GUÍA DE IMPLEMENTACIÓN DEL ESTÁNDAR IEC61970 PARA EL MANEJO DE INFORMACIÓN DE SISTEMAS DE DISTRIBUCIÓN DE ENERGÍA

Presentado por
Edinson Enrique Sanchéz Galán
Aura Milena Rueda Díaz

ESCUELA DE INGENIERÍAS ELÉCTRICA, ELECTRÓNICA Y DE TELECOMUNICACIONES
FACULTAD DE INGENIERÍAS FISICOMECÁNICAS
UNIVERSIDAD INDUSTRIAL DE SANTANDER
BUCARAMANGA
2015

GUÍA DE IMPLEMENTACIÓN DEL ESTÁNDAR IEC61970 PARA EL MANEJO DE INFORMACIÓN DE SISTEMAS DE DISTRIBUCIÓN DE ENERGÍA

Presentado por

Edinson Enrique Sanchéz Galán

Aura Milena Rueda Díaz

Trabajo de grado para optar por el título de

Ingeniero Eléctricista

Director

Iván David Serna Suárez

Magister en Ingeniería Eléctrica

Codirector

Gilberto Carrillo Caicedo

Profesor Titular Laureado Émerito Universidad Industrial de Santander

Doctor Ingeniero Industrial

Codirector

Gabriel Ordoñez Plata

Profesor Titular Laureado Universidad Industrial de Santander

Doctor Ingeniero Eléctricista

ESCUELA DE INGENIERÍAS ELÉCTRICA, ELECTRÓNICA Y DE TELECOMUNICACIONES

FACULTAD DE INGENIERÍAS FISICOMECÁNICAS

UNIVERSIDAD INDUSTRIAL DE SANTANDER

BUCARAMANGA

2015

Este trabajo de grado está dedicado especialmente a nuestros padres, quienes son responsables de las personas que somos hoy en día, sin su apoyo y amor la realización de este sueño hoy no sería realidad, a Dios porque sin su voluntad nada de lo que pasa en nuestras vidas sería posible, a nuestros docentes los cuales a través de sus enseñanzas, conocimientos y experiencias inculcaron una actitud crítica y de trabajo para con la academia y la vida misma, a nuestros amigos y familiares ya que gracias a su amistad y cariño alimentan día a día los deseos de seguir adelante y hacen más alegre el mundo en el que vivimos.

– Edinson y Aura

// AGRADECIMIENTOS

A Dios porque gracias a su amor y su voluntad pudimos sacar nuestra carrera adelante, brindándonos fuerza, entendimiento y por guiarnos en el camino de la verdad y de la alegría, a nuestros padres por darnos la vida, por formarnos con valores y principios, fomentar en nosotros las capacidades y sueños que Dios nos regaló y porque gracias a su apoyo somos personas de bien. A nuestros compañeros y profesores en especial a nuestro director el cual destaco en nosotros un potencial el cual aprovecharemos durante nuestra vida profesional.

Agradecimientos Edinson

Agradezco especialmente a mi compañera Aura Rueda porque sin ella esta gran labor no habría sido posible su pasión y trabajo fueron los aportes más fuertes en la realización de este proyecto.
A mi amiga Andrea Llanos porque ella estuvo siempre conmigo es una excelente amiga y compañera.
A mi amiga y compañera Katerin Osorio porque su amistad es un tesoro invaluable y será así por siempre.
A Daniel Palacios mi mejor amigo y compañero de la universidad, gracias por ser incondicional para conmigo.
A mis compañeros: Neify Pulgar, Natalia Cortes, Andres Tellez, Jair Meneses, Lesly Muñoz, Jhon Montañez, Andrea Sarmiento, Jennyfer Pulgar, Jose Perez, Nixon Ortiz, Felipe Pardo, Carlos Cordero, Carlos Aguilar y demás que me acompañaron en alguna etapa de mi formación profesional, gracias por todo su apoyo y amistad.

Contenido

Introducción 18

1. Programación Orientada a Objetos 20

1.1. Diagramas UML . . . 21
1.2. Clases . . . 21
1.2.1. Objetos o instancias de clases . . . 22
1.3. Atributos y métodos . . . 22
1.3.1. Atributos . . . 22
1.3.2. Métodos . . . 22
1.4. Relaciones entre clases . . . 23
1.4.1. Multiplicidad . . . 23
1.4.2. Asociación . . . 23
1.4.3. Composición . . . 24
1.4.4. Agregación . . . 24
1.4.5. Herencia o Generalización . . . 25
1.4.6. Ejemplo . . . 25
1.5. Especificación de datos y comportamientos . . . 26
1.5.1. Datos . . . 26
1.5.2. Comportamientos . . . 26
1.6. Abstracción . . . 27

2. Introducción a Python 28

2.1. Listas . . . 29
2.1.1. Funciones de una lista . . . 29
2.2. Diccionarios . . . 31
2.2.1. Funciones de los diccionarios . . . 32
2.2.2. Otras formas de crear diccionarios . . . 37
2.3. Clases . . . 37

2.3.1. Cadenas de caracteres . . . 38
2.3.2. Iniciación del objeto . . . 38
2.3.3. Identación en Python . . . 38
2.4. Atributos . . . 39
2.5. Herencia . . . 39
2.6. Métodos . . . 40
2.7. Instancias u objetos de clase . . . 41
2.8. Relaciones entre clases en Python . . . 41
2.9. Llamar atributos o métodos . . . 42
2.10. Módulos y paquetes . . . 43
2.10.1. Módulos . . . 43
2.10.2. Paquetes . . . 43
2.11. Importaciones . . . 44
2.12. Ejemplo general . . . 44

3. Introducción al estándar IEC 61970-301 CIM 50

3.1. Paquetes CIM . . . 51
3.1.1. Domain . . . 52
3.1.2. Core . . . 52
3.1.3. Topology . . . 52
3.1.4. Wires . . . 52
3.1.5. Outage . . . 52
3.1.6. Protection . . . 52
3.1.7. LoadModel . . . 52
3.1.8. Generation . . . 53
3.1.8.1. Production . . . 53
3.1.8.2. GenerationDynamics . . . 53
3.1.9. SCADA . . . 53
3.2. Clases y relaciones del CIM . . . 53
3.3. Estructura de las clases . . . 54
3.4. Otros conceptos importantes . . . 57
3.4.1. Clase <IdentifiedObject> . . . 57
3.4.2. Concepto de conectividad . . . 58

4. Organización de la información 59

4.1. Identificadores usados en el proyecto . . . 60

4.2. Distribución de la información 62
4.2.1. Clases 64
4.2.2. API 64
4.2.3. Diccionarios 64
4.2.4. Main 64
4.2.5. Interfaz 65

5. Interfaz de usuario 66

5.1. Construcción de la interfaz 67
5.2. Interfaz gráfica de usuario 67
5.2.1. Bloque de búsqueda 68
5.2.2. Radio-botones 69
5.2.3. Subventana de atributos 70
5.2.4. Tabla de datos 71
5.2.5. Ubicación geográfica y diagrama unifilar 72
5.3. Manejo de datos por consola 74
5.3.1. Creación de instancias 74
5.3.2. Consulta de atributos por medio de consola 79
5.3.3. Actualización de información a través de los diccionarios 79
5.3.4. Actualización de información a través de las instancias 81

6. Caso de aplicación 85

6.1. Creación de clases CIM 86
6.1.1. Creación del paquete Core 86
6.1.2. Creación del paquete Domain 87
6.2. Pasos a seguir en la implementación del estándar 87
6.3. Implementación del circuito de prueba 91
6.3.1. Circuito de prueba 92
6.3.2. Proceso de implementación 92
6.4. Archivos XML para el intercambio de la información 106

7. Conclusiones 107

Bibliografía 109

Anexos 110

LISTA DE FIGURAS

1.1. . En este caso la clase es <Conductor> y posee los atributos de susceptancia, resistencia, longitud. 21
1.2. Ejemplo de multiplicidad. Ninguno o muchos conductores estan relacionados con cero o un tipo de conductor. 23
1.3. Ejemplo de asociación. Conductor esta asociado con un tipo de conductor por medio de la multiplicidad descrita. 24
1.4. Ejemplo de composición. Un contenedor de equipos se compone de uno o más equipos. 24
1.5. Ejemplo de agregación. Una línea puede tener uno o más segmentos de línea. . 25
1.6. Ejemplo de herencia. La clase <ConductingEquipment> es padre de <Conductor>. 25
1.7. Ejemplo de diagrama UML 26

2.1. Esquema de módulos y paquetes. 44
2.2. Diagrama UML del ejemplo general. 45
2.3. Distribución de clases del ejemplo general. 46

3.1. Esquema de paquetes en el CIM. 51
3.2. Agregación 1 de la clase PowerTransformer. 56
3.3. Agregación 2 de la clase PowerTransformer. 56

4.1. Esquema de organización de la información 63

5.1. Esquema de la interfaz gráfica de usuario. 68
5.2. Esquema del bloque de búsqueda dentro de la interfaz. 68
5.3. Esquema para la obtención de la ruta. 69
5.4. Indicaciones para cargar la información. 69
5.5. Utilización de los radio-botones en la interfaz. 70
5.6. Subventana de atributos para un elemento. 71
5.7. Tabla de datos para unos parámetros determinados 72

5.8. Ubicación geográfica y diagrama unifilar de un elemento seleccionado 73
5.9. Diagrama unifilar de un elemento del sistema 74
5.10. Actualización de instancias a partir de pickle. La figura muestra las acciones y el flujo que se sigue al actualizar datos por medio de diccionarios. 81

6.1. Flujo de información a través de API para el uso de las clases CIM en los distintos Main. 90
6.2. Esquema del proceso general para la instanciación de elementos 91
6.3. Diagrama unifilar del sistema de 4 barras 92
6.4. a.) Estructura línea 115 kV b.)Estructura línea 34.5 kV 93
6.5. Diagrama UML de las líneas de transmisión 98
6.6. Diagrama UML del sistema 99

Lista de tablas

4.1. Código de equipos. 60

6.1. Características eléctricas de las líneas de transmisión. 93
6.2. Características de los conductores. 93
6.3. Disposición de las líneas. 94
6.4. Características eléctricas del generador GEN1. 94
6.5. Características eléctrica de la carga CAR1. 94
6.6. Características eléctricas del transformador TRF1. 94

Lista de anexos

Anexo A Manejo de archivos en Python 111

Anexo B Uso del espacio de nombres 118

RESUMEN

TÍTULO:

GUÍA DE IMPLEMENTACIÓN DEL ESTÁNDAR IEC61970 PARA EL MANEJO DE INFORMACIÓN DE SISTEMAS DE DISTRIBUCIÓN DE ENERGÍA[1]

AUTOR:

EDINSON ENRIQUE SÁNCHEZ GALÁN

AURA MILENA RUEDA DÍAZ[2]

PALABRAS CLAVE:

Objeto, interfaz, Cim, Python, espacio de nombres, atributos, modelo de información común.

DESCRIPCIÓN:

En este trabajo se presenta inicialmente todo lo relacionado con la programación orientada a objetos, la cual representa una parte esencial en este proyecto debido a que el estándar IEC 61970-301 está diseñado bajo este paradigma de programación. Posterior a esto, se materializa la estructura del estándar mediante un lenguaje de programación, que en este caso es Python. De igual manera se podrá ver una síntesis de la norma, la cual contiene todas las clases junto con los atributos y relaciones necesarias para construir objetos que emulen a los equipos dentro de un sistema de energía eléctrica.

Al final de este proyecto de investigación se muestra el proceso paso a paso de la construcción de la aplicación informática de un circuito de prueba escogido para la implementación de la norma a través de Python. En las secciones dedicadas a mostrar dicha elaboración se detallan elementos como: sintaxis utilizada en el espacio de nombres de los objetos, organización de la información, caracterización de los datos recopilados del circuito de prueba, manejo de archivos para la correcta gestión de la información y la forma apropiada de usar la interfaz gráfica de usuario desarrollada para la comunicación con la estructura de objetos del circuito de prueba, generando así una buena guía para la implementación del estándar en cuestión.

[1] Trabajo de grado.

[2] Facultad de Ingenierías Físico-Mecánicas. Escuela de Ingenierías Eléctrica, Electrónica y de Telecomunicaciones. Director: Iván David Serna Suárez. Codirector: Gilberto Carrillo Caicedo. Codirector: Gabriel Ordoñez Plata.

ABSTRACT

TITLE:

GUIDE TO THE IMPLEMENTATION OF THE IEC 61970 STANDARD FOR THE INFORMATION MANAGEMENT IN ENERGY DISTRIBUTION SYSTEMS

AUTHORS:

EDINSON ENRIQUE SÁNCHEZ GALÁN

AURA MILENA RUEDA DÍAZ[3]

KEY WORDS:

Object, interface, CIM, Python, namespace, attributes, common information model.

DESCRIPTION:

This document first presents the entire meant of the objects-oriented programming, and is an essential part of this project because the standard IEC 61970-301 is designed based on that type of programming paradigm. Following this structure is materialize using a standard programming language, which in this case is Python. Similarly you can see a summary of the standard, it contains all classes with attributes and relationships needed to build objects that emulate the teams within a power system.

At the end of this research project, it is shown a step-by-step model of a computational application building for a trial circuit that was chosen for the implementation of the standard by Python language. In the sections dedicated to that purpose, is possible to find specific details about: syntaxes used in the objects name space, information's organization, a characterization of collected data about the trial circuit, archives treatment for the correct information management and appropriate way to use the graphic user's interface developed essentially for the communication with the structure and the objects of the trial circuit, generating a good guide for the implementation of this standard.

[3]Faculty of Physical-Mechanic Engieneering. School of Electrical, Electronical and Telecommunications Engineering. Advisor: Iván David Serna Suárez. Co-advisor:Gilberto Carrillo Caicedo. Co-advisor:Gabriel Ordoñez Plata.

INTRODUCCIÓN

A través de los años las empresas del sector eléctrico se han visto en la necesidad de intercambiar información acerca de sus redes eléctricas, para soportar sus propias bases de datos y lograr que los sistemas de energía respondan a los eventos a los que son sometidos de una forma eficiente, confiable y rápida. La información que se transmite de un lugar a otro para la coordinación de dichas redes, debe ser precisa, pero muchas veces esto no es posible, debido a la degradación de la misma. Como la mayoría de empresas manejan distintos formatos para la gestión de la información contenida en sus bases de datos, es difícil entenderse de manera directa entre ellas, por lo que se hace necesario el uso de traductores, haciendo lento el proceso de intercambio de datos y degradando la calidad de la información que se comparte.

El modelo de información común, o por sus siglas en inglés CIM (Common Information Model) comprendido en el estándar IEC 61970-301, es un modelo informático orientado a la organización y administración de los elementos que componen un sistema de energía eléctrica. Este permite manejar la información presente en una empresa de forma segura, rápida y eficaz, pero sobre todo, de manera sencilla, entregando una solución al problema de entre áreas de trabajo al interior y exterior de una empresa del sector eléctrico (McMorran, 2007).

El objetivo primordial de este trabajo de investigación es aplicar el estándar IEC 61970-301 a un circuito de prueba, para así comprender la norma, caracterizarla y desarrollar buenas prácticas de implementación de la misma. A lo largo de este trabajo se muestra la información recopilada acerca de las herramientas usadas para la implementación del estándar. De la misma forma se proponen ciertas metodologías a seguir para la correcta gestión de los datos recopilados dentro de la aplicación informática creada.

En este trabajo no solo se abordaron los parámetros más influyentes al momento de mani-

pular la norma si no también se crearon nuevas estrategias de gestión que no afectan la forma en la que el CIM da manejo a la información. La aplicación más importante es la interfaz gráfica de usuario diseñada para la consulta de datos de los equipos que conforman un circuito de potencia, que incluye: generación, transmisión, y distribución. Esta aplicación GUI (Graphical User Interface) también permite agregar y/o modificar la información contenida en la base de datos previamente construida con el CIM, dando como resultado un software de alto nivel que se comunica con el estándar sin alterar las características propias de la norma.

Para finalizar se hizo un primer acercamiento con el lenguaje de marcas extensibles o en su forma abreviada XML, el cual almacena la información contenida en la base de datos en un formato legible que puede ser aprovechado por otras aplicaciones CIM, es decir, logra una interoperabilidad con otros sistemas.

A pesar de la gran cantidad de información recopilada en este trabajo acerca del funcionamiento, comportamiento y desarrollo de aplicaciones basadas en el CIM, no se alcanza a revizar todo lo que el estándar IEC 61970 ofrece para la gestión de información del sector eléctrico; sin embargo, se deja plasmado un documento con experiencias satisfactorias, abriendo un espacio para la estandarización de los datos y de esta forma una mejorar el continuo flujo de información en las empresas.

CAPÍTULO 1

PROGRAMACIÓN ORIENTADA A OBJETOS

Una parte importante para la comprensión de este proyecto es la programación orientada a objetos (POO), la cual proporciona las herramientas necesarias para modelar sistemas mediante un paradigma que desarrolla aplicaciones informáticas gracias al modelado de objetos que comprenden características y acciones propias y que les permiten interactuar entre sí para lograr un objetivo común.

La programación orientada a objetos permite modelar los sistemas de una manera más cercana a la realidad. Piénsese por ejemplo en un automóvil: Un objeto que puede ser de un color, tiene un fabricante (o comercializador) y una serie de acciones específicas como ponerse en marcha, detenerse y pitar. En la programación orientada a objetos el automóvil sería una clase de objeto, que al darle valores concretos a cada una de las características (color, fabricante) se convertiría en una instancia de la clase automóvil, es decir, un objeto específico de éste. Las características de color, modelo y nombre, serían los atributos de dicho objeto y las acciones como parar, moverse o pitar serían los métodos o funciones a realizar por el objeto.

A continuación se abordarán más a fondo cada uno de los conceptos básicos de la programación orientada a objetos como: clases, objetos, atributos, métodos, herencia, entre otros y serán una síntesis de (Phillips, 2010) y (McMorran, 2007). Esto se hace para tener una visión mucho más clara de la importancia de la POO en la aplicación de este proyecto.

1.1 Diagramas UML

Para poder abordar el tema de programación orientada a objetos de una manera más clara, se utilizará una herramienta llamada: "diagramas UML”. Este es un lenguaje de modelado de sistemas de software que sirve para especificar, visualizar y documentar esquemas de dichos sistemas orientados a objetos, además de describir gráficamente su estructura, las interacciones, las relaciones y los comportamientos de las distintas clases que los componen. Las principales ventajas de este tipo de diagramas radican en su fácil lectura, interpretación y comprensión de la manera como se encuentra organizado.

Este tipo de diagramas muestran las diferentes clases que hacen parte del modelado de un sistema a partir de programación orientada a objetos con sus respectivos métodos y atributos, las relaciones que estas guardan con otras y el orden de jerarquía que señale el sistema. Todos los diagramas UML mostrados a lo largo de este proyecto son creados con el software “StarUML”(StarUMLSoftware, 2014).

1.2 Clases

Una clase es un conjunto de características y acciones, que representan un objeto de una forma general. A partir de la clase, es posible crear diferentes objetos, relacionados solo por sus características y acciones en común. Para entender esto de forma más clara se podría pensar en la clase <Conductor>, la cual contiene varias cualidades o características como: susceptancia, resistencia, longitud. La clase <Conductor>, no es un objeto específico sino un ente que contiene todo lo que representa a un conductor eléctrico. La clase <Conductor> almacena toda la información que haría que un objeto fuera considerado como un conductor eléctrico. En un diagrama UML las clases están representadas por un rectángulo dentro del cual se haya los atributos propios (Ver Figura 1.1).

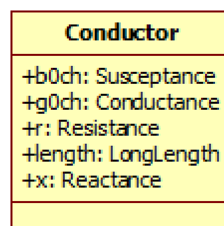

Figura 1.1.: . En este caso la clase es <Conductor> y posee los atributos de susceptancia, resistencia, longitud.

1.2.1 Objetos o instancias de clases

Referirse a un objeto o a una instancia de clase en programación orientada a objetos es equivalente. El término objeto se puede definir sobre cualquier cosa existente dentro de un código de POO, al igual que en el mundo real todo lo que es tangible se puede considerar como un objeto. Las instancias de clase son objetos que se pueden asociar entre sí y tienen su propio conjunto de datos y comportamientos, es decir, puede tener diferentes valores de datos de las características indicadas.

1.3 Atributos y métodos

Los atributos y métodos como ya se ha mencionado antes, son las características que definen a un objeto, estas contienen la información requerida para la formación de los mismos y especifican sus comportamientos ante la interacción con otros objetos de clase.

1.3.1 Atributos

Al tipo de características definidas por la clase también se le conoce como atributo, o en algunos casos como propiedades. Estos atributos pueden ser implícitos o explícitos, es decir, pueden ser de naturaleza primitiva por parte de la clase o ser adquiridos mediante una relación de herencia con otra clase.

1.3.2 Métodos

Los métodos son similares a las funciones de programación estructurada, solo que tienen acceso a todos los datos asociados con el objeto, y al igual que las funciones, pueden aceptar parámetros y valores de retorno. Los parámetros de un método son una lista de objetos que se transfieren al método que se está llamando, estos objetos son utilizados por el mismo para llevar a cabo la conducta o tarea que se pretende hacer. Por ejemplo, si los objetos son números, la clase `<Número>` podría tener un método `<sumar>` que acepta un segundo número como parámetro.

Dados el nombre del objeto y del método, un objeto de llamada puede invocar el método de un objeto de destino, esto se puede hacer si se pasan los parámetros necesarios como argumentos. Cuando dos objetos se unen por medio de una relación, uno puede adquirir un atributo que sea necesario para llamar un método en el objeto de destino, de esta forma se transmiten los parámetros necesarios para llamar al método que se necesite (vease, concepto de relación en la secccion 1.4).

Para los objetos individuales es posible adicionar modelos y métodos que permitan crear sistemas de objetos que interactúan entre sí. Cada objeto dentro del sistema es un miembro de una determinada clase, estas especifican que tipos de datos del objeto pueden estar en un estado diferente de otros objetos en una misma clase y cada uno de estos puede reaccionar de

formas distintas a las llamadas de sus métodos debido a esta diferencia de estado.

1.4 Relaciones entre clases

Las clases pueden tener algún tipo de relación entre ellas, es decir, puede existir un medio para que dos o más clases puedan interactuar entre sí para lograr un fin específico. Existen los siguientes tipos de relaciones entre clases: herencia, asociación, agregación o composición. A continuación se mostrarán más a fondo estas relaciones, pero, primero se debe entender un concepto llamado "multiplicidad" el cual es vital para la construcción de algunas relaciones.

1.4.1 Multiplicidad Es el número de objetos de una clase que se pueden relacionar con otro en el extremo opuesto de la relación, (ver Figura 1.2).

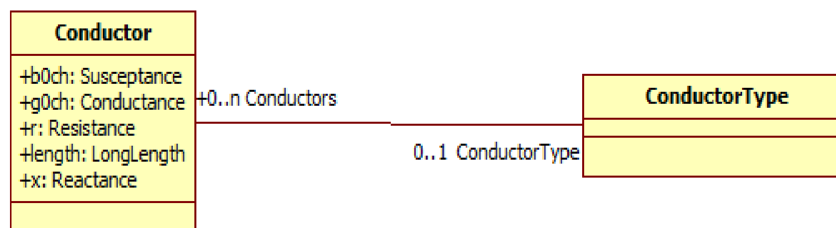

Figura 1.2.: Ejemplo de multiplicidad. Ninguno o muchos conductores estan relacionados con cero o un tipo de conductor.

1.4.2 Asociación Define una relación existente entre dos clases propias del sistema. Este tipo de relaciones se puede presentar cuando ninguna de las dos clases dependa o se componga directamente de la otra. Se representa en el diagrama mediante una línea que une dichas clases. Este tipo de relación es una manera de complementar y obtener un mejor análisis de la información y se muestra como un rango. Cuando este último tiende a infinito se representa con un asterisco (*) o con una "n" (ver Figura 1.3).

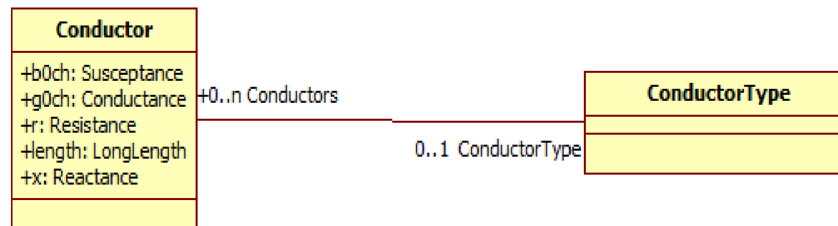

Figura 1.3.: Ejemplo de asociación. Conductor esta asociado con un tipo de conductor por medio de la multiplicidad descrita.

1.4.3 Composición

Es la acción de recoger varios objetos para componer uno nuevo. La composición es una buena opción cuando se trata de modelar un objeto que se compone de otros que ya existen. Si un objeto que es componente de otro se destruye, el objeto conjunto desaparecerá. Esta relación es representada por una línea que une las dos clases y en el extremo donde se encuentre la clase conjunto se ubica un diamante o rombo sólido, además se debe también especificar la multiplicidad de cada clase (ver Figura 1.4).

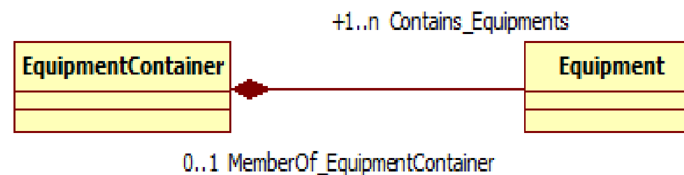

Figura 1.4.: Ejemplo de composición. Un contenedor de equipos se compone de uno o más equipos.

1.4.4 Agregación

Esta es una relación de acumulación la cual define como una clase puede ser parte de otra sin llegar a crear una interdependencia entre las dos, es decir, si en un punto determinado desaparece la clase agregada, la clase contenedora no se destruye. En estas relaciones también se especifica el rango de multiplicidad que posee cada clase. En el diagrama UML, este tipo de relación se representa mediante una línea que une las dos clases y en un extremo se halla un diamante o rombo sin relleno al lado de la clase que contiene la otra (ver Figura 1.5).

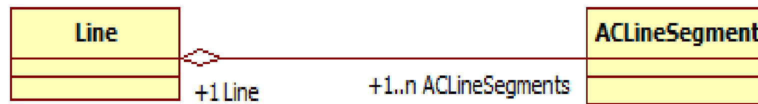

Figura 1.5.: Ejemplo de agregación. Una línea puede tener uno o más segmentos de línea.

1.4.5 Herencia o Generalización Este concepto representa la base para la jerarquía de clases. La herencia define una clase como una subclase de otra, a la cual se le llama "clase padre". La subclase hija hereda todos los atributos y métodos de la clase padre, además puede contener atributos y métodos propios. En el diagrama UML la herencia está representada por una flecha que une las dos clases relacionadas, donde la punta de la flecha se encuentra ubicada al lado de la clase padre (ver Figura 1.6). Solo en la relación de herencia se transmiten atributos a otras clases, esto no ocurre en las otras relaciones vistas anteriormente.

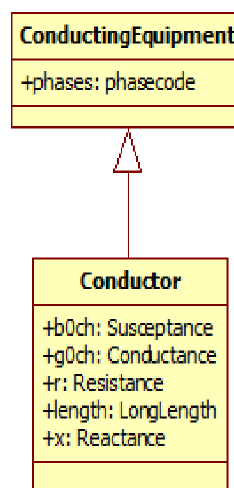

Figura 1.6.: Ejemplo de herencia. La clase <ConductingEquipment> es padre de <Conductor>.

1.4.6 Ejemplo A partir de los anteriores conceptos se puede modelar y tener un orden de jerarquía de clases de cualquier sistema haciendo más efectiva la comprensión del mismo. La unión de todos los esquemas anteriores conforma el diagrama UML (ver Figura 1.7).

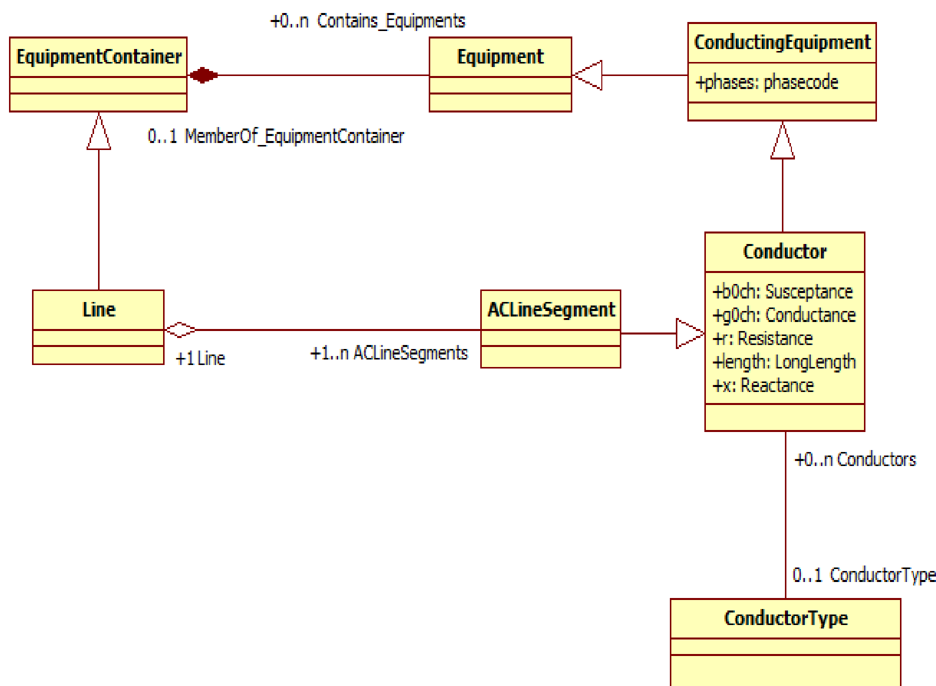

Figura 1.7.: Ejemplo de diagrama UML

1.5 Especificación de datos y comportamientos

Como ya se había mencionado, los objetos son instancias de clases que se pueden relacionar entre sí y contienen una colección de datos y comportamientos. Estas cualidades son la base de la programación orientada a objetos, por lo que se hace necesario conocerlas muy bien para poder comprender otros conceptos más especializados de este campo.

1.5.1 Datos Los datos en general describen las características individuales de un determinado objeto. Para entender mejor esto, se debe recordar que una clase define las características específicas que deben tener todas las instancias de esta, pero cada una puede tener valores diferentes de datos en esas características indicadas por la clase, de aquí se puede ver claramente que los datos, son valores específicos que adquiere una instancia en los atributos impuestos por la clase.

1.5.2 Comportamientos Los comportamientos de un objeto se determinan a partir de las acciones que pueda realizar este, por ejemplo, en un juego de ajedrez existe una clase llamada

<Ficha> y cada objeto de ficha puede tener un comportamiento dependiendo del tipo de acción que realice al moverse, es decir, una instancia de clase <Ficha> no necesariamente tendrá el mismo comportamiento al moverse, ya que un peón no se mueve igual que un caballo o una torre.

Los comportamientos son una tarea fundamental en el diseño de una aplicación informática, ya que se debe establecer la manera apropiada cómo se comportarán los objetos de las clases ante determinadas circunstancias o interacciones con otros objetos.

1.6 Abstracción

La abstracción se refiere al nivel de detalle con el que se diseña y está estrechamente ligada con la encapsulación y la ocultación de información, ya que ésta, también se puede describir como un proceso de encapsulación de información con interfaces públicas y privadas independientes estando ésta última sujeta a la ocultación de la información.

CAPÍTULO 2

INTRODUCCIÓN A PYTHON

En esta sección se explicarán a detalle los elementos básicos de la programación orientada a objetos en el lenguaje de programación Python. Como se vió en secciones anteriores los diagramas UML están conformados por: asociaciones, relaciones, clases, objetos y herencias. Estos pueden ser de gran ayuda al momento de conformar la estructura de una base de datos en general y así mismo subdividirla en otras más detalladas; pero por si solos estos diagramas no son funcionales, deben soportarse mediante un lenguaje de programación, el cual permita aplicar la estructura conformada en UML a uno o muchos casos específicos con características y comportamientos propios.

A continuación se explicará cómo se construyen algunos de los componentes de un diagrama UML en el lenguaje de programación Python 2.7 guiados por (Phillips, 2010) y (PythonOrg, 2014).

2.1 LISTAS

Una lista es un conjunto de elementos, los cuales ocupan un lugar específico dentro de este. Para construir una lista en Python basta con seguir la siguiente sintaxis:

```
lista=['elemento1','elemento2','elemento3']
lista=[1,2,3,4,5]
```

Se pueden declarar listas con elementos de diferentes tipos de datos ya sean numéricos, de cadenas de caracteres, booleanos, incluso objetos instanciados. Las listas son muy útiles en la programación orientada a objetos ya que permiten una flexibilidad en el tratamiento de la información gracias a su simplicidad en construcción y manejo; además, debido a las funciones propias que manejan las listas, se pueden manipular los datos de una manera más rápida y eficiente, agregando, borrando o modificándolo sin la necesidad de llegar al sitio exacto en donde se creó una lista específica. Las listas son una herramienta poderosa para encapsular datos que en conjunto se comportan como un solo elemento, es por eso que se debe tener especial cuidado al momento de seleccionar los elementos de una lista, ya que si dichos elementos no comparten una característica o un fin específico, en lugar de obtener mejoras se puede incurrir en errores que degradarían la eficacia del código.

2.1.1 Funciones de una lista

Función de búsqueda Muchas veces se quiere o se necesita saber si un elemento se encuentra en una lista, para esto se usa una línea de código que mostrará mediante un dato lógico si el elemento se encuentra o no en ella, por ejemplo:

```
lista=['elemento1','elemento2','elemento3']
print 'elemento1' in lista
True
```

Para este ejemplo es claro observar que <lista> si contiene a <'elemento1'> por lo que el resultado será `True` (verdadero). En aplicaciones tan sencillas como esta no es necesario el uso de esta función pero en códigos donde existen listas con muchos elementos, puede ser una gran ayuda para encontrar datos y manipularlos si se desea.

Index() En ocasiones no basta con conocer si un elemento se encuentra en una lista sino que también se requiere el índice que indica la posición del elemento en la lista para usar dicho dato en otras acciones del código. Para esto se usa la siguiente función:

```
print lista.index('elemento1')
0
```

El resultado (es decir el número cero) indica que <'elemento1'> se encuentra en el índice cero (0) de la lista ubicándose en la primera posición de la misma, comprobando esto se puede ver claramente que el resultado es acertado.

Append() Cuando se requiere agregar datos a la lista, se pensaría en buscar el fragmento de código en donde se encuentra esta y añadir cuantos datos sean necesarios, pero existe una forma más eficiente de lograr esto y es mediante el uso de la función <append()> la cual añade un nuevo elemento en la posición n+1 de la lista (siendo una lista de n subíndices llenos hasta ese momento).

```
lista.append('elemento4')
print lista
['elemento1', 'elemento2', 'elemento3', 'elemento4']
```

El elemento cuatro (<'elemento4'>) fue añadido de manera correcta comprobando así la funcionalidad de <append()>.

Count() Cuando los códigos se vuelven extensos, cabe la posibilidad de que por error se ingrese más de una vez un mismo dato en una lista, y esto no es recomendable. Si se desea saber cuántas veces se encuentra un elemento en una lista se usa la función <count()>.

```
print lista.count('elemento1')
1
```

El resultado indica que el elemento <'elemento1'> se encuentra una sola vez en la lista, lo cual es verdadero.

Insert() Si se quiere ingresar un dato en una posición específica de la lista se usa la siguiente función:

```
lista.insert(2,'nuevo elemento')
print lista
['elemento1', 'elemento2', 'nuevo elemento', 'elemento3']
```

Gracias a la función <insert()> se puedo agregar el elemento <'nuevo elemento'> en el índice dos (2) de la lista, pero el elemento anterior que estaba en dos se corre a la posición 3 y así hacia la derecha con los demás elementos.

Pop() Para borrar un elemento se usa la función <pop()>, la cual saca un dato de la lista en la posición del índice que se desee; sino se coloca el índice la función asume que se quiere sacar el elemento de la última posición.

```
lista.pop(2)
print lista
['elemento1', 'elemento2']
```

El elemento del índice 2 era 'elemento3' y como se puede apreciar no se encuentra ya en la lista.

Remove() La última función que se tocará para las listas es la función <remove()>. Esta acción es muy similar a <pop()> solo que no tiene nada que ver con los índices de una lista sino que elimina la primera ocurrencia de la lista, es decir, si se quiere eliminar <'elemento2'> se coloca en los paréntesis de <remove()>, automáticamente buscará la primer coincidencia de <'elemento2'> de la lista, si dicho elemento se encuentra varias veces y se quisieran eliminar todos se podría hacer uso de un <for> combinado con esta función.

```
lista.remove('elemento2')
print lista
['elemento1', 'elemento3']
```

2.2 Diccionarios

Un diccionario es un objeto de programación el cual se encarga de almacenar un grupo de datos en una variable al igual que una lista, solo que en los diccionarios los elementos pueden estar asociados a un índice de posición no numérico, a estos índices en los diccionarios se les conoce como "claves". Una clave es una señal que guarda o tiene asociado un tipo de dato, ya sea numérico, booleano, de cadena de texto, lista, incluso otro diccionario.

Los diccionarios tienen una ventaja muy grande sobre las listas, y es la ubicación de los datos en un conjunto, ya que al tener las claves se puede trabajar más fácil el dato requerido. Las claves de los diccionarios siempre deben ser datos numéricos o cadenas de texto, nunca una instancia, lista o tupla.

Para construir un diccionario basta con declarar una variable la cual contendrá los datos y las claves asociadas. A continuación se muestra un ejemplo de diccionario y las distintas formas de definir las claves y valores en el mismo.

```
Diccionario={ 'clave1':'elemento1',
2:'elemento2',
'clave3':49,
'clave4':['a','b','c'],
5:[1,2,3,4] }
```

El anterior es un ejemplo completo donde se puede apreciar distintas combinaciones de datos, existen claves que son cadenas de caracteres y otras que son valores numéricos, también se encuentran datos como cadenas de texto, números e incluso listas. Una diferencia importante con respecto a las listas es el uso de llaves en lugar de paréntesis planos. Los datos se separan de las claves por medio de dos puntos, y cada pareja conformada por clave y dato se separa una de la otra por medio de comas. Cabe resaltar que las parejas no necesariamente deben estar identadas una bajo la otra, el diccionario se puede escribir de forma lineal, es decir continuo utilizando una sola línea de código, pero para efectos de estética y una mejor apreciación se construirán de esta manera.

Al igual que en el uso de las listas, los diccionarios también poseen funciones propias que facilitan el trabajo con ellos. A continuación se mostrarán algunas de éstas, importantes para la correcta comprensión de este trabajo.

2.2.1 Funciones de los diccionarios

has_key() Se comenzará con una función muy sencilla la cual permite al usuario saber si una clave ya fue utilizada en un diccionario. Es muy importante que una clave no se encuentre dos veces en un mismo diccionario con datos distintos, ya que si se requiere trabajar con un dato que se encuentra en una de las dos claves repetidas puede que no se acceda al él directamente debido a que Python traerá a aclaración el valor de la primera ocurrencia con la clave requerida. La función <has_key()> busca una clave en un diccionario e indica mediante un valor lógico si se encuentra en uso dicha clave. Para efectos de practicidad se usará el siguiente diccionario ejemplo y a él se aplicarán las distintas funciones requeridas por este trabajo.

```
Diccionario={ 'clave1':'elemento1',
'clave2':'elemento2',
'clave3':'elemento3',
'clave4':'elemento4'}
```

Ejemplo de aplicación de la función <has_key()>:

```
print Diccionario.has_key('clave1')
True
```

Es verdad que <'clave1'> ya se encuentra en uso, es por eso que el resultado es <True>.

Keys() Si el programador desea ver las claves de un diccionario e inspeccionar visualmente si todo es correcto, se puede utilizar la función <keys()>. Esta función entrega como resultado en forma de lista todas las claves usadas en el momento por un diccionario .

```
print Diccionario.keys()
['clave4', 'clave1', 'clave3', 'clave2']
```

El resultado es el esperando; aunque puede notarse que las claves no se encuentran en el mismo orden en el que se introdujeron al diccionario.

Values() Del mismo modo que el programador puede solicitar las claves del diccionario puede querer solo los valores asignados a cada una de las claves, para esto se usa la función <values()> la cual entregará como resultado una lista con todos los valores de las claves del diccionario.

```
print Diccionario.values()
['elemento4', 'elemento1', 'elemento3', 'elemento2']
```

Al igual que con la función <keys()> la función <values()> entrega los datos desordenados.

Pop() Si se necesita borrar una clave, existe una función <pop()> similar a la utilizada en la subsección de listas, el funcionamiento de esta consiste en ingresar la clave que se desea borrar, cabe resaltar que el valor asociado a dicha clave también se perderá. A diferencia de la función <pop()> de las listas, esta tiene una utilidad extra en los diccionarios y es averiguar si una clave se encontraba en el diccionario, es decir, puede pasar que el programador desee borrar una clave que no existe en el diccionario, al aplicar la función <pop()> el resultado en consola será un mensaje de error. Para evitar que aparezca ese incómodo comunicado se puede agregar un argumento más a los paréntesis de la función <pop()> separado por coma, y será este argumento el que aparezca en pantalla de consola si la clave a borrar nunca existió en el diccionario. Para ver esto con más detalle se presentan dos ejemplos a continuación:

Ejemplo 1: borrando una clave existente en el diccionario

```
Diccionario.pop('clave3')
print Diccionario
{'clave4':'elemento4', 'clave1':'elemento1', 'clave2':'elemento2'}
```

Acertadamente se ha borrado la clave <`'clave3'`> y su valor asociado.

Ejemplo 2: borrando una clave inexistente en el diccionario

```
print Diccionario.pop('clave5','no se encuentra la clave')
no se encuentra la clave
```

Si se quiere verificar que la acción de borrado no realizó ningún cambio en el diccionario se puede imprimir el diccionario nuevamente.

Del Existe otra forma diferente a la función <`pop()`> para eliminar una clave de un diccionario, esta es la función <`del`>, funciona exactamente igual solo que no se tiene la posibilidad de conocer si la clave existía o no en el diccionario.

```
del Diccionario['clave4']
print Diccionario
{'clave1':'elemento1', 'clave3':'elemento3', 'clave2':'elemento2'}
```

De igual manera se ha borrado satisfactoriamente la clave <`'clave4'`>.

Clear() Si el programador, por alguna razón, desea borrar el contenido del diccionario basta con usar la función <`clear()`>, de esta forma el diccionario seguirá existiendo pero como un conjunto de elementos vacío.

```
Diccionario.clear()
print Diccionario
{}
```

El resultado es un diccionario vacío.

Añadir elementos a un diccionario o cambiar el valor de una clave Añadir elementos a un diccionario es mucho más sencillo que en una lista. Para esto no se necesita una función específica, basta simplemente con escribir la siguiente línea de código:

```
Diccionario['clave6']='elemento6'
```

De esta manera se agregará el nuevo elemento al diccionario o si la clave ya existe se cambiará su valor.

Copy() En muchas ocasiones se requiere aprovechar la existencia de ciertos diccionarios para determinadas acciones dentro del código sin llegar a modificar el contenido de los mismos, por esta razón sería ideal crear una copia de estos. Esto se logra con la función <`copy()`>.

```
Diccionario2=Diccionario.copy()
```

La copia del diccionario original se guarda en una nueva variable, logrando así aprovechar la información contenida en este sin temor a alterar de alguna forma los datos contenidos en dicho diccionario.

Len() Se puede determinar el tamaño de un diccionario por el número de elementos que contiene, esto se logra usando la función <`len()`>. Esta función regresa el número de claves que contiene el diccionario.

```
print len(Diccionario)
4
```

Efectivamente el diccionario contiene 4 elementos.

Get() En numerales anteriores ya se ha hablado un poco sobre como verificar si una clave existe o no en un diccionario, pero también existen otras para mostrar esto y quizá la mejor de todas sea la función <`get()`>, este método permite verificar de una forma más didáctica si una clave existe o no.

Ejemplo 1:

```
print Diccionario.get('clave2','no asignada')
elemento2
```

En este caso la clave existe, así que la función <`get()`> obtendrá el valor asociado a dicha clave.

Ejemplo 2:

```
print Diccionario.get('clave6','no asignada')
no asignada
```

Como la clave `'clave6'` no existe, el resultado será la frase `'no asignada'`.

Setdefault() Este método actúa de igual forma que la función <get()> con la única diferencia que crea una nueva clave en el diccionario cuando esta no existe en el momento y asigna un valor ingresado previamente. Para entender mejor lo mencionado se muestran dos ejemplos a continuación:

Ejemplo 1:

```
print Diccionario.setdefault('clave4','no asignada')
print Diccionario
elemento4
{'clave4': 'elemento4', 'clave1': 'elemento1',
'clave3': 'elemento3', 'clave2': 'elemento2'}
```

Como la clave existe en el diccionario la función devuelve el valor contenido en esta y no afecta la estructura original del diccionario.

Ejemplo 2:

```
print Diccionario.setdefault('clave5','no asignada')
print Diccionario
no asignada
{'clave5': 'no asignada', 'clave4': 'elemento4',
'clave1': 'elemento1', 'clave3': 'elemento3',
'clave2': 'elemento2'}
```

En este caso la clave no existía antes de invocar la función <setdefault()>. Este método lo que hace es construir dicha clave al interior del diccionario y asignar el valor de <‘no asignada’>, tal y como se puede apreciar en el resultado del ejemplo 2.

Update() Otro método muy utilizado es el <update()> este actualiza un primer diccionario con todas las parejas (clave, valor) de un segundo diccionario. Para las claves que son comunes a ambos, los valores del segundo diccionario sustituyen a los del primero. Para entender mejor esta función se muestra el siguiente ejemplo:

```
z = {1: ‘Uno’, 2: ‘Dos’}
x = {0: ‘Cero’, 1: ‘Un’}
x.update(z)
print x
```

```
{0: ‘Cero’, 1: ‘Uno’, 2: ‘Dos’}
```

Como se puede ver el diccionario <z> actualiza al diccionario <x> reemplazando el valor de la clave <‘1’> y agregando la clave <‘2’> con su respectivo valor.

2.2.2 Otras formas de crear diccionarios

Dict() A partir del constructor <dict()> se inicializa el diccionario con claves y valores:

```
Diccionario=dict(clave1='elemento1',
clave2='elemento2',
clave3='elemento3')
```

Zip() A partir de dos listas separadas se crea una lista con las claves y otra con los valores, la función <zip()> asocia cada elemento de igual subíndice y con ayuda del constructor <dict()> se crea el diccionario.

```
claves=['clave1','clave2','clave3']
valores=['elemento1','elemento2','elemento3']
Diccionario=dict(zip(claves,valores))
```

2.3 Clases

Las clases son un conjunto de atributos y métodos generales para las instancias creadas a partir de ella, también contienen las relaciones de herencia, asociación, composición, agregación, etc.

Para crear una clase en Python basta con establecer la siguiente sintaxis:

```
class NombreDeUnaClase():pass
```

Como se pudo observar, crear una clase en Python es relativamente sencillo. La palabra <pass> al final de la clase indica que ésta no contiene ningún atributo o método a ejecutar. Cabe resaltar que las clases se nombran con la convención genérica de la programación orientada a objetos donde siempre cada palabra que compone el nombre de la clase debe comenzar en mayúscula (por ejemplo, <Nombredeunaclase>).

2.3.1 Cadenas de caracteres

Python es un lenguaje de programación muy fácil de leer; sin embargo, al hacer programación orientada a objetos es importante reconocer claramente lo que hace cada objeto. Python admite este tipo de aclaraciones mediante la utilización de cadenas de caracteres. Cada clase, función o método puede tener una de estas cadenas estándar de Python en la primera línea después de su definición.

Las cadenas de caracteres es texto que admite letras, signos y números en código ASCII encerradas con apóstrofo (‘) o comillas (“), en estas se debe resumir de forma clara y concreta el propósito esencial de la clase o método que se está describiendo, debe explicar cualquier parámetro cuyo uso inmediato no es evidente, también pueden incluirse ejemplos cortos de cómo debe utilizarse, o pueden ser usadas como portadoras de información de las clases en sus atributos o funciones propias.

```
“creación de la clase”
```

2.3.2 Iniciación del objeto

En la programación orientada a objetos comúnmente existe un concepto llamado “constructor” el cual crea e inicializa un objeto cuando el programador lo requiera; por el contrario Python tiene un constructor y un inicializador por separado. La función constructora es poco usada, a menos que se quiera construir algo mucho más complejo, así que, primero todo se centraliza alrededor del inicializador.

El método de inicialización en Python es similar a cualquier otro método, excepto que tiene un nombre especial llamado: <__init__>. Los guiones a piso dobles iniciales y finales hacen referencia a que es un método especial propio del lenguaje.

```
“creación de la clase”
class NombreClase():
    def __init__(self,variableAtributo):
        self.atributo=variableAtributo
```

2.3.3 Identación en Python

La palabra identación en informática significa mover un bloque de texto hacia la derecha insertando espacios vacíos para así distinguirlo mejor del texto adyacente. La identación en ciertos lenguajes de programación solo se usa para mejorar la legibilidad del código por parte del programador y rara vez es de utilidad para el intérprete o el compilador; sin embargo, Python si hace uso de la identación para delimitar la estructura del programa permitiendo establecer bloques de código. A lo largo de este capítulo se podrá observar cómo influye la identación en el funcionamiento del código y se tendrá una visión más clara acerca de este concepto.

2.4 Atributos

Los atributos, como se vió anteriormente, son características propias de cada clase que le permiten almacenar la información que las describe. Para agregar atributos a una clase se usa el método de inicialización <__init__>. La estructura básica para agregar atributos a una clase es la siguiente:

```
class NombreClase():

    def __init__(self,variableAtributo):
        self.atributo=variableAtributo
```

La variable <variableAtributo> se define en el <__init__> con el objetivo de invocar las variables usadas en los atributos en el momento de instanciar una clase determinada, además permite a las clases hijas (es decir las clases que heredan de esta) copiar sus atributos posteriormente.

Los nombres de los atributos siempre comienzan con minúscula. Si el nombre es compuesto, la primera palabra que compone el nombre deberá comenzar con minúscula y las demás con mayúscula, de igual forma se declaran las variables de entrada para los valores de los atributos.

La palabra <self> que se antepone a las variables de los atributos dentro del inicializador <__init__> y de los nombres de los atributos, es un parámetro añadido automáticamente por el intérprete de Python por convención en las llamadas a los métodos y atributos de una clase, además, contiene una referencia al objeto que recibe la señal (el que recibe una llamada a un método o atributo), es decir <self> permite acceder a las propiedades y acciones de un objeto determinado, además representa al objeto mismo al que pertenece.

Nota: la palabra self no es esclusiva, puede ser reemplazada por cualquier otra siempre y cuando se siga usando posteriormente.

2.5 Herencia

Para crear la herencia basta con introducir el nombre de la clase padre en los paréntesis de la clase hija, esto sí y solo sí, la clase padre no contiene atributos propios o heredados de una clase superior; si por el contrario la clase padre contiene atributos propios que entregará a su o sus clases hijas se deben declarar en dichas subclases para poder ingresar valores a estos atributos si la aplicación lo demandase, ya que si no se hace de la manera correcta (que a continuación se mostrará) dichas subclases no podrán usar los atributos heredados por su padre. A continuación se observa una estructura sencilla para crear la herencia en Python.

```
class ClasePadre():
def __init__(self,variableAtributoPadre):
self.atributoPadre=variableAtributoPadre)
class ClaseHija (ClasePadre):
def __init__(self,variableAtributoHija,variableAtributoPadre):
ClasePadre.__init__(self, variableAtributoPadre)
self.atributoHija=variableAtributoHija
```

Como se pudo observar la clase <`ClaseHija`> hereda de la clase <`ClasePadre`>. Al momento de introducir esta última clase dentro de los paréntesis de <`ClaseHija`> automáticamente esta hereda el atributo <`atributoPadre`> pero no su variable de entrada <`variableAtributoPadre`> es por esto que debe ser definida en el inicializador <`__init__`> de <`ClaseHija`> , así pues, se debe poner dentro de la identación de los atributos la siguiente línea de código:

```
<ClasePadre.__init__(self, variableAtributoPadre)>
```

De esta forma ya se pueden ingresar valores a los atributos heredados de la instancias u objeto a construir de la clase hija.

2.6 MÉTODOS

Para definir un método en una clase se debe seguir la siguiente estructura:

```
class Nombreclase (self):
def __init__(self,variableAtributo):
self.atributo=variableAtributo
def nombreMetodo(self):
print ‘acciones a realizar por el metodo’
```

Como puede verse, el método se ingresa muy fácilmente colocando la palabra <`def`> seguido del nombre del método, paréntesis, dos puntos y dentro de la identación se declaran las acciones del método. En el ejemplo anterior la acción a realizar por el método será imprimir o mostrar: `‘acciones a realizar por el método’`.

Los métodos pueden aceptar argumentos provenientes de otros métodos u otro tipo de funciones o clases para aprovecharlos en la ejecución de las acciones a realizar por el mismo.

2.7 Instancias u objetos de clase

Una instancia de clase es un objeto particular de una clase. Para instanciar una clase se tiene la siguiente estructura de código:

```
NombreInstancia=ClasePadre()
```

En este caso <NombreInstancia> es una instancia de la clase <ClasePadre>, los paréntesis vacíos indican que la clase de la cual se desprende la instancia no contiene atributos, ya que de lo contrario se deberían llenar las variables de entrada de los atributos (esto se verá en detalle más adelante).

2.8 Relaciones entre clases en Python

Como se mencionó en el capítulo anterior de programación orientada a objetos, las clases pueden tener algún tipo de relación con otra. Las relaciones descritas como composición, agregación, asociación, permiten al programador obtener parámetros o datos de una clase a partir de otra.

Para lograr una asociación en Python no existe una convención estricta, la solución al problema es de libre consideración para el programador, este debe encontrar la mejor manera de relacionar las clases para alcanzar el objetivo que se requiera. Para el caso específico de este trabajo, la asociación debería poder darle al usuario y al mismo código en si las herramientas o rutas necesarias para la adquisición de datos a partir de una clase de referencia, es decir, al ubicarse en una clase determinada se debería poder tomar la información de las clases relacionadas con esta.

Para poder entender mejor este concepto, se mostrará un ejemplo tomado del capitulo anterior. Para lograr la asociación de las clases se hace lo siguiente:

Inicialmente se declaran las clases relacionadas, además se ingresan atributos a cada una de la clase con la cual está relacionada, y se asigna en el inicializador <__init__> con un valor base <None>.

```
class Clase1():
def __init__(self,variableAtributo1,variableAtributo2=None):
self.atributo1=variableAtributo1
self.clase2=variableAtributo2
class Clase2():
def __init__(self,variableAtributo2,variableAtributo1=None):
```

```
self.atributo2=variableAtributo2
self.clase1=variableAtributo1
```

Luego, al instanciar los objetos de las clases relacionadas, solo se ingresarían los datos de los atributos propios o heredados que se requieran para la construcción de dichas instancias.

```
Instancia_Clase1=Clase1('valorAtributo1')
Instancia_Clase2=Clase2('valorAtributo2')
```

Finalmente se debe agregar la instancia al objeto correspondiente en el atributo que se ha creado para almacenar dicha información.

```
Instancia_Clase1.clase2=Instancia_Clase2
Instancia_Clase2.clase1=Instancia_Clase1
```

No necesariamente un objeto debe tener solo una instancia asociada, puede tener múltiples de estas asociadas a un único atributo de relación. Para lograrlo, estos se pueden añadir mediante el uso de listas, por ejemplo, si se crea una nueva instancia de `<Clase2>` esta deberá asociarse al tributo de `<Clase1>` llamado `<Clase2>` como se muestra:

```
Instancia_Clase1=Clase1('valorAtributo1')
Instancia_Clase2_1=Clase2('valorAtributo2_1')
Instancia_Clase2_2=Clase2('valorAtributo2_2')
Instancia_Clase1.clase2=[Instancia_Clase2_1, Instancia_Clase2_2]
Instancia_Clase2_1.clase1=Instancia_Clase1
Instancia_Clase2_2.clase1=Instancia_Clase1
```

De esta forma se puede ver que no solo se puede asociar de 1 a 1 clases sino que también se puede establecer la multiplicidad requerida para un caso particular.

2.9 Llamar atributos o métodos

Para invocar o llamar atributos de una instancia de clase específica se deben seguir las siguientes estructuras de código o sintaxis:

```
NombreInstancia.nombreAtributo
NombreInstancia.nombreMetodo()
```

Como se puede ver, basta con poner el nombre de la instancia de clase seguido de punto y el nombre del atributo o método a invocar, la única diferencia radica en los paréntesis que se colocan al final cuando se desea instanciar un método. Si se quiere ver en consola el valor del atributo o los productos que ejecuta el método se debe anteponer `<print>` para que se imprima la información.

2.10 Módulos y paquetes

A medida que los códigos crecen se vuelve tedioso buscar fragmentos de este para corregirlo o actualizarlo, por lo tanto es importante o vital el tener organizada lo mejor posible la información sin perder la claridad en la lógica de la estructura del programa. A continuación se tratarán los dos factores que ayudan a organizar de una mejor manera la información y estructurar adecuadamente el programa que se está construyendo con Python.

2.10.1 Módulos Un módulo es simplemente un archivo de extensión ".py" en el cual existe parte o la totalidad del código que se esté formando en su momento, esto puede ayudar a que no toda la estructura esté sobre un mismo sitio y se pueda perder, además contribuye a identificar partes del código para su posterior intervención por actualización o corrección, es decir, los módulos actúan como lugares donde habita el código fraccionado para su fácil ubicación, además de permitirles actuar entre sí, siempre y cuando lo requieran.

2.10.2 Paquetes Un paquete no es más que una carpeta que contiene uno o más módulos con el único detalle que ésta debe tener un archivo (si se quiere vacío) de extensión ".py" con el nombre de `__init__.py` . Al igual que en los módulos, los paquetes se usan para separar la información de código y permitir una identificación mucho más eficiente de la estructura del programa (ver Figura 2.1).

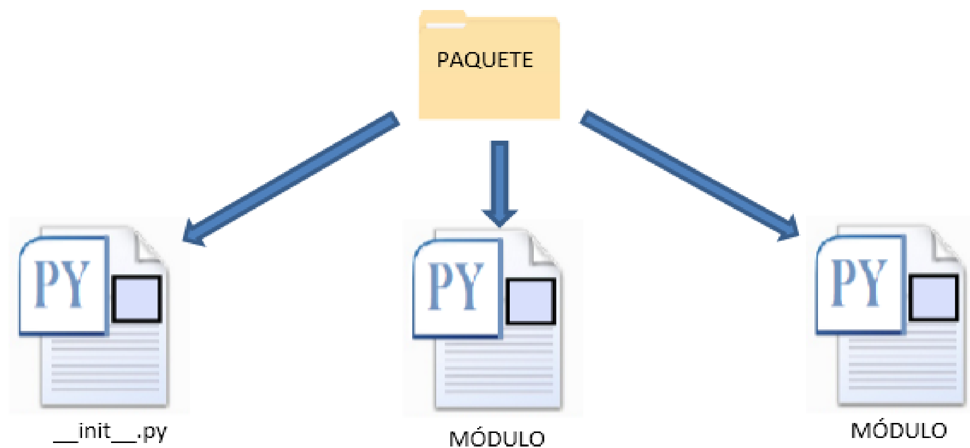

Figura 2.1.: Esquema de módulos y paquetes.

2.11 Importaciones

Cuando se usan paquetes y/o módulos en la programación de un código casi siempre se necesita que estos interactúen entre si de una forma óptima para que el programa realice las funciones requeridas por el usuario, esta relación se logra mediante el uso de importaciones las cuales se dividen en dos:

- Absolutas
- Relativas

Las importaciones en términos absolutos especifican la ruta completa al módulo, función, o el camino que se quiere importar, mientras que las importaciones relativas permiten adquirir un paquete de funciones dentro de las cuales se utilizarán un número específico de estas.

2.12 Ejemplo general

Ahora se mostrará un ejemplo para ver cómo funciona todo lo mencionado hasta el momento en este capítulo. El ejemplo se basa en crear una jerarquía de clases, con atributos propios y heredados, además se crearán algunas instancias donde se agreguen valores a los atributos requeridos por el ejemplo.

El ejemplo establecido es una base del mostrado en el capítulo anterior (ver figura 2.2), con un nivel de jerarquía establecido siendo la clase `<Equipment>` la de mayor nivel. También se tienen dos relaciones una de agregación entre las clases `<Line>` y `<ACLineSegment>` y una

de asociación entre <Conductor> y <ConductorType>. Las clases están distribuidas en dos paquetes: Core y Wires, (ver figura 2.3).

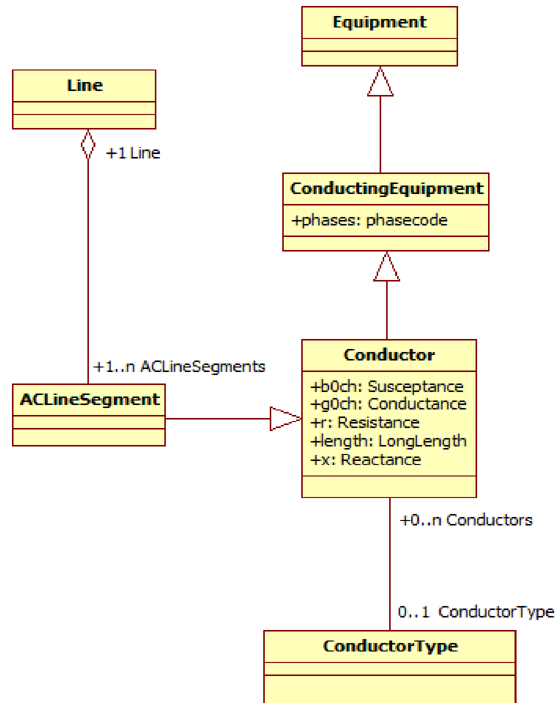

Figura 2.2.: Diagrama UML del ejemplo general.

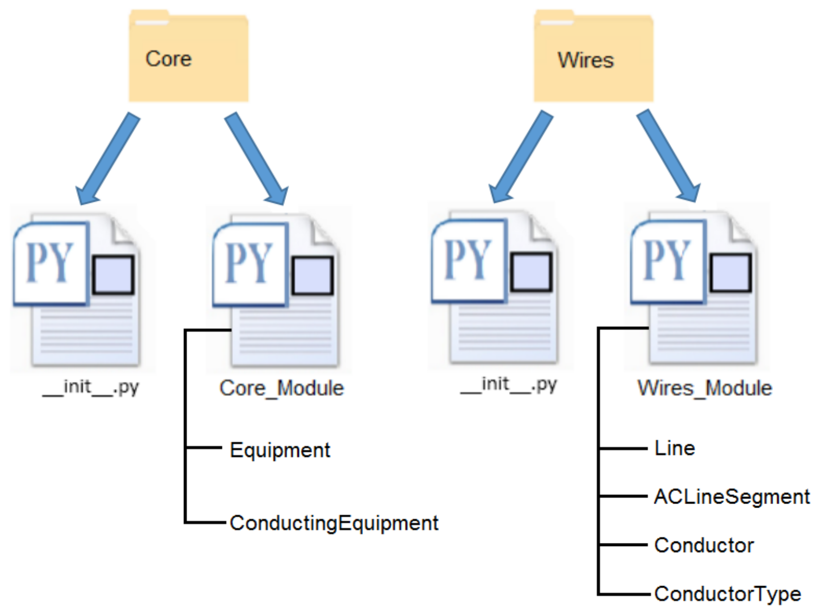

Figura 2.3.: Distribución de clases del ejemplo general.

El primer paso lógico es crear cada uno de los paquetes que contendrán a las clases involucradas en el ejemplo, nuevamente se resalta la importancia del modulo incializador `__init__.py`. Dentro de cada paquete se debe crear un módulo donde se van a construir cada una de las clases, en la figura 1.3 se puede observar que Core contiene a Core_Module.py y que este contiene a las clases <Equipment> y <ConductingEquipment>, de igual manera el paquete Wires contiene a Wires_Module.py y este a las clases <Line>, <ACLineSegment>, <Conductor> y <CoductorType>.

Una vez construidos los paquetes y módulos correspondientes se crean las clases en su lugar, por ejemplo: la clase <Equipment> se construirá en el paquete Core y en el módulo Core_Module.

Clases de Core En las clases de Core se encuentra la clase <Equipment> la cual es la de mayor nivel jerárquico en el ejemplo, no contiene ningún atributo propio por lo que se crea de manera sencilla y se cierra haciendo uso de la palabra `pass`. La clase <ConductingEquipment> tiene un atributo propio <phaseCode> así que la creación de esta clase tambien resulta sencilla.

```
class Equipment():pass
class ConductingEquipment(Equipment):

    def __init__(self,phaseCode):
```

```
        self.phaseCode=phaseCode
```

Clases de Wires Observando el diagrama UML del ejemplo se puede observar que la clase <Conductor> hereda de la clase <ConductingEquipment> la cual se encuentra en el paquete Core por lo que se debe hacer uso de una importación. En este paquete tambien se encuentran las relaciones de asociación y de agregación las cuales se establecieron como se explicó en secciones anteriores.

```
from Core.Core_Module import ConductingEquipment
class Conductor(ConductingEquipment):

    def __init__(self,b0ch,g0ch,r,length,x,conductorType=None):

        ConductingEquipment.__init__(self,phaseCode)
        self.b0ch=b0ch
        self.g0ch=g0ch
        self.r=r
        self.length=length
        self.x=x
        self.conductorType=ConductorType

class ConductorType():

    def _init__(self,conductor=None):

        self.conductor=conductor

class Line():

    def __init__(self,aCLineSegment=None):

        self.aCLineSegment=aCLineSegment

class ACLineSegment(Conductor):

    def __init__(self,line=None):

        Conductor.__init__(self,b0ch,g0ch,r,length,x)
        self.line=line
```

Ya terminadas las clases involucradas en el diagrama UML del ejemplo tal y como se explicó en este capítulo, se procede a instanciar las clases requeridas por el usuario o el sistema que se este programando. Para este caso se crearon instancias para todas las clases del ejemplo en un módulo y paquete diferentes a los ya instaurados para la creación de las clases, por lo

que se debe recurrir nuevamente a las importaciones para llevar la información de las clases y poder construir instancias de un caso particular que contenga información específica.

Creación de Instancias Primero se construirán las instancias y se darán los datos a cada uno de los atributos presentes en cada una de ellas. Para efectos de sencillez en el ejemplo las instancias serán llamadas por el mismo nombre de la clase, un identificador "inst" y un número para diferenciarlas de otras similares.

```
from Core.Core_Module import*
from Wires.Wires_Module import*
Inst_Equipment_1=Equipment()
Inst_ConductingEquipment_1=ConductingEquipment(ABC)
```

ABC corresponde al codigo de fases del equipo.

```
Inst_Conductor_1=Conductor(
                            b0ch=0.15,g0ch=0.028,
                            r=0.76, length=100,
                            x=0.09,phaseCode=ABC)
```

Se hace uso de la identación para mejorar el orden de los atributos de la clase <Conductor> además se especifican las igualdades para asiganr correctamente los valores.

```
Inst_ConductorType_1=ConductorType()
Inst_ACLineSegment_1=ACLineSegment(
                             b0ch=0.12,g0ch=0.045,
                            r=0.34, length=50,
                            x=0.056,phaseCode=ABC)
Inst_ACLineSegment_2=ACLineSegment(
                             b0ch=0.145,g0ch=0.057,
                            r=0.38, length=50,
                            x=0.059,phaseCode=ABC)
```

Se crearon dos segmentos de línea los cuales conforman una sola línea de transmisión.

```
Inst_Line_1=Line()
```

Relaciones Una vez terminadas las instancias se puede proceder a relacionarlas. La asociación entre <Conductor> y <ConductorType> se lleva a cabo introduciendo como parámetro las instancias en un atributo.

```
Inst_Conductor_1.conductorType=Inst_ConductorType_1
Inst_ConductorType_1.conductor=Inst_Conductor_1
```

Ahora para la agregación los segmentos de línea se agregan a la línea en forma de lista.

```
Inst_Line_1.aCLineSegment=[Inst_ACLineSegment_1,Inst_ACLineSegment_2]
Inst_ACLineSegment_1.line=Inst_Line_1
Inst_ACLineSegment_2.line=Inst_Line_1
```

Los fundamentos de este capítulo son necesarios para la correcta comprensión de la forma en la que se aplicó el estándar IEC 61970 a un circuito prueba. La creación de los elementos involucrados por la norma para la formación de una base de datos universal con programación orientada a objetos debe hacerse por medio de un lenguaje de programación bien estructurado, Python ofrece muy buenas herramientas para ello, posee un lenguaje limpio y sencillo, una amplia flexibilidad para que el programador pueda construir cada detalle impuesto por el estándar y de esa manera no incurrir en faltas hacia este. De lo propuesto aquí se halla la razón del porqué se entrega un espacio en este documento para tratar aspectos, métodos y características del lenguaje de programación Python, y así el lector de este artículo pueda entender, y si es necesario manipule el código generado para la creación de la base de datos del circuito de prueba.

CAPÍTULO 3

INTRODUCCIÓN AL ESTÁNDAR IEC 61970-301 CIM

El estándar IEC 61970-301 comprende la base del proyecto en cuestión, ya que el objetivo general de este trabajo, es generar buenas prácticas para la aplicación de dicha norma a los distintos sistemas de energía eléctrica; sin embargo el desarrollo de dichas prácticas para la implementación no son aprovechables si no se tiene una concepción básica de lo que es la norma, cómo se interpreta y cómo debe aplicarse. Debido a que el estándar ofrece una variedad muy grande para la implementación de la misma en los diferentes campos de acción del sector eléctrico como: generación, transmisión o distribución y sus combinaciones, puede resultar tedioso tratar de aplicar las prácticas aquí desarrolladas sin antes tener un manejo óptimo de la norma. De aquí nace la importancia de este capítulo, en el cual se muestra una síntesis de ésta y se dan las herramientas y guías necesarias para su correcta comprensión.

De acuerdo a lo señalado en el mismo, este es un modelo estándar orientado a objetos que busca representar mediante clases, atributos y relaciones los diferentes elementos que conforman un sistema de energía eléctrica, facilitando de esta forma el desarrollo y la integración de los diferentes mecanismos utilizados en la planificación, la gestión y la interoperabilidad de los datos que se tengan de dicho sistema.

Cada una de las clases y las diferentes relaciones que hacen parte de esta norma, son representadas mediante diagramas UML, mencionados y descritos en secciones anteriores, los cuales señalan el orden jerárquico de las clases del modelo en un caso particular. El estándar cuenta con una gran cantidad de diagramas UML que representan los principales elementos u objetos que se tienen en un sistema de energía y que pueden ser aprovechados para la construcción de sistemas más complejos. A continuación se presenta una síntesis de los principales conceptos y componentes de la norma (IEC, 2009).

3.1 Paquetes CIM

El estándar IEC 61970-301 cuenta con un conjunto de paquetes cuya función es clasificar las clases para identificar, comprender y revisar fácilmente cada una de estas al momento de implementar dicha norma. Cada paquete contiene las clases con sus respectivos atributos, las relaciones de herencia, asociación y agregación de cada una, que describen los elementos asociados al nombre del paquete. Las relaciones entre clases de diferentes paquetes no son un problema en este caso, debido a que la misma estructura del estándar lo especifica en muchas de las clases que la componen, queriendo mostrar de esta forma que el CIM es el conjunto completo de dichos paquetes.

Los paquetes contenidos en el estándar IEC 61970-301 se muestran en la Figura 3.1. Allí se muestran las relaciones de dependencia entre los paquetes que componen la norma. El paquete que está en el extremo de origen de la flecha depende de aquel al cual llega dicha flecha, es decir, muchas de las clases del paquete dependiente poseen relaciones con las clases del otro paquete, lo cual se hace evidente cuando se está creando la estructura del estándar mediante un lenguaje de programación. Observando con más detalle, en este diagrama puede verse claramente que existen dos paquetes base para los demás en toda la norma “Core” y “Domain”.A continuación se da una breve descripción de cada uno los paquetes del estándar.

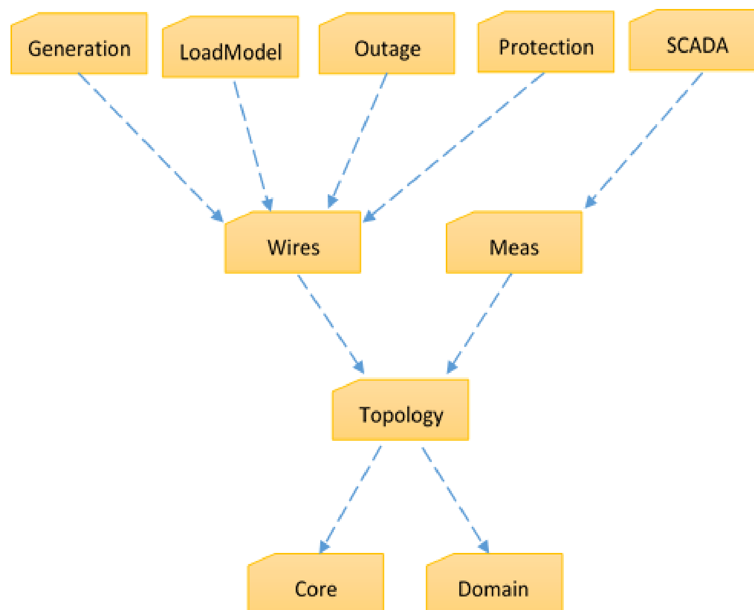

Figura 3.1.: Esquema de paquetes en el CIM.

3.1.1 Domain Las clases contenidas en este paquete hacen referencia a los diferentes tipos de atributos que contendrán las demás clases, es decir las propiedades de los objetos que modela el CIM también son objetos de otras clases las cuales se encuentran contenidas en este. El paquete "Domain" actúa como un diccionario de datos que señala las cantidades y unidades admisibles para cada atributo de las clases contenidas en los otros paquetes.

3.1.2 Core En este paquete se hallan las clases que actúan como soporte de las demás contenidas en otros paquetes. Contiene clases como: `<IdentifiedObject>`, `<PowerSystemResource>`, `<ConductingEquipment>`, `<Equipment>`, `<EquipmentContainer>`, entre otras, las cuales representan la base para el modelado de sistemas a implementar. Este paquete sólo se relaciona con el paquete "Domain" para asignar a los atributos los tipos de dato que manejan en cada una de las clases.

3.1.3 Topology Este paquete gestiona toda la información concerniente con los puntos de conexión de los equipos eléctricos a través de las relaciones con la clase `<Terminal>` contenida en el paquete "Core". También muestra la forma en la cual los equipos son conectados entre sí a través de interruptores.

3.1.4 Wires Este paquete es utilizado, en su mayoría, para mostrar las características eléctricas de las redes de transmisión y distribución. Posee clases específicas que representan los equipos eléctricos, algunas de estas como: `<PowerTransformer>`, `<Line>`, `<EnergyConsumer>`, `<Breaker>`, entre otras.

3.1.5 Outage Con este paquete se busca crear modelos de información sobre el estado actual o futuro de la red. Contiene clases como: `<ClearanceTag>`, `<OutageSchedule>`, entre otras, en las cuales se muestran las etiquetas de los equipos y/o máquinas que se encuentren fuera de servicio y el tiempo que estarán en este estado.

3.1.6 Protection Este paquete muestra la forma en la que se gestiona la información referente a las diversas protecciones que se tienen en un sistema determinado. Cuenta con clases como: `<CurrentRelay>`, `<SynchrocheckRelay>`, `<RecloseSequence>`, entre otras, con las cuales se modela el sistema de protecciones de un circuito eléctrico.

3.1.7 LoadModel En este paquete se describen las características de los usuarios de la red eléctrica, esto en su mayoría, a través de curvas de demanda. Este paquete, guarda una

estrecha relación con la clase <EnergyConsumer> debido a que está posee una gran cantidad de atributos para el modelado de una carga eléctrica y muchas de las clases contenidas en el paquete en cuestión heredan de esta clase. De igual forma este paquete contiene clases para modelar factores que afecten directamente el estado de la carga como es el caso de las estaciones climáticas y el tipo de día, lluvioso o seco.

3.1.8 Generation Este paquete se divide en dos sub-paquetes que son: "Production" y "GenerationDynamics" los cuales se describen a continuación:

3.1.8.1 Production Es un paquete que contiene los modelos usados para diferentes tipos y fuentes de generación. Además cuenta con algunas clases las cuales gestionan la información de costos y unidades de reserva.

3.1.8.2 GenerationDynamics Este paquete brinda modelos para describir las características de elementos como turbinas y calderas, así como los factores que de alguna forma los afectan. Posee una serie de clases con las cuales se pueden caracterizar o modelar diferentes tipos de estos elementos como es la clase <CombustionTurbine>, <DrumBoiler>, <FossilSteamSupply>, entre otras.

3.1.9 SCADA Contiene las clases <CommunicationLink>, <RemotePoint>, <RemoteControl>, entre otras, con las que se busca gestionar la información de unidades remotas, sistemas de control de subestaciones y centros de control.

Para obtener una descripción más detallada de cada una de las clases que hacen parte de estos paquetes es necesario acudir al estándar, el cual describe por separado cada uno de los paquetes, así como a las clases que los componen con sus respectivos atributos y relaciones.

3.2 Clases y relaciones del CIM

Como se mencionó anteriormente, una clase vista desde la programación orientada a objetos describe una serie de características de un tipo de objeto que estaría representado como una instancia de clase de la misma, por ejemplo: para un sistema de potencia determinado los principales objetos a modelar serán los transformadores de potencia, los generadores, las cargas y las líneas de transmisión, los cuales estarán representados en forma de objetos de alguna clase presente en los distintos paquetes de la norma, de esta forma se busca analizar, almacenar y facilitar el intercambio de los datos que contienen los equipos y las relaciones entre ellos.

Cada clase posee atributos ya sean propios o heredados que señalan las características del objeto o instancia de la misma. El estándar IEC 61970-301 hace una descripción de cada una de las clases contenidas en los paquetes junto con los atributos propios y heredados que estas puedan tener, además de la especificación del tipo de atributo en cada caso que se representa como instancias de las clases del paquete “Domain” lo cual se explicará con más detalle más adelante. De igual forma se describen las relaciones que posee cada clase con otra, buscando tener una mejor organización de la información.

El estándar muestra en cada sección diagramas UML para algunos equipos, subsistemas aplicados a los circuitos eléctricos de potencia o simplemente de la estructura jerárquica que presenta la mayoría de clases contenidas en un determinado paquete, señalando de igual forma, si una de esas clases proviene de un paquete diferente y se hace necesaria en la estructura.

3.3 Estructura de las clases

Para entender e identificar con mayor facilidad una clase determinada dentro del estándar, a continuación se mencionan algunas pautas referentes a la organización y descripción que hace este de todas las clases que lo componen.

Como se mencionó anteriormente la norma está compuesta por una serie de paquetes dentro de los cuales se describen cada una de las clases. Estos paquetes se resumen en un listado que se muestra a continuación:

- Core
- Domain
- LoadModel
- Meas
- Outage
- Topology
- Wires
- Generation
- Production
- GenerationDynamics

- SCADA
- Protection

Dentro de cada uno de estos paquetes se encuentran las clases presentadas en orden alfabético con sus respectivos atributos propios y/o heredados. A continuación se muestra la estructura en la que se ha organizado toda la información relacionada con las clases:

Nombre de la Clase
Descripción de la clase
Atributos Nativos

nombreAtributo	Tipo de dato	Definición del atributo

Atributos heredados

nombreAtributoHeredado	Tipo de dato	Clase a la que pertenece dicho atributo

En lo referente a las relaciones, el estándar IEC 61970-301 las presenta de la siguiente manera para cada una de las clases:

Asociaciones Nativas

Multiplicidad propia de la clase en cuestión	Nombre dado a la asociación vista desde esta clase	Multiplicidad de la clase relacionada	Clase con la que se relaciona	Descripción de la asociación

Relaciones heredades desde su Clase padre

Multiplicidad propia de la clase padre	Nombre dado a la asociación vista desde la clase padre	Multiplicidad de la otra clase	Clase con la que existe la relación	Clase padre

Relaciones heredades desde clases superiores en el orden de jerarquía

Multiplicidad de la clase de donde se está heredando	Nombre dado a la asociación vista desde la clase desde donde se hereda	Multiplicidad de la otra clase	Clase con la que existe la relación	Clase heredada y dueña de esta asociación

La mejor forma de explicar con detalle estos esquemas es a través de un ejemplo, directa-

mente del estándar IEC 61970-301. Se toma como ejemplo la clase <PowerTransformer> que hace parte del paquete Wires página 137 de la norma.

La primera parte de la arquitectura de la norma señala el nombre de la clase que en este caso es <PowerTransformer>, posteriormente una breve descripción de la clase y continúa con los atributos nativos o propios de esta, especificando el tipo de atributo, los cuales, como se puede observar son instancias de clase del paquete "Domain" y seguido a esto se encuentra la descripción propia del atributo. Lo mismo ocurre para los heredados con la única diferencia que en la descripción se indica la clase de donde se heredan dichos atributos.

En la segunda parte de la arquitectura se muestran las relaciones propias de la clase que en este caso se entiende de la siguiente manera:

- Un objeto de <PowerTransformer> está asociado con cero o almenos un objeto de la clase <HeatExchanger>, lo cual se enuncia en la descripción de la asociación y lleva a determinar que es una relación de tipo agregación. Lo dicho anteriormente se puede ver en la figura 3.2 :

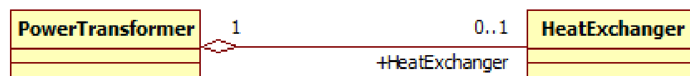

Figura 3.2.: Agregación 1 de la clase PowerTransformer.

- Un objeto de <PowerTransformer> está asociado con uno o más objetos de la clase <TransformerWinding>, es decir, que un transformador de potencia puede tener uno o más devanados de transformación (ver figura 3.3):

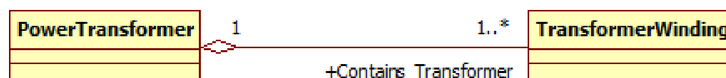

Figura 3.3.: Agregación 2 de la clase PowerTransformer.

Es importante entender el significado de cada una de las relaciones que posee la clase, para determinar si son de tipo asociativo o de agregación debido a que el estándar no lo muestra de forma explícita en la descripción de las clases. Aunque cabe recalcar que todas estas relaciones se pueden ver claramente en los diferentes diagramas UML con los que cuenta la norma y que

ayudan a una mejor compresión de la misma. Las relaciones heredadas por parte de otras clases, con un rango mayor en la jerarquía, hacen referencia a las relaciones que posee la clase de la que se está heredando, por ejemplo: en este caso, <PowerTransformer> hereda directamente de la clase <Equipment> la cual posee una relación con <EquipmentContainer> de esta forma existe una relación heredada entre <PowerTransformer> y <EquipmentContainer>. A su vez <Equipment> hereda de <PowerSystemResource> a la cual se le especifican las relaciones que posee, y así se continúa estableciendo cada relación que tienen las clases que participan en el orden de la jerarquía. Con esto se busca que pueda existir una conexión entre la información propia de las clases y la de otras a través de las relaciones existentes con su clase padre.

Algo importante en la comprensión del estándar es identificar la clase de la cual se está heredando directamente, esto se logra observando la estructura que se mostró anteriormente de la clase. La clase padre será aquella de la cual se desprenda el primer conjunto de relaciones que se encuentre al revisar la clase en cuestión. Para este ejemplo se observa que las primeras relaciones heredadas pertenecen a <Equipment> por ende <PowerTransformer> es una sub-clase de <Equipment>, lo cual se puede verificar acudiendo al estándar y observando los diagramas UML que este posee para el modelado de un transformador de potencia; sin embargo puede suceder que una clase no tenga ninguna relación heredada pero si tenga atributos heredados, en este caso la clase padre será aquella que se presente en primer lugar cuando se esté describiendo el conjunto de propiedades que contiene la clase.

3.4 Otros conceptos importantes

Para comprender de manera correcta la norma es necesario conocer algunas de las clases principales junto con ciertos conceptos que giran alrededor de estas y que representan la base para el modelado de la mayoría de los sistemas de energía eléctrica.

3.4.1 Clase <IdentifiedObject> Esta clase contenida en el paquete "Core", representa la base de la jerarquía del estándar, ya que la mayoría de las clases de la norma heredan atributos por parte de esta, ya sea de forma directa o a través de otras subclases dentro de la jerarquía de clases del estándar. La clase <IdentifiedObject> posee seis atributos de tipo "string"[1]con los cuales se busca dar un nombre y ciertas especificaciones o descripciones generales a la instancia u objeto que se esté creando. En la página 45 de la norma, se puede observar en detalle esta clase junto con la descripción de cada uno de sus atributos.

[1]Cadenas de caracteres, concepto explicado en la sección "Introducción a Python"

3.4.2 Concepto de conectividad En los sistemas de energía eléctrica el término conexión no es desconocido, por el contrario es un elemento vital para el funcionamiento de las redes eléctricas. La forma en la que se conectan los equipos a través de nodos tiene su espacio en el modelado del CIM.

Primero que todo para entender este concepto es necesario conocer que existe una relación entre la clase <ConductingEquipment> y la clase <Terminal>, y que además, esta última también posee una relación con la clase <ConnectivityNode> formando así el modelo de conectividad de un sistema. Es vital aclarar que la clase <ConductingEquipment> ocupa un nivel alto en la estructura jerárquica de la norma, permitiéndoles a todos los equipos de una red de energía eléctrica que puedan tener acceso a atributos como voltaje o número de terminales, etc.

Como la misma norma lo especifica, un elemento de la clase <Terminal> está asociado con uno de <ConductingEquipment>, pero un <ConductingEquipment> puede tener muchos elementos de <Terminal>. De igual forma un terminal puede ser conectado a un objeto de la clase <ConnectivityNode> pero este puede estar asociado a muchos objetos de <Terminal>, además hace parte de un <TopologicalNode>. La descripción de cada una de estas clases y estas relaciones se encuentra en el estándar. En la página 25 literal 4.4.2.1 de la norma se puede observar un ejemplo con el cual se hace una explicación más detallada de dicho concepto.

Esta sección es solo una breve introducción al estándar, aquí solo se mostró como está escrita, como está clasificada la información y como están estructuradas las clases del CIM, todo esto es necesario para la construcción de las clases del estándar en un lenguaje de programación para la posterior instanciación de distintos elementos que en conjunto se comportan como los sistemas reales de energía eléctrica.

CAPÍTULO 4

ORGANIZACIÓN DE LA INFORMACIÓN

En este capítulo se mostrará el uso del espacio de nombres para la correcta clasificación y organización de la información en la herramienta informática construida para la base de datos del CIM, es decir, se podrá ver como los archivos (o módulos de Python), los objetos instanciados y otras variables a usar dentro del proyecto, fueron nombrados para facilitar la gestión de los mismos y de esta forma alcanzar los objetivos propuestos.

También se muestra en forma clara como se distribuye la información en las distintas carpetas existentes dentro del proyecto y como se relacionan entre sí; no específica en detalle uno a uno los elementos que la componen, pero si da una idea general, la cual es suficiente para entender la estructura completa de este trabajo. Entender como está organizado este proyecto facilita y da una idea de las secciones posteriores las cuales comprenden el tema de aplicación y de interfaz gráfica de usuario para la gestión de los datos la cual se podrá ver en detalle en la sección 5.

4.1 IDENTIFICADORES USADOS EN EL PROYECTO

Como se mencionó en ocasiones anteriores, las instancias de clase del CIM y los diccionarios fueron creados con una estructura determinada en su nombre, esto se hizo para aprovechar el espacio de nombres y generar aplicaciones más pequeñas que ayudasen a la gestión optima de dichos elementos. En el estándar IEC 61970-301 las clases ya tienen un espacio de nombres determinado para cada una, al igual que sus atributos. A continuación se describirán los identificadores utilizados..

Identificadores: "instance" y "código del equipo" para instancias de clase Al momento de crear una nueva instancia de clase del CIM, se debe seguir una estructura general establecida por los autores de este proyecto para el correcto funcionamiento de cada una de las aplicaciones desarrolladas. Esto no implicaría en si un esfuerzo adicional ya que las reglas siguen las convenciones descritas en la IEC61970-301 y si permite una foirmaeficiente de organizar la información. La estructura genérica se muestra a continuación:

```
<Instance_NombreClaseCIM_CODIGODELEQUIPO+numerodelequipo>
```

Siempre se debe colocar en primer lugar la pablara "Instance" al crear un objeto, seguido de un guion al piso, luego el nombre de la clase de la cual es instancia el objeto, se coloca un nuevo guion al piso, posteriormente el código del equipo del circuito que se esté creando y el número que le corresponda a este. El código del equipo también es una convención establecida por los autores para usarlos en subfunciones dentro del código y se muestran en la tabla 4.1:

TABLA 4.1.: Código de equipos.

Equipo	Código
Línea	LN
Transformador	TRF
Carga	CAR
Generador	GEN
Subestación	SUB

Los códigos de los equipos siempre deben ingresarse no solo en el nombre de una instancia, sino que también debe estar contenido dentro del atributo `<localName>` heredado de la clase `<IdentifiedObject>` esto se hace para tener un nivel de identificación más complejo y poder clasificar todas las instancias pertenecientes a un equipo particular. No se hizo necesario crear más códigos a excepción de estos cinco y es de vital importancia que se utilicen en la creación

de cualquier objeto en el que se involucre alguno de estos equipos. Por ejemplo, al crear los terminales de una línea se debe usar la clase `<Terminal>` y siguiendo la estructura anterior los terminales se llamarían:

```
<Instance_Terminal1_LN1>
<Instance_Terminal2_LN1>
```

O bien:

```
<Instance_Terminal_LN1_1>
<Instance_Terminal_LN1_2>
```

Lo anterior se podría leer como: instancia 1 de terminal de la línea 1 e instancia 2 de terminal de la línea 1. Puede verse que en el nombre del objeto existe un número al lado del nombre de la clase de la cual es instancia, esto se hace para diferenciar los terminales, pero no influye en el manejo del espacio de nombres ya que la parte del nombre de la clase solo sirve para identificar más fácilmente los objetos creados a partir de esta. Si para el programador esto resulta incómodo puede valerse de la segunda forma y no habría inconveniente. Concluyendo esta subsección se puede establecer que se debe respetar rigurosamente la palabra "Instance" y el "código del equipo" al momento de crear una nueva instancia.

Identificador "dic" para diccionarios Existen dos tipos de diccionarios en la creación de este proyecto uno es usado en la instanciación de atributos de una clase, ya que como se vió en capítulos anteriores, los atributos pueden ser a su vez instancias de clase del paquete de "Domain" contenido en el estándar, y el otro es usado para ingresar los valores de los atributos de una clase en la creación de una nueva instancia. Para el primer caso en el que los atributos son instancias, el diccionario debe seguir la siguiente sintaxis:

```
<dic_nombreDelAtributo_NombreDeLaClase_CodigoDelEquipo+numerodelequipo>
```

Cada vez que se cree este tipo de diccionario debe comenzar con las siglas "dic" seguido por un guion al piso, luego el nombre del atributo en cuestión, guion al piso, el nombre de la clase a la cual pertenece el atributo, nuevamente un guion al piso y por último el código del equipo especificado en la tabla 1.1 más el número asignado a este.

Por ejemplo al crear el diccionario para el atributo `<powerFactor>` de `<EnergyConsumer>` el cual es instancia de la clase `<PowerFactor>` este se debe llamar de la siguiente forma:

```
<dic_powerFactor_EnergyConsumer_CAR1>
```

La anterior linea de codigo se puede leer de la siguiente forma: diccionario del atributo `<powerFactor>` de la clase `<EnergyConsumer>` perteneciente a la carga número 1.

Para el segundo tipo de diccionario la estructura es idéntica a la anterior solo que no lleva el nombre del atributo ya que no es un elemento enfocado a este, sino que pertenece directamente a una clase en particular. Por ejemplo si se quiere crear un diccionario para ingresar los atributos de la carga 1, se debería escribir lo siguiente:

```
<dic_EnergyConsumer_CAR1>
```

Lo mostrado en la anterior línea de código se lee como: diccionario de la instancia de la clase `<EnergyConsumer>` de la carga 1.

Por último para que la interfaz gráfica de usuario pueda reconocer los diagramas unifilares en la ubicación de los diferentes equipos, estos deben llamarse como se define a continuación:

```
<unifilar+CodigoDelEquipo+NumeroDelEquipo>
```

Por ejemplo si se creó una línea cuyo número de equipo es 3, es decir se creó la línea 3, el unifilar que define la ubicación del nuevo elemento deberá llamarse: `<unifilarLN3>`, de esta forma la aplicación GUI reconocerá dicho diagrama y lo mostrará en el widget dispuesto para esta funcionalidad.

4.2 Distribución de la información

Cuando se trabaja con aplicaciones informáticas extensas, se debe llevar un orden y una distribución de la información de los bloques de programación necesarios para la construcción de un software. A continuación se explica cómo se organizó el código generado con base en el estándar IEC 61970-301, para generar un ambiente de trabajo óptimo claro y conciso.

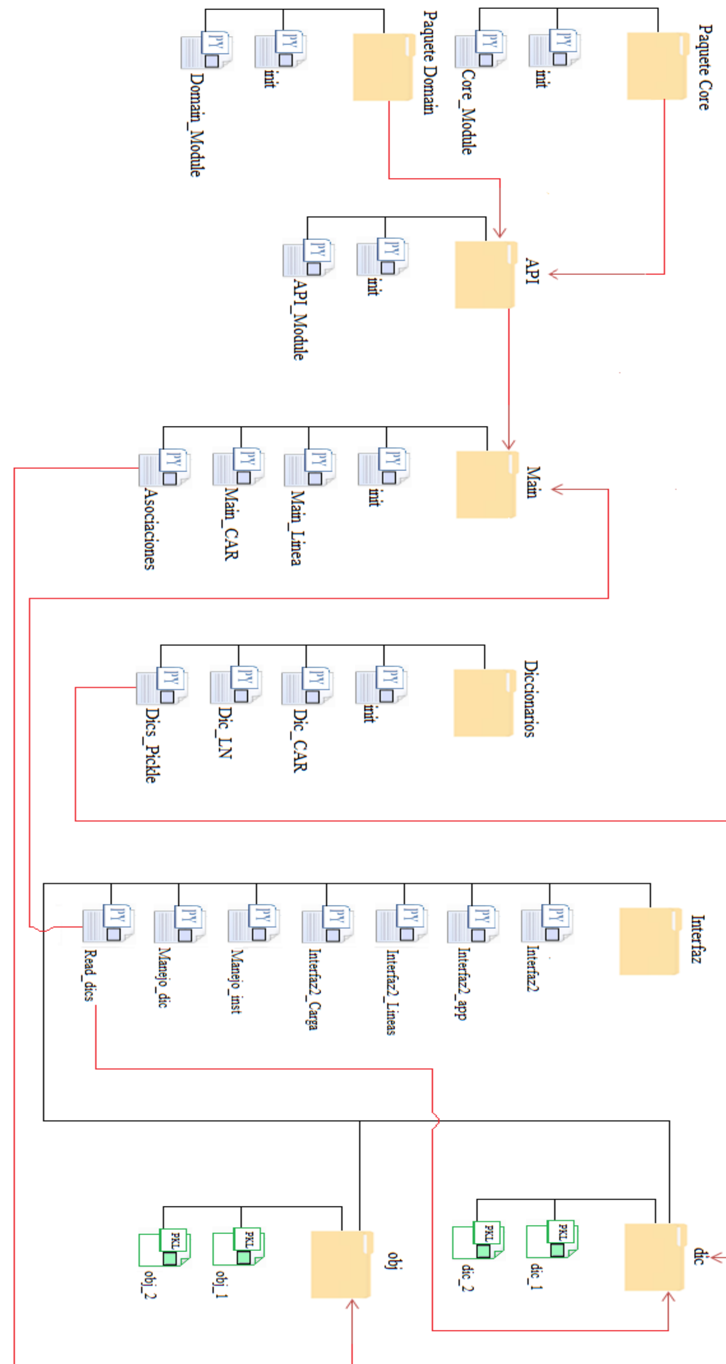

Figura 4.1.: Esquema de organización de la información

En la figura anterior se pueden apreciar los distintos paquetes de los cuales está compuesto el proyecto en cuestión, mostrando algunos de los módulos o archivos más importantes

contenidos en ellos para la construcción de la base de datos para el modelo de información común.

4.2.1 Clases

Las clases contenidas en la norma se encuentran en paquetes como se había mencionado anteriormente. En la gráfica anterior, se muestran solo dos paquetes: Core y Domain, como ejemplo de la organización de los mismo dentro del proyecto, además de los módulos de los que están compuestos, uno de estos es el inicializador init.py y el otro contiene las clases de dicho paquete. Como se ve, estas carpetas son aparentemente independientes; pero, cuando sus clases se aplican a sistemas concretos se verán relacionadas gracias a la estructura jerárquica que propone el estándar, es por esto que deben estar contenidas dentro del mismo directorio ya que de lo contrario puede incurrirse en errores al momento de instanciar los objetos requeridos por el usuario.

4.2.2 API

La carpeta API es un paquete denominado en Python para encerrar las distintas rutas que llevan a las clases de la norma las cuales están distribuidas en los diversos paquetes ya conocidos. De igual forma, contiene un módulo inicializador init.py y un módulo que contendrá las rutas hacia las distintas clases existentes del CIM. En la gráfica puede verse la relación entre los paquetes de la norma y la carpeta API mediante una flecha de color rojo indicando que suministran información hacia esta.

4.2.3 Diccionarios

La carpeta de diccionarios encerrará todos los módulos destinados a la creación de los diferentes diccionarios utilizados en la encapsulación de la información del sistema, necesaria para la creación de los objetos del CIM. En la gráfica 4.1 pueden verse dos ejemplos: en uno se encuentran los diccionarios de las líneas y en el otro el de la carga, también se dispone de un módulo extra el cual recoge todos los diccionarios creados y realiza el proceso de serialización establecido en el apéndice A. Dicha información codificada se almacenará en una carpeta llamada “dic” contenida dentro de la carpeta dedicada a la interfaz gráfica de usuario.

4.2.4 Main

El Main es un paquete que recoge todos los objetos instanciados requeridos por el sistema de distribución de energía y por el usuario en cuestión, este encierra tres tipos de módulos: inicializador init.py, objetos y asociaciones. En la figura 4.1 se muestran dos módulos de objetos, uno para líneas y otro para cargas, los equipos instanciados se pueden subdividir tanto como se requiera y de esto dependerá el número total de archivos contenidos en el paquete Main.

Este paquete, al contener objetos, necesita de la información almacenada en los diccionarios, esta información es entregada desde la carpeta “dic” a través del módulo “read_dics” el cual deserializa la información. Esta ruta puede verse en la figura 4.1 mediante el camino descrito por las flechas rojas.

4.2.5 Interfaz

En la carpeta llamada “Interfaz” se almacenan todos los módulos que en conjunto forman la interfaz gráfica de usuario, así como otras funciones informáticas creadas para la gestión de los recursos y de la información del sistema, como: “Manejo_inst”, ”Manejo_dic”, “read_dics”, el uso y descripción a fondo de estos tres ficheros se explica en secciones siguientes. La carpeta de interfaz también encierra la información serializada de los objetos y diccionarios la cual se almacenará en nuevas carpetas denominadas “obj” y “dic” respectivamente.

La distribución de la información mostrada, es el resultado del desarrollo de las prácticas para la implementación de la norma IEC 61970-301 mediante el uso del lenguaje de programación Python, esta fue la mejor opción para gestionar la información y construir una aplicación informática de modo que sea entendible para los interesados en este proyecto, también sirve como base para comprender las secciones posteriores que complementan el estudio del estándar en cuestión.

CAPÍTULO 5

INTERFAZ DE USUARIO

Una interfaz es un conjunto de herramientas visuales que le permiten al usuario interactuar con una aplicación informática, ya sea para consulta o edición de datos, manipulación o comunicación de sistemas, etc, es decir, la interfaz será el medio por el cual el usuario se podrá comunicar con el software desarrollado para cumplir con un objetivo específico. Para una aplicación informática es vital tener un medio de comunicación con el usuario, ya que la autónomia de la aplicación no es suficiente y no se puede eximir la interacción humana para su funcionamiento.

Existen muchos tipos de interfaces, entre estas se encuentran las de texto plano y las interfaces gráficas. La diferencia entre estas dos últimas radica en la forma en la que se manipula la aplicación por parte del usuario, la primera, de texto plano, requiere que el usuario ejecute ciertos comandos que activen y desactiven comportamientos en los objetos utilizados para la creación de la interfaz; por otro lado, la interfaz gráfica de usuario, es mucho más didáctica, fácil de comprender y por ende es más dinámica la comunicación decon el usuario ya que no requiere de comandos sino que establece botones, líneas de texto, gráficas, tablas y demás herramientas visuales para la manipulación de un código de programación presentando así mayores ventajas.

A continuación se explica el funcionamiento de la interfaz gráfica de usuario creada para la manipulación de la base de datos del sistema de potencia generado a partir del estándar IEC 61970-301 y el lenguaje de programación Python. Como usar correctamente la interfaz, sus componentes principales y cada una de las herramientas, además de su funcionamiento en conjunto es el tema principal de este capítulo.

5.1 Construcción de la interfaz

Para la construcción de la interfaz se uso el software para el modelado de interfaces gráficas de usuario “Qt Designer” la cual provee una amplia gama de widgets[1] . Originalmente esta herramienta está escrita en el lenguaje de programación C++, pero puede ser utilizada con código Python a través de un adaptador de bibliotecas llamado PyQt que enlaza código Python a Qt, así pues, se pueden desarrollar aplicaciones GUI (Graphical User Interface) utilizando Python y no abandonar lo que se ha hecho hasta el momento.

Qt es una herramienta muy buena y flexible, ya que es un desarrollador de aplicaciones multiplataforma, es decir, puede ser utilizada en sistemas operativos como Windows, Linux, Mac, etc. Además como se vió, no solo puede ser utilizada con C++ sino que también puede usarse con otros lenguajes de programación a través de adaptadores de biblioteca (Summerfield, 2007).

5.2 Interfaz gráfica de usuario

En esta sección se muestran los distintos elementos que componen la interfaz gráfica de usuario, para la observación y manipulación de una base de datos de un sistema de energía eléctrica, que contenga toda la información, requerida por un usuario, de los distintos equipos eléctricos que componen una o más redes eléctricas.

A continuación se muestra la aplicación GUI desarrollada para este proyecto (ver figura 5.1) mediante el uso de Qt Designer y PyQt basados en el libro “Rapid GUI programming with Python and Qt” (Summerfield, 2007). Se pueden ver todos los widgets utilizados, los cuales se explicarán poco a poco en esta sección, como un primer acercamiento para la manipulación de una base de datos basada en CIM.

[1] Los widgets son aplicaciones informáticas pequeñas que brindan fácil acceso a funciones que son frecuentemente usadas y proveen información de forma visual, por ejemplo: un botón o una tabla de datos.

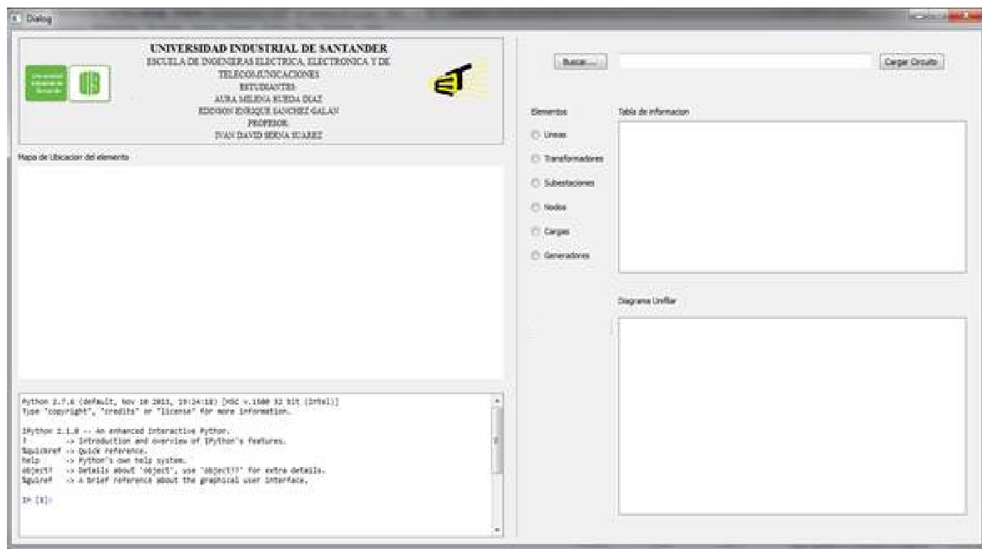

Figura 5.1.: Esquema de la interfaz gráfica de usuario.

La interfaz gráfica de usuario desarrollada se compone de tres medios para el llamado de datos y/o manipulación de los mismos, uno de estos es un conjunto de radio-botones los cuales permiten visualizar los datos disponibles en un widget, otro es la consola de Python dispuesta en la interfaz para la manipulación y edición de datos, y por último el bloque de búsqueda de los datos a visualizar.

5.2.1 Bloque de búsqueda

El bloque de búsqueda permite encontrar los datos serializados de las instancias de clase en las cuales se encuentra almacenada la información a consultar. El programa desarrollado contiene una carpeta por defecto para guardar dichos archivos; sin embargo la ruta a estos documentos no es igual en cada ordenador por lo que se requiere suministrar dicha ruta mediante el uso del bloque de búsqueda (ver figura 5.2).

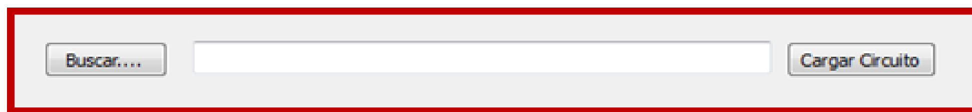

Figura 5.2.: Esquema del bloque de búsqueda dentro de la interfaz.

Primero se debe oprimir el botón llamado "Buscar" el cual ejecuta una sub-ventana de búsqueda para encontrar manualmente las instancias y así obtener la ruta para cargar la información necesaria (ver figura 5.3).

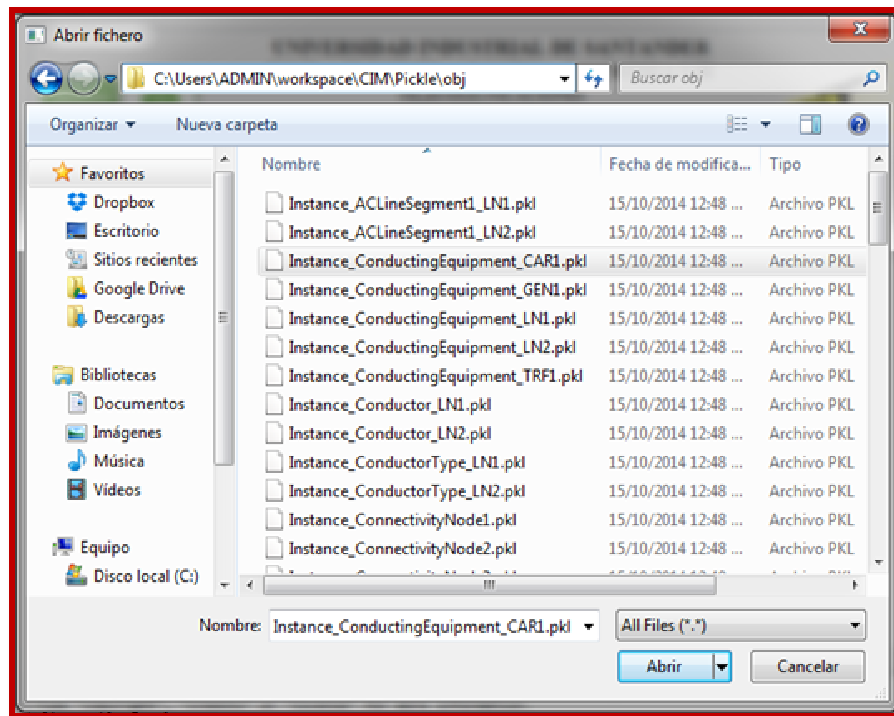

Figura 5.3.: Esquema para la obtención de la ruta.

Una vez encontradas las instancias en el directorio apropiado se puede dar clic al botón "abrir" y luego en el bloque de búsqueda al botón "Cargar Circuito" (ver figura 5.4).

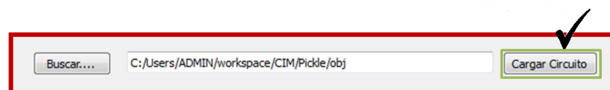

Figura 5.4.: Indicaciones para cargar la información.

Dispuesto esto, se puede proceder a seleccionar la información a consultar mediante los radio-botones.

5.2.2 Radio-botones

Al seleccionar un radio-botón se despliega una sub-ventana la cual contiene un grupo de "checkboxes" para la selección de los datos a observar posteriormente, esta sub-ventana puede apreciarse en la figura 5.5.

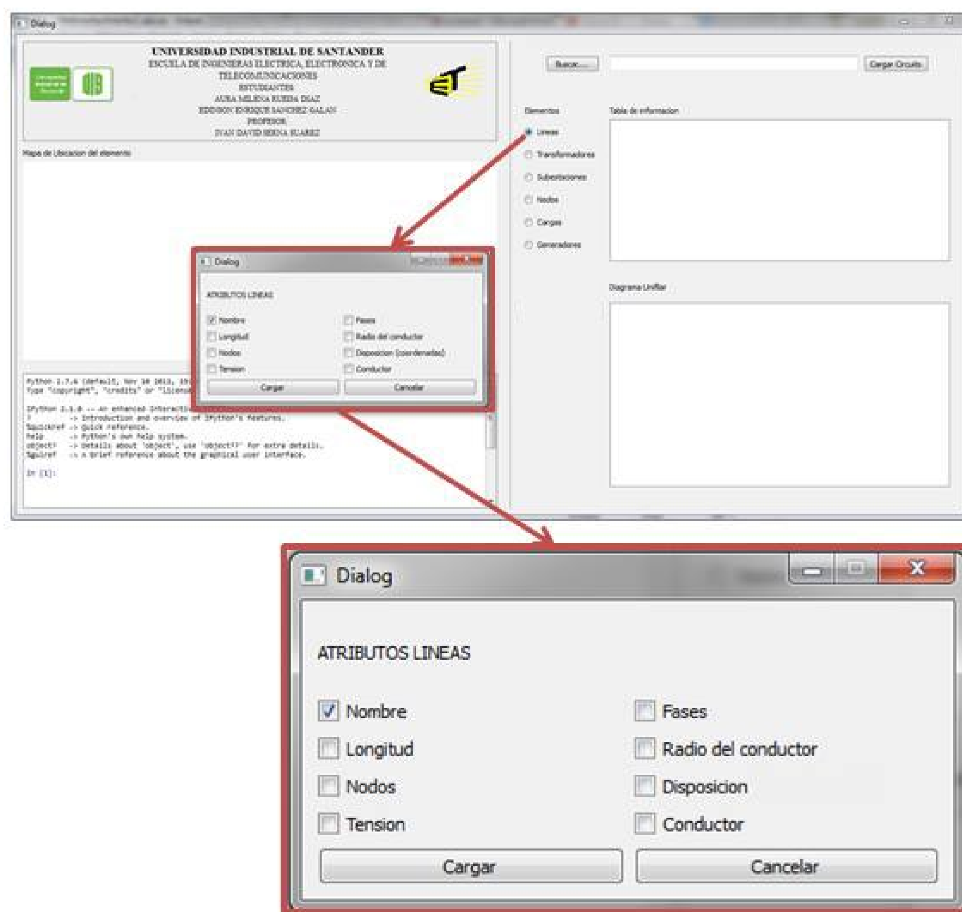

Figura 5.5.: Utilización de los radio-botones en la interfaz.

Al oprimir el radio-botón con la etiqueta de "Línea" se despliega la subventana, la cual contiene un menú de atributos o datos propios de una línea de transmisión de energía. Para este proyecto en particular se escogieron los más comunes en el campo de la ingeniería; sin embargo pudo haberse agregado otros diferentes dependiendo de la aplicación o el requerimiento del usuario final.

5.2.3 Subventana de atributos

En esta sub-interfaz, por decirlo de alguna manera, se pueden escoger los parámetros de la línea, transformador, subestación, etc que se deseen apreciar en un widget de tabla. El parámetro de "Nombre" siempre está disponible y no existe posibilidad de cambiarlo ya que este proporcionará el dato necesario para identificar de forma más clara a que equipo se está haciendo referencia.

Los botones en la parte inferior llamados "Cargar" y "Cancelar" ejecutan dos acciones posteriores, la primera muestra en pantalla la información que se ha escogido mediante los "check-boxes" y la segunda simplemente cierra la ventana impidiendo que algo se ejecute, ya que

existe la posibilidad de equivocarse al abrir una sub-ventana que no era requerida en ese momento (ver figura 5.6).

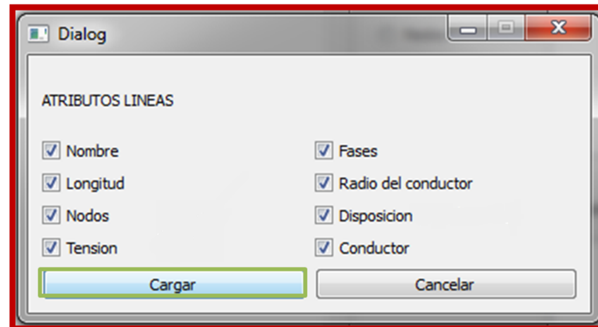

Figura 5.6.: Subventana de atributos para un elemento.

Una vez terminado el proceso de selección de atributos se procede a cargar la información de las líneas existentes en la base de datos.

5.2.4 Tabla de datos

La información escogida anteriormente se muestra en una tabla que se encuentra ubicada en la ventana principal como se aprecia en la figura 5.7. La información se organiza en columnas y filas, en las columnas se muestran los datos, es decir los valores de las características seleccionadas en la sub-ventana de atributos y por filas se pueden ver los equipos existentes en la base de datos. Para este caso específico se muestran todas las líneas de transmisión.

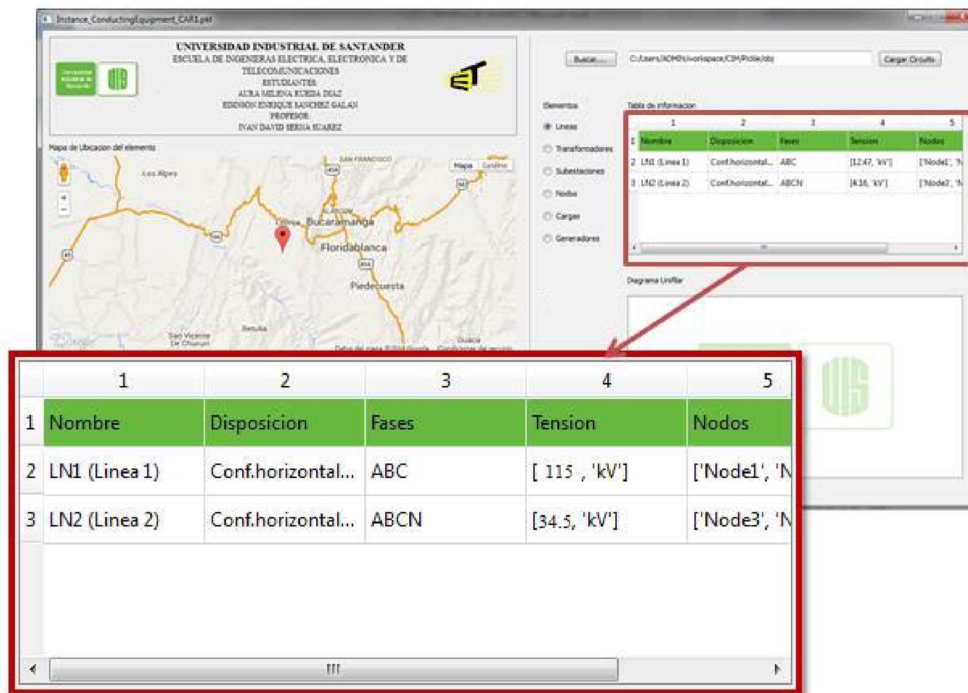

Figura 5.7.: Tabla de datos para unos parámetros determinados

El widget de tabla tiene como función principal visualizar información, pero también puede ser usado como una fuente de interacción para mostrar otros tipos de datos. Existen dos widgets más para visualizar parte de la información presente en la base de datos CIM de este sistema de potencia, pero dependen de un elemento particular, por ejemplo: una línea tiene una ubicación geográfica específica y no es compartida por ninguna otra, de igual manera se encuentra situado en forma única dentro de un diagrama unifilar, es por esto que si se selecciona con un clic sencillo el nombre del elemento contenido en la tabla de datos, se despliega dicha información en los widgets de la siguiente subsección. Como se pudo observar la tabla de datos también puede recibir órdenes del usuario y ejecutar determinadas acciones.

5.2.5 Ubicación geográfica y diagrama unifilar

Para poder observar la ubicación geográfica de un elemento basta con hacer clic sobre el nombre de este en la tabla de datos, automáticamente se cargará ésta en el widget correspondiente, dicho procedimiento puede verse detallado en la figura 5.8.

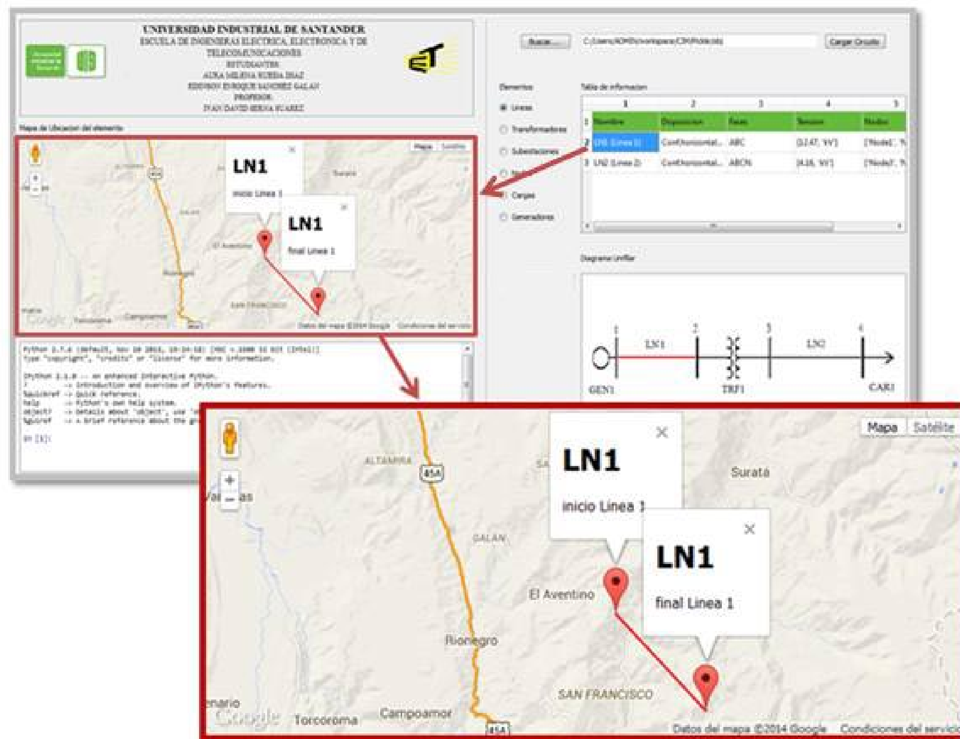

Figura 5.8.: Ubicación geográfica y diagrama unifilar de un elemento seleccionado

La ubicación geográfica se puede ver gracias a las API de “google maps” que se usaron en el código de la interfaz. La ubicación de los elementos se logró mediante la longitud y latitud. Los marcadores ubicados en los extremos de la línea pueden tener más información sobre la ubicación de la misma y simplemente haciendo clic sobre ellos se pueden ver dichos datos.

Para la ubicación en el diagrama unifilar no se necesita realizar ninguna otra acción, con el clic sencillo sobre el nombre del elemento también se carga el diagrama unifilar de los equipos cercanos a dicho objeto y lo muestra resaltándolo en “rojo”.

En la figura 5.9 se puede ver el resultado de la acción al seleccionar del nombre del elemento. Para esta aplicación resulta sencillo mostrar todo el diagrama unifilar del sistema, pero para aplicaciones mucho más grandes con una gran cantidad de barras y demás elementos, se podría usar secciones del diagrama para mostrar en este widget. Los diagramas unifilares deben estar contenidos en la misma carpeta de interfaz y utilizar un espacio de nombres establecido en el capítulo 4, esto se hace con la idea de ampliar posteriormente la base de datos con nuevos elementos y poder visualizarlos de la manera correcta.

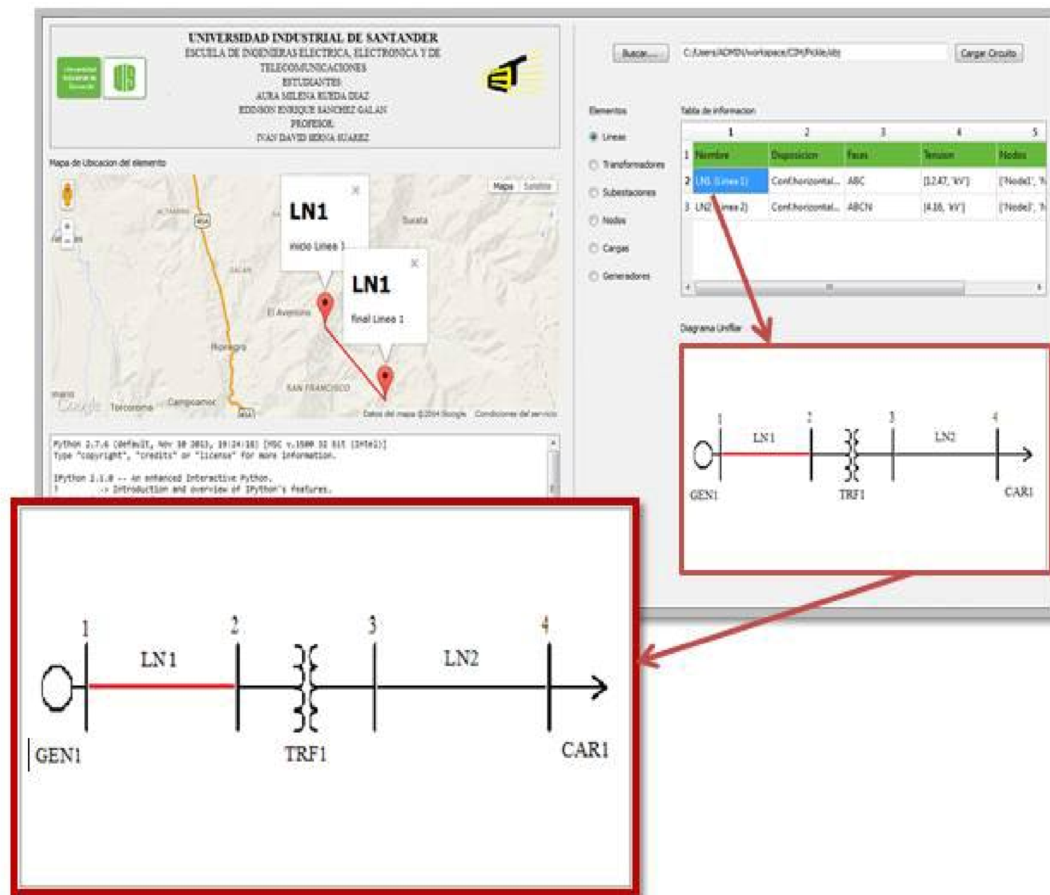

Figura 5.9.: Diagrama unifilar de un elemento del sistema

Hasta el momento solo se ha mencionado la parte de visualización de la información, es decir, solo para la lectura y verificación de los datos contenidos. Ahora se mostrará como modificar datos mediante la consola de Python que se encuentra en la interfaz.

5.3 Manejo de datos por consola

A continuación se muestra una serie de pasos a seguir en la manipulación de la base de datos CIM por medio de la consola de Python embebida en la interfaz generada para este proyecto, describiendo los comandos y líneas de código necesarias para lograr cada una de las finalidades planteadas en los numerales siguientes.

5.3.1 Creación de instancias

Para la creación de nuevas instancias mediante la consola de Python en la aplicación GUI desarrollada para el manejo de la base de datos del CIM, se

deben seguir unos sencillos pasos que a continuación se van a mostrar. Es importante tener en cuenta las recomendaciones sobre el manejo del espacio de nombres del capítulo 4, si aún no se ha familiarizado con este concepto debe referirse primero a dicha sección para poder entender lo que se va a hacer a continuación.

Importaciones La aplicación GUI desarrollada contiene todos los paquetes del estándar los cuales poseen las clases necesarias para la creación de cualquier componente de una red de energía eléctrica. Para la creación de un nuevo objeto de una clase CIM se deben importar las clases necesarias para la conformación del mismo, es decir, si en un caso determinado un objeto depende de otros se deben importar todas las clases necesarias para la creación de dichos objetos; si por el contrario éste es único dentro del sistema que se está desarrollando solo se importará la clase a utilizar.

Si la persona no conoce o no está relacionada con la estructura de la norma puede importar las clases que necesite a través del módulo de las API (Interfaz de Programación de Aplicaciones) el cual contiene las rutas de todas las clases involucradas en el estándar, de esta manera no es necesario conocer el nombre del paquete que contiene la o las clases a instanciar.

Creación de diccionarios En los capítulos 2 y 4 se orienta al usuario en la construcción de un diccionario y la correcta asignación del espacio de nombres siguiendo las sintaxis apropiadas según los autores del proyecto, de esta forma no se tendrán inconvenientes al momento de utilizar la aplicación GUI para posteriores consultas sobre nuevas instancias creadas a partir de esta guía.

El usuario debe crear tantos diccionarios como lo requiera el objeto a instanciar, es decir, deberá construir los diccionarios de los atributos que son instancias de las clases del paquete de “Domain” además de los diccionarios propios del objeto en cuestión.

Serialización de diccionarios Una vez creados los diccionarios, se deben serializar aquellos que son instancias de clase del paquete “Domain” para lograr la persistencia de la información, además de permitir la edición de estos de una forma más sencilla, gracias a la aplicación “Manejo_dic.py” la cual se explicará a detalle más adelante. La serialización de los diccionarios se puede hacer mediante el sencillo código que se muestra a continuación:

```
import pickle
import re
import os
dyc=locals()
```

```
lista_dyc=[]
for item in sorted(dyc.items ()):
    if re.search('dic', item[0]):
        lista_dyc.append(item[0])
print len(lista_dyc)
pkl_dyc=[]
for i in range(len(lista_dyc)):
    pkl_dyc.append(lista_dyc[i]+".pkl")
for j in range(len(lista_dyc)):
    fil= os.path.abspath(".\dic")+'/'+pkl_dyc[j]
    with open(fil, 'wb')as output:
        exec("pickle.dump("+lista_dyc[j]+",output)")
        output.close()
```

El diccionario serializado será creado en la carpeta o directorio correspondiente de forma automática utilizando el mismo nombre del diccionario y una extensión .pkl, tal y como se exige en el capítulo 4 de este proyecto.

Des-serialización de diccionarios Como se sabe los diccionarios contienen los datos que se ingresarán a los atributos propios de la clase a instanciar, y es por esto que se requiere obtener la información guardada en ellos a través de los archivos serializados para ligarla con dichas instancias. Para cargar un archivo y trabajar con su información, se debe poner el siguiente código:

```
import pickle
import glob
import os
import pickle
list_read=glob.glob(os.path.abspath(".\dic")+ "\*.pkl")
list_names=os.listdir(os.path.abspath(".\dic"))
lista_nombres=[]
for k in range(len(list_names)):
    lista_nombres.append(os.path.splitext(list_names[k])[0])
for z in range(len(lista_nombres)):
    with open(list_read[z], 'rb') as f:
```

```
        exec(lista_nombres[z]+"=pickle.load(f)")
        f.close()
```

Una vez ejecutado este código ya se puede trabajar con los diccionarios previamente creados para la construcción de la o las instancias de clase que harán parte de la base de datos del CIM.

Construcción de una nueva instancia La creación de un objeto es muy sencilla y ya se ha explicado en secciones anteriores, así que este paso no debe representar un obstáculo para el usuario interesado en ampliar la base de datos. Se tiene que recordar que algunos atributos de las clases son instancias de las clases del paquete "Domain" y deben ser ingresados a un nuevo diccionario, por lo que se repite el proceso anterior para finalmente llegar a la construcción del objeto final.

Es muy probable que el objeto que se está creando mantenga una relación directa con otro objeto que ya se encuentra contenido dentro de la base de datos, esto hace necesario llamar a dicha instancia para actualizar su asociación y agregar también la información de dicho objeto al nuevo que se está construyendo. Para esto se debe realizar un sub-proceso de des-serialización de todos los objetos que se relacionen de alguna forma con el objeto en cuestión.

```
Import pickle
Import os
with open(os.path.abspath(".\obj")+'/'+'NombreDeLaInstancia.pkl','rb')as f:
    NombreDeLaInstancia=pickle.load(f)
    f.close()
NombreDeLaInstancia.atributo=NuevoValor
With open(os.path.abspath(".\obj")+'/'+'NombreDeLaInstancia.pkl','wb') as file:
    pickle.dump(NombreDeLaInstancia,file)
    file.close()
```

Con la estructura de código anterior es posible cargar de forma individual todas las instancias necesarias para asociar el nuevo objeto y modificar las relaciones existentes de las demás clases que interactúan directamente con este. Primero se carga para lectura, se realiza el cambio en el atributo de asociación y luego se guarda el cambio, esto puede verse en el Apéndice A sobre manejo de archivos en Python.

Hasta el momento todo lo que se ha hecho puede ser utilizado en la creación masiva de objetos y diccionarios, el procedimiento no se limita a un solo elemento, sino que resulta un

método general que aprovecha el ingreso de una gran cantidad de información, ya que los códigos para la serialización y la des-serialización toman todos los diccionarios existentes (ingresados hasta ese momento) y crea la persistencia de la información en ficheros con extensión .pkl.

Serialización de objetos CIM Por último se deben crear los objeto en forma serializada para obtener nuevamente la persistencia de dicha información y tenerla disponible para la interfaz si esta lo requiere, además de poder modificar dicha información a través de la aplicación "Manejo_inst.py" la cual se verá mas adelante. Para convertir todos los objetos creados hasta el momento en ficheros .pkl se debe ejecutar en consola el siguiente código:

```
import pickle
import os
import re
dyc=locals()
lista_dyc=[]
for item in sorted(dyc.items()):
    if re.search('Instance', item[0]):
        lista_dyc.append(item[0]) print len(lista_dyc)
print lista_dyc
pkl_dyc=[]
for i in range(len(lista_dyc)):
    pkl_dyc.append(lista_dyc[i]+".pkl")
for j in range(len(lista_dyc)):
    fil= os.path.abspath(".\dic")+'/'+pkl_dyc[j]
    with open(fil, 'wb') as output:
        exec("pickle.dump("+lista_dyc[j]+",output)")
        output.close()
```

Una vez serializado los objetos modificados y los nuevos que se hayan creado, puede decirse que se ha finalizado con éxito el procesos de construcción de las nuevas instancias que ahora harán parte de la base de datos del CIM y estarán disponibles para su consulta y/o edición por medio de la aplicación GUI de este proyecto.

5.3.2 Consulta de atributos por medio de consola

La aplicación GUI está limitada a mostrar ciertos atributos de los equipos que allí se muestran, por lo que no es posible consultar otro tipo de información que pueda contener la base de datos CIM sujeta a dicha aplicación por medio de las herramientas visuales contenidas en esta. Para solucionar esto se debe hacer la consulta por medio de la consola de Python disponible en la interfaz.

Identificación de objetos

Toda la información sin excepción está contenida en los atributos de las clases del CIM, así que el primer paso en la búsqueda de la información es determinar que o cuales instancias de clase deberían contener los datos a consultar, una vez hecho esto se procede a ejecutar en consola los comandos que llevarán a ver en pantalla toda la información requerida en el momento.

Des-serialización de los objetos

Una vez establecidos los objetos que contienen la información requerida, se procede a des-serializarlos, esto se logra mediante el uso en consola del siguiente código:

```
import pickle
with open(os.path.abspath(".\obj")+'/'+"NombreDeLaInstancia.pkl",'rb') as file:
    NombreDeLaInstancia=pickle.load(file)
    file.close()
```

Consulta

Ahora solo falta realizar la consulta del atributo que se requiera, esto se logra llamando dicha característica tal y como se da a conocer en el capítulo de introducción a Python 2.7; sin embargo se muestra nuevamente a continuación:

```
<Print NombreDeLaInstancia.atributo>
>>>> valorDelAtributo
```

5.3.3 Actualización de información a través de los diccionarios

El manejo de diccionarios para el ingreso de la información es muy útil debido a que estos ofrecen una estructura de datos que puede ser fácilmente identificable gracias a las claves y al espacio de nombres asociados a estos. A continuación se muestra el proceso a seguir cuando se desea actualizar una gran cantidad de información utilizando los diccionarios como fuente generadora de datos.

Actualización de datos contenidos en un diccionario Para la actualización de la información a través de los diccionarios existentes de los cuales se desprende los datos contenidos en las instancias de clase, se debe correr en la consola de la interfaz el script llamado “Manejo_dic.py” el cual se ejecuta mediante la siguiente línea de código:

```
<import Interfaz.Manejo_dic>
```

Al ejecutarse la aplicación se despliega inmediatamente un mensaje con instrucciones para el uso de dicho programa, además pide al usuario ingresar un valor determinado para escoger la acción a realizar sobre los diccionarios.

```
TODAS LAS OPCIONES QUE INGRESE DEBEN ESTAR ENTRE COMILLAS ” EXCEPTO SI EL VALOR ES NUMERICO ()
Seleccione la opcion e para editar, r para leer y s para salir: "e"
```

Luego de escoger la opción “editar” se ingresa el nombre del diccionario más la extensión .pkl. Además pide al usuario ingresar el número de claves a modificar y el nuevo valor que se le asignará.

```
Ingrese el nombre del diccionario a modificar .pkl: "dic_g0ch_Conductor_LN1.pkl"
Cuantas claves desea modificar: 1
Ingrese la clave del elemento: "value"
Nuevo valor del elemento: 0.4365
```

Una vez terminado el procedimiento se mostrará en pantalla un mensaje indicando que la actualización fue exitosa y brinda la posibilidad de seguir editando o salir de la aplicación.

**** EL DICCIONARIO HA SIDO ACTUALIZADO ****

()

**PARA MODIFICAR MAS DICCIONARIOS OPRIMA 1 PARA TERMINAR Y SALIR OPRIMA 0.

Actualización de instancias a partir de los diccionarios Los diccionarios por si solos no pueden generar algún tipo de objeto o parámetro que pueda servir para adquirir información acerca de un sistema de energía eléctrica, ya que esta función es propia de las instancias de las clases CIM. Los diccionarios solo proveen la información de forma más estructurada a los atributos de dichos objetos, por lo cual, editar la información en los diccionarios no sirve si no

se generan nuevamente las instancias de clase que contienen esa información, por esta razón es recomendable que el usuario mantenga un registro en ficheros de extensión .py que contengan los objetos de construcción inicial y que estos estén ligados a la información serializada de los diccionarios, de esta forma, se pueden generar nuevamente los archivos .pkl de las instancias de clase existentes y renovar la información que contienen. La dependencia de las instancias con los diccionarios serializados de este trabajo se pueden ver en el capítulo orientado a la organización de la informacón.

Para poder realizar esta operación de regeneración de los archivos serializados de las instancias de clase, el usuario debe tener un intérprete de Python que le permita organizar los objetos en módulos de extensión .py, además de ligar los diccionarios serializados a estos. Con la fundamentación vista hasta el momento debe resultar sencillo para el usuario cargar los ficheros .pkl de los diccionarios y a su vez generar nuevamente las instancias modificadas a través de estos. El resultado no debería ocasionar problemas en la ejecución de la interfaz siempre y cuando dichos ficheros sean ubicados dentro del directorio reservado para el almacenamiento de información de la aplicación GUI.

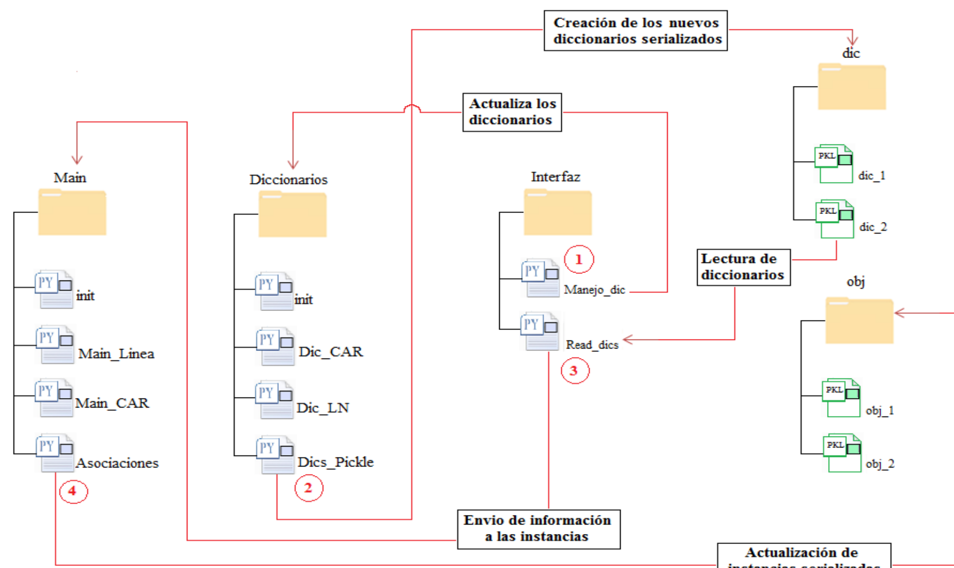

Figura 5.10.: Actualización de instancias a partir de pickle. La figura muestra las acciones y el flujo que se sigue al actualizar datos por medio de diccionarios.

5.3.4 Actualización de información a través de las instancias

Inicialmente el cambio de datos por consola puede resultar complicado debido a la gran cantidad de líneas de código a escribir o cuando no se tiene un manejo óptimo del tema en cuestión. Es por esto, que se

diseñó una aplicación interna que permite introducir comandos simples ayudados por cadenas de documentación para actualizar la información de la base de datos de una forma más fácil y rápida a traves de los atributos de los objetos. Una de estas aplicaciones lleva por nombre "Manejo_inst.py" y se ejecuta desde la consola mediante la siguiente línea de código:

```
<import Interfaz.Manejo_inst>
```

El funcionamiento de la aplicación es muy simple, solo se deben seguir, de la manera correcta, los pasos que está misma va señalando .

Cabe resaltar que esta acción no requiere ejecutar el módulo de búsqueda ya que es independiente del resto de funciones de la interfaz, pero si se quiere observar el resultado de forma inmediata es recomendable haber ejecutado este subproceso antes de poner en funcionamiento la herramienta de actualización de la información.

A continuación se muestra un ejemplo de actualización de un dato en una instancia de clase del CIM usada en la interfaz para la visualización de un parámetro de un determinado equipo. El parámetro a modificar será el nombre de la única carga presente en el sistema de potencia, actualmente el nombre de la carga es "Carga Trifasica" pero se considera que el nombre no es el indicado para dicho elemento y se quiere cambiar a "Carga 1", asi pues en consola el procedimiento sería el siguiente:

El primer paso, como ya se mencionó con anterioridad, es ejecutar la aplicación de nombre "Manejo_inst.py" mediante la línea de código que se muestra:

```
import Interfaz.Manejo_inst
```

Como primer evento aparece una instrucción de las condiciones establecidas para el ingreso de los datos, donde toda la información a ingresar debe estar en comillas a excepción de los datos numéricos. Al mismo tiempo entrega opciones para escoger la acción que se desea realizar. Esta aplicación tiene dos funciones: modificar atributos y leer el listado de los mismos asociados a una determinada instancia de clase. Como lo que se quiere es modificar el nombre de la carga se debe ingresar como parámetro "e".

```
TODAS LAS OPCIONES QUE INGRESE DEBEN ESTAR ENTRE COMILLAS ” EXCEPTO SI EL VALOR ES NUMERICO
()
Seleccione la opcion e para editar, r para leer y s para salir: "e"
```

Una vez se introduce el valor de la opción, la aplicación pide al usuario ingresar el nombre de la instancia a modificar más la extensión del archivo donde se encuentra serializado. Como se sabe, cada instancia se serializó con el mismo nombre con el que fue creada (ver Apéndice A). La carga es una instancia de clase de <`EnergyConsumer`> del paquete "Wires" y el nombre de esta se encuentra en su atributo <name> heredado de la clase <`IdentifiedObject`> perteneciente al paquete de "Core", además, gracias al espacio de nombres establecido en el capitulo 4 de este trabajo, se puede conocer el nombre con el que fue creado el elemento. Es por esto que el dato a ingresar en la aplicación de actualización será: <`"Instance_EnergyConsumer_CAR1.pkl"`>.

```
Ingrese el nombre de la instancia a modificar .pkl:
"Instance_EnergyConsumer_CAR1.pkl"
```

Continuando con los pasos, lo siguiente resulta sencillo, ingresar el número de atributos a modificar que para el caso será "1", el nombre del atributo y el valor correspondiente el cual se estableció en "Carga 1".

```
Cuantos atributos desea modificar: 1
Ingrese el atributo: "name"
Nuevo valor del atributo: "Carga 1"
**** LA INSTANCIA HA SIDO ACTUALIZADA ****
()
**PARA MODIFICAR MAS INSTANCIAS OPRIMA 1 PARA TERMINAR Y SALIR OPRIMA 0.
```

Al final de la aplicación mostrará un mensaje indicando que el proceso ha sido satisfactorio y concederá la opción de seguir actualizando o cerrar la aplicación.

Como se pudo observar, es un procedimiento muy sencillo y se logra en poco tiempo, a diferencia si se hace por medio de la utilización de algún lenguaje de programación para la manipulación de archivos serializados. Si el atributo actualizado no se puede observar en la tabla de datos se debe recurrir a la subsección anterior llamada "consulta de atributos por consola" para llamarlo de la forma correcta.

Como última anotación de este capítulo cabe resaltar que los demás radio-botones como: transformadores, cargas, subestación o nodos, trabajan de la misma forma que en el ejemplo mostrado con el radio-botón de Líneas, también poseen sub-ventanas de atributos, botones de carga y de cancelación y reaccionan al clic sencillo sobre el nombre del equipo para mostrar

la ubicación en el diagrama unifilar y en el mapa de Google.

Todo el trabajo de la interfaz fue pensado en crear una herramienta muy sencilla de manejar, que sea flexible y pueda comunicarse a un nivel más profundo con la información de la base de datos conformada a partir del CIM, manteniendo una confiabilidad en las operaciones que se realicen, ya sean de actualización, lectura o eliminación de datos. Los elementos gráficos como botones, pantallas, sub-ventanas, menús y demás partes de la interfaz gráfica de usuario, son una poderosa arma para el manejo este tipo de aplicaciones informáticas, las cuales debido a la gran cantidad de objetos que poseen resultan complejas a la hora de tratarlas.

CAPÍTULO 6

CASO DE APLICACIÓN

La generación de buenas prácticas para la implementación de la norma IEC 61970-301, se inició al establecer un entorno de trabajo y al escogier un optimo lenguaje de programación por el cual se construyeran todas las clases descritas en el CIM, además de obtener un buen interprete que facilitara el trabajo de programación y pudiese manejar apropiadamente todos los paquetes establecidos por la norma.

Como se ha mostrado a lo largo de este proyecto el lenguaje de programación escogido para materializar la norma a través de la programación orientada a objetos es Python versión 2.7 y se usó en el intérprete: Eclipse, Version: Luna Release (4.4.0), el cual permite la ejecución de códigos en lenguaje Python a través de un plugin llamado “Pydev”. Gracias a estas herramientas fue posible programar todas las clases de la norma y por ende tener siempre a disposición la estructura jerárquica del CIM para el desarrollo de una base de datos determinada.

Los pasos a seguir para la creación de las clases de la norma, mencionadas con más detalle en secciones anteriores (Introducción al estándar IEC 61970-301) se describen a continuación utilizando parte del código generado a partir de Python.

6.1 Creación de clases CIM

En la sección anterior se pudo observar que las clases están divididas en paquetes con el fin de establecer una clasificación y dar un orden a estas para una mejor comprensión de la norma. A su vez, los paquetes están relacionados entre sí a través de sus clases, esto conlleva, a realizar un análisis del orden que se debe llevar al momento de crear cada uno de los paquetes del estándar. Como se estableció en la sección anterior los paquetes base de la norma son "Core" y "Domain" así que se empieza por construirlos a partir de las clases contenidas en estos.

6.1.1 Creación del paquete Core

Con anterioridad se explicó el concepto de paquete y módulo en Python, por lo que su creación debe ser trivial en este punto, como también la construcción de todas las clases del CIM. Gracias al interprete Eclipse Luna, la ejecución de paquetes y módulos se simplifica mucho, ya que este los construye por si solo y no existe necesidad de crear manualmente la carpeta y el archivo inicializador de extensión .py. Una vez hecho lo anterior se crea un módulo en el cual existirán todas las clases del paquete "Core", esta es una ventaja de Python ya que a diferencia de otros lenguajes de programación un solo archivo puede contener múltiples clases. A continuación se muestra la estructura de la clase más básica e importante del estándar la clase `<IdentifiedObject>`.

```
class IdentifiedObject():
    def__init__(self,dic_IdentifiedObject):
       self.mRID=dic_IdentifiedObject['mRID']
       self.name=dic_IdentifiedObject['name']
       self.localName=dic_IdentifiedObject['localName']
       self.pathName=dic_IdentifiedObject['pathName']
       self.aliasName=dic_IdentifiedObject['aliasName']
       self.description=dic_IdentifiedObject['description']
```

En la construcción de clases, los valores de los atributos son ingresados a las instancias de clase por medio de diccionarios, es por esto que la variable que contiene los datos de los atributos es una sola y aprovecha su característica de claves para asignar el valor correspondiente a los distintos atributos de la clase. El funcionamiento de esta estrategia se describe de una manera sencilla: para asignar un valor a un atributo este busca dentro de la variable (el diccionario asignado a la clase) una clave que coincida con el nombre de dicho atributo y lo

asigna a este, de esta forma se evita el ingreso manual de los datos al momento de instanciar los objetos, además se estructura mucho mejor la entrada de la información.

6.1.2 Creación del paquete Domain La creación de las clases del paquete "Domain" no representa complejidad, son clases sencillas con pocos atributos y son independientes de todo, es decir no poseen ningún tipo de relación con otras clases.

Los demás paquetes son creados a partir del mismo análisis de relación que se hizo al comienzo de la construcción de las clases, especificando en el código Python los atributos propios, los heredados y aquellos que relacionan las diferentes clases de cada uno de los paquetes. En esta parte se debe prestar especial atención, ya que por medio de la asociación que se pueden determinar los valores de los atributos de una clase específica, es decir, a partir de una clase de referencia se puede llegar a conocer la información almacenada en otra, si y solo si existe una ruta conformada por asociaciones entre clases, este concepto se verá en detalle más adelante cuando se esté explicando el desarrollo de la implementación.

Finalmente al tener todas las clases construidas dentro de sus respectivos paquetes gracias a Python, se crea un paquete extra de nombre "API" y un módulo "API_Module" el cual contendrá la ruta de todas las clases hechas en esta primera etapa de la implementación de la norma, de esta forma se reduce el tiempo de búsqueda de una clase ya que las "API" facilitan dicho trabajo. Aquí ya se dispone de la estructura y la base necesaria para instanciar clases, es decir construir los objetos que emulan los equipos de un sistema de potencia sencillo, el cual se describe posteriormente.

6.2 PASOS A SEGUIR EN LA IMPLEMENTACIÓN DEL ESTÁNDAR

En un determinado lenguaje de programación, cada uno de los módulos y paquetes que contienen las diferentes clases, atributos y relaciones, propios del estándar, representan la base para la implementación del mismo y por ende la gestión de la información que se tenga. Para aplicar esta estructura de datos a un sistema que contiene diferentes objetos o equipos, se debe seguir un proceso que permita construir dicho sistema con orden, para una buena gestión y evitar así la pérdida de información en algún punto. Los pasos que hacen parte de este proceso son los siguientes:

1. Recopilación de la información: en esta primera etapa se debe tener claro el tipo de información que se desea gestionar. Es necesario seleccionar la información requerida que se manejará dentro del CIM y tener una organización y clasificación de acuerdo a los datos que contenga cada elemento que haga parte del sistema, es decir, si se desea

modelar un circuito o parte de un sistema de potencia lo primero sería identificar los equipos a modelar y extraer la información que se desea gestionar por parte del usuario final.

2. Clasificación de la información: cuando se cuenta con toda la información a gestionar es importante darle una clasificación más detallada de acuerdo a las clases que contiene el estándar. En esta parte del proceso se deben identificar las clases y los atributos que harán parte del modelado del sistema, es decir, el diseñador deberá definir que atributos representan a cada uno de los datos que se tienen y por ende las instancias de clases que conformarán el modelo en cuestión. Para hacer más fácil y rápida la identificación de atributos es importante definir a qué clase del estándar pertenece cada elemento del sistema de energía, posterior a ello, dirigirse a la norma y buscar los atributos propios y heredados que posee dicha clase, así como las relaciones que ésta tenga con otras. De esta manera el programador puede dar un mejor orden a la clasificación de la información.
   Una vez identificadas las clases que harán parte del modelado de un sistema de potencia determinado, es necesario crear una estructura jerárquica con estas, y así construir los diagramas UML.

3. Creación de los diagramas UML: con esta etapa se busca mostrar el orden jerárquico que poseen las clases que hacen parte del modelado del sistema. A través de estos diagramas UML se puede ver en forma clara el tipo de relaciones que posee cada clase, sus atributos y el nivel de detalle que se tiene de la información para poder crear en forma satisfactoria las instancias de clase que representaran a los objetos reales del sistema y en cuya estructura estará encapsulada la información.

4. Creación de los objetos en Python: este paso representa la etapa final del proceso, en este punto ya se cuenta con la estructura general del CIM, es decir, ya se han programado todos los paquetes de la norma, sus clases, atributos y relaciones como esta lo especifica. Se comienza a gestionar la información que se tiene creando cada uno de los elementos que hacen parte del sistema como objetos o instancias de las clases ya construidas. Para la creación de objetos en Python según la norma, se deben seguir ciertas pautas para hacer efectiva la implementación.

   a) Se definen los diccionarios con los que se describe cada uno de los atributos propios y heredados de las clases que hacen parte del modelado de los elementos que conforman el sistema y dentro de los cuales se guarda la información clasificada con anterioridad. Estos diccionarios se deben serializar para generar persistencia en la información y de esta forma hacer que el acceso y la modificación de los mismo se

haga de una forma más eficiente. Para este proyecto se utilizaron las bibliotecas de pickle propias de Python para dicha serialización de archivos.

b) Se crea un módulo principal en el cual se realiza la instanciación de cada una de las clases requeridas de acuerdo a la información que se tiene y a la estructura jerárquica elaborada del sistema en cuestion, a este modulo se le llamó "Main" en este se realiza todo lo mencionado a continuación. El proceso de instanciación inicia importando a través de las API todas las clases involucradas en el modelado a los modulos (ver figura 6.1), posterior a ello se crean los atributos de la clase que se desea instanciar como objetos de clase del paquete "Domain" en estos mismos. Una vez realizado esto, se elabora el diccionario que contendrá todos los atributos propios de la instancia de clase importando sus valores de los diccionarios bases creados en el numeral anterior, y finalmente se crea la instancia de clase determinada, incluyendo en ella el diccionario que contiene los atributos propios y los diccionarios con atributos heredados.

c) Cuando se crea un objeto de una determinada clase que posee relaciones con otras, es necesario realizar dicha asociación entre las instancias de clase involucradas. La manera de realizar este tipo de relaciones se muestra más claramente en el ejemplo de implementación que se explicará mas adelante.

d) Al igual que en el caso de los diccionario, estas instancias también deben ser serializadas para tener un manejo mas eficaz de las mismas.

Estos son los pasos generales para realizar una correcta implementación del éstandar. En numerales posteriores se podrá ver en forma mas detallada la construccion de las instancias de las clases CIM. La forma en la que se nombraron los objetos y los diccionarios, los cuales se rigen por un patron común que se establecieron en el capitulo 4 de este proyecto.

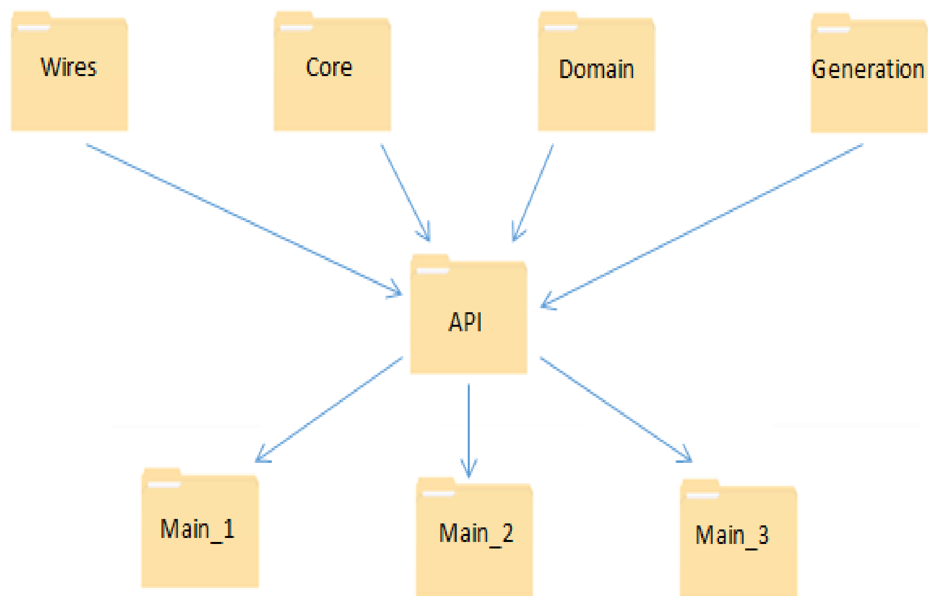

Figura 6.1.: Flujo de información a través de API para el uso de las clases CIM en los distintos Main.

[1]

[1] El concepto de Main se puede ver en el capitulo 4 sección 4.2.4

Figura 6.2.: Esquema del proceso general para la instanciación de elementos

La figura 6.2 muestra los diferentes paquetes que se deben crear para un manejo óptimo del proceso de instanciación. El paquete "Dic" contiene dos módulos, uno para la creación de los distintos diccionarios básicos que almacenarán la información de los atributos, y otro para la serialización de la información. El paquete "Main" contiene dos módulos el primero para la creación de instancias de la clase CIM el cual recibe la información de los diccionarios serializados, genera los atributos como instancias del paquete "Domain" y crea a partir de estas, el diccionario final para los objetos del estándar, posteriormente en el segundo módulo se serializan las instancias y son enviadas a una carpeta de nombre "obj" en donde se podrá acceder a los distintos atributos de las clases por medio de una interfaz gráfica de usuario.

6.3 Implementación del circuito de prueba

Para una correcta implementación de la norma en un determinado sistema es importante conocer a detalle sus componentes y características para realizar una efectiva clasificación de la información que este contiene. En esta sección se busca seguir cada uno de los pasos señalados anteriormente para gestionar adecuadamente la información que se extraiga de dicho sistema, teniendo en cuenta la estructura y los criterios que señale la norma. Inicialmente se mostrará

el sistema de forma completa con cada uno de los elementos que lo componen y posterior a ello se iniciará con el proceso de implementación del estándar manteniendo un orden en la construcción de cada elemento.

6.3.1 Circuito de prueba

El sistema de prueba en este caso es una base del "Standard 4-bus case" de "IEEE Distribution Test Feeders" (IEEE, 2010) modelo por el cual se rige la aplicación del presente proyecto. El sistema se puede apreciar en el diagrama unifilar mostrado en la figura 6.3.

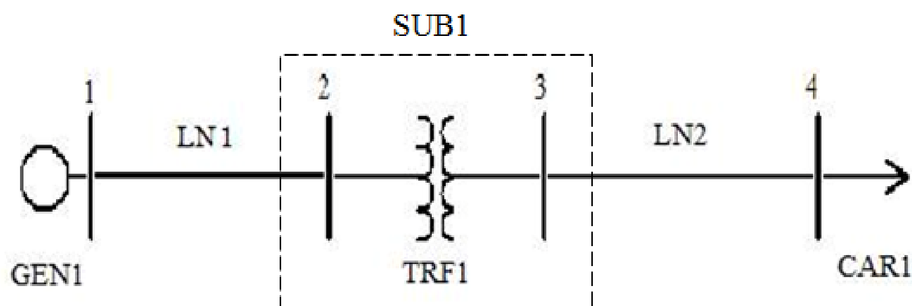

Figura 6.3.: Diagrama unifilar del sistema de 4 barras

Este sistema cuenta con 4 barras, una unidad de generación conectada a la barra 1, dos líneas de transmisión, un transformador entre las barras 2 y 3 y una carga conectada a la barra 4. Además se plantea la existencia de una subestación reductora conformada por las barras 2 y 3, las líneas de entrada y salida LN1 y LN2 y el transformador TRF1.

6.3.2 Proceso de implementación

A continuación se realizará el proceso de aplicación del estándar a cada uno de los elementos que conforman el sistema siguiendo los pasos mostrados anteriormente.

Recopilación de la información

De acuerdo con los datos suministrados por el ejemplo base "Standard 4-bus case" y las diferentes modificaciones realizadas al mismo por parte de los autores con datos obtenidos del libro "Líneas de transporte de energía" (Checa, 1988) , se tiene la siguiente información para cada elemento del sistema:

- Líneas de transmisión LN1 y LN2

TABLA 6.1.: Características eléctricas de las líneas de transmisión.

Nombre	Impedancia sec(+) [Ω/km]	Impedancia sec(0) [Ω/km]	Longitud [km]
Línea 1	0,1932 + 0,4110j	0,3407 + 1,26j	60
Línea 2	0,2331 + 0,3665 j	0,5321 + 2,0398j	18

TABLA 6.2.: Características de los conductores.

Nombre	Cap. Amp	Conductor	Radio [cm]	GMR [m]
Línea 1	300 [A]	Dove	1.175	0.004785
Línea 2	200 [A]	Hawk	1.089	0.440435

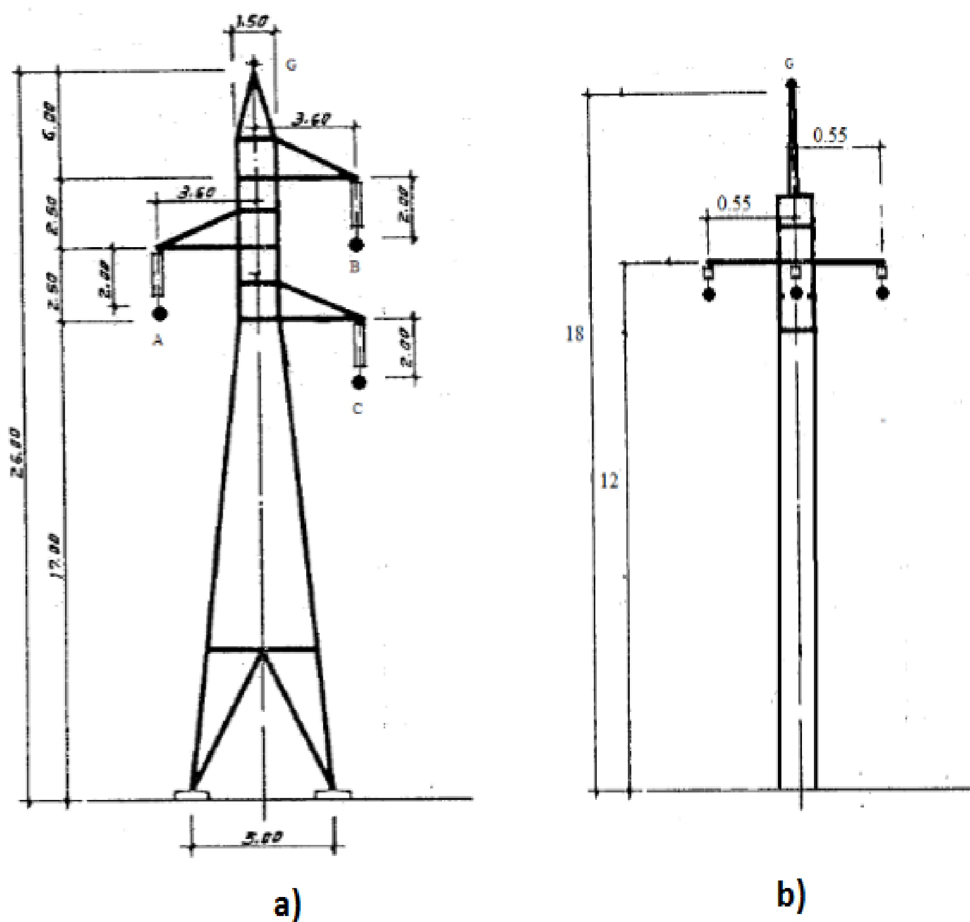

Figura 6.4.: a.) Estructura línea 115 kV b.)Estructura línea 34.5 kV

TABLA 6.3.: Disposición de las líneas.

Nombres	Coordenadas [m]			
	Fase A	Fase B	Fase C	Guarda
Línea 1	(-3.6; 17)	(3.6; 19.5)	(3.6; 15)	(0; 28)
Línea 2	(-0.55; 12)	(0; 12)	(0.55; 12)	(0; 18)

- Generador GEN1

TABLA 6.4.: Características eléctricas del generador GEN1.

Nombre	Pot Nom [MVA]	Tensión de operación [kV]	X de eje directo [pu]	X de cuadratura [pu]
Generador 1	70	13.8	1.5	0.75

- Carga trifásica balanceada CAR1

TABLA 6.5.: Características eléctrica de la carga CAR1.

Nombre	Potencia [MVA]	Factor de potencia
Carga 1	40	0.9

- Transformador TRF1

TABLA 6.6.: Características eléctricas del transformador TRF1.

Nombre	R[%]	X[%]	Potencia MVA	Conexión	Tensión alta [kV]	Tensión baja [kV]
Transformador 1	1	6	60	D-y	115	34.5

- Subestación SUB1: Esta subestación de tipo reductora tiene como líneas de entrada y salida LN1y LN2 respectivamente. Posee el transformador de potencia TRF1 descrito anteriormente. Su configuración es barraje sencillo.

El código que acompaña a cada elemento es decir el LN1, LN2, TRF1, etc se le conoce como código de equipo, este es creado por los autores y su funcionalidad se específica en el capítulo “Organización de la información”.

Clasificación de la información Conociendo cada uno de los datos de los elemento del sistema de potencia, es importante identificar los atributos que representan estos datos y la clase a la que pertenecen. El estándar cuenta con una serie de diagramas UML de los principales elementos que hacen parte de un sistema de potencia y que sirven de referencia al momento de iniciar con dicha identificación de atributos cuando aún no se maneja con destreza la norma. A continuación se muestran las principales clases que hacen parte del modelado de cada elemento del sistema teniendo en cuenta el nivel de abstracción considerado en este caso particular. Si se requiere observar en detalle estas clases se debe recurrir a la norma (IEC, 2009).

- Líneas de transmisión LN1 y LN2
  - Clase Conductor (Wires): los atributos contenidos en esta clase representan las características eléctricas propias de una línea de transmisión.
  - Clase WireType (Wires): representa las propiedades fundamentales de los cables conductores que componen la línea.
  - Clase ConductorType (Wires): es una especificación técnica de un determinado tipo de conductor donde se almacenan datos acerca de la chaqueta protectora del cable.
  - Clase WireArrangement (Wires): representa la distribución o ubicación de los conductores en una estructura determinada.
  - Clase ConductingEquipment (Core): cuenta con un atributo que señala la disposición de fases de cada equipo o elemento del sistema. Esta clase es, en la mayoría de los casos, la conexión para relacionar los equipos a las clases de <`Terminal`> y demás que conforman el modelo de conexión.
  - Clase ACLineSegment (Wires): es la clase que conforma el objeto de línea en cuestión, ya que posee sus características eléctricas a través de la clase <`Conductor`> y permite crear una ruta para acceder a otros datos relacionados con el objeto "línea".
  - Clase Line (Wires): es la clase de la cual se da origen a la instancia fundamental en este caso, es decir, a partir de ésta se creará el objeto Línea de transmisión LN.
- Transformador TRF1
  - Clase TransformerWinding (Wires): Contiene gran número de atributos que representan las características eléctricas de los devanados de transformación.
  - Clase HeatExchanger (Wires): busca representar la carcasa del transformador.

  - Clase PowerTransformer (Wires): a partir de esta clase se origina el objeto Transformador señalando a través de sus atributos las características propias de dicho elemento.

- Carga CAR1
  - Clase EnergyConsumer (Wires): posee atributos propios y relaciones con las cuales se representan las propiedades de una carga en el sistema.

- Generador GEN1
  - Clase SynchronousMachine (Wires): cuenta con las principales características de una máquina síncrona como elemento generador.

- Subestación SUB1
  - Clase Bay: a través de esta clase se puede determinar el tipo de configuración de la subestación.
  - Clase EquipmentContainer: es una clase que busca mostrar el conjunto de elementos que hacen parte de una determinada instancia a partir de las relaciones que posee con otras, se usa para asociar los objetos de líneas y transformador a la clase `<Substation>`.
  - Clase Substation (Wires): instancia el objeto "subestación" para acceder a los distintos atributos de las demás clases asociados a esta.

Para todos los equipos mencionados que hacen parte de este ejemplo se utilizan las clases `<Equipment>` y `<ConductingEquipment>`, las cuales relacionan los equipos a los nodos del sistema y permiten asociar las líneas y el transformador a la clase `<EquipmentContainer>` de la subestación.

De igual manera en este tipo de sistemas es importante gestionar la información de los nodos o barras que lo componen, teniendo en cuenta que cada equipo contiene 1 o más terminales asociados a un nodo, a su vez cada nodo está asociado a un objeto de la clase `<TopologicalNode>`. Este tipo de relaciones y datos se gestionaron a través de las siguientes clases señaladas en el estándar.

- Clase Terminal(Core): permite relacionar los equipos a los nodos de conectividad del sistema.

- ConnectivityNode (Topology): modela los puntos en los cuales se conectan los equipos.

- TopologicalNode (Topology): considera el nivel de tensión en el punto donde se encuentran uno o más nodos de conectividad.

Realizado el proceso de identificación de los atributos y las clases que formarán parte del modelado del circuito de prueba, es importante observar de igual forma en el estándar, el orden jerárquico de dichas clases y de las relaciones que estas pueden guardar con otras. De esta manera se puede realizar en forma más eficiente el siguiente paso en el proceso de implementación de la norma.

Creación del diagrama UML Conociendo las clases, sus atributos y relaciones se puede crear el diagrama UML de cada elemento del sistema y generar a partir de estos un diagrama general que muestre, a través de las clases del CIM, todo el sistema a implementar.

A continuación se muestra el diagrama UML de las líneas de transmisión del sistema de acuerdo a la información que se tiene y en el cual se observa claramente las relaciones de herencia y asociación que existe entre cada una de las clases de acuerdo a lo establecido en el estándar. Este diagrama, al igual que el de varios de los elementos del sistema en cuestión, se puede ver de forma más completa en el estándar IEC 61970-301 (IEC, 2009).

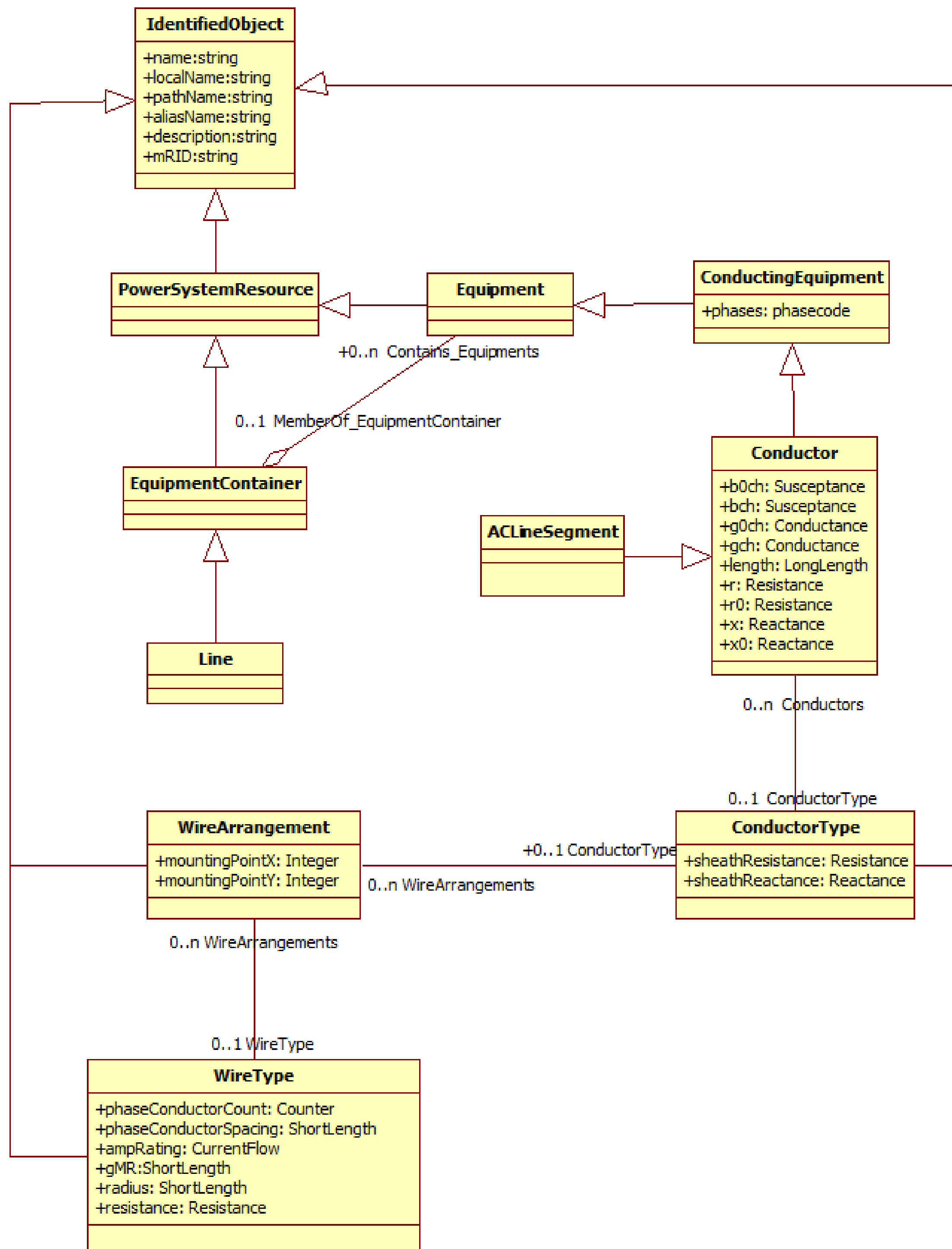

Figura 6.5.: Diagrama UML de las líneas de transmisión

A partir de los diagramas UML de cada uno de los elementos, se crea un diagrama general

donde se modela todo el sistema y de esta forma observar todas las clases que lo componen, ademas del orden de jerarquía en las mismas.

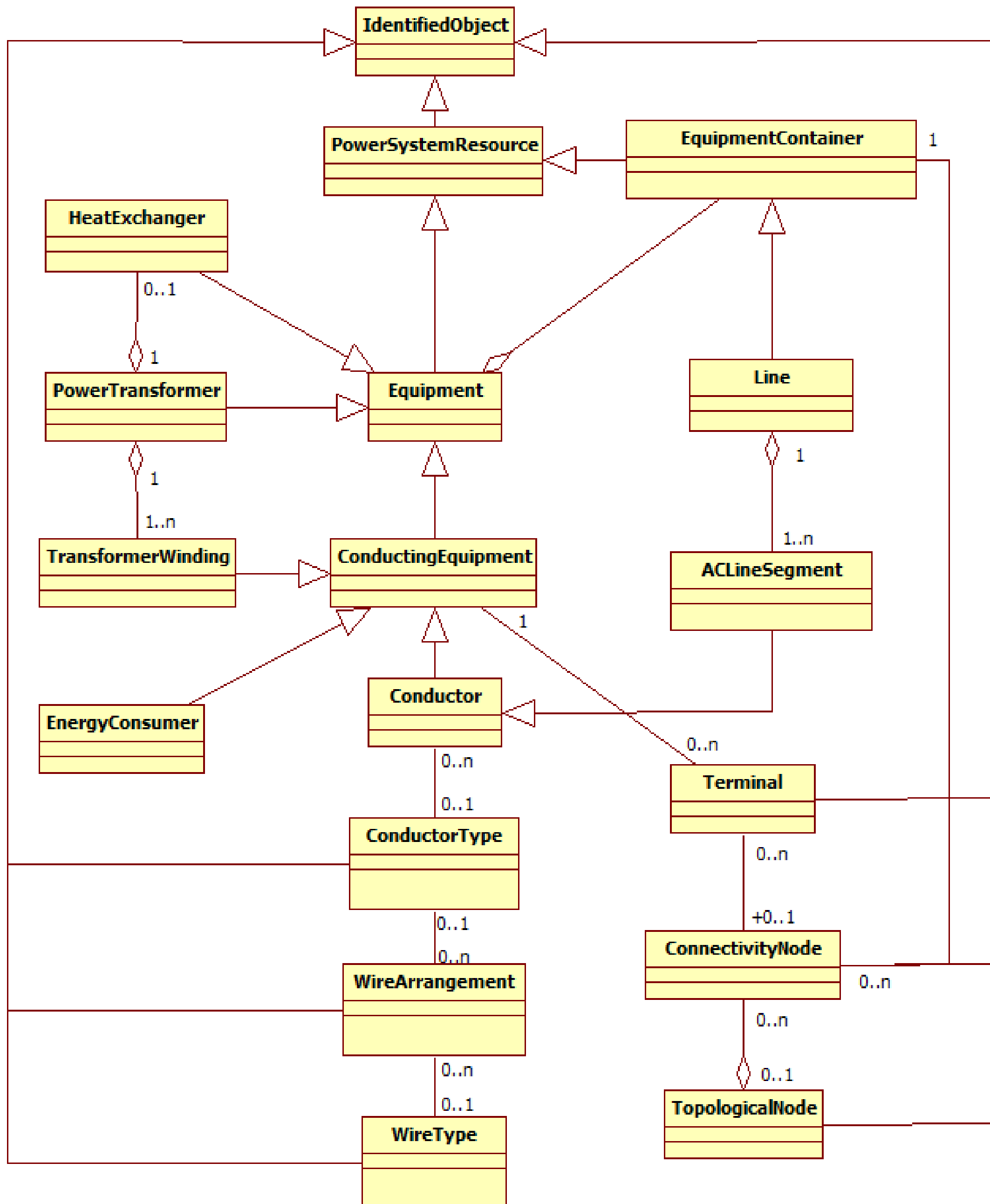

Figura 6.6.: Diagrama UML del sistema

Programación en Python Finalmente se llega a la última etapa que hace referencia a la implementación del estándar IEC 61970-301 mediante el lenguaje de programación Python. Para mostrar cada uno de los pasos que componen esta etapa, se tomará la línea de transmisión LN1 como base para la ejecución de los mismos, teniendo en cuenta que de igual forma se realizará este proceso para todos los demás elementos del sistema. Con relación a esto cabe resaltar que el espacio de nombres para los siguientes casos, está señalado en el capítulo 4 en donde se explica la manera en la que se nombran cada una de las variables que se construyen en Python al modelar cualquier elemento del sistema.

Para la implementación de la línea LN1 se unició creando un nuevo paquete llamado "Dic"que contiene un módulo "Dic_LN1" (ver figura 4.1 del capitulo 4) dentro del cual se definen los diccionarios basicos necesarios para la instanciacion de los atributos que representan los datos o la información que se tiene de la línea. En la creación de estos diccionarios, para cada atributo, se debe tener en cuenta el tipo dato de acuerdo con lo señalado en el paquete "Domain". A continuación se muestra una síntesis de lo planteado anteriormente en la línea de transmisión:

Por ejemplo: la línea tiene asociada una reactancia inductiva propia, este dato se almacena en un atributo de nombre: <b0ch> el cual pertenece a la clase <Conductor>. Este atributo es una instancia de la clase <Susceptance> del paquete "Domain" y tiene dos atributos propios: <value> y <units>. Por lo tanto, la forma de crear el diccionario para este atributo de la líneas es:

```
dic_b0ch_Conductor_LN1={'value':-0.85, 'units':'siemens'}
```

De igual forma se deben crear los diccionarios para los atributos heredados como por ejemplo:

```
dic_phases_ConductingEquipment_CAR1={'value':'ABCN'}
```

Y así se crean todos los diccionarios para los atributos de cada clase.

Es importante tener un orden en la creación de los mismos para identificarlos con mayor facilidad

Una vez creados los diccionarios requeridos, se serializaron utilizando la herramienta "pickle" de Python la cual se detalla en el Apéndice A. Para la serialización de cada diccionario se creó una carpeta de nombre "dic" en un paquete llamado "Interfaz" el cual se mostró en la sección 4 y que se explicará en detalle en el capitulo 5. En esa carpeta se ubican todos los diccionarios que han sido serializados con el mismo nombre del diccionario y utilizando la extensión .pkl, señalado en el espacio de nombres contenido en la sección "Organizacion

de la informacion". La manera de generar los archivos .pkl de estos diccionarios se muestra a continuación:

```
with open(os.path.abspath(".\dic")+"/dic_b0ch_Conductor_LN1.pkl”, ’wb’) as f:
    pickle.dump(dic_b0ch_Conductor_LN1, f)
    f.close()
```

La primera línea de código permite buscar la carpeta creada y generar allí el archivo .pkl del diccionario en cuestión, con la segunda línea se registra la información del diccionario en el archivo creado y por último se cierra el archivo.

Teniendo cada uno de estos diccionarios como archivos .pkl se hace más fácil su gestión en otras aplicaciones del código. La manera de acceder a ellos ya sea para leer su contenido o modificarlo, se puede ver con más detalle en el ejemplo propuesto en el Apéndice A.

Creados todos los diccionarios en los cuales se almacena la información, se continúa con la creación de todas las instancias de clase que hacen parte del modelado de la línea. Para este punto, se creó un nuevo módulo “Main_LN1” el cual contendrá cada uno de los objetos llamados según lo determina el espacio de nombres.

Antes de crear cada una de las instancias en el módulo “Main_LN1” se necesitó importar cada una de las clases que ya han sido programadas bajo la estructura del estándar. Estas importaciones se hacen a través del “API_Module” como se muestra:

```
from API.API_Module import ConductingEquipment, Line, ACLineSegment,\
     Susceptance, Reactance
```

la anterior importación no contiene todas clases usadas en el modulo donde se crearon las instancias de clase de las líneas, solo es una parte para indicar como se llamaron a traves de las API.

Se declararon los atributos de la clase como instancias de clase del paquete Domain teniendo en cuenta que en esta instanciación se requiere definir los valores correspondientes al atributo, los cuales se fijaron con anterioridad en los diccionarios. Es importante tener en cuenta que para definir los valores del atributo a partir de los diccionarios creados en el paquete “Dic” se necesitó cargar dichos diccionarios pero esta vez como archivos .pkl para así poder leerlos en este módulo sin perder los ajustes realizados a los mismos y que queden ligados directamente a la instancia que se crea. El proceso de carga de los diccionarios se realizó en un módulo

diferente llamado “read_dics” con el fin de mantener un mejor orden en el manejo de los paquetes. La manera de cargar los archivos .pkl en el módulo “read_dics” se realizó siguiendo el siguiente esquema:

```
with open(os.path.abspath(".\dic")+"/dic_b0ch_Conductor_LN1.pkl”, ’rb’) as f:
    dic_b0ch_Conductor_LN1=pickle.load(f)
    f.close()
```

Así en el “Main_Module” se importó el módulo “read_dics” para conectar los diccionarios con las instancias. Este primer paso en el proceso se muestra a continuación:

```
from Interfaz.read_dics import*

b0ch_Conductor_LN1=Susceptance(dic_b0ch_Conductor_LN1)
phases_LN1=PhaseCode(dic_phases_ConductingEquipment_LN1)
```

El codigo anterior muestra la instanciación de dos atributos de la clase <Conductor> como ejemplo para la creación de los demas.

Un segundo paso en esta etapa, es la creación del diccionario para la instanciación de la clase. Este diccionario contiene todos los atributos pertenecientes a dicha clase y se asocian a la instancia de clase del atributo que se creó en el literal anterior. Si no se cuenta con un dato del sistema asociado a un atributo determinado no se crea la instancia del mismo y por ende se deja vacio en el diccionario de la clase. Para comprender mejor lo dicho, se muestra a continuación el diccionario para la instancia de clase de <Conductor> para la línea 1:

```
dic_Conductor_LN1={’b0ch’:[b0ch_Conductor_LN1.value,b0ch_Conductor_LN1.units],
                   ’bch’:[bch_Conductor_LN1.value,bch_Conductor_LN1.units],
                   ’g0ch’:[g0ch_Conductor_LN1.value,g0ch_Conductor_LN1.units],
                   ’gch’:[gch_Conductor_LN1.value,gch_Conductor_LN1.units],
                   ’length’:[length_Conductor_LN1.value,length_Conductor_LN1.units],
                   ’r’:[r_Conductor_LN1.value,r_Conductor_LN1.units],
                   ’r0’:[x0_Conductor_LN1.value,x0_Conductor_LN1.units],
                   ’x’:[x_Conductor_LN1.value,x_Conductor_LN1.units],
                   ’x0’:[x0_Conductor_LN1.value,x0_Conductor_LN1.units], }
```

Finalmente se declararon las instancias de clase requeridas. En esta instanciación se deben incluir todos los diccionarios que definan a los atributos propios o heredados de la clase. Un ejemplo de esto se muestra a continuación utilizando la clase <Conductor>:

```
Instance_Conductor_LN1=Conductor(dic_Conductor_LN1,dic_ConductingEquipment_LN1,
                                 dic_IdentifiedObject_Conductor_LN1)
```

Donde <dic_Conductor_LN1> corresponde al diccionario creado en el ejemplo anterior, la clase <dic_ConductingEquipment_LN1> representa el diccionario creado para la instancia de clase de <ConductingEquipment_LN1> y <dic_IdentifiedObject_Conductor_LN1> contiene los datos de los atributos heredados de la clase <IdentifiedObject>.

Para cada elemento del sistema se debe realizar un análisis del concepto de conectividad e instanciar las clases necesarias para dicho proceso. En este caso se parte de que la línea LN1, posee dos terminales cada uno conectados a un nodo perteneciente a la clase <TopologicaNode>. A continuación se muestran algunas instancias de clase requeridas para llevar a cabo el diseño de conexión de la línea, teniendo en cuenta que antes se debieron crear sus diccionarios y seguir los pasos correspondientes para instanciar una clase determinada.

En esta primera parte se instanciaron los dos terminales que componen la línea LN1:

```
Instance_Terminal1_LN1=Terminal(dic_IdentifiedObject_Terminal1_LN1)
Instance_Terminal2_LN1=Terminal(dic_IdentifiedObject_Terminal2_LN1)
```

Con las siguientes líneas de código se determinan los terminales de los diferentes equipos existentes en el sistema que están unidos a los elementos de <ConnectivityNode> y posterior a esto se crearon los objetos de esta última clase.

```
Terminales_ConnectivityNode1=[Instance_Terminal1_LN1,Instance_Terminal1_GEN1]
Terminales_ConnectivityNode2=[Instance_Terminal2_LN1,Instance_Terminal1_TRF1]

Instance_ConnectivityNode1=ConnectivityNode(dic_IdentifiedObject_ConnectivityNode1
Instance_ConnectivityNode2=ConnectivityNode(dic_IdentifiedObject_ConnectivityNode2)
```

Lo mismo se realizó con los objetos de la clase <TopologicalNode>.

```
TopologicalNodes1=[Instance_ConnectivityNode1]
```

```
TopologicalNodes2=[Instance_ConnectivityNode2]

Instance_TopologicalNode1=TopologicalNode(dic_TopologicalNode1,
                                          dic_IdentifiedObject_TopologicalNode1)
Instance_TopologicalNode2=TopologicalNode(dic_TopologicalNode2,
                                          dic_IdentifiedObject_TopologicalNode2)
```

De esta manera se crearon todos los objetos correspondientes a cada una de los terminales de los elementos y los nodos que conforman el sistema de potencia.

Una vez construidas todas las instancias de cada una de las clases que componen el modelado, se realizó la asociación existente entre las diferentes clases según lo señala la norma y la mejor manera de hacerlo es gestionando la estructura de la programación de cada clase es a través sus mismas instancias. Para entender este proceso se muestran a continuación algunos ejemplos.

Para este primer caso se muestran las relaciones existentes entre la clase `<Terminal>` y las clases `<ConductingEquipment >`y `<ConnectivityNode>`.

```
Instance_Terminal1_LN1.conductingEquipment=Instance_ConductingEquipment_LN1
Instance_Terminal2_LN1.conductingEquipment=Instance_ConductingEquipment_LN1
Instance_Terminal1_LN1.connectivityNode=Instance_ConnectivityNode1
Instance_Terminal2_LN1.connectivityNode=Instance_ConnectivityNode2

Instance_ConductingEquipment_LN1.terminal=Terminales_LN1

Instance_ConnectivityNode1.terminal=Terminales_ConnectivityNode1
Instance_ConnectivityNode2.terminal=Terminales_ConnectivityNode2
```

De igual forma la relación entre las clases `<Conductor>` y `<ConductorType>`.

```
Instance_ConductorType_LN1.conductor=Instance_Conductor_LN1
Instance_Conductor_LN1.conductorType=Instance_ConductorType_LN1
```

Este mismo proceso se aplicó para las demás clases que tenían una asociación con otra para que de esta manera se pueda obtener cualquier atributo de la línea que pertenezca a otra clase llamándola desde la instancia propia de la clase `<Line>`.

Con esto y los ítems anteriores se pueden crear las diferentes instancias de clases necesarias

de acuerdo a la información que se quiera gestionar. Una vez instanciada cada clase con sus atributos, una manera de poner a prueba la efectividad del código principal es imprimir cualquier dato que se tenga en el programa y que se haya ingresado como atributo. Un ejemplo de esto es:

- Cuando el atributo es heredado o propio de la instancia

```
print Line.name
>>>> Linea 1 (LN1)
```

- Cuando se desea llegar hasta el atributo de otra clase a partir de la instancia de clase de <Line> usando las asociaciones realizadas anteriormente.

```
print Instance_Line_LN1.aCLineSegment.conductorType.wireArrangement[0].wireType.radius

>>>> [1.177, 'centimetros']
```

Con los objetos ya creados y hechas las asociaciones para cada uno de estos, se serializó cada instancia, a través del mismo proceso realizado para los diccionario. Para este caso se creó una carpeta con nombre “obj” donde estarán contenidas todas las instancias de clase como archivos .pkl. Estos archivos mantendrán el nombre de la instancia que se serializó seguido de la extensión .pkl. Esto se hizo para hacer más eficiente el sistema de ajuste o modificación de la información, es decir, cuando se desea cambiar tan sólo unos cuantos datos de la instancia basta con abrir su archivo .pkl y realizar el ajuste correspondiente, pero si el proceso de modificación es extenso, se recurre a los archivos .pkl de los diccionarios y se realiza el ajuste allí haciendo que cambien también las instancias que se encuentren ligadas con los mismos, estos procesos se pueden ver en detalle en la sección "Interfaz gráfica de usuario".

La serialización de una instancia de clase se realiza de la misma manera que en el caso de los diccionarios cambiando únicamente el nombre del archivo a generarse. También se pueden modificar los atributos de los objetos cuando están serializados, un ejemplo del esquema para modificar una instancia de clase se muestran en el apéndice A.

Para este proyecto se crearon dos módulos llamados “Manejo_dic” y “Manejo_inst” los cuales permiten leer o realizar un ajuste a un diccionario o instancia de clase respectivamente de una forma fácil y dinámica. Su elaboración se basó en los esquemas de ajuste presentados en el apéndice A. Estas aplicaciones se encuentran dentro de la carpeta “Interfaz” tal y como se puede ver en el capítulo 4.

6.4 ARCHIVOS XML PARA EL INTERCAMBIO DE LA INFORMACIÓN

Para darle continuidad al concepto de interoperabilidad de información que plantea el estándar, es necesario generar archivos de texto plano donde se almacene esta información y pueda ser tratada por cualquier intérprete. Una manera de realizar el intercambio de datos a través de este tipo de archivos es utilizar el lenguaje de marcas extensibles XML, eXtensible Markup Language, siendo este un sistema universal para la codificación de información haciendo de esto una de sus principales ventajas. El lenguaje XML es un conjunto de reglas bajo las cuales se crea un orden jerárquico de los datos allí contenidos y una serie de etiquetas que definen la estructura del mismo. Es un lenguaje que puede ser interpretado por muchos otros para ser entendible por la empresa o personas que requiera la información (Steve Widergren & JunZhu, 1999).

Los programas o sistemas que utilizan el formato XML pueden intercambiar fácilmente sus datos debido a que responden a una misma lógica interna. En este caso se busca crear un intercambio de información entre las empresas del sector eléctrico.

Para la creación de estos archivos de texto plano basados en XML se utilizó la librería "xml_marshaller" en Python, la cual posee una clase llamada <xml_marshaller> y a través del método dump se da origen a los archivos xml del sistema en cuestión. A continuación se muestra un ejemplo para la creación de estos archivos xml:

```
from xml_marshaller import xml_marshaller
a=xml_marshaller.dumps(Instance_EnergyConsumer_CAR1)
with open('Instance_EnergyConsumer_CAR1.xml','wb') as f:
    f.write(a)
    f.close()
```

Con el código anterior se generan estos archivos y se puede realizar el intercambio de la información allí contenida y aso ser usada en aplicaciones futuras.

Como pudo verse en este capítulo, la creación de objetos de clase, atributos y relaciones que exige la norma no son tareas arduas ni complejas, solo se debe llevar un orden para gestionar la información y crear secuencialmente todos los parámetros mencionados. Al aplicarse de esta manera la norma, se ve en forma más clara la simplicidad que maneja esta, permitiéndole al programador crear una base de datos limpia, sencilla y fácil de comprender por los usuarios que deseen comunicarse con esta a través de una interfaz.

CAPÍTULO 7

CONCLUSIONES

- Mediante la programación orientada a objetos se pudo construir de forma satisfactoria una base de datos que almacena la información requerida de un sistema de energía eléctrica. Las clases presentes en el estándar IEC 61970-301, fueron suficientes y eficaces al momento de diseñar el circuito de prueba, por lo que la norma cumplió con las exigencias planteadas al inicio de esta investigación.
- El uso del lenguaje de programación Python ofrece todas las herramientas necesarias para materializar la aplicación informática para el almacenamiento de información del circuito de prueba pensada hacia el CIM, a pesar de actuar como un pseudocódigo, no es necesario adentrarse en constructores propios del lenguaje, existen suficientes elementos para generar una aplicación de alto nivel, capaz de responder con los requerimientos del usuario final.
- La interfaz gráfica de usuario elaborada para interactuar con la base de datos CIM, responde de manera eficiente a las distintas acciones que puede ejecutar un usuario para comunicarse con el código fuente. Gracias a la simplicidad en la estructura del lenguaje de programación Python, resulto agradable el trabajo para el desarrollo de aplicaciones GUI que pudiesen manipular ciertos comportamientos de los objetos instanciados para el modelado del circuito de prueba.
- Se caracterizó el estándar IEC 61970-301 para su correcta comprensión y aplicación, para futuras implementaciones, arrojando como resultado una síntesis de la norma y una guía para la lectura y comprensión de la misma, especificando características básicas, clases principales y conceptos esenciales.

- Se establecieron buenas prácticas para la implementación de un modelo de información común, mediante la elaboración de aplicaciones que manipulan en forma eficiente la información contenida en los objetos tipo creados a partir de las clases de los diferentes paquetes que componen la norma. También se logró esto gracias a la información recopilada sobre las herramientas utilizadas en la creación de la aplicación informática final, dejando así, un legado de conocimiento para posteriores aplicaciones o continuación del estudio del estándar en otros campos de la ingeniería eléctrica.

Bibliografía

Checa, Luis María. 1988. *Líneas de transporte de energía.*

IEC. 2009. *Energy management system application program interface(EMS-API)-Part 301:Common information model (CIM) base. INTERNATIONAL ELECTROTECHNICAL COMMISSION.* INTERNATIONAL STANDARD.

IEEE. 2010. *Distribution Test Feeders. Power and Energy Society. Copyright 2000-2010,.* Disponible en línea el día 05 de septiembre 2014 en: http://ewh.ieee.org/soc/pes/dsacom/testfeeders/.

McMorran, Dr Alan W. 2007. An Introduction to IEC 61970-301 & 61968-11: The Common Information Model. *University of Strathclyde Glasgow, UK*, 42.

Phillips, Dusty. 2010. *Python 3 Object Oriented Programming.* Packt Publishing Ltd.

PythonOrg. 2014. *Python 2.7.8 documentation.Python Software Foundation.* Disponible en línea el día 02 Junio 2014 en: https://docs.python.org/2.7/.

StarUMLSoftware. 2014. *StarUML 2 BETA. MKLab. V5.0.2.1570.* Disponible en línea el día 10 Junio 2014 en: http://staruml.io/.

Steve Widergren, Arnold deVos, & JunZhu. 1999. XML for Data Exchange. *IEEE ASSP Magazine.*

Summerfield, Mark. 2007. *Rapid GUI Programming with Python and Qt. The Definitive Guide to PyQt Programming.* Prentice Hall.

ANEXOS

ANEXO A

MANEJO DE ARCHIVOS EN PYTHON

Cuando se tiene una serie de información en Python ésta es guardada en ficheros o archivos para ser leída o modificada posteriormente. De allí surge la necesidad de aprender cómo se pueden crear y además acceder de forma fácil a dichos archivos para ser leídos o modificados, esto es lo que se conoce en Python y en los demás lenguajes de programación como manejo de archivos.

A.1 PASOS FUNDAMENTALES EN EL MANEJO DE ARCHIVOS

Este manejo de archivos se realiza en tres pasos fundamentales:

A.1.1 Apertura del archivo o fichero

Para abrir un archivo en Python se tiene la función `open()`. Al llamar esta función se requieren dos argumentos: el primero hace referencia al nombre del archivo que se quiere abrir con su respectiva extención, es decir, .txt, .py, .pkl etc, indicando el path o camino si dicho fichero no se encuentra en la misma carpeta donde se halla el módulo sobre el cual se está trabajando. Y el segundo es el modo de acceso, el cual determina el fin para el cual se abrirá el archivo. Los modos de acceso fundamentales en el manejo de archivos son los siguientes:

- r: indica que el archivo se abrirá para lectura. Si se señala este modo de acceso para un archivo que no existe de inmediato se mostrará un error.
- w: este se establece cuando se quiere escribir en el archivo. Cuando el archivo ya existe se escribe la nueva información sobre la que se encuentre en dicho fichero; en el caso de que el archivo no esté creado y se llama este modo de acceso se creará el fichero.

- a: este modo indica que se agregará información a un determinado archivo. Como en el caso anterior, si el archivo no existe se creará de forma inmediata, pero si existe, este modo hará que se escriba lo deseado después del último carácter que contenga dicho archivo.

  A partir de estos modos de acceso básicos surgen otros utilizados en los diferentes lenguajes de programación que realizan la misma función pero además proporcionan mayor especificación del caso. Estos se muestran a continuación:

- rb: este modo abre un archivo de sólo lectura en formato binario.
- r+: se abre el archivo para lectura y escritura ubicando el puntero en el principio del archivo.
- rb+: abre el archivo para lectura y escritura en formato binario.
- wb: se abre el archivo para escribir en formato binario.
- w+: abre el fichero para lectura y escritura.
- wb+: se abre el archivo tanto para lectura como para escritura en formato binario.
- ab: abre el archivo en formato binario para agregar información.
- a+: con este se abre el fichero para anexar información y para la lectura del mismo.
- ab+: abre el archivo para leer y agregar información en formato binario.

Por lo tanto la estructura para la apertura de archivos se muestra a continuación:

```
with open(nombre del archivo,modo) as varible:
```

Otra forma sería:

```
variable= open(nombre del archivo,modo)
```

Un ejemplo de esto es:

```
with open(“diccionarios.txt”,”w”)) as f:
```

A.1.2 Lectura y escritura del archivo De acuerdo al fin para el cual fue abierto el archivo o el modo de acceso al mismo se tienen los siguientes métodos:

- Para escritura
  - `write()`: con este método se escribe determinada información sobre la que se encuentre en esos momentos en el archivo. Si se desea escribir una cadena de string basta con colocar dicha cadena dentro de este método, o en el caso de que se desee registrar algún tipo de dato en forma de diccionario o lista se coloca la variable que contenga dichos datos dentro de los paréntesis del método. Un ejemplo de esto se muestra a continuación:

    ```
    f.write(“Common Information Model”)
    ```

    En el caso de tener un diccionario sería de la forma:

    ```
    dic={‘potencia’:3.6 , ‘unidad’:MW}
    f.write(dic)
    ```
- Para lectura
  - `read()`: permite leer la totalidad del contenido del archivo. Cuando a este método se le agrega un valor n permite leer los primeros n bytes del archivo.
  - `readline()`: Con este método se lee una línea completa del archivo y se ubica el puntero interno al inicio de la siguiente línea.
    Un ejemplo de ejecutar estos métodos sería:

    ```
    f.read()
    ```

A.1.3 Cierre del archivo Una vez finalizado la ejecución de los métodos y realizadas las acciones a los archivos, es importante realizar el cierre de los mismos. Esto se hace para asegurar los cambios efectuados en él y además liberar el archivo para que pueda ser usado en otro caso o programa. La manera de efectuar dicho cierre es a través del método <.close()>. Una vez ejecutado este método no se podrá acceder a dicho archivo a menos que se vuelva a realizar la apertura. La manera de hacerlo es como se muestra:

```
f.close()
```

Por lo tanto el esquema completo al manejar un archivo se muestra con el siguiente ejemplo:

```
with open('diccionario.txt', 'wb')as f:
    f.write("Este es un diccionario")
    f.close()
```

A.2 Pickle

En cientos de aplicaciones de la programación orientada a objetos surge la necesidad de transformar el estado de un objeto con el fin de almacenarlo, recuperar y transportarlo, a este proceso de transformación se le conoce como serialización. La serialización hace referencia al proceso por el cual un objeto en Python se convierte en una cadena de bytes o caracteres. Esta cadena, de igual manera, debe tener toda la información necesaria para reconstruir el objeto en otro script en Python y realizar el proceso inverso cuando el caso lo requiera.

Para este tipo de situaciones Python tiene varios módulos o librerías que permiten la realización de dicho proceso. En este caso se seleccionó la librería pickle como herramienta para la serialización y de-serialización de los elementos, manteniendo así una persistencia en los datos que se estén gestionando. El concepto de persistencia hace referencia a almacenar determinados objetos en archivos para poder recuperarlos posteriormente sin llegar a perderse los diferentes ajusten presentados en el proceso.

En este módulo se debe identificar dos conceptos importantes:

- Pickling: es el proceso en el cual un objeto Python se convierte en un flujo de bytes.
- Unpickling: es la operación inversa, mediante el cual una cadena de bytes se convierte de nuevo en un objeto.

Al momento de implementar pickle se tienen dos métodos fundamentales bajo los cuales se realiza su función principal. Estos métodos son:

- `<dump()>`: Esta función tiene como argumentos el objeto a serializar y el archivo con su respectiva extensión en el cual se guardará dicho objeto siempre y cuando este archivo ya esté abierto con anterioridad y allí mismo se muestre el método para el cual fue abierto ('w' ,'a') . Una vez ejecutada esta función ya se tiene serializado un objeto en un archivo .pkl.
- `<load()>`: esta función es utilizada para cargar un objeto serializado que esté guardado en un archivo determinado. Eso se ejecuta siempre y cuando antes se tenga abierto el archivo bajo el método de lectura (´r´).

A continuación se muestra la forma de serializar un objeto utilizando pickle.

Aplicación de la librería pickle Antes de iniciar la serialización de un objeto determinado se debe tener claro el concepto de manejo de archivos en Python que se mostró anteriormente y en donde se describe como leer o crear un fichero determinado debido a que estos elementos serán utilizados en la creación de un archivo serializado pickle.

Para la serialización de un objeto Python, a través de la librería pickle, se siguen estos pasos:

1. Importar la librería: una vez ubicado en el módulo python la importación se debe hacer de la siguiente manera:

   ```
   Import pickle
   ```

2. Creación del objeto a serializar: En este punto se crea la lista, el diccionario o la instancia que se desea guardar en un archivo pickle. Como por ejemplo:

   ```
   Dic1={‘clave1’:’valor1’ , ‘clave2’:’valor2’ , ‘clave3’:’valor3’}
   ```

3. Abrir el archivo donde se va a guardar: Se debe tener en cuenta la sintaxis utilizada en el manejo de archivos.

   ```
   file=open(’Dic1.pkl’,’wb’)
   ```

   Si el este archivo se desea crear en una carpeta diferente es necesario indicar el path en el primer argumento de la función <open()>.

4. Se ejecuta la función dump: con la cual se guarda el objeto creado en el archivo abierto para escritura.

   ```
   pickle.dump(Dic1,file)
   ```

5. Se cierra el archivo

   ```
   file.close()
   ```

De esta manera ya se tiene el diccionario <Dic1> serializado en un archivo.pkl el cual podrá emplearse en otros módulos y además guardará cualquier tipo de ajuste aplicado al mismo. Para poder cargar este archivo .pkl creado se utiliza la función <load> como se muestra a

continuación:

```
f=open('Dic1.pkl', 'rb')
dic=pickle.load(f)
f.close()
```

En este caso, inicialmente se debe abrir en modo lectura el archivo donde se encuentra el objeto, posterior a ello cargarlo con la función <load> haciendo que esa cadena de bytes que se tenían se conviertan nuevamente a un objeto Python. Para finalizar, se debe cerrar el archivo como se mostraba anteriormente. Si en este caso se pidiera imprimir el valor de `Dic1` mostrará:

```
Print dic
{'clave1':'valor1' , 'clave2':'valor2' , 'clave3':'valor3'}
```

Guardando cualquier ajuste que se le haya aplicado al diccionario y por ende manteniendo la persistencia del objeto.

Un ejemplo completo de esto se muestra a continuación teniendo en cuenta que el archivo se crea en la misma carpeta:

```
dic_r_Conductor_LN1={ 'value':0.1159, 'units':'Ohms', }
with open('dic_r_Conductor_LN1.pkl','wb') as file:
pickle.dump(dic_r_Conductor_LN1,file)
file.close()
```

Para cargar este diccionario sería de la siguiente forma:

```
with open('dic_r_Conductor_LN1.pkl', 'rb')as f:
dic_r_Conductor_LN1=pickle.load(f)
f.close()
```

Así como se serializó el anterior diccionario así también se puede generar un archivo pickle de cualquier lista o instancia que se tenga.

Si se deseará modificar la clave 'value' de este diccionario se debe registrar las siguientes líneas de código:

```
with open('dic_r_Conductor_LN1.pkl', 'rb')as f:
dic_r_Conductor_LN1=pickle.load(f)
f.close()
dic_r_Conductor_LN1 ['value'] = 0,225
with open('dic_r_Conductor_LN1.pkl','wb') as file:
pickle.dump(dic_r_Conductor_LN1,file)
file.close()
```

Quedando así guardado el nuevo valor del atributo.

Si en otro caso se deseará cambiar el atributo de una instancia de clase que se encuentra como archivo .pkl sería de una manera muy similar que el caso anterior:

```
with open('Instance_EnergyConsumer_CAR1.pkl', 'rb')as f:
Instance_EnergyConsumer_CAR1=pickle.load(f)
f.close()
Instance_EnergyConsumer_CAR1.name="Carga trifásica 1"
with open('Instance_EnergyConsumer_CAR1.pkl','wb') as file:
pickle.dump(Instance_EnergyConsumer_CAR1,file)
file.close()
```

Manteniendo de esta manera la persistencia de la información.

ANEXO B

USO DEL ESPACIO DE NOMBRES

El espacio de nombres en un concepto de programación que ayuda a identificar objetos instanciados y poderlos clasificar o hacer uso de ellos en aplicaciones posteriores del código, esta clasificación se logra a través de los llamados "identificadores". Un identificador es único dentro de un espacio de nombres pero también puede ser utilizado en otros, es decir, puede tener una única connotación dentro del nombre de una instancia o estar asociado a múltiples instancias que poseen un parámetro en común y que por ende, deben ser diferenciadas del resto de objetos particulares. A continuación se dará una introducción teórica hacia los espacios de nombres, su utilización en la programación orientada a objetos y el uso particular de este concepto en la aplicación de la norma IEC 61970-301.

B.1 ESPACIO DE NOMBRES

Un espacio de nombres tiene como concepción básica un conjunto en el cual, todos los nombres son únicos. Cada instacia dentro de una aplicación de software orientada a objetos debe poder identificarse fácilmente gracias a la correcta construcción de su espacio de nombres a esta propiedad que los hace unicos se le conoce como: "nombre", sin embargo, este puede tener similitudes con otros gracias a los identificadores, los cuales otorgan a un objeto una cualidad para que puedan pertenecer a un grupo determinado dentro del código, esto se hace con el fin de identificar grandes cantidades de elementos que comparten una característica en común y necesiten ser asociados con algún método, función general o atributo universal. Todo lo anterior depende de la aplicación que se esté realizando por el diseñador, por ejemplo: Juan estudia una carrera "X" y tiene el código de estudiante "123" mientras que Andrea estudia

una carrera "Y" y tiene el mismo código "123", pero la razón por la que Juan y Andrea pueden ser identificados es el hecho que estudian carreras distintas y la característica que los une es el hecho de ser estudiantes.

En Python el espacio de nombres ayuda mucho a la eficiencia del código en cuanto a la utilización de variables. Un ejemplo implícito dentro de Python son las variables de tipo local y global. Cuando una variable se declara como global puede ser utilizada por distintos módulos de Python y asociar de esta manera dichos archivos .py. por ejemplo, se tienen dos funciones cada una con una variable de nombre "var" a la cual se le asigna un valor, en la primera función a esta se le da un valor de tipo string y en la segunda se le asigna un valor de tipo numérico, al ejecutar ambas funciones y llamar al valor de dicha variable se puede ver que sus valores son distintos aún llamándose igual.

```
def HolaMundo():
    var='Hola mundo'
    print var

def Numero():
    var=10
    print var

HolaMundo()
Numero()

>>>> 'Hola mundo'
>>>> 10
```

Este resultado se debe a que las variables se usan de forma local es decir su identificador implícito es el método al que pertenecen, pero, si la variable "var" se declarase como global, estaría siendo compartida por los dos métodos, es decir seria la misma para ambos casos. Por ejemplo, si ahora solo se le da valor a "var" dentro del método `<HolaMundo>` y se imprime en el método `<Número>` se mostrará en consola el valor asignado en la primera función.

```
def HolaMundo():
    global var
    var='Hola mundo'
```

```
    print var

def Numero():
    global var
    print var

HolaMundo()
Numero()

>>>> 'Hola mundo'
>>>> 'Hola mundo'
```

Puede verse claramente que al ejecutar los dos métodos el valor de "var" es el mismo ya que al ser de tipo global, esta no contiene ningún identificador que pueda diferenciarla en cualquier método en el cual se declare o se use dicha variable. Este es un buen ejemplo de la importancia del uso correcto del espacio de nombres en los programas informáticos, ya que como se mostró puede ser de utilidad al momento de reducir el número de variables utilizadas en una aplicación.

Otro caso implícito del uso del espacio de nombres y de los identificadores está en la declaración de atributos en las clases de una estructura jerárquica, es decir, dos clases distintas pueden tener un mismo atributo pero ligado directamente a la clase que los contiene, por ejemplo: la clase `<A>` tiene un atributo `<real>` y la clase `<B>` contiene al atributo `<real>`, ósea que `<A.real>` es diferente de `<B.real>` debido a que `<real>` es local a la clase que lo contiene.

En Python existen muchos ejemplos implícitos del manejo del espacio de nombres para el beneficio de la memoria de un software y para la facilidad de programación del mismo; es evidente que en estos casos es casi imposible adecuar estas normas (refiriendose a manipular las reglas internas del lenguaje Python), pero lo que sí se puede hacer es manipular el espacio de nombres de los objetos y funciones creados para asignar comportamientos y generar resultados a gran escala gracias a la participación de algunos elementos requeridos para dicho fin en una aplicación informática. Para este proyecto en particular se crearon unas convenciones o identificadores en el espacio de nombres de las instancias de clase, valores de los atributos, paquetes, módulos y algunas funciones, para mejorar la eficiencia del código, poder asignar comportamientos y generar resultados gracias a la interacción de algunos objetos.

B.2 Importancia del espacio de nombres en la serializacion de objetos

La explicación de porque se utilizó dichas sintaxis en las instancias y los diccionarios corresponde a los métodos instaurados para la serializarían y des serialización de los mismos. La aplicación necesita identificar todas las instancias de clase y diccionarios para generar sobre estos los archivos serializados que conseguirán la persistencia de la información, estos ficheros serializados poseen igual nombre que las instancias y diccionarios tratados y su extensión será .pkl (la cual también fue determinada por los autores del presente proyecto). Al tener toda la información serializada y siguiendo un patrón determinado gracias al uso adecuado del espacio de nombres, es posible serializar objetos (diccionarios e instancias de clase) durante su creación y cargar la información en los archivo .pkl para la lectura de los objetos y funciones por medio de la interfaz.

En el Apéndice C se puede apreciar mucho mejor lo mencionado en esta sección, ya que es allí donde se explican los pasos a seguir para la creación de nuevas clases y por ende se deben serializar para poder ser usadas por la interfaz, mostrando el potencial y la necesidad del uso adecuado de las sintaxis presentadas en este anexo.