
Cargar(Datos *model_aux, QString nom, Principal *p);
bool extraer_archivo()const;
```

Protected:
```
void extraer_interruptor(QStringList list)const;
void extraer_enchufe(QStringList list)const;
void extraer_empalme(QStringList list)const;
void extraer_luminaria(QStringList list)const;
void extraer_plc(QStringList list)const;
void extraer_icp(QStringList list)const;
void extraer_pias(QStringList list)const;
void extraer_magnetotermico(QStringList list)const;
void extraer_texto(QStringList list)const;
void extraer_cable(QStringList list)const;
bool cabecera()const;
```

Figura 5.12 Métodos de la clase Cargar

5.11.1. Constructor

```
Cargar::Cargar(Datos *model_aux, QString nom,Principal *p)
{
   nombre=nom;
   model=model_aux->getModel();
   scene=p;
}
```

En el constructor se recibe la dirección de la tabla de datos, el nombre del fichero a cargar la información y un puntero que apunta al área de diseño. El puntero del área de diseño es necesario para poder acceder a métodos de la clase *Principal*, cambiar el estado de esta clase y enviar la información. Se recuerda que es el método *cargar_items()* de la clase *Principal* el encargado de restablecer la información directamente enviándole en un vector toda la información.

5.11.2. Public

```
bool extraer_archivo()const;
```

Solicita al método *cabecera()* comprobar si la primera línea del archivo se corresponde con **"dat.est.proyecto.pfc**", de ser así, se restablecen los valores iniciales del EDEE invocando al método *valores_iniciales()* de principal. A continuación se abre el archivo, se lee línea por línea la información que contiene dicho archivo. Esta información se guarda en una variable de tipo vector. Primero se extrae la información del texto, después la de los ítems y por último la del cable invocando a los métodos protegidos. Este orden se debe a que primero se inserta un ítem para poder enganchar un cable, el EDEE esta programado para no quedar un cable enganchado solo de una punta.

5.11.3. Protected

```
void extraer_texto(QStringList list)const;
void extraer_cable(QStringList list)const;
void extraer_interruptor(QStringList list)const;
void extraer_enchufe(QStringList list)const;
void extraer_empalme(QStringList list)const;
void extraer_luminaria(QStringList list)const;
void extraer_plc(QStringList list)const;
void extraer_icp(QStringList list)const;
void extraer_pias(QStringList list)const;
void extraer_magnetotermico(QStringList list)const;
```

Todos los métodos tienen el mismo código variando simplemente algunos parámetros. Para los métodos *extraer_texto() y extraer_cable()* se actualiza el estado de *Principal* a *insertarTexto() e insertaCable()* respectivamente, y se llama al método *cargar_items()* a la que se le pasa el vector que contiene la información de cada elemento. El resto de métodos realizan el mismo proceso que las anteriores pero además añaden el tipo de ítem a insertar actualizando *Principal*.

Ninguna de estos métodos devuelve un valor.

```
bool cabecera()const;
```

Abre el archivo lee la primera línea y comprueba si coincide la siguiente línea **"dat.est.proyecto.pfc",** si es así devuelve ***true*** en caso contrario devuelve ***false***.

5.12. Clase *Propiedades*

La clase Propiedades descarga de código la clase *MainWindow*, aunque también aporta alguna que otra funcionalidad que no seria posible sino se realiza en una clase nueva. Proporciona un widget que se puede acoplar en *MainWindow* o flotar como una ventana en el escritorio.
Es una pequeña ventana que se coloca alrededor de la ventana principal, en nuestro caso se coloca debajo, pero puede colocarse donde el usuario prefiera. Este pequeño menú emergente consiste en una barra de titulo donde aparece el nombre del elemento y un área con etiquetas y recuadros donde el usuario puede cambiar las propiedades de cada elemento. Dispone de dos botones, uno para mover el menú por la ventana principal y otro para cerrar. En el código toma el nombre de Dock Windows.

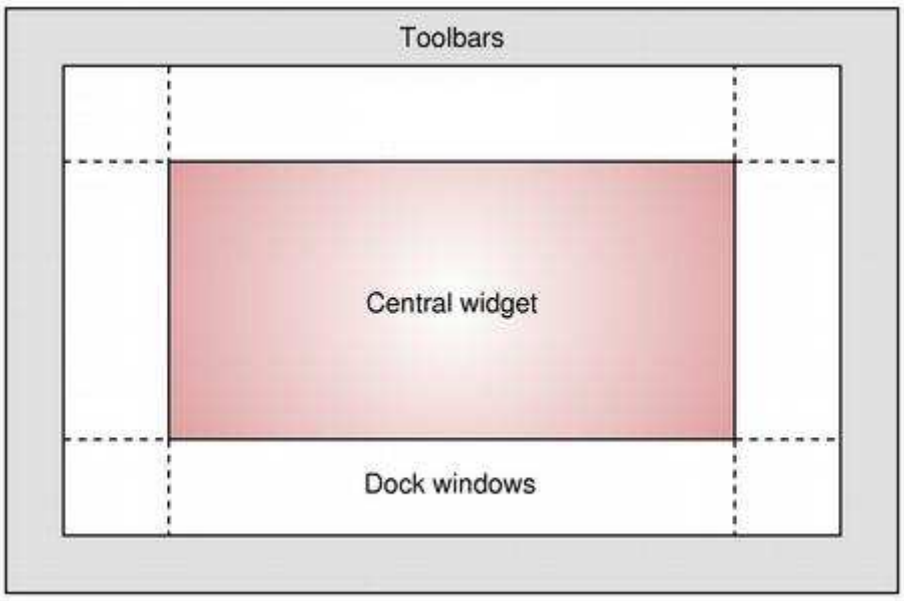


Figura 5.13 Diseño en ventana de un Dock

Public:
```
Propiedades(Datos *model,int referencia,QString nom,int id_tabla);
void cable(int id_model);
void enchufe(int id_model);
void luminaria(int id_model);
void otro_elemento(int id_model);
void PLC(int id_model);
int getId() const { return id; }
```

Signals:
```
void restar(int id);
```

Public Slots:
```
void closeDock();
void habilitar_botones(bool habilitar=true);
```

```
void guardar();
void volver();
void visible(bool vi);
```

Figura 5.14 Métodos de la clase Propiedades

5.12.1. Constructor

```
Propiedades::Propiedades(Datos *tabla, int referencia,QString nom,int
id_tabla)
{
    tipo=tabla;
    id=referencia;
    nombre=nom;
    dock= new QDockWidget();
    if(nom=="Cable"){
       cable(id_tabla);
    }else if(nom=="Enchufe"){
       enchufe(id_tabla);
    }else if(nom=="Luminaria"){
       luminaria(id_tabla);
    }else if(nom=="PLC"){
       PLC(id_tabla);
   }else{
       otro_elemento(id_tabla);
   }
}
```

Esta clase recibe en el constructor diferentes variables. Al recibir estas variables se invoca a los métodos correspondientes para crear el menú Dock. Las diferentes variables son:

- Tipo: Guarda la tabla de datos para acceder a ella directamente.
- Id: Este id identifica al Dock, sirve para tener un control de los menús que se abren y que se cierran.
- Nombre: Es el nombre al elemento el que se desea abrir.
- Dock: Se crea un objeto del tipo *QDockWidget*.
- Id_tabla: Es el identificador que tiene cada ítem o cable para buscarlo en la tabla de datos.

5.12.2. Públic

```
void cable(int id_model);
void enchufe(int id_model);
void luminaria(int id_model);
void otro_elemento(int id_model);
void PLC(int id_model);
```

Estos métodos son los encargados de sostener el menú, es decir ponen el titulo del elemento, rellenan el área de botones, etiquetas y recuadros para la edición de cada propiedad. Conecta las señales de los botones con los diferentes *slots*.

Figura 5.15 Imagen de un Menú con propiedades

```
int getId() const { return id; }
```

Devuelve el id del menú. Este valor se utiliza en el método de *MainWindow restar_contDockWidget(int id)* explicado en el apartado de *MainWindow*.

5.12.3. Signals

```
void restar(int id);
```

Esta señal se utiliza para enviar el id del menú al objeto *MainWindow*, de esta manera cada vez que el usuario cierra un menú esta señal es invocada automáticamente y el objeto *MainWindow* es advertido de dicha situación.

5.12.4. Public Slots

```
void closeDock();
void visible(bool vi);
```

Estos dos métodos tienen la misma acción. Cierran el menú al pinchar el usuario sobre el botón cerrar o al pinchar en la “x” de la esquina superior. Además se envía la señal *restar(int id).*

```
void habilitar_botones(bool habilitar=true);
void guardar();
void volver();
```

Estos *slots* gestionan la actividad de los botones cuando el usuario inserta o edita información de las propiedades. Guardan la información al pulsar el botón guardar, volver restablece los valores iniciales y *habilitar_botones()* marca los botones para poder ser pulsados o los desmarca.

5.13. Clase *Datos*

Para la realización de esta clase se ha utilizado o se ha recurrido al sistema Modelo-Vista-Controlador. Esta manera de programar gestiona la relación entre los datos y la forma en que se presenta al usuario.

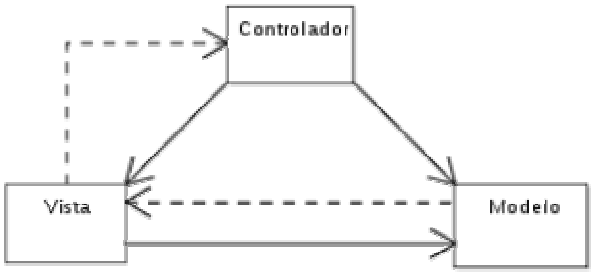


Figura 5.16 Un diagrama sencillo que muestra la relación entre el modelo, la vista y el controlador. Nota: las líneas sólidas indican una asociación directa, y las punteadas una indirecta.(Wikipedia)

MVC se define en las clases abstractas y en los métodos que ofrece Qt.
MVC consiste en tres tipos de objetos. El modelo es la representación específica de la información con la cual el sistema opera, la vista es la presentación en pantalla, y el controlador responde a eventos, usualmente acciones del usuario e invoca peticiones al modelo y, probablemente, a la vista. MVC no pretende discriminar entre capa de negocio y capa de presentación pero si pretende separar la capa visual gráfica de su correspondiente programación y acceso a datos.

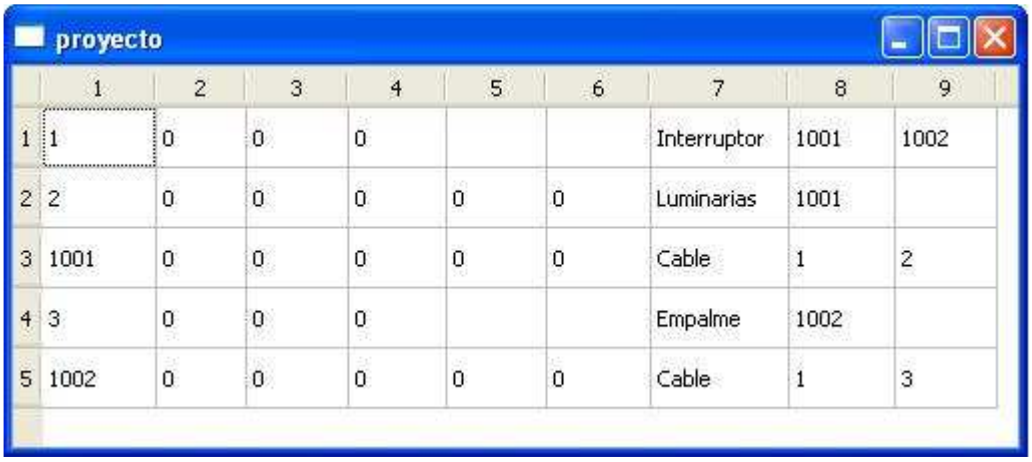
La separación hace posible la visualización de los mismos datos en varias vistas diferentes, y aplicar nuevas vistas al mismo modelo, sin cambiar las estructuras de datos.

La vista, el modelo y el controlador se comunican entre sí mediante señales y *slots*. Las señales del modelo informa sobre sus cambios de datos, las señales de la vista informa sobre la interacción del usuario, y por último las señales del controlador se utilizan durante la edición del modelo y la vista.

Todos los modelos de Qt se basan en la clase *QAbstractItemModel*. Se ha escogido la clase *QStandardItemModel* ya que ofrece los métodos que más interesan a la hora de programar el código.

QStandardItemModel maneja estructuras en árbol más compleja, cada uno de los cuales puede contener datos arbitrarios.

Se utiliza *QDataWidgetMapper* que es el encargado de hacer de controlador lo que nos ofrece los métodos necesarios para interactuar entre la vista y los datos.



	1	2	3	4	5	6	7	8	9
1	1	0	0	0			Interruptor	1001	1002
2	2	0	0	0	0	0	Luminarias	1001	
3	1001	0	0	0	0	0	Cable	1	2
4	3	0	0	0			Empalme	1002	
5	1002	0	0	0	0	0	Cable	1	3

Figura 5.17 Esta tabla es invisible para el usuario, pero muestra como se estructura los datos. La primera columna guarda el id de cada elemento, hasta la columna 6 son las propiedades. La columna 8 en adelante guarda el id del elemento al que está conectado. Este ejemplo es de una conexión entre una bombilla, un interruptor y un empalme.

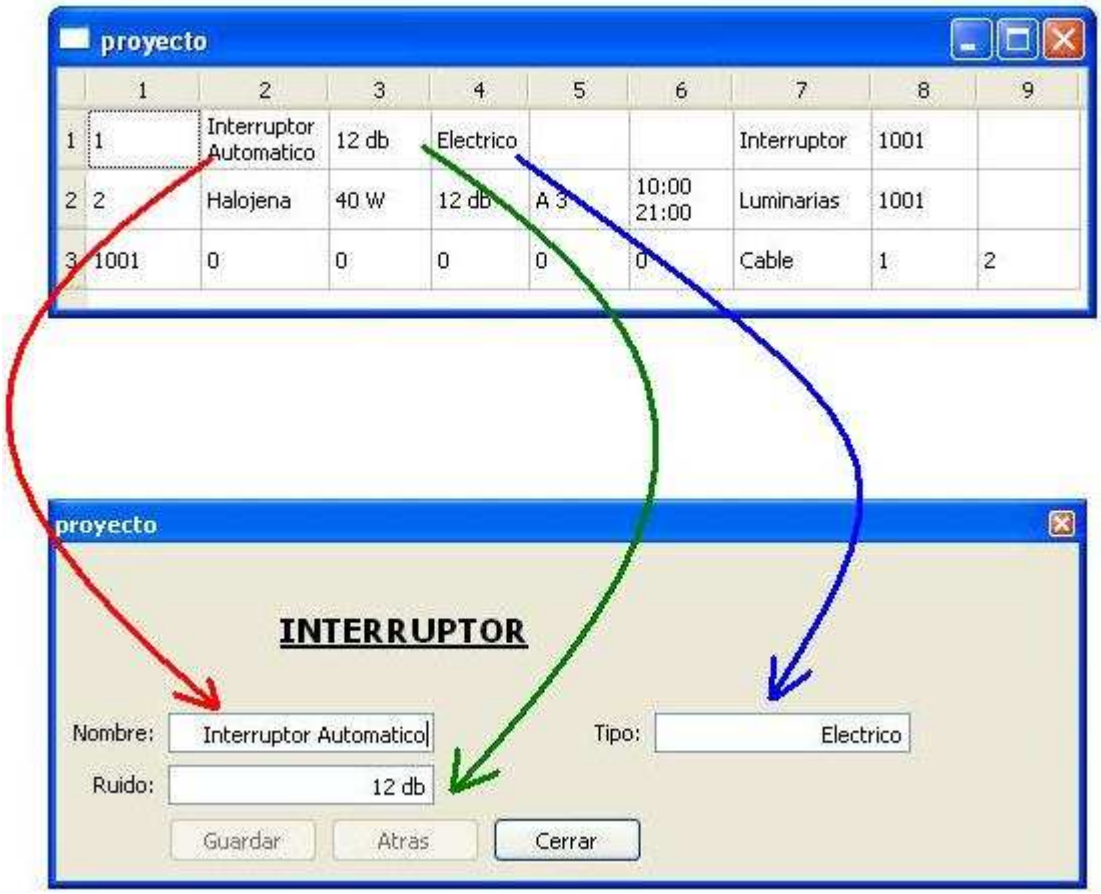


Figura 5.18 Muestra la tabla de datos y cómo se representa en el menú propiedades.

La clase *Datos* hereda de *QStandardItemModel*, proporciona un modelo general para el almacenamiento de datos personalizados.

QStandardItemModel ofrece todo tipo de métodos para trabajar con el modelo de datos. Aun así la clase *Datos* contiene otros métodos personalizadas a propósito para el beneficio del EDEE.

```
Public:
   Datos(QObject *parent=0);
   int  borrar_item( int id)const;
   void eliminar_cable(int id)const;
   void insertar_datos(Item *item,QString str,QString var1,
                  QString var2,QString var3,QString var4,
                  QString var5,int cont)const;
   void insertar_datos_cable(Cable *cables,QString str,
                  QString var1,
                  QString var2,QString var3,QString var4,
                  QString var5,int cont)const;
   bool elemento_sin_conectar()const;
   int extraer_datos(int indi)const;
   QStandardItemModel *getModel()const {return dat; }
   void borrar_datos()const;
   bool getModificado() { return modificado; }
   void setModificado(bool mod) { modificado=mod; }

Private Slots:
   void datos_modificados();

Protected:
   void ordenar_columnas(int fila)const;

};
```

Figura 5.19 Métodos de la clase Datos

5.13.1. Constructor

```
Datos::Datos(QObject *parent)
             : QStandardItemModel(parent)
 {
     dat=new QStandardItemModel();
     modificado=false;
     connect(dat, SIGNAL(dataChanged(QModelIndex,QModelIndex)),
             this, SLOT (datos_modificados()));
 }
```

El constructor de esta clase simplemente crea la tabla ilustrada en la figura 5.13 y conecta la señal a los *slots*.

5.13.2. Public

```
int  borrar_item( int id)const;
void eliminar_cable(int id)const;
```

Eliminan las entradas de los ítems y los cables en la tabla de datos. El método *borrar_item()* contiene un algoritmo que busca en la tabla el id que corresponde con el ítem a eliminar, una vez encontrado obtiene el cable al que está conectado y lo busca en la tabla para eliminarlo también, por eso invoca al método *eliminar_cable()*. El algoritmo que se implementa en el método *eliminar_cable()* busca el id del cable, pero antes de borrarlo de la tabla busca los id´s de los dos elementos a los que está conectado, los busca en la tabla y borra el id del cable de estos elementos. Una vez hecho esto elimina por completo la fila entera del cable.

```
void insertar_datos(Item *item,QString str,QString var1, QString var2,
                    QString var3,QString var4, QString var5,int cont)const;
void insertar_datos_cable(Cable *cables,QString str, QString var1, QString var2,
                          QString var3,QString var4, QString var5,int cont)const;
```

Estos métodos reciben los datos que el usuario ha insertado para cada elemento. El método *insertar_datos_cable()*, además de guardar la información del cable, busca los ítems a los que se conecta y añade a sus entradas el id del cable.

```
bool elemento_sin_conectar()const;
```

Devuelve ***true*** si existe elemento en el diseño que no está conectado.

```
int extraer_datos(int indi)const;
```

Devuelve la entrada en la tabla que coincide con el valor recibido (indi).

```
QStandardItemModel *getModel()const {return dat; }
```

Retorna la dirección de la tabla datos.

```
void borrar_datos()const;
```

Borra todo la información que contiene la tabla.

```
bool getModificado() { return modificado; }
void setModificado(bool mod) { modificado=mod; }
```

“Modificado” es una variable que nos indica cualquier cambio ocurrido en la tabla de datos, así se conoce en todo momento si el diseño ha cambiado o el usuario ha cambiado alguna propiedad de un cable o un ítem.

5.13.3. Private Slots

```
void datos_modificados();
```

Pone a **true** la variable modificado.

5.13.4. Protected

```
void ordenar_columnas(int fila)const;
```

El algoritmo de este método tiene un papel importante para el buen funcionamiento del sistema, ya que si produjera un error la aplicación no funcionaria correctamente. Al eliminar un cable, el id de éste se elimina también de la entrada de los ítems a los que está conectado. Para una buena gestión de la tabla es conveniente no dejar espacios libres en las columnas ya que eso hace más complejo el código que se realiza después. Es por ello que se ha implementado este método, para tener ordenada todas las entradas.

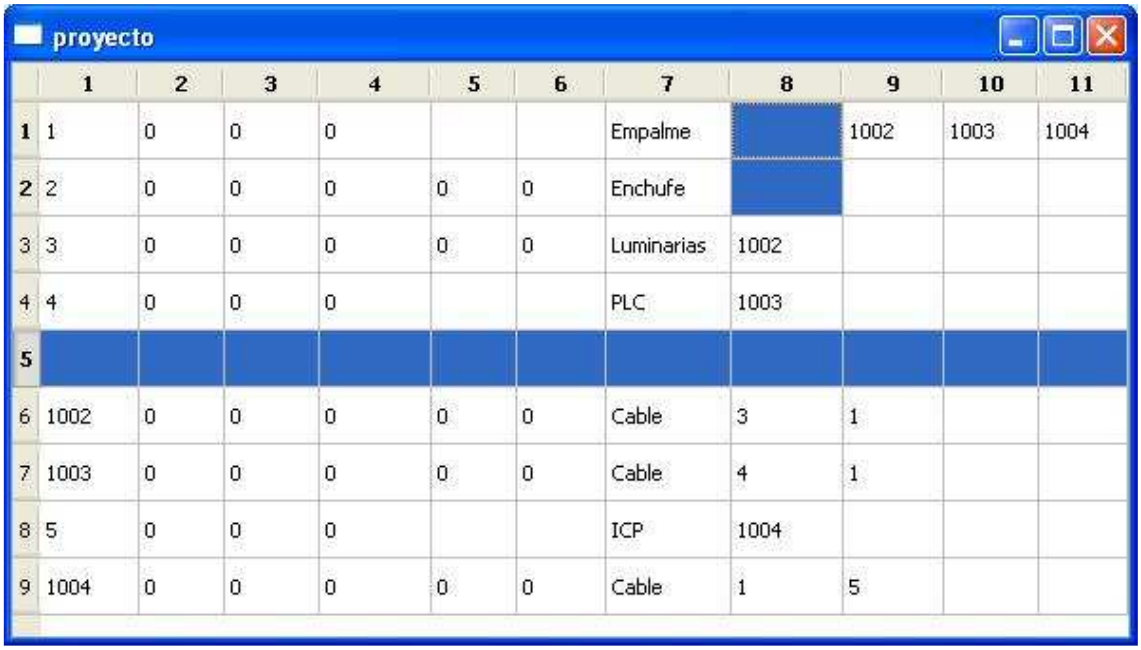
En las siguientes ilustraciones se explica este método:

El cable con el id 1001 es el eliminado.

proyecto

	1	2	3	4	5	6	7	8	9	10	11
1	1	0	0	0			Empalme	1001	1002	1003	1004
2	2	0	0	0	0	0	Enchufe	1001			
3	3	0	0	0	0	0	Luminarias	1002			
4	4	0	0	0			PLC	1003			
5	1001	0	0	0	0	0	Cable	2	1		
6	1002	0	0	0	0	0	Cable	1	3		
7	1003	0	0	0	0	0	Cable	1	4		
8	5	0	0	0			ICP	1004			
9	1004	0	0	0	0	0	Cable	1	5		

Aquí se muestra como queda la tabla antes de ordenar.



	1	2	3	4	5	6	7	8	9	10	11
1	1	0	0	0			Empalme		1002	1003	1004
2	2	0	0	0	0	0	Enchufe				
3	3	0	0	0	0	0	Luminarias	1002			
4	4	0	0	0			PLC	1003			
5											
6	1002	0	0	0	0	0	Cable	3	1		
7	1003	0	0	0	0	0	Cable	4	1		
8	5	0	0	0			ICP	1004			
9	1004	0	0	0	0	0	Cable	1	5		

Aquí se muestra una vez ordenada.

	1	2	3	4	5	6	7	8	9	10	11
1	1	0	0	0			Empalme	1002	1003	1004	
2	2	0	0	0	0	0	Enchufe				
3	3	0	0	0	0	0	Luminarias	1002			
4	4	0	0	0			PLC	1003			
5	1002	0	0	0	0	0	Cable	1	3		
6	1003	0	0	0	0	0	Cable	1	4		
7	5	0	0	0			ICP	1004			
8	1004	0	0	0	0	0	Cable	1	5		

5.14. Clase *Exportar*

La clase *exportar* tiene como misión primordial extraer toda la información que se encuentra en la tabla de datos y escribirla en un archivo de texto. De la manera en la que se gestiona la tabla de datos, extraer la información no es muy complicada, aunque a primera vista puede parecer que sí. Se recorre la tabla de datos fila a fila extrayendo cada elemento y exportándolo al archivo de texto. Al invocar el usuario la acción exportar en la clase *MainWindow* se crea un objeto de la clase *Exportar*.

```
Public:
   Exportar(Datos *model,QString nom);
   bool crear_archivo_texto()const;

Protected:
   void escribir_enchufe(int indice)const;
   void escribir_luminaria(int indice)const;
   void escribir_otro_elemento(int indice, QString nom_elemento)const;
   void escribir_cable(int indice)const;
   void escribir_PLC(int indice)const;
};
```

Figura 5.20 Métodos de la clase Exportar

5.14.1. Constructor

```
Exportar::Exportar(Datos *model_aux, QString nom)
 {
     model=model_aux->getModel();
     nombre=nom;
 }
```

En el constructor de esta clase se recibe el nombre del archivo y la dirección de la tabla de datos.

5.14.2. Public

```
bool crear_archivo_texto()const;
```

El método *crear_archivo_texto()* devuelve ***true*** si el archivo se ha creado sin problemas o ***false*** si se ha producido algún error. Con un bucle se recorre la fila y se comprueba la columna 6 (sabiendo que la primera columna es la 0), esta columna indica el tipo de elemento (enchufe, luminaria, cable etc…) se extrae el tipo de elemento se compara y se invoca a los métodos *protected* dependiendo del elemento que sea.

5.14.3. Protected

```
void escribir_enchufe(int indice)const;
```

El método *enchufe(int indice)* como las demás tienen el mismo algoritmo, la única diferencia es las propiedades que se escriben en el archivo. Para un enchufe las propiedades son: Nombre, Potencia, Ruido, Elemento y Horario. Aquí un ejemplo:

Enchufe 1: Nombre 0, Potencia 0, Ruido 0, Elemento 0, Horario 0, Conector 1001;

```
void escribir_luminaria(int indice)const;
```

Una luminaria (bombilla) contiene estas propiedades: Nombre, Potencia, Ruido, Tipo y Horario. Ejemplo:

Luminaria 6: Nombre 0, Potencia 0, Ruido 0, Tipo 0, Horario 0, Conector 1005;

```
void escribir_PLC(int indice)const;
```

PLC contiene las siguientes propiedades: Nombre, Tipo y Horario. Por ejemplo:

PLC 4: Nombre 0, Tipo 0, Horario 0, Conector 1004;

```
void escribir_otro_elemento(int indice, QString nom_elemento)const;
```

Este método abarca los demás elementos ya que contienen las mismas propiedades, por lo tanto era innecesario hacer más métodos que hicieran el mismo proceso. Las propiedades son: Nombre, Ruido y Tipo. Ejemplo de un interruptor:

Interruptor 2: Nombre 0, Ruido 0, Tipo 0, Conector 1003;

```
void escribir_cable(int indice)const;
```

Por último el método *escribir_cable()* una vez copiado todos los datos al archivo busca en la tabla el nombre de los id´s a los que está conectado. Se recuerda que en la tabla de datos los cables y los demás elementos guardan las conexiones que tienen. En el caso del cable como es un número no se sabe que elemento es al que está conectado, por ello se realiza una nueva búsqueda para encontrarlo y copiarlo al archivo de texto. Por ejemplo:

Conector 1004: Nombre 0, Ruido 0, Tipo 0, Grosor 0, Longitud 0, Conexion con: Empalme 3,PLC 4;

Capítulo 6. Conclusiones y líneas futuras

6.1. Conclusiones

Los diferentes capítulos presentes en este trabajo han tratado de cubrir cada una de las necesidades iniciales del PFC. En primer lugar se ha realizado una breve referencia a Qt, explicando qué es el lenguaje Qt así como sus principales características. Por otro lado se ha redactado una definición de esquema eléctrico haciendo especial enfásis en sus usos y los componentes a considerar en una red PLC. Por último se han desarrollado capítulos en los que se describe la aplicación desde dos puntos de vista distintos: 1)qué apariencia tiene la aplicación y cómo debe usarla un usuario y 2) cómo se ha implementado el código, los métodos y las clases.

Sin lugar a dudas se puede concluir que se han cumplido los objetivos iniciales marcados ya que se ha realizado una aplicación que ofrece al usuario la posibilidad de diseñar un circuito eléctrico basado en componentes. Cada componente contiene una serie de propiedades editables por partes del usuario mediante los menús que ofrece la aplicación. Además, la aplicación permite la exportación de toda la información contenida en el esquema eléctrico a un archivo de texto que identifique los componentes, sus propiedades y sus conexiones. Además, la forma en la que está creada la aplicación permite modificaciones, añadir nuevas opciones y mejoras de una manera relativamente sencilla.

Por último, Qt permite que la aplicación creada puede utilizarse en Windows, Linux y sistemas Mac sin ningun problema

6.2. Aspectos a destacar de la programación

Es conveniente resaltar que a la hora de programar la aplicación se han encontrado diversos problemas. Al ser un código complejo, la representación de los datos de cada ítem eléctrico o cable ha dado alguna que otra dificultad. Se ha recurrido al sistema Modelo-Vista-Controlador, que permite realizar cambios en el código de forma sencilla y sin modificar la estructura. La separación (Modelo-Vista-Controlador) hace posible la visualización de los mismos datos en varias vistas diferentes, y aplicar nuevas vistas al mismo modelo sin cambiar las estructuras de datos. Una vez se realiza el formato de la tabla de datos, se hace muchísimo más cómodo continuar con el proyecto De esta forma acciones como guardar, exportar o ver las propiedades de cada elemento, resultan bastante más fácil de realizar ya que la tabla contiene la información ordenada y organizada.

Es muy importante comprender y conocer los *widgets* que ofrece Qt. Cada uno ofrece una serie de características y propiedades que hay que entender perfectamente para conseguir el objetivo que se plantea. Un ejemplo puede ser donde están colocados los botones con los componentes eléctricos en la parte izquierda. Casi todos los *widgets* encargados de sostener estos botones mostraban problemas que de alguna manera no se solventaban fácilmente o no hacían lo que se esperaba de ellos.

Utilizar clases que manejen objetos gráficos, ya sea moverlos, rotarlos, cambiar de color etc… conlleva una gran dificultad. Para solucionar los diversos problemas que aparece al

hacer algo sobre este tema ha sido necesario comprender la descripción del entorno *Graphics View Framework* desarrollada en la documentación que ofrece Qt Creator.

Guardar toda la información en un principio podía ser una operación difícil, pero gracias a la manera en que los datos se gestionan en la tabla el desarrollo de esta acción no ha sido tan complejo como se preveía.

Para la comprensión de Qt, se ha optado por libros y ejemplos que ofrece Qt Creator, en él se pueden encontrar multitud de ejemplos que ayudan a entender diversas partes que utiliza la aplicación.

Las librerías de Qt son muy extensas, por ello ses recomendable estudiar ejemplos que hagan la función que se desea para solucionar los diversos problemas que se plantean.

6.3. Líneas futuras

Como cualquier software, la aplicación creada es susceptible de mejoras, tanto estéticas como de comportamiento, no obstante el grado de madurez de la aplicación es el suficiente para que pueda emplearse tal y como está.

Como posibles mejoras o líneas futuras se podrían incluir las siguientes:

Por un lado se podría añadir una opción que fuese capaz de calcular si un elemento PLC puede tener interferencias o si las provoca. Es decir, la propia aplicación realizaría ciertas labores del simulador ofreciendo una previsión acerca de si los PLCs del esquema sufrirán interferencias y la magnitud aproximada de éstas.

Por otro lado se podría mejorar la vista de los ítems e iconos. Sería intereante añadir opciones como copiar, cortar, pegar y deshacer. Poder rotarlos y alguna que otra opción que de más sencillez al diseño del usuario serían también útiles.