SPICEX, SIMULACIÓN GRÁFICA DE CIRCUITOS ELECTRÓNICOS EN LINUX

Memoria para optar al Título Profesional de Ingeniero Civil Electrónico
Mención: Control Automático

RICARDO FRANCO MENDOZA GARCÍA

Profesor Guía: Ramón Guirriman Carrasco
Profesor Informante: Raúl Sanhueza Hormazabal

Arica, Chile 2006
Dedicatoria

Al señor Ricardo Mendoza Mamani y la señora Inelia García Rojas: mis padres; quienes, sin exigirme nada, me enseñaron a dar lo máximo.
Agradecimientos

A mis padres, mis hermanos Ruth y Victor, mi cuñado Frankie, y sobrinitos Tamara y Lucas: mi familia, y al profesor Raúl Sapiain Araya, quienes me animaban permanentemente a concluir el presente trabajo, cuando la situación dificultaba su desarrollo.

A la señorita Magdalena Dobrajska, mi compañera, quien se ha encargado de mantener mi salud mental y quien ha sacrificado incontables horas de su tiempo libre, para hacerme compañía durante el desarrollo del proyecto.

A todo el resto de mi familia, en el extranjero y en Chile, quienes siempre se han encargado de atenuar la melancolía que periódicamente aparece al encontrarme tan distante de mi tierra natal.

A mi permanente grupo de amigos con quienes he compartido penas y alegrías y a los que tengo siempre presente al momento de forjar mi futuro.

A mi permanente grupo de amigos con quienes he compartido penas y alegrías y a los que tengo siempre presente al momento de forjar mi futuro.

Al señor John Hoffner, mi supervisor en la compañía que dio soporte económico durante un largo período de tiempo a mis estudios: Damixa; quien, sin obligación alguna, siempre soportó mis demandas; especialmente cuando la memoria consumía gran parte de mi tiempo.

A mi profesor guía, el señor Ramón Guirriman Carrasco, y a mi profesor informante, el señor Raúl Sanhueza Hormazabal, quienes tuvieron la paciencia de guiar, revisar, y corregir un trabajo a distancia; y durante un período de tiempo excepcional.

A la señorita Magdalena Campusano Vergara, siempre dispuesta a resolver los problemas de los alumnos de la carrera de Ingeniería Civil Eléctrica-Electrónica.

A todos los profesores que confiaron en mí al momento de seleccionar uno de los alumnos que viajarían a perfeccionar sus conocimientos al extranjero, y que hicieron posible concluir mi proyecto final de carrera.
# Índice general

<table>
<thead>
<tr>
<th>Sección</th>
<th>Página</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dedicatoria</td>
<td>I</td>
</tr>
<tr>
<td>Agradecimientos</td>
<td>II</td>
</tr>
<tr>
<td>Resumen</td>
<td>VIII</td>
</tr>
<tr>
<td><strong>1. Introducción</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1. Motivación</td>
<td>1</td>
</tr>
<tr>
<td>1.1.1. El plus de las universidades</td>
<td>1</td>
</tr>
<tr>
<td>1.1.2. De la teoría a la práctica</td>
<td>2</td>
</tr>
<tr>
<td>1.1.3. Linux: el camino legal</td>
<td>3</td>
</tr>
<tr>
<td>1.2. Objetivo General</td>
<td>3</td>
</tr>
<tr>
<td>1.3. Alcances del Proyecto</td>
<td>4</td>
</tr>
<tr>
<td><strong>2. Linux y la Electrónica</strong></td>
<td>5</td>
</tr>
<tr>
<td>2.1. Linux y el usuario final</td>
<td>5</td>
</tr>
<tr>
<td>2.2. Alternativas gratuitas</td>
<td>6</td>
</tr>
<tr>
<td>2.2.1. Oficina</td>
<td>6</td>
</tr>
<tr>
<td>2.2.2. Matemática</td>
<td>6</td>
</tr>
<tr>
<td>2.2.2.1. Cálculo numérico</td>
<td>7</td>
</tr>
<tr>
<td>2.2.2.2. Cálculo simbólico</td>
<td>7</td>
</tr>
<tr>
<td>2.2.3. Electrónica</td>
<td>8</td>
</tr>
<tr>
<td>2.2.3.1. Grupos EDA</td>
<td>8</td>
</tr>
<tr>
<td>Sección</td>
<td>Página</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>2.2.3.2. Captura esquemática</td>
<td>8</td>
</tr>
<tr>
<td>2.2.3.3. Diseño de placas impresas</td>
<td>9</td>
</tr>
<tr>
<td>2.2.3.4. Simulación electrónica</td>
<td>10</td>
</tr>
<tr>
<td>2.3. Captura esquemática y simulación electrónica</td>
<td>11</td>
</tr>
<tr>
<td>2.3.1. Oregano</td>
<td>12</td>
</tr>
<tr>
<td>2.3.2. Qucs</td>
<td>13</td>
</tr>
<tr>
<td>2.3.3. KTechlab</td>
<td>13</td>
</tr>
<tr>
<td>3. Orientación a Objetos</td>
<td>15</td>
</tr>
<tr>
<td>3.1. Principios</td>
<td>15</td>
</tr>
<tr>
<td>3.1.1. Abstracción</td>
<td>16</td>
</tr>
<tr>
<td>3.1.1.1. Abstracción de un objeto</td>
<td>16</td>
</tr>
<tr>
<td>3.1.1.2. Clases de objetos</td>
<td>18</td>
</tr>
<tr>
<td>3.1.1.3. Composición de clases</td>
<td>18</td>
</tr>
<tr>
<td>3.1.2. Herramientas</td>
<td>19</td>
</tr>
<tr>
<td>3.1.2.1. Encapsulación</td>
<td>19</td>
</tr>
<tr>
<td>3.1.2.2. Herencia</td>
<td>21</td>
</tr>
<tr>
<td>3.1.2.3. Polimorfismo</td>
<td>22</td>
</tr>
<tr>
<td>3.2. Metodología</td>
<td>24</td>
</tr>
<tr>
<td>3.2.1. Enfoque</td>
<td>24</td>
</tr>
<tr>
<td>3.2.2. Procesos de desarrollo</td>
<td>25</td>
</tr>
<tr>
<td>3.2.3. Pautas a seguir</td>
<td>26</td>
</tr>
<tr>
<td>4. Rational Unified Process - RUP</td>
<td>28</td>
</tr>
<tr>
<td>4.1. Principios</td>
<td>28</td>
</tr>
<tr>
<td>4.1.1. Proceso de desarrollo iterativo</td>
<td>29</td>
</tr>
<tr>
<td>4.1.2. Administración de los requerimientos</td>
<td>29</td>
</tr>
<tr>
<td>4.1.3. Estructura del RUP</td>
<td>31</td>
</tr>
</tbody>
</table>
ÍNDICE GENERAL

4.2. Metodología ......................................................... 31
  4.2.1. Fases ......................................................... 31
    4.2.1.1. Concepción ........................................... 31
    4.2.1.2. Elaboración .......................................... 32
    4.2.1.3. Construcción .......................................... 32
    4.2.1.4. Transición ............................................. 33
  4.2.2. Iteraciones ................................................. 33
  4.2.3. Disciplinas .................................................. 34
    4.2.3.1. Administración del Proyecto ......................... 34
    4.2.3.2. Requerimientos (Análisis) .......................... 35
    4.2.3.3. Modelo de Negocio (Análisis) ....................... 35
    4.2.3.4. Diseño .................................................. 35
    4.2.3.5. Implementación ........................................ 35
  4.2.4. Artefactos .................................................. 36
    4.2.4.1. Modelo de Casos de Uso .............................. 36
    4.2.4.2. Plan de Iteración .................................... 41
    4.2.4.3. Modelo del Dominio .................................. 42
    4.2.4.4. Modelo del Diseño .................................... 46
    4.2.4.5. Modelo de la Implementación ......................... 54

5. Desarrollo de SpiceX .................................................. 55
  5.1. Concepción .................................................... 55
    5.1.1. Disciplina de Requerimientos .......................... 55
      5.1.1.1. Modelo de Casos de Uso ............................ 55
    5.1.2. Disciplina de Administración del Proyecto ............. 59
      5.1.2.1. Plan de Iteración ................................... 59
  5.2. Elaboración: Primera Iteración ............................... 60
ÍNDICE GENERAL

5.2.1. Disciplina de Modelo de Negocio .................................. 60
  5.2.1.1. Modelo del Dominio ........................................... 60
5.2.2. Disciplina de Diseño .................................................. 62
  5.2.2.1. Modelo del Diseño: Diagramas de Interacción .......... 62
5.2.3. Disciplina de Implementación ..................................... 66
  5.2.3.1. Modelo de la Implementación ............................... 66
5.2.4. Disciplina de Administración del Proyecto .................... 67
  5.2.4.1. Plan de Iteración ............................................. 67
5.2.5. Disciplina de Requerimientos .................................... 67
  5.2.5.1. Modelo de Casos de Uso .................................... 67
5.3. Elaboración: Segunda Iteración ...................................... 69
  5.3.1. Disciplina de Modelo de Negocio ............................... 69
    5.3.1.1. Modelo del Dominio ....................................... 69
  5.3.2. Disciplina de Diseño ............................................. 69
    5.3.2.1. Modelo del Diseño: Diagramas de Interacción ....... 69
  5.3.3. Disciplina de Implementación .................................. 72
    5.3.3.1. Modelo de la Implementación ............................. 72
  5.3.4. Disciplina de Administración del Proyecto .................. 74
    5.3.4.1. Plan de Iteración .......................................... 74
  5.3.5. Disciplina de Requerimientos .................................. 74
    5.3.5.1. Modelo de Casos de Uso .................................. 74
5.4. Elaboración: Tercera Iteración ...................................... 76
  5.4.1. Disciplina de Modelo de Negocio ............................... 76
    5.4.1.1. Modelo del Dominio ....................................... 76
  5.4.2. Disciplina de Diseño ............................................. 76
    5.4.2.1. Modelo del Diseño: Diagramas de Interacción ....... 76
  5.4.3. Disciplina de Implementación .................................. 78
    5.4.3.1. Modelo de la Implementación ............................. 78
ÍNDICE GENERAL

6. Conclusiones 82
   6.1. EDA en Linux .......................... 82
   6.2. Webometrics .......................... 83
   6.3. Alcances logrados ....................... 84

Apéndice 86

A. Código de SpiceX 86

B. Diagramas de Interacción 87

Bibliografía 96
Resumen

El siguiente trabajo implementa una interfaz gráfica para asistir la creación de topologías de circuitos electrónicos en formato de texto: netlist; que satisfagan la sintaxis específica de diversos núcleos de simulación de línea de comandos disponibles en Linux.

Además, el contenido trata de graficar, en general, el estado actual de los programas para la automatización del diseño electrónico en Linux, y de presentar el marco teórico necesario para entender y mejorar el sistema a futuro.

La metodología de desarrollo seguida al momento de implementar la aplicación es el Rational Unified Process, el cual aborda los requerimientos del proyecto de una forma rápida, directa, organizada y eficaz.

Independiente del pequeño avance alcanzado en las fases del proceso de desarrollo, y gracias a las pautas de implementación propuestas por el mismo, la arquitectura principal de la aplicación está prácticamente finalizada. Es así que el programa es funcional, puede ser y ha sido fácilmente testead, y permite recibir realimentación efectiva. Todo esto, en conjunto con un código y funcionalidad de fácil expansión por parte de los usuarios.

Los alcances técnicos logrados son: implementación de circuitos electrónicos de forma gráfica y asistida, con posibilidades de modificación y almacenado; simulación de análisis transiente de los circuitos, disponiendo para ello de una librería de componentes básicos, pero suficientes para una amplia gama de configuraciones; el soporte de los núcleos de simulación Spice y Ngspice, para efectuar dichas simulaciones; y la posibilidad de incrementar la cantidad de componentes soportados, mediante la importación de librerías de componentes para Spice.
Capítulo 1

Introducción

1.1. Motivación

1.1.1. El plus de las universidades

Es de conocimiento general la preocupación de las universidades por mejorar y destacar sus carreras, logrando así prestigio y distinción entre sus pares. Así como algunas universidades se destacan por la sobresaliente capacidad administrativa que presentan los alumnos que egresan de éstas, otras lo hacen por el bagaje tecnológico que poseen los profesionales que en ellas se forman.

Mirando los criterios de evaluación de excelencia, aplicados por importantes medios de comunicación al momento de elaborar rankings de universidades a nivel mundial, queda de manifiesto la importancia de las publicaciones generadas en las instituciones de educación superior[7]. En este contexto, no existe una ley que limite el desarrollo y proliferación de investigaciones, manuales y cualquier tipo de publicaciones científico-técnicas, a los académicos de las universidades.

Asumiendo esto, el alumno tiene la capacidad de aportar en forma directa al prestigio que posee la institución que a futuro respaldará su nivel de educación, usando como herramienta la publicación del conocimiento[4]. Es decir, el estudiante tiene la posibilidad de generar ese plus tan conveniente, tomando en sus manos la responsabilidad hasta ahora delegada a las administraciones universitarias.
CAPÍTULO 1. INTRODUCCIÓN

1.1.2. De la teoría a la práctica

¿Cómo lograr esto? La respuesta a esta interrogante es tema de interesantes debates. Una medida inicial puede ser la generación de espacios de tiempo para desarrollar esta capacidad, tan limitada por la saturación de asignaturas presente en las carreras de Ingeniería. Sin embargo, con un alumno desmotivado, cualquier optimización curricular sería infructuosa. Independiente de la forma como se impulse este macro objetivo, sea como resultado de una buena política de acción de parte de los académicos o como producto de la motivación interna del estudiante, éste último debería incursionar, en el transcurso de su formación profesional, en el ámbito de la investigación, innovación y publicación. Posteriormente, si la consecuencia del esfuerzo son buenas críticas, es muy probable que el afán de superación genere en el alumno una motivación extra.

Sin embargo, se debe tener presente que llevar a cabo una publicación en formato electrónico requiere una serie de programas, entre los cuales están: sistema operativo, editores de texto, hojas de cálculo, editores de ecuaciones, editores gráficos, sistemas para administración de bases de datos, etcétera. Si la publicación está relacionada con el mundo de la Ingeniería Electrónica, se podrían necesitar, además, programas para realizar las siguientes tareas: cálculo simbólico, cálculo numérico, captura esquemática, diseño de PCBs, y simulación electrónica, entre otros.

No se puede pensar en reconocimiento institucional, partiendo de una base ilegal. Lamentablemente, muchas universidades carecen de recursos suficientes para la adquisición de las licencias necesarias para que sus alumnos realicen publicaciones. De más esta decir, que para un alumno en particular esto resulta imposible.

Esto último significa un gran impedimento para el objetivo propuesto, siempre y cuando, se considere la utilización de programas comerciales como herramientas de trabajo.
CAPÍTULO 1. INTRODUCCIÓN

1.1.3. Linux: el camino legal

Es en este punto donde hace su aparición Linux. Este sistema operativo provee todas las herramientas necesarias para llevar a cabo cualquier tipo de publicación en forma totalmente gratuita, sin temor a enfrentar problemas de patentes y demandas tanto a nivel personal como institucional.

También es cierto que la migración hacia un sistema operativo distinto supone una curva de aprendizaje, la cual se relaciona con el proceso de adaptación a una nueva forma de hacer las cosas; no más difícil, pero sí diferente. Por tal motivo, en la actualidad, la mayoría de las aplicaciones en Linux disponen de interfaces gráficas, similares a las que utiliza Windows; las cuales facilitan la migración y el acercamiento a las nuevas herramientas informáticas.

Sin embargo, existe una excepción, que la conforman las aplicaciones CAD\(^1\), particularmente, las de uso en Ingeniería Eléctrica y Electrónica. Estas herramientas de línea de comandos, si bien han sido el núcleo de muchas aplicaciones comerciales de Windows, aún son poco intuitivas.

La posibilidad de eliminar esa carencia presente en Linux, muchas veces clave en la decisión de un electrónico de no utilizar este sistema operativo, y el sueño de aportar con algo al mejoramiento de la excelencia académica en la universidad; no sólo por la institución, sino, además, por el país y la región... es motivante.

1.2. Objetivo General

Por todo lo expuesto anteriormente, el objetivo principal de este proyecto es la creación de una interfaz gráfica para el simulador electrónico Spice\(^2\) de Linux: SpiceX; que se familiarice con las herramientas de trabajo actuales de los estudiantes de Ingeniería Electrónica.

\(^1\)CAD: Computer Aided Design.

\(^2\)Spice es la herramienta de simulación electrónica que dio origen a Pspice, un popular simulador electrónico para Windows, considerado un standard en la industria.
1.3. Alcances del Proyecto

Se desea poner a disposición del alumno, todas las herramientas necesarias para llevar a cabo un trabajo de investigación, sin producir cambios drásticos en su metodología de trabajo. Considerando la complejidad del objetivo, el desarrollo deberá enfocarse de tal forma que el programa presente las siguientes características:

1. código de fácil desarrollo y mantención,
2. funcionalidad suficiente para comenzar su distribución,
3. y librerías electrónicas de fácil expansión por parte de los usuarios.
Capítulo 2

Linux y la Electrónica

Al momento de escribir el presente capítulo, Linux se encuentra compitiendo seriamente con otros sistemas operativos que, por haberse enfocado desde su génesis en la facilidad de uso, continúan ocupando la mayoría de los escritorios de usuarios.

2.1. Linux y el usuario final

En Enero del 2006, la organización de consumidores y usuarios de España, publicó un interesante artículo que compara las ventajas y desventajas de los sistemas operativos: Windows XP, Mac OS X, y Ubuntu Linux, al momento de ser utilizados en computadores personales[3].

Independiente de los resultados finales de aquella comparación, queda de manifiesto que Linux está dejando de ser el sistema operativo utilizado únicamente por expertos en informática. De hecho, la distribución\(^1\) evaluada busca facilitar el acercamiento del usuario final, el cual no está dispuesto a dedicar mucho tiempo a la solución de tareas triviales.

\(^1\)Existen diversas compañías o agrupaciones, las cuales empaquetan el kernel Linux con distintos tipos de programas, para así ser distribuido a los usuarios. Dicho conjunto de programas, presentando pequeñas diferencias en torno a la gestión del sistema, recibe el nombre de distribución.
Aportando al mismo objetivo de las distribuciones orientadas al escritorio, cierto tipo de páginas web se dedica a reunir información de interés para usuarios que desean comenzar a utilizar opciones gratuitas, frente a los programas de pago que les presiona a utilizar la popularidad del sistema operativo Windows.

Dichas herramientas informáticas, conocidas generalmente como alternativas libres, si bien no se restringen al sistema operativo Linux, en muy pocos casos lo excluyen.

2.2. Alternativas gratuitas

Sería imposible mencionar todas las aplicaciones gratuitas que están disponibles para el sistema operativo Linux. Es por eso que en esta sección se mencionarán, únicamente, algunas de las alternativas disponibles en las áreas que competen a un estudiante de Ingeniería Eléctrica y Electrónica.

2.2.1. Oficina

Al momento de escribir reportes que acompañen trabajos prácticos o de laboratorio, se dispone del conjunto ofimático: OpenOffice. Este grupo de programas, que comenzó como un proyecto privado y comercial de Sun Microsystems, presenta todos los componentes disponibles en su análoga comercial Microsoft Office; incluyendo: procesador de texto, hoja de cálculo, editor y presentador de diapositivas, editor de ecuaciones, editor de gráficos y base de datos.

Aunque no perfecta, la compatibilidad con la contra parte comercial es casi total, permitiendo la lectura y escritura de los archivos generados por cada uno de los módulos que componen Microsoft Office.

2.2.2. Matemática

En el estudio de la ingeniería, existen dos importantes áreas de las matemáticas que demandan el uso del computador para agilizar su aplicación: el cálculo numérico y el cálculo simbólico.

---


\(^{3}\)http://www.openoffice.org/
CAPÍTULO 2. LINUX Y LA ELECTRÓNICA

2.2.2.1. Cálculo numérico

Para agilizar el cálculo numérico, el estudiante necesita de un programa estilo Matlab, y las alternativas presentes son: Scilab\(^4\) y Octave\(^5\). En los dos casos, la interacción con el usuario se efectúa de manera similar a Matlab, a través de sentencias de línea de comandos, y utilizando prácticamente el mismo grupo de instrucciones.

Cabe mencionar que Scilab dispone de SciCos, un módulo estilo Simulink de Matlab, que podría determinar una ventaja con respecto a Octave. Sin embargo, ambos programas buscan ambiciosamente, y para beneficio de los usuarios insertos en entornos comerciales, la compatibilidad con Matlab; apuntando principalmente, a la ejecución de programas escritos para esta aplicación.

2.2.2.2. Cálculo simbólico

Para agilizar el cálculo simbólico, el estudiante necesita de un programa estilo Mathematica, y las alternativas presentes son: Maxima\(^6\) y Mupad\(^7\). Nuevamente, la interacción con el usuario se efectúa a través de un intérprete de comandos, pero se debe tomar en cuenta que, en esta ocasión, los lenguajes utilizados por las alternativas gratuitas difieren entre ellos y del lenguaje implementado en su análogo comercial.

Sin embargo, Maxima es la evolución de uno de los primeros programas creados para el cálculo simbólico: Macsyma; del cual también derivan algunos programas comerciales, entre ellos Mathematica. Por esta razón, los lenguajes implementados en Maxima y Mathematica presentan ciertas similitudes, lo que podría ser un punto a favor de esta alternativa libre.

Lamentablemente, Mupad ha descontinuado la entrega de licencias gratuitas para estudiantes, y en estos momentos se trata de un programa de pago más. No obstante, algunas versiones antiguas pueden continuar siendo utilizadas de forma gratuita.

\(^4\)http://www.scilab.org/
\(^5\)http://www.octave.org/
\(^6\)http://maxima.sourceforge.net/
\(^7\)http://www.mupad.de/
2.2.3. Electrónica

En el mundo de la electrónica, Linux puede ser utilizado de dos formas: como sistema operativo incorporado en un dispositivo electrónico (embedded), o como plataforma para la ejecución de tareas relacionadas con la automatización del diseño electrónico: EDA\(^8\).

Considerando los alcances del proyecto, en esta sección se analizarán las aplicaciones EDA para Linux, sin considerar el grupo de programas con funcionalidades similares a SpiceX: a los cuales se dedicará una sección en detalle.

2.2.3.1. Grupos EDA

Frente a la preponderante necesidad de contar con un conjunto completo de herramientas para realizar tareas de EDA en Linux, se han formado varios grupos de programadores que buscan proveer la tan solicitada suite informática. De entre estos grupos se pueden nombrar: gEDA\(^9\) y Open Circuit Design\(^10\).

La diferencia entre estos grupos está determinada por el conjunto de programas que se han seleccionado, para suplir las diferentes necesidades que aparecen durante los procesos de diseño, prueba, e implementación de circuitos electrónicos.

Lamentablemente, más allá de reunir los paquetes, empaquetarlos bajo un único nombre, y promocionarlos de manera insuficiente, el trabajo de desarrollo de cada uno de los componentes de software sigue siendo una tarea, prácticamente, individual.

2.2.3.2. Captura esquemática

Los programas de captura esquemática son aplicaciones creadas para esbozar circuitos electrónicos y exportar los esquemas como archivos netlist; los cuales satisfacen la sintaxis de algún núcleo de simulación en particular.

---

\(^8\)EDA: Electronic Design Automation.
\(^9\)http://www.geda.seul.org/
\(^10\)http://opencircuitdesign.com/
Dentro de este grupo de programas, que delegan al usuario la tarea de enlazar los archivos *netlist* generados con el núcleo de simulación, se encuentran: *XCircuit*\(^\text{11}\), y *gschem*\(^\text{12}\); pertenecientes a los grupos *Open Circuit Design* y *gEDA*, respectivamente.

La principal característica de ambos programas, es que pueden generar dibujos de circuitos de gran calidad, debido a la tecnología utilizada en su implementación: el lenguaje *Postscript* para la descripción de diagramas.

Sin embargo, se aprecia una ligera diferencia entre estas dos aplicaciones, lo cual podría ayudar a discriminar entre una y otra. Por una parte, el autor de *gschem* reconoce que *XCircuit* produce mejores salidas gráficas; y por otra, *gschem* está más orientado al diseño de circuitos que a dibujos de calidad en sí, lo cual se aprecia en su funcionalidad.

Lamentablemente, la desventaja de programas de este tipo, es el hecho de tener que enlazar explícitamente el núcleo de simulación; lo cual incrementa la curva de aprendizaje de usuarios provenientes de ambientes más amigables.

### 2.2.3.3. Diseño de placas impresas

Al momento de diseñar las placas impresas que conectarán eléctricamente los circuitos diseñados, se puede recurrir a programas como: *PCB*\(^\text{13}\) o *MUCS-PCB*\(^\text{14}\). *PCB* forma parte de ambos grupos *EDA*: *gEDA* y *Open Circuit Design*; mientras que *MUCS-PCB* es soportado únicamente por *gEDA*.

El principal objetivo de ambos programas es la creación de *layouts*, que son esquemas de rutas que señalizarán la trayectoria a seguir por el cobre en un sustrato. Considerando esto, la principal funcionalidad provista es la automatización en la creación de las rutas, dejando unos pocos segmentos del circuito a ser detallados de forma manual por el diseñador.

Los archivos generados por estas herramientas pueden especificar las dimensiones de la placa y los lugares donde se taladrarán los hoyos, entre otras características. Además, es importante destacar que las placas diseñadas pueden ser exportadas como archivos *Gerber*, el cual es el formato estándar utilizado por los fabricantes de placas impresas.

\(^\text{11}\)http://opencircuitdesign.com/xcircuit/index.html  
\(^\text{13}\)http://pcb.sourceforge.net/  
\(^\text{14}\)http://www.cs.manchester.ac.uk/apt/projects/tools/mucs-pcb/
La gran diferencia entre ambos programas radica en la interfaz de usuario, siendo gráfica en el caso de PCB, y de línea de comandos en el caso de MUCS-PCB. Esto último determina una ventaja para PCB, al momento de ser utilizado por usuarios provenientes de ambientes gráficos.

Otra ventaja a favor de PCB es la posibilidad de importar un tipo especial de archivos netlist, los cuales difieren ligeramente de los archivos netlist generados por los programas de captura esquemática, para las herramientas de simulación. De esta forma, no es difícil encontrar aplicaciones de captura esquemática con la habilidad de exportar esquemas de circuitos como archivos netlist para PCB: XCircuit es un ejemplo.

2.2.3.4. Simulación electrónica

Una etapa fundamental en el diseño de un circuito electrónico es la prueba del funcionamiento. En los años previos a la aparición de los primeros programas de simulación electrónica, los diseñadores tenían que lidiar con lentos y costosos procesos de prueba, en los cuales se utilizaba directamente un prototipo del hardware a producir[1].

De entre los programas para realizar simulación electrónica en Linux, se pueden mencionar: Spice15, Xspice16, y Ngspice17.

Spice es un simulador de circuitos de propósitos generales, cuyas siglas significan Simulation Program with Integrated Circuit Emphasis. Esta aplicación fue uno de los primeros programas de simulación electrónica disponibles en el mercado, lo que la llevó a convertirse en un estándar de la industria en los años ochenta.

Entre las capacidades de Spice se encuentran los análisis: DC, de pequeña señal AC, transiente, de polos y ceros, de distorsión de pequeña señal, de sensibilidad, y de ruido. Además, los circuitos pueden contener resistencias, condensadores, inductores, inductores mutuos, fuentes independientes de voltaje y corriente, cuatro tipos de fuentes dependientes, líneas de transmisión con pérdida y sin pérdida (dos implementaciones separadas), interruptores, líneas RC uniformemente distribuidas,

15http://bwrc.eecs.berkeley.edu/Classes/IcBook/SPICE/
16http://users.ece.gatech.edu/ mrichard/Xspice/
17http://ngspice.sourceforge.net/
y cinco de los más comunes dispositivos semiconductores: diodos, BJT, JFET, MESFET, y MOSFET[12].

Sin embargo, se debe tener en mente las limitaciones de Spice, como son: el grado de precisión de las simulaciones de circuitos depende de los comportamientos modelados en los componentes; las simulaciones están libres de ruido e interferencia, a no ser que se modelen explícitamente; y, finalmente, el simulador no predice el fallo de los componentes, aunque se excedan los parámetros de tolerancia máxima[5].

Continuando, Xspice es una extensión de Spice que provee la habilidad de usar técnicas de modelado por código para la adición de nuevos modelos. La librería de modelos de Xspice contiene más de cuarenta nuevos bloques funcionales, incluyendo: sumadores, multiplicadores, integradores, modelos magnéticos, limitadores, funciones de transferencia en el dominio S, compuertas digitales, elementos de almacenamiento digital, y una máquina de estados digitales generalizada.

Finalmente, Ngspice es una mezcla entre Spice, Xspice, y Cider. Esta última aplicación mejora la precisión de las simulaciones de dispositivos críticos, soportando la descripción de estos últimos en términos de estructuras y materiales.

En todos los casos, la interacción con el usuario se realiza a través de archivos de descripción de circuitos: netlist; los cuales incluyen una descripción textual de los circuitos a simular, las instrucciones que indican el tipo de simulación a realizar, y los parámetros necesarios para efectuar los análisis.

2.3. Captura esquemática y simulación electrónica


Bajo el lema de “diseño desde el principio a fin”, Microsim elaboró DesignLab; el primer sistema integrado para EDA, uniendo la captura esquemática con la simulación electrónica, entre otras etapas del diseño de circuitos.

18PCs: Personal Computers.
En esta sección se nombrarán las ventajas y desventajas de los programas disponibles para realizar captura esquemática y simulación electrónica de forma integrada en Linux; siendo ésta, la misma tarea que busca asistir SpiceX.

2.3.1. **Oregano**

*Oregano* ha pasado por dos grandes ciclos de desarrollo durante su existencia. El primer período transcurrió desde su creación el año 1999 hasta su abandono el año 2001; y el segundo periodo comenzó el año 2003, cuando el grupo LUGFI retomó su desarrollo hasta el día de hoy.

Esta aplicación tiene características que se acercan bastante a los alcances propuestos por *SpiceX*: es sencillo de utilizar, presenta una obvia similitud con la interfaz de *Pspice-Schematics*, y dispone de una buena cantidad de componentes en sus librerías. Además, esta diseñado para trabajar con cualquiera de los siguientes núcleos de simulación derivados de *Spice*: *Ngspice* o *Gnucap*.

Su principal desventaja, sin embargo, es el lenguaje de programación utilizado en su implementación: C. Esta característica hace difíciles y tediosas las modificaciones del código fuente, obstaculizando el impulso grupal del proyecto.

El problema a llegado a tal punto que, actualmente, su desarrollo se encuentra detenido. Los programadores del grupo LUGFI están evaluando el rendimiento de segmentos de código en *Python* y C#, para así decidir el posible lenguaje a utilizar en futuras versiones del programa. Los resultados parciales indican una ventaja para el segundo lenguaje evaluado[10].

Lamentablemente, el abandono del proyecto es otra de las posibilidades a seguir por el grupo de programadores, debido, principalmente, a la carencia de una metodología de diseño en la génesis del programa[10].

---


20 Grupo de Usuarios de GNU/Linux de la Facultad de Ingeniería de la Universidad de Buenos Aires.

21 http://www.gnu.org/software/gnucap/
2.3.2. **Qucs**

**Qucs**\(^\text{22}\) es otro intento por crear un entorno integrado de trabajo para la esquematización y simulación de circuitos electrónicos en Linux. Su desarrollo se encuentra en etapas iniciales, debido a los ambiciosos objetivos propuestos por su grupo de desarrolladores.

A diferencia de **SpiceX**, **Qucs** no busca ser una interfaz de captura esquemática para ser utilizado con uno de los núcleos de simulación actualmente disponibles; éste busca la implementación de un núcleo propio de simulación, con toda la complejidad que esto implica. A futuro, **Qucs** podría ser una opción a considerar por electrónicos ya acostumbrados al sistema operativo Linux.

En la actualidad, la aplicación presenta las siguientes funcionalidades: análisis AC, análisis DC, análisis transiente, análisis de ruido AC, entre otras. Además, ya es posible efectuar algunas simulaciones mixtas, entre componentes análogos y digitales. Todo lo anterior de forma gráfica y asistida.

Finalmente, vale decir que el código de **Qucs** está implementado en el lenguaje de programación C++, con todas las ventajas que esto acarrea desde el punto de vista de evolución del software.

2.3.3. **KTechlab**

**KTechlab**\(^\text{23}\) es un nuevo intento por crear un entorno integrado de trabajo para la esquematización y simulación de circuitos electrónicos en Linux. Su desarrollo se encuentra en etapas iniciales, debido a los objetivos propuestos por su grupo de desarrolladores: los cuales son incluso más ambiciosos que los de **Qucs**.

Esta aplicación, está orientada a la simulación análoga y digital mixta, con especial énfasis en la inclusión de dispositivos microcontroladores **PICs**\(^\text{24}\) en los circuitos. Además se pretende interactuar con dispositivos de hardware directamente conectados al computador mediante los puertos serie y paralelo.

\(^{22}\)http://qucs.sourceforge.net/index.html

\(^{23}\)http://ktechlab.org/

\(^{24}\)http://www.microchip.com/
El tipo de simulación realizada por este programa se aleja de los profundos análisis efectuados por las aplicaciones de simulación electrónica previamente vistas. El análisis así realizado se conoce como: simulación de tiempo real. Dicha simulación podría compararse con el tipo de simulación conseguida mediante otro de los programas pioneros en la simulación electrónica: *Electronic Workbench*.25

Actualmente, la aplicación presenta las siguientes características: simulación electrónica mixta, incluyendo la simulación de PICs; disponibilidad de un osciloscopio para la visualización de señales eléctricas; simulación lógica de alta velocidad; interacción con hardware a través de los puertos serie y paralelo; y la programación de microcontroladores PIC, mediante los lenguajes *FlowCode*26 o *Microbe*.

---

25http://www.electronicsworkbench.com/
26Diagrama de flujos.
Capítulo 3

Orientación a Objetos

El primero de los alcances del proyecto enfatiza que SpiceX debe disponer de un código de fácil desarrollo y mantención. Se pretende que la comunidad electrónica, con conocimientos de programación, pueda continuar su desarrollo; convirtiéndolo en la utilidad que pretende ser, más allá del resultado parcial elaborado con fines académicos.

3.1. Principios

Un programa es un conjunto de instrucciones, escritas en un lenguaje de programación, destinadas a satisfacer distintas necesidades de usuarios. Cuando se implementa una aplicación en particular, se relaciona el entorno donde se presenta el problema y el computador, que es la herramienta ejecutora de las acciones que proveen la solución.

El entorno que agrupa todos los objetos o entidades\(^1\), que son administrados, monitoreados o controlados por un sistema, recibe el nombre de dominio del problema. De igual forma, el entorno donde se modela el problema, por ejemplo un computador, recibe el nombre de dominio de la solución[11].

El paradigma orientado a objetos es un enfoque de análisis, diseño, e implementación de una aplicación, que provee una manera distinta de relacionar el dominio del problema con el dominio de la solución, produciendo las siguientes ventajas:

\(^1\)Colectividad considerada como unidad. Lo que constituye la esencia o la forma de una cosa.
facilidad de mantención en las aplicaciones,

- expansibilidad del código implementado,

- simplificación del proceso de desarrollo;

lo cual concuerda con el primer alcance preciso para SpiceX.

3.1.1. Abstracción

Las capacidades de los lenguajes de programación imperativos, paradigma que precede a la orientación a objetos, inducen a relacionar el problema que se quiere solucionar con la solución en sí, pensando en términos de instrucciones computacionales. El esfuerzo requerido para realizar esta asociación, y el hecho de que esto es extrínseco al lenguaje de programación, produce programas difíciles de escribir y mantener[6].

Por otra parte, los lenguajes orientados a objetos proveen herramientas para representar objetos del problema que se está atacando, mediante objetos de software. Es decir, se describe el problema a solucionar en términos del problema, y no en términos de instrucciones computacionales.

La asociación necesaria entre los dominios del problema y la solución, recibe el nombre de abstracción. La abstracción producida mediante lenguajes orientados a objetos es más flexible y poderosa que la producida mediante los lenguajes precedentes, debido a que cuando se lee el código describiendo la solución, se están leyendo palabras que también expresan el problema. Esto último se traduce en facilidad de mantención y expansión del software.

3.1.1.1. Abstracción de un objeto

Si se observa por algunos segundos un entorno cualquiera, se puede identificar la vasta cantidad de objetos que lo componen. Una lámpara, una mesa, una cama, dos espejos, una puerta, etc. Pensando en modelar objetos de software, se puede enfocar la atención en solo un objeto, por ejemplo: la lámpara. Una posible descripción de éste objeto, basada en la figura 3.1, puede ser:
La lámpara es de color azul, tiene 30 cm de altura, y puede ser encendida o apagada posicionando el interruptor en on u off, respectivamente.

Esta descripción se basa en las características (datos), y comportamientos (funcionalidades) del objeto lámpara. Si bien los lenguajes imperativos proveen herramientas para describir grupos de características (variables miembro) en una sola entidad: la estructura; los lenguajes orientados a objetos añaden la capacidad de incluir comportamientos (funciones miembro) a esas estructuras. Este nuevo tipo de entidad de software recibe el nombre de clase.

La figura 3.2 muestra la abstracción computacional del objeto lámpara. Esta abstracción no profundiza en otras características de la lámpara como son: voltaje, peso, volumen, etc. La selección de características y comportamientos a ser modelados recibe el nombre de nivel de abstracción.

El nivel de abstracción requerido, para relacionar objetos del dominio del problema con objetos del dominio de la solución, dependerá de la aplicación en particular. Si se quiere modelar una lámpara para un sistema de ventas en una tienda comercial, puede ser necesario abstraer sólo el modelo y el precio; en cambio, si se quiere modelar la lámpara en el contexto de una aplicación para administrar cargas en una compañía de envío de paquetes, puede resultar necesario modelar, además, el peso y volumen de la lámpara.

Figura 3.1: Lámpara real. Figura 3.2: Representación en software.
3.1.1.2. Clases de objetos

La lámpara utilizada para realizar la abstracción fue comprada en una tienda de artículos para el hogar. Es así que existen cientos de lámparas con el mismo modelo en el mercado, y ésta viene a ser una más entre tantas. Se puede decir que la abstracción realizada al objeto lámpara fue de la clase o modelo de ésta, y no del elemento en particular; ya que no se especifican las características propias del objeto en observación: color azul, altura igual a 30 cm, etc.

La ventaja de esto último es que se pueden crear tantas lámparas como se desee en nuestro programa, con el mismo patrón de características y comportamientos, y utilizando una misma porción de código. Así como existen los tipos de variables predefinidos (por ejemplo: int), ahora se pueden crear clases de objetos (por ejemplo: Lampara). Cuando se crea un objeto de la clase lámpara con características específicas, así como se crean variables int que almacenan números enteros específicos, se dice que se crea una instancia de la clase. La instancia de una clase es lo que se conoce como objeto.

3.1.1.3. Composición de clases

El tratamiento que los lenguajes orientados a objetos dan a estas instancias de clases es similar al de las variables normales; posibilitando la creación de arrays de objetos, objetos como parámetros de funciones o procedimientos, etcétera. Incluso se pueden utilizar objetos de una clase escrita previamente, en la composición de nuevas clases. Estos objetos reciben el nombre de objetos miembros de una clase.

Esta capacidad es un paso más en la optimización de la abstracción del dominio del problema, ya que, generalmente, los objetos presentan esta característica. La lámpara esta compuesta por distintos objetos: ampolleta, enchufe, cable, pedestal, pantalla, etcétera. El nombre de esta característica es composición o agregación, y determina un tipo de relación “tiene-un”, como en la frase: “una lámpara tiene una ampolleta”. La diferencia entre los términos composición y agregación esta determinada solo por la utilidad del objeto miembro cuando no pertenece a la clase.
Si la clase del objeto miembro fue diseñada solo para pertenecer a una clase, y no tiene utilidad alguna trabajando de forma aislada, se dice que la característica es una agregación. Un ejemplo de esto es el objeto cuello, el cual no tiene utilidad al no pertenecer a la clase camisa. Por el contrario, si la clase del objeto miembro presenta utilidad propia, se dice que la característica es una composición. Un ejemplo de esto es el objeto trabajador, el cual tiene utilidad propia, independiente si pertenece o no a la clase compañía.

En este punto se justifica que la orientación a objetos produzca programas más fáciles de entender, debido a la abstracción mediante objetos; más compactos, debido a la reutilización del código que proveen las clases; y, finalmente, más fáciles de programar, debido a la posibilidad de descomponer el problema en objetos sencillos de entender que luego pueden componer clases más complejas.

3.1.2. Herramientas

Las herramientas que diferencian los lenguajes orientados a objetos de los lenguajes imperativos, buscan proveer medios realistas de representación para los objetos que componen el entorno habitual de las personas, lugar donde se presentan los problemas a solucionar.

3.1.2.1. Encapsulación

Cuando se utiliza una lámpara, generalmente, no es necesario conocer en detalle el mecanismo interno que gobierna el funcionamiento del objeto. Desde el punto de vista del usuario de la lámpara, solo es necesario saber como conseguir o prescindir de iluminación: interruptor en On o interruptor en Off, respectivamente.

Los lenguajes orientados a objetos proveen un mecanismo para representar esta simplicidad en el manejo de la información, al momento de trabajar con objetos: la encapsulación. Así como en el dominio del problema existen usuarios y creadores de objetos, en el dominio de la solución existen programadores clientes y creadores de clases.

2Se utiliza la palabra creadores y no fabricantes, debido a la existencia de objetos que no son producto de procesos de fabricación. Personas, animales, etc.
La encapsulación separa la interfaz de la implementación de una clase. La interfaz es la información accesible por el programador cliente, necesaria para conocer la funcionalidad que presenta el objeto. Por otra parte, la implementación es el conjunto de instrucciones que determinan las acciones a realizar, para entregar dicha funcionalidad.

![Diagrama de una clase]

Figura 3.3: Interfaz del objeto de software Lámpara.

Esta característica adiciona dos ventajas más al simple hecho de mejorar la abstracción del mundo real. La primera es la de controlar el acceso de los programadores clientes a la información crucial que mantiene la coherencia del objeto. Los clientes solo pueden obtener o modificar características de un objeto: las variables miembro; mediante procedimientos bien definidos previamente por el creador de la clase: las funciones miembro. Para lograr este control se hace uso de los especificadores de acceso: `private`, `protected`, `public`, al momento de programar una clase.

Los miembros, variables o funciones que conforman una clase, son los bloques de código que pueden ser marcados con alguno de los modificadores de acceso. Si el miembro es una variable, por ejemplo la altura, se puede controlar el acceso a la información; por otra parte, si el miembro es una función, por ejemplo la función `encender`, se puede controlar el acceso a la funcionalidad.

Si el miembro está marcado como `private`, el programador cliente no tiene acceso directo a éste. Si el miembro está marcado como `protected`, sólo los programadores que utilicen la clase como base para programar otras clases, podrán acceder directamente a éste. Si el miembro está marcado como `public`, todos los programadores clientes podrán acceder directamente a éste.

---

3Las clases bases se utilizan mediante la herencia, mecanismo explicado en la siguiente sección.
Finalmente, la segunda ventaja extra de la encapsulación, es la de permitir la modificación u optimización de la implementación, manteniendo intacta la interfaz. De esta forma, las aplicaciones de los programadores clientes no se verán afectadas, por mejoras en cualquiera de los algoritmos que conforman la funcionalidad del objeto.

3.1.2.2. Herencia

Hasta ahora, se sabe que a través de las clases de objetos se puede reutilizar código previamente escrito; aprovechando la similitud en las características y funcionalidades de los distintos elementos que componen el mundo real. Pero, ¿qué sucede si se quiere modelar un objeto ligeramente distinto, es decir, con alguna característica y/o funcionalidad extra?.

Para reutilizar código y expandir características y funcionalidades de un objeto, los lenguajes orientados a objetos disponen de la herencia. Siguiendo con el ejemplo de la lámpara, se puede decir que ésta hereda las características y funcionalidades de un electrodoméstico. Éste último se puede enchufar, desenchufar, presenta un voltaje de operación, etcétera. Es así que se podría haber creado una clase electrodoméstico, agrupando todas las características y comportamientos comunes de este tipo de elementos, y después heredar los miembros en futuras clases especializadas, como la lámpara.

La clase de la cual se heredan las características y funcionalidades recibe el nombre de: clase base, super clase, o clase padre. Asimismo, la clase que hereda las características o funcionalidades recibe el nombre de: clase derivada, clase heredada, subclase, o clase hija.

La herencia posibilita no sólo la adición de nuevos miembros a las clases, reutilizando el código de las clases bases, sino que también permite la modificación de miembros ya existentes. Si se quiere cambiar alguna característica o comportamiento de un objeto derivado, solo hace falta sobreescribir ⁴ el miembro, en la implementación de la nueva clase.

⁴Por ejemplo, en algunos lenguajes de programación esto se logra mediante la utilización de las palabras claves `override` o `new` (C++ o C#, respectivamente).
La relación que la herencia determina entre la clase base y la derivada recibe el nombre de relación “es-un”, en el caso de mantener intacta la interfaz de la clase derivada; y el nombre de relación “es-como-un”, en el caso de adicionar nuevos miembros a la interfaz de la clase derivada. Obviamente, se asume que en el caso de presentarse una relación “es-un”, se sobreescribe algún miembro. De no ser así la aplicación de la herencia no tendría sentido.

Al igual que la encapsulación, esta herramienta de los lenguajes orientados a objetos también adiciona dos ventajas extras a la inherente mejora en la abstracción de la realidad. La primera de éstas es la obvia reutilización de código por parte de los creadores de clases, los cuales implementan nuevas clases a través de la herencia; y la segunda es que posibilita la aplicación del polimorfismo, herramienta de la orientación a objetos explicada en la siguiente sección.

Generalmente se utiliza la frase reutilización de la implementación, haciendo referencia a la ventaja de implementar clases con objetos miembros, además de la creación de objetos a partir de clases genéricas; y la frase reutilización de la interfaz, para hacer referencia a la ventaja de heredar las características y funcionalidades de un objeto.

3.1.2.3. Polimorfismo

Previamente, se dijo que la lámpara es un electrodoméstico. Una de las funcionalidades comunes en cada uno de estos elementos es que se pueden prender o apagar, y aunque no todos los electrodomésticos operan de la misma forma, se tiene una idea general de como proceder para lograr estas funcionalidades.

Generalmente, no es necesario describir a los usuarios de una sala de computadores: “Cuando se abandone la sala, apagar: el computador presionando el botón rojo en la torre, el monitor presionando el botón blanco, los parlantes girando la perilla en contra sentido de reloj, y la luz de la habitación posicionando el interruptor en off”.

Un instructivo de uso más realista podría ser: “Cuando se abandone la sala, dejar todo apagado”. 
CAPÍTULO 3. ORIENTACIÓN A OBJETOS

Los lenguajes orientados a objetos disponen del polimorfismo para abstraer esta versatilidad de acción en las personas, con la diferencia de que en el dominio de la solución, las acciones son efectuadas por el computador. Esto último permite escribir código (el instructivo de operación en el mundo real) independiente de tipos o clases específicas (cada uno de los equipos electrónicos en la sala de computadores).

Pensando en términos del dominio de la solución, se podría crear un array de objetos tipo electrodoméstico, y después aplicar la función apagar a cada uno de los objetos que componen el vector. Lo increíble en esta funcionalidad, es que los objetos que conforman el array podrían pertenecer a diferentes clases derivadas de la clase base electrodoméstico, con acciones específicas en la implementación del procedimiento de apagado.

Para lograr esto, al momento de implementar una clase base, que se piensa presentará una función genérica pero diferente en cada una de las clases derivadas, se debe marcar dicha función miembro como virtual. Luego, al aplicar herencia para crear las clases derivadas, se debe implementar las funciones virtuales con acciones específicas que se adapten a cada objeto.

La diferencia entre esta forma de implementación y la simple sobre-escritura de funciones, además de la posibilidad de aplicar polimorfismo, radica en la determinación de la función a ejecutar. Mientras que en la simple sobre-escritura de funciones se decide en tiempo de compilación, en la sobre-escritura de funciones virtuales, se decide en tiempo de ejecución. Por lo tanto, al momento de implementar polimorfismo, se debe considerar el costo en velocidad que determina la aplicación de este mecanismo.

La principal ventaja del polimorfismo es que: así como se podría adicionar un nuevo electrodoméstico a la sala de computadores, sin necesidad de modificar el instructivo de uso; se podría adicionar una nueva clase derivada al array de electrodomésticos, sin modificar las funciones que operan con la clase base del vector original. Es decir, se facilita la evolución del software.
3.2. Metodología

La metodología a seguir será aplicada dentro de las distintas disciplinas que conforman el proceso de desarrollo, es por esto que conviene tener claro los objetivos de cada una de éstas. Primero que nada, se conoce por análisis a la investigación enfocada al problema y sus requerimientos. Para seguir, el término diseño enfatiza una solución conceptual que satisface dichos requerimientos. Y finalmente, la programación es la implementación, mediante algún lenguaje de programación, del diseño previamente creado.

3.2.1. Enfoque

El programador que pretenda sacar provecho a los lenguajes orientados a objetos, deberá comenzar por cambiar el enfoque de análisis, diseño, e implementación de las aplicaciones a desarrollar. De no ser así, los programas implementados carecen de ventajas frente a los desarrollados mediante lenguajes imperativos.

A continuación se presentan cinco consejos de enfoque para aplicar la orientación a objetos:

1. Cada cosa es un objeto: En teoría, se puede tomar cualquier componente conceptual del problema y representarlo como un objeto en el programa.

2. Un programa es un manojo de objetos intercambiando instrucciones, mediante el envío de mensajes: Para solicitar alguna funcionalidad a un objeto, se debe de enviar un mensaje a éste.

3. Cada objeto está compuesto por otros objetos: De esta forma, se puede implementar la complejidad de un programa, escondida detrás de la simplicidad de los objetos.

---

5 Se dice que en el análisis se construye un modelo del dominio del problema, mientras que en el diseño se construye un modelo del dominio de la solución.
6 Generalmente, la programación recibe el nombre de implementación.
7 La carencia de ventajas es especialmente cierta cuando se trabaja con lenguajes orientados a objetos que permiten, además, la programación imperativa. C++ es un ejemplo.
CAPÍTULO 3. ORIENTACIÓN A OBJETOS

4. Cada objeto pertenece a una clase: La principal característica que distingue a una clase es: ¿Qué mensajes se pueden enviar a ésta?.

5. Todos los objetos de una clase en particular pueden recibir los mismos mensajes: La capacidad de reemplazar los objetos es uno de los conceptos más poderosos de la orientación a objetos.

3.2.2. Procesos de desarrollo

Para comenzar el desarrollo de cualquier programa informático, se debe elegir una metodología de trabajo que se adecue a los alcances del proyecto. Informalmente, un proceso de desarrollo de software es una metodología para la construcción, entrega\(^8\), y posible mantención del software\(^9\).

Cuando se programa en lenguajes imperativos, es difícil efectuar cambios en el código una vez que se ha comenzado la implementación. Es por esto que se requiere de metodologías detallistas, al momento de analizar y diseñar aplicaciones complejas. El problema surge cuando se presenta una parálisis del análisis, debido a que hay detalles en la aplicación que no serán observables sino hasta el diseño, la implementación, e incluso la prueba del programa\(^9\).

Por el contrario, los lenguajes orientados a objetos facilitan la comprensión, reutilización, y expansión del código, simplificando los procesos de desarrollo requeridos. Las iteraciones, pequeños ciclos de análisis, diseño, implementación, y prueba de módulos de la aplicación final, conforman, hoy en día, el mecanismo de trabajo más recomendado. Los metodologías basadas en iteraciones, reciben el nombre de procesos de desarrollo iterativos.

Gracias a la orientación a objetos, al trabajar con iteraciones se produce información relevante y código utilizable en los siguientes ciclos de desarrollo, es decir, en las futuras iteraciones.

\(^{8}\)Entrega de versiones de una aplicación.
3.2.3. Pautas a seguir

La experiencia acumulada por programadores experimentados en orientación a objetos, genera ciertas pautas a considerar al momento de aplicar el paradigma[6].

1. Primero se debe hacer que el código trabaje, luego que sea rápido. Esto es cierto aún si se sabe que una pieza de código es realmente importante y que será uno de los principales cuellos de botella del sistema. Se debería comenzar a trabajar con el diseño más simple que se pueda; luego, si el código no es lo suficientemente rápido, se debería mejorar. Casi siempre se descubrirá que el cuello de botella no es lo importante. Se debe ocupar el tiempo en las cosas realmente importantes.

2. Recordar el principio “divide y conquistarás”. Si el problema que se está tratando de solucionar es demasiado confuso, se debe intentar imaginar cuál es la operación básica que el programa debería llevar a cabo, suponiendo la existencia de una pieza mágica del programa que manejará las partes complicadas. Esa pieza de programa será un objeto. Se debe escribir el código que utilizará ese objeto, y después, al enfocarse en éste, se deben encapsular sus partes complicadas en otros objetos, etcétera.

3. El control de acceso permite modificar la clase tanto como sea posible, sin dañar el código del cliente en el cual la clase está siendo utilizada. Con esto en mente, se debe hacer private cada cosa que sea posible, y hacer sólo la interfaz de la clase public, usando funciones en vez de datos. Los datos deberían ser public solo cuando sea totalmente necesario. Si una clase no necesita acceder a una función, se debería hacer private. Si una parte de la clase será expuesta a la herencia como protected, se debe proveer una interfaz de funciones mejor que exponer los datos. De esta forma, los cambios en la interfaz tendrán un mínimo impacto en las clases derivadas.

4. No caer en la parálisis del análisis. Hay algunas cosas que no se comprenderán hasta que se comience a codificar y/o tener algún tipo de sistema corriendo. Los errores en una clase o un set de clases no destruirán la integridad del sistema completo.
5. El análisis y diseño debe producir, al menos, las clases del sistema, sus interfaces públicas, y sus relaciones con otras clases; especialmente las clases bases. Si la metodología de diseño produce más que esto, se debe cuestionar si todas las piezas producidas por esa metodología tienen valor durante el ciclo de vida del programa. Si no la tienen, la mantención de éstas tendrá un costo para el programador. Los miembros de grupos de desarrollo tienden a no mantener nada que no contribuya a su productividad; es totalmente cierto que muchos métodos de diseño no valen la pena.
Capítulo 4

Rational Unified Process - RUP

El segundo de los alcances del proyecto enfatiza la utilidad de SpiceX al momento de finalizar la memoria. Aunque la aplicación no pretende cubrir cada una de las posibilidades del núcleo de simulación, desde el punto de vista de librerías electrónicas disponibles en Spice, ésta debería ser una interfaz ejecutable, testable, y funcional.

4.1. Principios

En el capítulo 3, se menciona que la orientación a objetos necesita ser presentada en el contexto de un proceso de desarrollo. El Rational Unified Process, ofrece una metodología de desarrollo de software para construir sistemas orientados a objetos, la cual satisface las características de un proceso de desarrollo iterativo y adaptativo1. Entre sus ventajas se encuentran:

- mitigación temprana de riesgos importantes,
- progreso de la aplicación visible desde un comienzo,
- realimentación y adaptación a los requerimientos de usuario, durante todo el proceso de desarrollo,
- simplicidad en la administración del proyecto,
- mejoramiento del proceso de desarrollo en sí, iteración por iteración;

1Adaptativo, va: adj. Perteneciente o relativo a la adaptación o la capacidad de adaptación. Artículo nuevo. Avance de la vigésima tercera edición del diccionario de la Real Academia Española.
lo cual concuerda con los dos primeros alcances precisados para *SpiceX*.

### 4.1.1. Proceso de desarrollo iterativo

El ciclo de desarrollo de una aplicación, es el período de tiempo que transcurre desde que se comienza la planificación del proyecto, hasta que se libera una versión estable o lista para la producción. En el *RUP*, el ciclo de desarrollo está basado en la sucesiva ampliación y refinamiento de un sistema a través de múltiples iteraciones, las cuales establecen sus bases en la realimentación y adaptación cíclica para converger a un sistema que satisfaga al usuario.

![Figura 4.1: Desarrollo Iterativo e Incremental.](image)

### 4.1.2. Administración de los requerimientos

La metodología del *RUP* se enfoca principalmente en afrontar cambios de los requerimientos, los cuales son capacidades y condiciones que conforman un sistema. Esta característica en un proceso de desarrollo se denomina: administración de los requerimientos[9].

Los procesos de desarrollo utilizados por los lenguajes imperativos, tratan de obtener los requerimientos finales de usuario cuando se comienza la planificación del sistema. Es por esto que se comienza con un largo período de análisis de requerimientos, lo cual acarrea dos inconvenientes:

- el riesgo de caer en la parálisis del análisis, y
■ la inflexibilidad frente a los cambios en los requerimientos de usuario, una vez comenzado el desarrollo.

Por el contrario, el *RUP* provee una muestra de lo que se está haciendo al usuario, para ver si coincide con lo que realmente desea. El proceso de desarrollo está preparado para enfrentar los cambios en las exigencias del usuario, debido a:

■ las facilidades de mantención y expansión del software que provee el paradigma orientado a objetos, y

■ al mecanismo de trabajo basado en iteraciones.

Esto no quiere decir que la dirección a la cual apunta el proyecto cambiará de forma continua durante todo el desarrollo. Sólo se tiene conciencia de la posibilidad de haber malentendido los requerimientos iniciales del usuario, y del cambio en las exigencias, una vez iniciado el desarrollo del sistema\(^2\). Al final del ciclo de desarrollo, la aplicación debería satisfacer completamente al usuario.

En el caso de haberlo, la diferencia entre lo que tiene en mente el usuario y lo que se está desarrollando, debería ser más grande en un comienzo del ciclo de desarrollo; período en el cual se obtienen las primeras realimentaciones y se realizan las primeras adaptaciones. Esta característica determina una temprana mitigación de riesgos importantes.

Además, no es necesario sobre dimensionar el análisis de los requerimientos, ya que las iteraciones proveen la flexibilidad para comenzar un desarrollo confiado y abierto a las correcciones. Esto disminuye enormemente los gastos en la administración del proyecto, la cual se encarga, principalmente, de realizar exhaustivos procesos de visualización de futuros requerimientos. En el *RUP* los requerimientos futuros no sólo se visualizan, sino que también se acogen.

---

\(^2\)Una vez que el usuario tiene las acceso a los adelantos del sistema final producidos por las iteraciones, podría, fácilmente, cambiar o adicionar exigencias que no visualizó en un comienzo.
4.1.3. **Estructura del RUP**

Para el *RUP*, un ciclo de desarrollo consta de cuatro fases: Concepción, Elaboración, Construcción y Transición; cada una compuesta por una serie de iteraciones. A la vez, en cada iteración se ejecutan disciplinas o *work-flows*, las cuales utilizan varios artefactos para lograr sus objetivos. Aunque no se conozca el significado y objetivo de cada uno de los términos aquí mencionados, se presentan de forma adelantada para estructurar el proceso, como se muestra en la figura 4.2.

4.2. **Metodología**

4.2.1. **Fases**

Las fases que componen el *RUP* se diferencian entre sí, básicamente, por la importancia de los requerimientos abordados. A continuación se presenta la descripción de cada una de las etapas que componen un ciclo de desarrollo:

4.2.1.1. **Concepción**

En esta fase se establece un punto vista común de los objetivos del proyecto, se determina si es factible de realizar, y se decide si merece una investigación más seria en la fase de elaboración. Se dice que en esta fase se mitigan los riesgos de negocios de la aplicación; aquellos relacionados con la factibilidad económica del proyecto.
CAPÍTULO 4. RATIONAL UNIFIED PROCESS - RUP

Si se ha decidido con anterioridad que el proyecto será realizado, y es claramente factible, entonces la fase de concepción podría ser sumamente corta³. En este caso, podría incluir las primeras actividades de análisis de requerimientos, y los planes a seguir durante la primera iteración. Después de esto, se debería comenzar de inmediato la fase de elaboración.

Los artefactos que se comiencen a utilizar en concepción se completarán sólo de forma parcial, debido a que serán refinados, una y otra vez, en las siguientes iteraciones.

4.2.1.2. Elaboración

Esta fase está compuesta por la serie inicial de iteraciones, durante las cuales el grupo realiza una investigación seria, implementa (programa y prueba) la arquitectura principal de la aplicación, clarifica la mayoría de los requerimientos, y aborda los asuntos de alto riesgo. Se dice que en esta fase se mitigan los riesgos técnicos del programa.

La elaboración a menudo consiste entre dos a cuatro iteraciones; cada una con una duración recomendada entre dos a seis semanas, al menos que se trate de un grupo de desarrolladores masivo. Además, cada iteración debería terminar siempre con una versión estable y probada de los requerimientos que implemente.

4.2.1.3. Construcción

Es la fase donde se abordan los riesgos menos importantes que quedan en el proyecto, y se prepara la entrega de la aplicación. Durante la construcción, la mayor parte del tiempo se dedica al perfeccionamiento del diseño y de la implementación, y a detallados tests de la aplicación; para así, finalmente, converger a un sistema completo.

Se debe recordar que hasta ahora, muchos casos de uso no han sido implementados, y los que han sido abordados, se han completado solo de forma parcial; lo suficiente para validar algunas hipótesis, o mitigar los riesgos principales. Se han definido subsistemas e implementado interfaces; sin embargo, sólo una pequeña parte del código final ha sido escrito (principales escenarios de casos de uso, flujos alternativos de eventos, y manejo de errores). Es por esto que la mayor parte del trabajo aún no está hecho[8].

³Esta es la situación en la que se encuentra SpiceX.
Los requerimientos implementados en construcción no deberían variar mayormente la arquitectura del sistema; de no ser así, se ha hecho un mal trabajo durante la etapa de elaboración. Debido a que se desarrolla un código de alta calidad, la mayor parte del tiempo se dedica a la fase de construcción; usualmente entre tres a seis iteraciones, de menor duración que las iteraciones de la fase de elaboración.

4.2.1.4. Transición

La fase de construcción termina con una versión beta del sistema, la cual debería ser completamente funcional, e incluir: instalador, documentación de ayuda, y algunos ejemplos de uso del programa. Sin embargo, se sabe que esta versión no es el producto final, y que necesita ser refinada en cuanto a funcionalidad, rendimiento y calidad en general.

Durante la transición normalmente se desarrollan una o dos iteraciones que incluyen: tests del producto, preparándolo así para la primera entrega; y ajustes menores basados en la realimentación de los usuarios. En esta fase también se pueden completar funcionalidades remanentes, que han sido postergadas debido a los plazos fijos de tiempo determinados para las iteraciones.

4.2.2. Iteraciones

En un proceso de desarrollo iterativo, el trabajo es organizado en iteraciones, las cuales son una serie de pequeños mini proyectos de duración fija. El producto de cada una de estas iteraciones es un sistema probado, integrado, y ejecutable. Cada iteración incluye sus propias actividades de análisis, diseño, implementación, y prueba.

El resultado de cada iteración es un sistema ejecutable pero incompleto; éste no está listo para la producción. El sistema puede necesitar varias iteraciones antes de estar listo para la producción; por ejemplo, 10 o 15 iteraciones. Sin embargo, el resultado producido por cada iteración no es descartable, éste se reutilizará, mejorará o ampliará en las siguientes iteraciones. La orientación a objetos posibilita la aplicación de este mecanismo de trabajo.
La duración de cada iteración debería ser de entre dos a seis semanas. Menos tiempo, implicaría producir sistemas carentes de suficiente información para obtener una realimentación de valor; más tiempo, implicaría aumentar los riesgos del proyecto, al no recibir realimentación a tiempo. En el caso de prever que no se alcanzarán a cumplir los objetivos planteados para la iteración en curso, es conveniente posponer el tratamiento de algunos requerimientos para futuras iteraciones. Nunca se debe posponer la fecha de término de la presente iteración.

Aunque, en general, cada iteración aborda nuevos requerimientos e incrementalmente extiende el sistema, una iteración puede ocasionalmente revisar el software existente y mejorararlo.

4.2.3. Disciplinas

Las disciplinas son conjuntos de actividades, y artefactos relacionados con éstas, enfocados a tratar un mismo tema en particular; por ejemplo, todas las actividades enfocadas a analizar los requerimientos del sistema. Todas las pautas de trabajo del RUP, caen dentro de alguna disciplina. Un artefacto es el término general empleado para describir el producto de algún trabajo; por ejemplo, esquemas de bases de datos, documentos de texto, diagramas, modelos, etcétera.

Existe una variada cantidad de disciplinas orientadas a diferentes áreas del desarrollo de un sistema; sin embargo, y basándose en las pautas de desarrollo presentadas en el capítulo 3, sólo se mencionarán las que van a ser utilizadas durante el desarrollo de SpiceX.

4.2.3.1. Administración del Proyecto

Es el conjunto de todas las actividades enfocadas a organizar el trabajo a realizar durante la ejecución del RUP.

No se debe confundir esta disciplina con un calendario estricto de actividades. No es el detalle todos los pasos a seguir durante la completa ejecución del ciclo de desarrollo. Para lograr esto, se deberían conocer por adelantado todos los requerimientos que tienen y tendrán los usuarios, lo cual contradice los principios del RUP.
Más bien se trata de dos clases de actividades agrupadas en una sola disciplina: las que describen los macro-objetivos del proyecto, dejando abiertas las especificaciones de cómo lograr esto; y las que describen los micro-objetivos del proyecto, qué son los pasos a seguir en el corto plazo para implementar requerimientos y funcionalidades. Estas dos clases de actividades se relacionan directamente, ya que los micro-objetivos buscan satisfacer de forma adaptativa e incremental los macro-objetivos.

4.2.3.2. Requerimientos (Análisis)

Es el conjunto de todas las actividades enfocadas al análisis de requerimientos de una aplicación, como son la escritura de casos de usos y la identificación de requerimientos no-funcionales.

El primer objetivo de un trabajo en el área de los requerimientos es encontrar, comunicar, y recordar lo que es realmente necesario, de forma clara para el usuario y el desarrollador.

4.2.3.3. Modelo de Negocio (Análisis)

Es el conjunto de todas las actividades enfocadas a modelar el contexto del negocio, y el alcance del sistema.

4.2.3.4. Diseño

Es el conjunto de todas las actividades enfocadas al diseño del proyecto, incluyendo la totalidad de la arquitectura, objetos, bases de datos, redes, etcétera.

4.2.3.5. Implementación

Es el conjunto de todas las actividades enfocadas a la programación y construcción del sistema, dejando de lado la entrega de la aplicación.

A medida que se avanza en las fases del desarrollo, las disciplinas cambian su nivel de importancia. Por ejemplo, en la fase de elaboración, puede que las disciplinas de requerimientos y diseño sean las más importantes; sin embargo, en la fase
de construcción, cuando los riesgos más importantes ya han sido mitigados y la arquitectura principal no se espera varíe mucho, la disciplina de implementación toma mayor importancia.

4.2.4. Artefactos

Como se mencionó con anterioridad, los artefactos son herramientas que ayudarán a cumplir los objetivos planteados por cada disciplina. Las iteraciones hacen uso reiterado de las distintas disciplinas para el desarrollo de cada mini-sistema; es por esto que sería incorrecto asociar una disciplina, y por ende, determinados artefactos a una fase en particular.

Se debe tener claro que los artefactos son una ayuda para el desarrollo del proyecto, y no una obligación. En la práctica, se debería seleccionar un subconjunto determinado de artefactos a implementar, los cuales aportarán valor al proyecto. Ni siquiera es necesario terminar en su plenitud los artefactos en curso; sólo se deberán madurar hasta un nivel suficiente, el cual permita tomar decisiones acertadas en base a éstos[8].

A continuación se presentarán algunos artefactos, y se indicará las disciplinas a las cuales asisten4.

4.2.4.1. Modelo de Casos de Uso

Este artefacto, utilizado dentro de la disciplina de requerimientos, describe los requerimientos funcionales de un sistema. Además, es utilizado para obtener los requerimientos no funcionales relacionados con éstos últimos.

Antes de profundizar en la descripción del artefacto, resulta conveniente dar a conocer algunas definiciones relacionadas con éste:

**Actor**: Es algo con comportamiento, como una persona (identificada por un rol), un sistema computacional, o una organización; por ejemplo, un cajero.

**Escenario**: Es una secuencia específica de acciones e interacciones entre actores y el sistema bajo discusión; también se conoce como instancia de caso de uso.

---

4Para ejemplos, dirigirse al Capítulo 5: Desarrollo de *SpiceX*. 
Pensando en SpiceX, simular con éxito un circuito electrónico, o fallar una simulación debido a una conexión eléctrica errónea, son ejemplos de escenarios.

Según el RUP, un caso de uso es un conjunto de escenarios, donde cada uno de éstos es una secuencia de acciones que un sistema realiza para otorgar un resultado observable y de interés para algún actor. Sabiendo esto, el modelo de casos de usos es el conjunto de todos los casos de usos de un sistema; un modelo de la funcionalidad del sistema y su contexto.

De acuerdo con la visibilidad en la descripción de un caso de uso, estos pueden clasificarse en los tipos: black-box y white-box. Los casos de uso del tipo black-box son los más usados y recomendados, y especifican lo que debe hacer el sistema (requerimientos funcionales) sin preocuparse de como lo hará (diseño). En los casos de uso con visibilidad white-box se describen, además, detalles que corresponden a la fase de diseño.

Debido a que este artefacto se presenta dentro de un proceso de desarrollo iterativo que busca construir sistemas orientados a objetos, resulta adecuado escribir los casos de uso con visibilidad del tipo black-box; la cual presenta una similitud directa con el concepto de encapsulación, estudiado en el capítulo 3.

Por otra parte, de acuerdo a las necesidades, los casos de usos pueden ser escritos con distintos grados de formalidad, los cuales se describen a continuación:

**Breve:** conciso resumen en un párrafo, que describe usualmente el escenario principal ejecutado con éxito.

**Casual:** varios párrafos informales, los cuales cubren varios escenarios.

**Completo:** formato más elaborado, que cubre todos los pasos y variaciones en detalle. 
Presenta secciones de soporte, como precondiciones y éxitos garantizados.
Descubriendo casos de uso

Cuando se trabaje con casos de uso, los esfuerzos deberían enfocarse en responder “¿Cómo se puede proveer valor observable al usuario, o satisfacer sus objetivos, utilizando el sistema?”, en vez de pensar en los requerimientos del sistema como una simple lista de características o funciones.

Existen innumerables interacciones entre los usuarios y un sistema. A continuación se describe una estrategia para discriminar casos de uso de poca utilidad, y seleccionar aquellos que conformarán el grupo de casos de uso base.

Un Proceso de Negocio Elemental, o EBP\(^5\), es un término del área de ingeniería de procesos de negocio, definido como:

| Una tarea realizada por una persona, en un lugar, y en un momento determinado, en respuesta a un evento de negocios, el cual adiciona valor de negocio medible y deja los datos en un estado consistente. |
| Por ejemplo, la aprobación de crédito o una orden de precio. |

Dicho esto, la pauta a seguir en el RUP, es: para el análisis de los requerimientos de una aplicación computacional, se deben enfocar los casos de uso a nivel de proceso de negocio elemental\(^9\).

Aunque la pauta debería seguirse de forma literal para encontrar los casos de uso base, existen algunas excepciones que llevan a considerar como casos de uso a interacciones de bajo nivel y que no satisfacen la descripción de un EBP.

Entre estas interacciones podrían considerarse: subfunciones comúnmente utilizadas por casos de uso base, y así evitar la duplicación de texto; y aquellas subfunciones que podrían formar parte de las precondiciones de casos de uso base, en repetidas ocasiones. Por ejemplo, el medio de pago a través de tarjeta de crédito, o la identificación de un usuario, en cada caso.

---

\(^5\) Elementary Business Process.
Casos de uso y objetivos de usuario

Pensando en satisfacer los objetivos de usuarios, los pasos recomendados para la escritura de casos de uso son:

1. escoger los límites del sistema,
   preguntándose si el sistema es sólo una aplicación de software, hardware y software como una unidad, todo esto más una persona usando el sistema, o una organización completa;

2. identificar los actores primarios,
   que son aquéllos que tienen objetivos de usuario satisfechos directamente a través del uso de servicios del sistema;

3. identificar los objetivos de los actores primarios,
   llevando éstos a un nivel que satisfaga la pauta del EBP;

4. y finalmente, definir los casos de usos,
   los cuales satisfarán los objetivos de usuario. Los casos de uso deberían ser nombrados de acuerdo a su objetivo, y comenzando por un verbo. Por ejemplo, para el objetivo: procesar una venta, el nombre del caso de uso podría ser: Procesar Venta.

Casos de uso y fases del RUP

Los casos de uso son centrales y de vital importancia para el RUP, el cual anima un desarrollo dirigido por éstos. Esto implica que:

- los requerimientos son primeramente capturados en casos de uso (en el modelo de casos de uso);
- los casos de uso son parte importante del plan de iteración. De hecho, el trabajo de una iteración es, en parte, definido por la elección de algunos escenarios, o
casos de uso completos. Además, los casos de uso son la fuente de información principal para realizar una estimación;

- la realización de casos de uso guía el diseño. Esto significa, que el grupo de desarrolladores diseña objetos y subsistemas colaborando entre sí, para ejecutar o realizar casos de uso;
- los casos de uso a menudo influencian la organización de los manuales de usuario.

A continuación se describe el grado de formalidad que deberían presentar los casos de uso, a medida que se completan las fases del RUP.

**Concepción:** dentro de esta fase, la mayoría de los casos de uso y actores son identificados. Además, se escribe entre el 10 y 20% de los casos de usos en forma completa y detallada.

**Elaboración:** dentro de esta fase se escribe entre el 80 y 90% de los casos de uso en forma completa y detallada. Los casos de uso abordados son aquéllos que capturan los requerimientos técnicos de alto riesgo, y que, generalmente, son los más difíciles de implementar. En cada iteración se realizan trabajos de casos de uso, escribiendo y reescribiendo éstos, basándose en la realimentación recibida de los adelantos del sistema.

**Construcción:** en esta fase, compuesta por múltiples iteraciones de tiempo limitado, sólo en el caso de ser necesario, se terminan de detallar la totalidad de los casos de uso. Sin embargo, no se debe considerar que el proceso de desarrollo congela los requerimientos en este punto. Se sigue abierto a los poco probables cambios en las exigencias, una vez llegado este maduro estado de la aplicación. De existir, los cambios en los requerimientos deberían ser de menor riesgo que en la fase de elaboración.

**Transición:** teniendo detallada la mayoría los casos de uso, esta fase utiliza dicha información para la implementación de funcionalidades remanentes.

---

6El caso de uso a seguir en la siguiente iteración, generalmente se escribe de manera completa y detallada en la disciplina de requerimientos de la iteración actual.
4.2.4.2. Plan de Iteración

Este artefacto, utilizado dentro de la disciplina de administración del proyecto, define el trabajo a realizar en la presente y la siguiente iteración. Junto al artefacto de Plan de Fase, forma parte de un artefacto más general, llamado Plan de Desarrollo de Software. En el RUP, generalmente, el trabajo es organizado en torno a la realización de casos de uso, por lo que se dice que es un proceso dirigido por éstos. Es decir, en una iteración, el desarrollador se dedica a implementar uno o más casos de uso, o algunos escenarios en particular, cuando el caso de uso completo es demasiado complejo. También se podrían dedicar iteraciones a implementar características que determinan requerimientos no funcionales del proyecto.

La decisión de qué requerimientos abordar en la siguiente iteración, debería estar basada en los siguientes criterios: riesgo, cobertura, valor de negocio, y habilidades de desarrollo, los cuales se describen a continuación:

- **Riesgo en los requerimientos** implica complejidad técnica y otros factores, como: incertidumbre de esfuerzo, especificación insuficiente, problemas políticos, o usabilidad.

- **Cobertura** se relaciona con el porcentaje de funcionalidad del sistema, que se alcanzaría al implementar un determinado caso de uso. En teoría, la mayor parte del sistema debería ser, al menos, tocado en las primeras iteraciones.

- **Valor de negocio** es un criterio que enfatiza la importancia de las funciones, que conforman los escenarios principales de los casos de uso. En teoría, se deberían implementar tempranamente al menos de forma parcial, aún si no presentan importantes desafíos técnicos.

- **Habilidades de desarrollo** es un criterio considerado solo por algunos proyectos. En el caso de que un equipo de desarrolladores carezca de habilidades en algún área en particular, se priorizan los casos de uso cuya implementación requiera de éstas.
Puntuación de Requerimientos

La puntuación de requerimientos proporciona una mecanismo guía para la elección de trabajos a realizar tempranamente. Éste consiste en una tabla en la cual se evalúan los requerimientos de acuerdo a determinados criterios, de manera cualitativa o cuantitativa.

En la evaluación cualitativa, se analiza el requerimiento (caso de uso o característica) teniendo todos los criterios en mente, y se le asigna un valor medio, bajo, o alto, de acuerdo a la importancia que presente.

En la evaluación cuantitativa, se analiza el requerimiento de acuerdo a cada criterio de forma independiente, y se le asigna una puntuación entre 1 y 3 (bajo, medio, o alto). Una vez terminado el proceso de evaluación para cada uno de los criterios, se realiza la suma total de puntajes, obteniendo así el parámetro de comparación.

4.2.4.3. Modelo del Dominio

Este artefacto, utilizado en la disciplina de modelo de negocio, ilustra clases conceptuales significativas (para los modeladores) en el dominio del problema. Es el principal artefacto a crear durante el análisis orientado a objetos, y de mucho valor al momento de efectuar trabajos de diseño e implementación.

Es muy importante comprender que el modelo del dominio es una representación visual de clases conceptuales que componen el mundo real, no de componentes de software[9].

El artefacto utiliza como medio de representación los diagramas de clases del UML, sin considerar operaciones en las entidades. Éste puede mostrar:

- objetos del dominio o clases conceptuales;
- asociaciones entre clases conceptuales;
- y atributos de clases conceptuales.

CAPÍTULO 4. RATIONAL UNIFIED PROCESS - RUP

En cada iteración, el modelo del dominio está limitado al escenario actualmente en consideración, y no pretende ser un modelo detallado de todas las posibles clases conceptuales y relaciones que compondrán el proyecto.

Teniendo en mente esta observación, el principal postulado para la identificación de clases conceptuales dice:

| Es mejor sobre detallar un modelo del dominio con una vasta cantidad de minuciosas clases conceptuales, que carecer de detalles. |

Consecuente con el enfoque recomendado por la orientación a objetos, no se debe pensar que un modelo del dominio es mejor si tiene menos clases conceptuales: lo contrario tiende a ser cierto.

Comúnmente se olvidan clases conceptuales durante los pasos de identificación iniciales, y se descubren después durante la consideración de atributos y asociaciones, o durante el trabajo de diseño. Cuando se encuentren, pueden ser adicionadas al modelo del dominio.

No se deben excluir clases conceptuales, simplemente porque los requerimientos no indican una obvia necesidad de recordar información relacionadas con éstas, o porque no presenten atributos. Es válido tener clases conceptuales sin atributos, o con un rol únicamente activo en vez de informativo.

Para crear un modelo del dominio, se aconseja seguir los siguientes pasos:

- listar las candidatas a clases conceptuales;
- dibujarlas en un modelo del dominio;
- adicionar las asociaciones necesarias;
- y adicionar los atributos.
CAPÍTULO 4. RATIONAL UNIFIED PROCESS - RUP

Identificación de Clases Conceptuales

Una técnica muy útil para la identificación de clases conceptuales es el análisis léxico: identificar los sustantivos y frases sustantivas en descripciones textuales del dominio, y considerar a ellas como candidatas a clases conceptuales o atributos. Los casos de uso en formato completo son una excelente descripción textual a considerar en este análisis. Se debe tener cuidado con la ambigüedad de algunas palabras del lenguaje, debido a que dos frases sustantivas podrían representar una misma clase conceptual.

Además, si se piensa en un concepto X, y no se visualiza como número o texto en el mundo real, es muy probable que X sea una clase conceptual y no un atributo. En caso de dudarlo demasiado, considerar el concepto como una clase.

Al momento de identificar clases conceptuales, se aconseja:

- usar el vocabulario del dominio para nombrar las clases conceptuales y atributos;
- excluir características irrelevantes;
- no adicionar conceptos ausentes en el dominio del problema.

Usualmente, el modelo del dominio es iniciado y terminado en elaboración.

Adicionando Asociaciones

Durante la creación del modelo del dominio, resulta de utilidad la identificación de asociaciones entre clases conceptuales necesarias para satisfacer requerimientos de información de los escenarios bajo desarrollo.

Una asociación es una relación entre clases, que representa alguna conexión de interés o significativa entre instancias de clases. Las asociaciones se adicionan al modelo del dominio mediante las relaciones entre clases del UML, y tienen un carácter bidireccional. Pueden incluir flechas de dirección para asistir la lectura del diagrama, y multiplicidad para representar restricciones de participación en un momento determinado.
Al momento de adicionar asociaciones al modelo del dominio, se deberían considerar sólo aquellas que necesitan ser preservadas en el tiempo⁸. La excesiva inclusión de asociaciones, por ejemplo con redundancias, podría generar un modelo confuso. Es más importante identificar clases conceptuales que identificar asociaciones⁹.

Se debe enfatizar las relaciones antes descritas, pero también adicionar relaciones que mejoran la comprensión del modelo del dominio.

**Adicionando Atributos**

Durante la creación del modelo del dominio, resulta de utilidad la identificación de atributos de clases conceptuales necesarios para satisfacer requerimientos de información de los escenarios bajo desarrollo.

Los atributos son valores de información lógica de objetos, y se adicionan al modelo del dominio mediante los atributos de clases del UML.

Los tipos de atributos del modelo del dominio deberían ser simples tipos de datos, como: Boolean, Date, Number, String, Time, etcétera. Por otra parte, los atributos complejos deberían ser representados como una clase separada, con la correspondiente asociación.

Al momento de adicionar atributos al modelo del dominio, se deberían considerar sólo aquéllos que representan información la cual se sugiere preservar, de forma explícita o implícita, en los requerimientos⁹.

Finalmente, se debería considerar la representación de tipos de datos simples, como clases separadas, si:

- está compuesto de secciones separadas, por ejemplo: números telefónicos, nombres de personas;
- existen operaciones asociadas con éste, por ejemplo: el carnet de identidad, el cual tiene asociada una operación de validación;

⁸Estas asociaciones se clasifican dentro del tipo “need to know”. Además, se recomienda el uso de listas de asociaciones comunes, las cuales muestran una serie de asociaciones típicas.
tiene otros atributos relacionados, por ejemplo: un precio en oferta puede tener relacionadas una fecha de inicio y otra de término;

- es una cantidad con una unidad, por ejemplo: velocidad con metros por segundos\(^9\);

- es una abstracción de uno o más tipos con algunas de las cualidades antes descritas.

La representación gráfica de estas clases de tipos de datos, debería estar condicionada por el valor adherido al modelo del dominio. No se debe olvidar que el modelo del dominio es una herramienta de comunicación\([9]\).

### 4.2.4.4. Modelo del Diseño

Durante el diseño de objetos, se desarrolla una solución lógica basada en el paradigma orientado a objetos. La principal actividad dentro de esta solución es la creación de diagramas de interacción, los cuales ilustran la colaboración entre objetos para satisfacer requerimientos.

Paralelamente a la creación de diagramas de interacción, pueden ser dibujados diagramas de clases de diseño. Éstos resumen la definición de clases de software e interfaces que serán implementadas en el software.

En términos del \textit{RUP}, estos dos subartefactos forman parte del modelo del diseño, artefacto más general utilizado dentro de la disciplina de diseño.

Se debe tener en consideración que el modelo del dominio provee un diccionario visual del vocabulario del dominio y conceptos, desde los cuales dibujar la inspiración para el nombramiento de algunas cosas en el diseño de software.

---

\(^9\)En este caso también se recomienda la representación de la unidad de medida, mediante una clase separada y la correspondiente asociación. Esto último facilita la conversión de unidades.
Diagramas de Interacción

Un problema común en los proyectos que utilizan la tecnología orientada a objetos, es que subestiman el valor de hacer el diseño de objetos a través de los diagramas de interacción. Se debe considerar que es durante este artefacto donde, primeramente, se requieren las habilidades de diseño de objetos; en términos de patrones, idiomas, y principios[9].

Para dibujar diagramas de interacción pueden utilizarse indistintamente: los diagramas de secuencia, o los diagramas de colaboración del UML; cada uno con sus ventajas y desventajas.

Los diagramas de secuencia son más fáciles de comprender, debido al orden que presentan; pero utilizan mayor espacio en documentos, debido a que los objetos se adicionan de forma secuencial a la derecha del diagrama. Al contrario, los diagramas de colaboración son más flexibles en cuanto a la ubicación de los objetos, pero más difíciles de entender.

Asignación de responsabilidades y métodos

La principal habilidad que un programador debe desarrollar, para crear diseños de objetos de buena calidad, es la asignación de responsabilidades.

Las responsabilidades son asignadas a los objetos a través de métodos o funciones. No se puede asegurar que una responsabilidad será siempre representada a través de un método, pero sí que los métodos son implementados para satisfacer las responsabilidades.

Por ejemplo, si se delega la responsabilidad de crear un circuito a la clase proyecto, ésta debería implementar uno o más métodos para la creación de dicho circuito. Esta asignación es representada en el diagrama de interacción mediante un mensaje.

Así mismo, los métodos utilizados para satisfacer responsabilidades pueden pertenecer a otros objetos, entre los cuales se divide la tarea asignada.
GRASP:
Patrones generales de software para la asignación de responsabilidades

Un patrón es un problema y su solución, identificados por un nombre, que puede ser aplicado en nuevos contextos. Éste dispone de una notificación acerca de su aplicación en nuevas situaciones y discusiones acerca de sus variaciones.

Los patrones GRASP\textsuperscript{10} describen principios fundamentales en el diseño de objetos y la asignación de responsabilidades. Es importante que el programador novato en las tecnologías de objetos entienda y aplique estos principios durante la creación de diagramas de interacción, debido a que se requiere que estas habilidades sean desarrolladas lo antes posible.

A continuación se presentan los cinco principales patrones GRASP:

Experto en Información o Experto

\textit{Problema}: ¿Qué principio aplicar para la asignación de responsabilidades a los objetos?

| Comenzar la asignación de responsabilidades indicando claramente la responsabilidad. |

\textit{Solución}: Asignar una responsabilidad al experto en información, la clase que tiene la información necesaria para hacerse cargo de la responsabilidad.

\textit{Discusión}: Aunque en el mundo real no todos los objetos pueden realizar operaciones por sí mismos, en el mundo del software orientado a objetos, todos los objetos tienen vida propia: pueden tomar responsabilidades y hacer cosas, dependiendo de la información de la cual dispongan.

\textit{Contraindicaciones}: No utilizar el patrón experto cuando se violen los principios de acoplamiento y cohesión.

\textit{Beneficios}:

\textsuperscript{10}General Responsibility Assignment Software Patterns.
Debido a que los objetos utilizan su propia información para realizar tareas, se mantiene la encapsulación de la información. 

Debido a que el comportamiento es distribuido a través de clases que satisfacen requerimientos de información, se anima la definición de clases “livianas” y más cohesivas; las cuales son más fáciles de entender y mantener.

Patrones o principios relacionados:

- Bajo Acoplamiento
- Alta Cohesión

También conocido como: “Colocar responsabilidades con los datos”, “el que lo conoce, lo hace”, “hacerlo por sí mismo”, o “poner los servicios con los atributos que se utilizan”.

Crearor

Problema: ¿Quién debería ser responsable de la creación de nuevas instancias de alguna clase?

Solución: Asignar a la clase B la responsabilidad de crear una instancia de la clase A, si se cumple al menos una de las siguientes afirmaciones:

- B agrega objetos de A.
- B contiene objetos de A.
- B graba instancias de objetos de A.
- B utiliza en detalle objetos de A.
- B tiene los datos de inicialización que serán pasados a los objetos de A, cuando éstos sean creados (en este caso B es un experto, con respecto a la creación de objetos de A).

Se dice que B es un creador de objetos de A. Si más de una clase satisface estas condiciones, preferir la clase B que agrega o contiene objetos de la clase A.

Discusión: La intención básica del patrón creador es encontrar un creador que necesite estar conectado al objeto creado en todo momento, o en cualquier evento.
Contraindicaciones: Cuando, por motivos de rendimiento, la creación de nuevas instancias utiliza una instancia de una clase similar basada en valores de propiedades externas (instancias recicladas), es conveniente la utilización de una clase Fábrica\textsuperscript{11}.

Beneficios: Se promueve el bajo acoplamiento, lo cual implica independencia en la mantención y altas oportunidades de reutilización de código.

Patrones y principios relacionados:
- Bajo Acoplamiento
- Fábrica

Bajo Acoplamiento

Problema: ¿Cómo mantener la independencia, el bajo impacto frente a las modificaciones, e incrementar la reutilización de los objetos?

Solución: Asignar una responsabilidad de manera que el acoplamiento se mantenga bajo.

Discusión: El acoplamiento mide el grado en que un elemento está conectado, tiene conocimiento, o confía en otros elementos. Estos elementos incluyen clases, subsistemas, sistemas, etcétera.

En la práctica, el nivel de acoplamiento no puede ser considerado independientemente de otros principios como Experto y Alta Cohesión. Sin embargo, éste es uno de los factores a considerar en el mejoramiento del diseño.

Una subclase está fuertemente acoplada a su superclase. Es por esto que la decisión de derivar de una superclase necesita ser cuidadosamente considerada.

En general, las clases que son de naturaleza inherentemente genérica, y con altas probabilidades de ser reutilizadas, deberían presentar, principalmente, un bajo acoplamiento.

\textsuperscript{11}Otro patrón de diseño GRASP.
El extremo contrario, es decir, la total ausencia de acoplamiento, tampoco es bueno. No se debe olvidar que una de las principales metáforas de la tecnología orientada a objetos es: un sistema de objetos conectados que se comunican vía mensajes.

Contraindicaciones: No debería existir problema alguno en acoplar nuevas clases con elementos estables y de fácil acceso. Un ejemplo de éstos son las clases incluidas en los lenguajes de programación orientados a objetos. El problema en sí no es un elevado acoplamiento, sino un elevado acoplamiento con elementos en algún sentido inestables.

Beneficios:

- Independencia frente a cambios en otros elementos.
- Facilidad en la comprensión de elementos aislados.
- Facilidad de reutilización de componentes.

Patrones y principios relacionados: Variación Protegida

Alta Cohesión

Problema: ¿Cómo controlar la complejidad de un sistema orientado a objetos?

Solución: Asignar una responsabilidad de manera que la cohesión se mantenga alta.

Discusión: Cohesión, o cohesión funcional, mide el grado en el cual las responsabilidades de un elemento están relacionadas. Estos elementos incluyen clases, subsistemas, etcétera.

Baja cohesión, una característica no deseable en los sistemas orientados a objetos, a menudo representa una insuficiente división del trabajo en la abstracción, o la asignación de responsabilidades que deberían haber sido delegadas a otros objetos. Desde este punto de vista, alta cohesión significa: divide lo suficiente, hasta que las clases tengan totalmente relacionados sus métodos.

En la práctica, el nivel de cohesión no puede ser considerado independientemente de otros principios como Experto y Bajo Acoplamiento.
La pauta a seguir es la siguiente: una clase con alta cohesión tiene un número relativamente pequeño de métodos, su funcionalidad está altamente relacionada, y no presenta sobrecarga de trabajo.

**Contraindicaciones:** Existen unas pocas clases en las cuales la baja cohesión es aceptable. Por ejemplo, cuando una clase debe ser mantenida por una sola persona, o cuando se implementan objetos en servidores distribuidos. En este último caso, el rendimiento disminuye si se divide demasiado el trabajo; debido a la dificultad que presenta el acceder a objetos remotos.

**Beneficios:**

- Incremento en claridad y facilidad de comprensión del diseño.
- Simplificación de la mantención y el mejoramiento.
- A menudo, se motiva el bajo acoplamiento.
- Se facilita la reutilización de las clases.

**Controlador**

**Problema:** ¿Quién debería ser el responsable de manejar un evento de entrada del sistema?

**Solución:** Asignar la responsabilidad de recibir o manejar un mensaje de evento de sistema, a una clase representando una de las siguientes elecciones:

- El sistema completo, dispositivo, o subsistema (controlador fachada).
- Un escenario de caso de uso dentro del cual ocurren eventos de sistema, a menudo nombrados: `<UseCaseName>Handler`, `<UseCaseName>Coordinator`, o `<UseCaseName>Session` (caso de uso o controlador de sesión).
- Usar la misma clase controlador para todos los eventos de sistema en el mismo escenario del caso de uso.

Las clases `window, applet, widget, view,` y `document` no caben dentro de esta lista. Generalmente, éstas delegan los eventos de sistema al controlador.
**Discusión:** Un evento de entrada del sistema es un evento generado por un actor externo. Estos eventos son asociados con operaciones del sistema, las cuales responden a éstos de igual forma que los métodos responden a los mensajes de otros objetos. Un controlador define un método para la operación del sistema (*handler*).

Un defecto común en el diseño de controladores, es la asignación de demasiadas responsabilidades. Normalmente, un controlador debería delegar trabajo a otros objetos; éste sólo coordina o controla la actividad.

¿Controladores fachada o controladores casos de uso?

- Los controladores fachada son convenientes cuando no se tienen demasiados eventos de sistema, o si no es posible que la interfaz de usuario, *UI*, envíe los mensajes de eventos de sistema a controladores alternativos, como un sistema de procesamiento de mensajes.
- Los controladores casos de uso son convenientes cuando los controladores fachada incrementan la baja cohesión o el alto acoplamiento. Cuando existe una gran cantidad de eventos de sistema a través de diferentes procesos; los controladores casos de uso dividen su manejo en clases separadas, más fáciles de administrar, y también proveen las bases para el conocimiento y razonamiento relacionado con el estado actual del escenario en progreso.

Las operaciones del sistema deberían ser manejadas en las capas de aplicación lógica o dominio de los objetos, y no en la capa de interfaz de un sistema (donde se encuentran las clases *windows, widgets*, etcétera).

**Beneficios:**

- Incremento en el potencial de reutilización y sustitución de interfaces. La lógica de la aplicación no está limitada a la capa de interfaz, por lo tanto, ésta puede ser reemplazada con una interfaz diferente.
- Conocimiento acerca del estado de un caso de uso.

**Patrones y principios relacionados:**

- Comando: En un sistema dirigido por mensajes, cada mensaje puede ser representado y dirigido por un objeto Comando separado.
• Fachada: Un controlador fachada es una clase de Fachada.
• Capas: Colocar la lógica del dominio en la capa del dominio, y no en la capa de presentación, es un patrón de Capas.
• Fabricación Pura: Es una clase de software que es una creación arbitraria del diseñador. No está inspirada por el modelo del dominio. Un controlador caso de uso es un tipo de Fabricación Pura.

4.2.4.5. Modelo de la Implementación

Este artefacto contiene el código fuente, las definiciones de bases de datos, páginas HTML, y otros artefactos que implementan el proyecto.

No se deberían excluir las actividades de diseño a medida que se ejecuta este artefacto. Es normal y conveniente corregir algunos diagramas del modelo del diseño, a medida que se programa. Después de actualizados, los diagramas pueden ser extendidos y utilizados en futuras iteraciones.
Capítulo 5

Desarrollo de SpiceX

En este capítulo, se documentará el desarrollo de SpiceX, siguiendo las pautas propuestas por el RUP. Al mismo tiempo, pretende ser el ejemplo y práctica de cada una de las teorías enunciadas en capítulos previos.

5.1. Concepción

En el caso de SpiceX, la fase de concepción incluirá las primeras actividades de análisis de requerimientos y los planes a seguir durante la primera iteración.

5.1.1. Disciplina de Requerimientos

5.1.1.1. Modelo de Casos de Uso

Para comenzar la investigación de los requerimientos funcionales del sistema, se identificarán los límites del programa, los actores primarios, sus objetivos, y los casos de uso relacionados directamente con estos últimos.

Límites del sistema

SpiceX será una aplicación de software utilizada directamente por el estudiante, profesor, o técnico, interesado en simular un circuito electrónico. Facilitará el mecanismo de diseño, y entregará información acerca de la topología del circuito a un núcleo de simulación: Spice; el cual trabaja, únicamente, a través de la línea de comandos.
Por lo tanto, el sistema estará compuesto por la interfaz gráfica de desarrollo de circuitos, con todos los módulos que la asisten, más el núcleo de simulación utilizado para realizar los cálculos matemáticos.

**Actores primarios**

En un principio, *SpiceX* estará enfocado en asistir el diseño de proyectos personales; por lo tanto, el actor primario será el estudiante, profesor, o técnico, identificado indiferentemente como Usuario\(^1\).

**Objetivos de usuario**

El Usuario busca satisfacer varios objetivos usando la aplicación, entre los cuales se encuentran:

- diseñar un modelo computacional de un circuito electrónico, teniendo la posibilidad de guardararlo en un archivo, y así poder: mejorararlo, terminarlo, o ampliarlo, a futuro;
- diseñar un modelo computacional de un circuito electrónico, teniendo la posibilidad de escalarlo a la medida y exportarlo como imagen, para así poder insertarlo en documentos de texto;
- simular un modelo de circuito electrónico previamente guardado, teniendo la posibilidad de comprobar el correcto funcionamiento del prototipo, antes de ser implementado en una placa de circuito impreso;
- desplegar el resultado de la simulación de forma gráfica, teniendo la posibilidad de analizar las señales de manera intuitiva;
- adicionar nuevos elementos a la librería de componentes gráficos, para así ser utilizados en futuros diseños.

**Definición de los casos de uso**

La definición de los casos de uso buscará satisfacer cada uno de los objetivos de usuario.

---

\(^1\) A futuro se podrían expandir las fronteras del sistema, considerando la creación de proyectos a nivel corporativo. En tal caso, entre los actores estarían: la corporación, los diseñadores, el coordinador de proyecto, etcétera.
FORMATO BREVE

1. Crear Circuito: El Usuario diseña un modelo de circuito electrónico, haciendo uso de una librería de componentes gráficos. Cuando el Usuario lo decida, el Sistema guardará el esquema de circuito en un archivo descriptivo.

2. Exportar Imagen: El Usuario diseña un modelo de circuito electrónico, haciendo uso de una librería de componentes gráficos. Cuando el Usuario lo decida, el Sistema escalará las magnitudes gráficas del modelo y exportará el esquema de circuito a un archivo de imagen.

3. Simular Circuito: El Sistema lee los parámetros de simulación previamente seleccionados por el Usuario, comprueba la integridad de las conexiones eléctricas que implementan el modelo diseñado, y, finalmente, simula el comportamiento del circuito.

4. Desplegar Simulación: El Sistema despliega la información generada por la simulación, de forma gráfica e inteligible.

5. Adicionar Componentes: El Usuario selecciona un archivo librería, de forma gráfica y asistida, para que luego el Sistema extraiga la información de cada componente y los inserte en la librería de componentes gráficos.

FORMATO CASUAL

Crear Circuito: El Usuario diseña un modelo del circuito electrónico, haciendo uso de una librería de componentes gráficos. Entonces…

Escenarios Alternativos:

- El Usuario termina el circuito de forma satisfactoria, e indica al Sistema que lo guarde en un archivo descriptivo;

- El Usuario elimina componentes que generan disconformidad desde el punto de vista de modelaje, para luego indicar al Sistema que guarde el circuito modificado en un archivo descriptivo.
el Usuario procede a cerrar el programa de forma abrupta, por lo que el Sistema pregunta si se quiere guardar o descartar el circuito hasta entonces implementado. De ser positiva la respuesta, el Sistema procede a guardar el circuito en un archivo descriptivo.

**Formato Completo**

**Primer Caso de Uso: Crear Circuito**

- Actor primario: Usuario
- Garantías de éxito (Poscondiciones): El modelo de circuito es guardado en un archivo descriptivo.
- Escenario principal ejecutado con éxito (flujo básico):
  1. El Usuario pone en funcionamiento el Sistema.
  2. El Usuario inicializa un proyecto.
  3. El Sistema crea un proyecto.
  4. El Sistema crea una superficie de conexión.
  5. El Sistema crea una rejilla de conexión.
  6. El Usuario selecciona un componente electrónico de una librería de componentes gráficos.
  7. El Usuario implementa la superficie de conexión, con el componente previamente seleccionado.
  8. El Sistema ubica el componente electrónico en posiciones discretas de la superficie de conexión, de acuerdo a una rejilla de conexión. 

  ***Usuario y Sistema repiten los pasos 6, 7 y 8, hasta que el circuito disponga de todos los componentes electrónicos deseados.***

  10. El Usuario interconecta pares de componentes electrónicos, previamente ubicados en la superficie de conexión.
11. El Sistema ubica el alambre en posiciones discretas de la superficie de conexión, de acuerdo a una rejilla de conexión.

*Usuario y Sistema repiten los pasos 10 y 11, hasta que el usuario decida terminar.*

12. El Usuario indica al Sistema que salve el proyecto.

13. El Sistema identifica y recuerda la posición de los componentes y los alambres en la superficie de conexión.

14. El Sistema salva el proyecto en un archivo descriptivo.

5.1.2. Disciplina de Administración del Proyecto

5.1.2.1. Plan de Iteración

En la puntuación de los requerimientos solo se incluirán los criterios que se consideren relevantes. Además, se dará prioridad a la implementación de los escenarios principales en los casos de uso analizados.

<table>
<thead>
<tr>
<th>Requerimiento</th>
<th>Tipo</th>
<th>Riesgo</th>
<th>Cobertura</th>
<th>Calificación</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crear Circuito</td>
<td>Caso de Uso</td>
<td>3</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>Exportar Imagen</td>
<td>Caso de Uso</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Simular Circuito</td>
<td>Caso de Uso</td>
<td>3</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>Desplegar Simulación</td>
<td>Caso de Uso</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Adicionar Componentes</td>
<td>Caso de Uso</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

Cuadro 5.1: Puntuación de los requerimientos en concepción.

De la puntuación de los requerimientos, se puede visualizar la importancia de los casos de uso Crear Circuito y Simular Circuito. Debido a que no se puede simular un circuito inexistente, la primera iteración de la fase de elaboración implementará el escenario principal del caso de uso: Crear Circuito.
5.2. Elaboración: Primera Iteración

5.2.1. Disciplina de Modelo de Negocio

5.2.1.1. Modelo del Dominio

Para la creación del modelo del dominio de la figura 5.1, se utilizó como descripción textual del dominio, el formato completo del caso de uso en desarrollo.

A continuación, en la figura 5.2 se adicionaron asociaciones consideradas dignas de recordar, las cuales serán de utilidad al momento de esquematizar la funcionalidad del sistema a través del modelo del diseño.

Finalmente, en la figura 5.3 se adicionaron atributos a las clases conceptuales y asociaciones existentes, además de nuevas asociaciones no visualizadas con anterioridad. Cabe recalcar que en esta figura se ha eliminado la clase conceptual Surface, debido a que representa el mismo concepto que Circuit.
Figura 5.2: Modelo del dominio del caso de uso Crear Circuito, incluidas las asociaciones relevantes.

Figura 5.3: Modelo del dominio del caso de uso Crear Circuito, incluidas asociaciones y atributos.
5.2.2. Disciplina de Diseño

5.2.2.1. Modelo del Diseño: Diagramas de Interacción

En los diagramas de interacción que se muestran a continuación, se pintaron de color gris los elementos gráficos que compondrán *SpiceX*. De esta forma se busca diferenciar claramente los elementos que están altamente acoplados a interfaces gráficas, de los elementos puramente lógicos.

Dicho esto, el primer patrón a considerar fue la creación de una clase *Controller*, la cual, si bien no está presente en el modelo del dominio, posibilita el manejo de la mayoría de los eventos generados por el Usuario.

Aunque se podría explotar la ventaja de flexibilidad en cuanto a la interfaz de usuario, la implementación de este patrón busca proveer un mecanismo para el seguimiento de los métodos ejecutados, durante la realización de los diferentes casos de uso.

Generalmente se dedica un caso de uso específico a la inicialización del sistema. Sin embargo, y tomando en cuenta la facilidad de implementación en este caso, se consideró como una precondición a implementar, rápidamente, en los pasos iniciales.

Una pauta para creación de inicializaciones es: crear el diseño de la inicialización después de haber estudiado algunos diseños de casos de uso, debido a que se tendrá una visión más detallada de los elementos que se deberían considerar en la puesta en marcha del sistema.

![Diagrama de interacción para la inicialización del sistema.](image-url)
CAPÍTULO 5. DESARROLLO DE SPICEX

A lo largo de los procesos representados en la figura 5.4 y la figura 5.5, la única intervención del usuario ha sido la ejecución del sistema. Sin embargo, una vez inicializado el programa, el usuario comenzará a trabajar con el sistema; por lo que se modeló tempranamente la funcionalidad de crear proyectos (figura 5.6).

Figura 5.5: Diagrama de interacción para la creación de los objetos gráficos.

Figura 5.6: Diagrama de interacción para la creación de un proyecto.
Dada su importancia, la creación de la clase *Schematic* se modeló en un diagrama independiente, representado por la figura 5.7. Esta clase abstrae el concepto Circuito, sugerido por el modelo del dominio.

![Diagrama de Interacción para la Creación de la Clase Schematic](image1)

Figura 5.7: Diagrama de interacción para la creación de la clase *Schematic*.

Considerando los últimos pasos del caso de uso, se sabe que en algún momento se guardará el proyecto. Es por esto que en la figura 5.8 se modeló la funcionalidad de abrir un proyecto previamente guardado, implementando así un mecanismo para probar la correcta realización del caso de uso.

![Diagrama de Interacción para la Apertura de un Proyecto](image2)

Figura 5.8: Diagrama de interacción para la apertura de un proyecto.
Después de esto, se modelaron la adición de componentes y la adición de alambres (figuras 5.9 y 5.10, respectivamente), para así implementar el circuito incluido en el proyecto previamente creado.

Figura 5.9: Diagrama de interacción para la adición de un componente.

Figura 5.10: Diagrama de interacción para la adición de un alambre.
Finalmente, se modeló el proceso de guardar el proyecto en un archivo de sistema.

![Diagrama de interacción para guardar un proyecto.](image)

Figura 5.11: Diagrama de interacción para guardar un proyecto.

### 5.2.3. Disciplina de Implementación

#### 5.2.3.1. Modelo de la Implementación

El lenguaje elegido para la implementación de *SpiceX* fue C#. En pocas palabras, este lenguaje es el resultado de mezclar la facilidad de utilización del lenguaje *Java*, con la rapidez de ejecución del potente C++. Dejando de lado toda la discusión idealista que se genera en torno a *Java* - *C#*, la decisión final estuvo influenciada por la productividad lograda en un par de pruebas, realizadas con anterioridad al comienzo del presente trabajo.

El proceso de implementación del caso de uso Crear Circuito, siguió los pasos inicialmente esquematizados en el modelo del diseño; sin embargo, el diseño fue actualizado a medida que se fueron descubriendo clases y adicionando características.

La figura 5.12 muestra el primer avance de *SpiceX*, después de adicionados algunos componentes electrónicos (representados mediante una única resistencia), los cuales se encuentran interconectados mediante alambres.

El código fuente que implementa el caso de uso Crear Circuito será incluido en un CD acompañando el presente trabajo, además de estar disponible online en la siguiente página web: [http://spicex.chileforge.cl](http://spicex.chileforge.cl).
Figura 5.12: Primer prototipo de SpiceX.

5.2.4. Disciplina de Administración del Proyecto

5.2.4.1. Plan de Iteración

Como se vio en la fase de concepción, y de acuerdo a la puntuación de los requerimientos, el segundo paso a ejecutar es la implementación del escenario principal del caso de uso: Simular Circuito.

5.2.5. Disciplina de Requerimientos

5.2.5.1. Modelo de Casos de Uso

A continuación se presenta el caso de uso Simular Circuito, en sus formatos casual y completo:

Formato Casual

Simular Circuito: El Sistema lee los parámetros de simulación previamente seleccionados por el Usuario, comprueba la integridad de las conexiones eléctricas que implementan el modelo diseñado. Entonces...

Escenarios Alternativos:

- el Sistema simula el comportamiento del circuito.
el Sistema detiene el proceso de simulación, debido a un error en las conexiones eléctricas.

Formato Completo

Segundo Caso de Uso: Simular Circuito

- Actor Primario: Usuario

- Precondiciones: El Usuario guarda el proyecto en un archivo descriptivo, incluyendo los parámetros de simulación, y luego presiona un botón para simular el circuito.

- Garantías de éxito (Poscondiciones): El Sistema crea un archivo para describir la topología electrónica del circuito: netlist; y la simulación es efectuada por el núcleo de simulación: Spice.

- Escenario principal ejecutado con éxito (flujo básico):

1. El Sistema lee el archivo descriptivo que almacena la descripción del proyecto implementado.
2. El Sistema lee los parámetros de simulación previamente seleccionados por el Usuario.
3. El Sistema genera el archivo de descripción topológica del circuito: netlist; basándose en reglas específicas de creación de este tipo de archivos para Spice\(^2\).
4. El Sistema ejecuta el Núcleo de Simulación, entregando el archivo netlist como parámetro.
5. El Sistema comprueba la integridad de las conexiones eléctricas del esquema de circuito, haciendo uso del Núcleo de Simulación y su mecanismo de detección de errores.

Como se puede ver en el formato completo del caso de uso, Núcleo de Simulación (Spice en este caso) es un actor de soporte para el sistema desarrollado.

\(^2\)Para utilizar otros núcleos de simulación, se deberían incluir reglas de creación para las netlists correspondientes.
5.3. **Elaboración: Segunda Iteración**

5.3.1. **Disciplina de Modelo de Negocio**

5.3.1.1. **Modelo del Dominio**

Al momento de crear este artefacto, no se han encontrado atributos relevantes a incluir en el modelo.

5.3.2. **Disciplina de Diseño**

5.3.2.1. **Modelo del Diseño: Diagramas de Interacción**

El primer paso en el diseño de este caso de uso, es el modelado de las precondiciones. Esto implica, entre otras cosas, la creación de una vía de interacción con el programa, al momento de ajustar los parámetros de simulación.

Dicho esto, el diagrama de interacción de la figura 5.14 muestra las clases que colaboran al momento de ajustar los parámetros de simulación. Nótese que el diseño busca imitar la forma de interacción con el usuario que presentan los simuladores del entorno Windows. Los alcances del sistema motivan este tipo de decisiones.
Figura 5.14: Diagrama de interacción para el ajuste de parámetros de simulación.

En el diagrama de la figura 5.14 se observa que la clase Project dispone de un nuevo miembro: la clase Parameters. Como se dijo en un comienzo, esta iteración implementará el segundo caso de uso en estudio y perfeccionará la implementación del trabajo previo. En el apéndice B se pueden observar los diagramas de interacción que describen las implementaciones actualizadas de los casos de uso.

A medida que se avanza en las iteraciones, se puede ver más claramente la aplicación de patrones de software en la implementación del sistema. Por ejemplo, la figura B.3 indica que la creación de los parámetros se delegó a la clase Project. En este caso la clase Project es el creador de la clase Parameters, debido a que utiliza en detalle la información almacenada en ella, y además mantiene una referencia constante como base contenedora.

En el futuro se podría habilitar la creación de más de un circuito por proyecto, con lo que se podría replantear el creador de la clase Parameters. Si se decidiese que cada circuito poseerá parámetros de simulación independientes del resto de los circuitos, dichos parámetros podrían estar contenidos en la clase Schematic, que es la clase que representa el circuito, y que utilizaría en forma detallada la información de simulación.

Una vez diseñado el ajuste de los parámetros de simulación, se puede visualizar la realización del caso de uso Simular Circuito. La figura 5.15 muestra las clases que colaboran en esta tarea.
Tomando en cuenta que a futuro se podrían generar netlist para distintos tipos de núcleos de simulación, la clase Core incluye y crea una instancia de la clase Syntax, la cual se encargará de cargar desde memoria las reglas específicas para la creación de este tipo de documentos (dependiendo del núcleo seleccionado en los parámetros de simulación). Dicha especificación requiere ser implementada en XML, para lo cual se dispondrá de reglas gramaticales bien definidas.

Entre las correcciones más importantes a la arquitectura del programa, las cuales se pueden observar en el apéndice B, se debe mencionar la eliminación de información redundante en varias clases. Si se observan las figuras 5.7 y 5.8 se puede observar que existen dos tipos de elementos de circuito: SchemElement y LogicElement, los cuales almacenaban, prácticamente, la misma información.

Para remediar esto se implementó una única clase Element, la cual incluye una clase vista o visual, que se encarga de representar la información de forma gráfica. El diseño se inspiró en una versión más avanzada del patrón controlador, llamada: MVC: model view controller[16], la cual motiva la separación de la información lógica, de la representación gráfica, y más aún, del manejo de eventos.
CAPÍTULO 5. DESARROLLO DE SPICEX

Considerando que no se desea disponer de una multitud de controladores, los cuales podrían dificultar la comprensibilidad del código, la aplicación del patrón MVC se reservó hasta la implementación de la clase **Schematic**. Esta clase se encarga de almacenar la información lógica de cada uno de los elementos de circuito, de desplegar sus vistas, y de controlar los eventos de usuario aplicados sobre la colectividad.

La figura 5.16, resume el diseño de la clase **Schematic**, con su vista y controlador:

![Diagrama de clases de la clase Schematic, con su vista y controlador.](attachment:image.png)

Figura 5.16: Diagrama de clases de la clase **Schematic**, con su vista y controlador.

5.3.3. Disciplina de Implementación

5.3.3.1. Modelo de la Implementación

Al momento de implementar estos nuevos requerimientos, se comenzó a utilizar el programa de documentación **Doxygen**. Esta aplicación permite utilizar los comentarios dentro del código fuente como documentación específica de una clase, método, miembro, o cualquier bloque de código en general[14].

Para posibilitar la extracción de la información, se deben seguir reglas específicas al momento de comentar el código. De esta forma, los comentarios serán transformados en documentación **HTML**, o cualquier otro formato soportado, una vez procesados los archivos que componen el sistema.

La figura 5.17 muestra la primera simulación realizada por **SpiceX**, después de implementados los requerimientos de la presente iteración.

---

Si bien la implementación de esta clase se inspiró en los postulados del patrón MVC, el código final presenta algunas variaciones que podrían mejorar el rendimiento del sistema.
CAPÍTULO 5. DESARROLLO DE SPICEX

Figura 5.17: Primera simulación de Spicex.
5.3.4. Disciplina de Administración del Proyecto

5.3.4.1. Plan de Iteración

Como se aprecia en el cuadro 5.1, el siguiente caso de uso a implementar es Adicionar Componentes. Considerando que al terminar la siguiente iteración se pretende que la funcionalidad del programa permita comenzar su utilización, se implementará, además, el segundo escenario del caso de uso Crear Circuito.

5.3.5. Disciplina de Requerimientos

5.3.5.1. Modelo de Casos de Uso

A continuación se presenta el caso de uso Adicionar Componentes, en sus formatos casual y completo.

Formato Casual

Adicionar Componentes: El Usuario selecciona un archivo de librería, de forma gráfica y asistida. Entonces...

Escenarios Alternativos:

- el Sistema reconoce los elementos descritos en el archivo, extrae la información de cada componente, y, finalmente, los inserta en la librería de componentes gráficos.
- el Sistema no reconoce la totalidad de los elementos descritos en el archivo, extrae únicamente la información de los componentes reconocidos, y los inserta en la librería de componentes gráficos.
- el Sistema no reconoce los elementos descritos en el archivo, y despliega un mensaje indicando el que el archivo no fue reconocido como librería de componentes para Spice.
Formato Completo

Tercer Caso de Uso: Adicionar Componentes

- Actor Primario: Usuario

- Precondiciones: El Usuario dispone de un archivo de librería de componentes para el Núcleo de Simulación Spice. El Usuario presiona un botón para la adición de librerías.

- Garantías de éxito (Postcondiciones): El Sistema genera un archivo de librería en formato SpiceX, y adiciona dicha librería al grupo de librerías gráficas disponibles.

- Escenario principal ejecutado con éxito (flujo básico):

  1. El Usuario selecciona el archivo de librería de componentes para Spice, haciendo uso de una ventana gráfica.

  2. El Sistema ejecuta el lector de librerías.

  3. El lector de librerías analiza la librería de componentes para Spice, y extrae la información de los componentes soportados.

  4. El lector de librerías entrega la información de los componentes extraídos a la librería de componentes de SpiceX.

  5. La librería de componentes de SpiceX genera un archivo de librería de componentes para SpiceX.

  6. La librería de componentes adiciona el archivo de librería recién creado al archivo de configuración de las librerías gráficas de SpiceX (libconf.xml).
5.4. Elaboración: Tercera Iteración

5.4.1. Disciplina de Modelo de Negocio

5.4.1.1. Modelo del Dominio

![Diagrama del modelo del dominio del caso de uso Adicionar Componentes.](image)

Figura 5.18: Modelo del dominio del caso de uso Adicionar Componentes.

Al momento de crear este artefacto, no se han encontrado atributos relevantes a incluir en el modelo.

5.4.2. Disciplina de Diseño

5.4.2.1. Modelo del Diseño: Diagramas de Interacción

La figura 5.19 muestra la realización del caso de uso Adicionar Componentes. Como ahí se aprecia, el proceso de exclusión de componentes será responsabilidad de la clase LibraryReader, por lo que la clase Library recibirá componentes que se sabe de antemano son compatibles con *SpiceX*. 
Figura 5.19: Diagrama de interacción del caso de uso Adicionar Componentes.
5.4.3. Disciplina de Implementación

5.4.3.1. Modelo de la Implementación

Para comenzar, en este paso se implementó el segundo escenario del primer caso de uso: Crear Circuito. Esto último adicionó la funcionalidad de eliminar componentes, alambres, y marcadores de voltajes, de circuitos previamente creados.

También se optimizaron los mecanismos de localización de los componentes, expresándolos en relación a los puntos que componen la grilla de conexiones, y no respecto a los pixels que los ubican en la pantalla gráfica. Este enfoque facilitará la futura implementación del mecanismo de zoom.

Además, se modificó el algoritmo de conexión eléctrica, mejorando la abstracción del proceso de alambrado de circuitos. Es así que, en el código actual, se pueden observar bloques de código que implementan: propagación nodal, conexión y desconexión de elementos (nodos y alambres), además del tratamiento especial del componente de tierra común; que se propaga a lo largo de los alambres con una mayor prioridad.

El caso de uso de Adicionar Componentes se implementó siguiendo los mismos pasos del proceso efectuado para la adición de nuevos componentes en Pspice[2]. Primeramente, se comienza con el reconocimiento de la descripción textual de los componentes; luego se asigna un símbolo genérico al tipo de componente adicionado (basándose en el tipo de modelo o en el número de nodos del subcircuito, según corresponda); y finalmente se deja abierta la posibilidad de asignar un símbolo especializado, y que describa mejor las características del nuevo componente.

La figura 5.20 muestra el incremento del número de componentes en las librerías de SpiceX, después de importar la librería: anlg_dev.lib.

A medida que se adicionen nuevos modelos y símbolos genéricos de subcircuitos, el número de componentes filtrados por la clase LibraryReader disminuirá. Por ahora, solo se han implementado los modelos de los diodos, y subcircuitos con 5 u 8 nodos.

Debido a que Pspice deriva directamente de Spice2, las mejoras en la sintaxis de Spice3 hacen incompatibles ciertos modelos creados de acuerdo al viejo formato Spice y actual formato Pspice. Por ejemplo, se necesita convertir todas las fuentes polinomiales (POLY) de Pspice a fuentes dependientes no-lineales en Spice3.
Figura 5.20: Adición de componentes en la librería gráfica de Spicex.
Por lo mismo, se han filtrado todos los circuitos que presenten fuentes polinomiales que especifiquen valores de parámetros mayores o iguales a los cuadráticos[15]. A medida que se investiguen más a fondo las ecuaciones que describen las fuentes polinomiales, esta limitación debería ser eliminada. Sin embargo, son raros los modelos que especifican dichos parámetros.

Finalmente, la figura 5.21 muestra la simulación realizada utilizando uno de los modelos importados de la librería eval.lib, y después de asignada su representación gráfica especializada. En este caso, el circuito es un amplificador inversor implementado mediante el amplificador operacional uA741[13].
Figura 5.21: Simulación de un amplificador inversor.
Capítulo 6

Conclusiones

Todo proyecto de título debería ser un aporte a la sociedad. Se trata de crear algo nuevo, ingenioso y que satisfaga una necesidad latente. Por lo mismo, junto con la proposición del proyecto siempre hay un conjunto de razones, argumentos, e hipótesis justificando la elección del tema.

La aplicación creada por este proyecto, el resultado del trabajo realizado, no comprueba o refuta directamente la tesis propuesta: la cual justifica que el proyecto sea digno de un candidato a título profesional. Sin embargo, en este caso, el ingenio fue aplicado al momento de identificar la necesidad y, más aún, visualizar las consecuencias de satisfacerla. Esto último, no quiere decir que el programa creado no haya presentado importantes desafíos técnicos.

6.1. **EDA en Linux**

Los argumentos que justificaron una nueva propuesta de aplicación para integrar captura esquemática y simulación electrónica en Linux, además de la facilidad de uso, fueron la arquitectura presente en los programas implementados previamente y el lenguaje elegido durante su creación.

A través del período de tiempo transcurrido desde el inicio del proyecto, se ha podido comprobar que dicha elección: la de comenzar un nuevo programa desde cero, y no continuar la expansión de programas anteriores; fue la adecuada.
En la página de ayuda de Oregano\(^1\), aplicación que presenta una funcionalidad bastante similar a la que busca implementar SpiceX, se puede leer que el grupo de programadores detendrá su evolución al término del presente ciclo de desarrollo. Esto último como consecuencia de la arquitectura que presenta el programa y el código que lo implementa: los cuales han hecho tedioso el proceso de implementación de nuevas funcionalidades.

El mejor candidato a lenguaje de programación, para implementar las futuras versiones de Oregano, es precisamente C#. Sería interesante averiguar qué proceso de desarrollo elegirán al momento de implementar la nueva arquitectura.

### 6.2. Webometrics

Webometrics\(^2\) es un término empleado para describir una nueva métrica de evaluación de referencias. El número de referencias a los artículos académicos de una institución, ha sido un importante criterio de evaluación de excelencia en reconocidos rankings de posicionamiento de universidades a nivel mundial.

Este nuevo tipo de métrica tiene la peculiaridad de ser fácilmente accesible por las instituciones, debido a que se pueden obtener sus magnitudes utilizando la World Wide Web. Por lo tanto, se puede prever que a futuro este criterio sea el más relevante al momento de posicionar las universidades, si se considera su complejidad y costo\(^3\).

La cantidad y calidad de las publicaciones en línea, por parte de las instituciones de educación superior, están tomando una importancia clave en el prestigio de las universidades. Si no existiese una alternativa gratuita o, a lo menos, más económica para publicar de forma legal, las universidades de los países menos desarrollados se verían fuertemente desprestigiadas.

En estos momentos, ya se encuentra implementado un ranking de universidades a nivel mundial, basado únicamente en webometrics\(^4\). Sin temor a equivocarse se podría decir que es el tercero en importancia a nivel mundial.

---

\(^1\)http://moin.lug.fi.uba.ar/Oregano
\(^2\)http://en.wikipedia.org/wiki/Webometrics
\(^3\)http://www.universitymetrics.com/tiki-index.php
\(^4\)http://www.webometrics.info/
Los criterios aplicados se han visto fuertemente acotados por este nuevo mecanismo de evaluación, provocando fuertes reacciones por parte de universidades con deficientes, inaccesibles, o desorganizados mecanismos de publicación.

6.3. Alcances logrados

Inicialmente, se determinó que los alcances del proyecto serían facilidad en la programación y evolución del sistema, funcionalidad suficiente para comenzar su utilización, y librerías de componentes de fácil expansión por parte de los usuarios. De acuerdo a esto, los alcances del sistema están logrados.

El código del sistema está implementado a través de patrones de software, como son: bajo acoplamiento y alta cohesión; los cuales permiten abordar fácilmente los nuevos desafíos o casos de usos que aparezcan a futuro. El primer beneficiado con esta arquitectura ha sido el propio autor del presente trabajo.

La funcionalidad de un sistema es un criterio independiente de la facilidad de uso de la aplicación. Por lo tanto, se podría decir que los núcleos de simulación en sí son completamente funcionales, debido a que permiten la simulación de una amplia gama de componentes electrónicos: suficientes para satisfacer cualquier requerimiento educacional.

Sin embargo, la funcionalidad de este sistema en particular no busca más que proveer una herramienta de fácil utilización y familiar a los programas de diseño y captura esquemática más populares. No se ha creado un programa de simulación electrónica, sino que una interfaz para explotar la funcionalidad previamente disponible que proveen de núcleos de simulación de línea de comandos.

Desde este punto de vista, la funcionalidad del sistema se ve reflejada en cada paso ejecutado a través del proceso de diseño y simulación de un circuito electrónico. Aunque no se ha provisto un mecanismo intuitivo para aprovechar la totalidad de las capacidades de los núcleos de simulación, esta tarea dejó de ser un desafío técnico y pasó a convertirse en una tarea que demanda, únicamente, disponibilidad de tiempo.
La adición de componentes no es totalmente automática, debido a que el usuario debe perfeccionar la representación gráfica de los componentes importados; sin embargo, la complejidad del escenario se puede comparar a los procesos de adición de componentes que implementan los programas comerciales de simulación electrónica.

Aunque no siempre se importe la totalidad de los componentes incluidos en una librería creada para Spice, la funcionalidad implementada permite incrementar el tamaño de las librerías SpiceX de forma significativa. Además, es interesante observar que aunque en un comienzo sólo se propuso importar librerías de componentes en formato Spice, se implementó una funcionalidad capaz de importar, además, librerías de componentes en formato Pspice.

Esto último es muy importante por dos motivos: primero, si se considera que en la actualidad el estándar de simulación electrónica es Pspice, se puede prever que la mayoría de los modelos de nuevos componentes electrónicos serán creados en este formato; y segundo, esta característica es un pequeño paso adelante en la compatibilidad con programas comerciales, lo cual es muy importante para el proceso de transición desde el sistema operativo Windows a Linux.

A futuro, los programadores y electrónicos con disponibilidad de tiempo, podrían, por un lado, incrementar las características de los núcleos de simulación soportadas; y por otro, aumentar la cantidad de modelos y representaciones gráficas disponibles. Aunque se trata de tareas totalmente independientes, sería recomendable que el programador que implemente nuevas funcionalidades disponga de conocimientos básicos en electrónica, y específicamente, en Spice.

Para colaborar con este sistema, los requerimientos de conocimientos en electrónica son mucho más altos que los requerimientos de conocimientos en programación. De hecho, la única tarea que podría demandar conocimientos básicos de programación, es la adición de nuevos mecanismos para la configuración de análisis. Sin embargo, estos mecanismos pueden ser copias idénticas del mecanismo de configuración del análisis transiente.

Se espera que esta aplicación sea autosuficiente, y que crezca a medida que se necesite. La arquitectura del sistema permite que los usuarios sean los principales contribuyentes a su funcionalidad.
Apéndice A

Código de *SpiceX*

El código y la documentación de las clases, además de los ejemplos mostrados durante el capítulo 5, se encuentran disponibles en el *CD* que acompaña el presente trabajo.
Apéndice B

Diagramas de Interacción

Las siguientes figuras representan los diagramas de interacción de SpiceX, acorde al código finalmente implementado.

Figura B.1: Diagrama de interacción para la inicialización del sistema.
Figura B.2: Diagrama de interacción para la creación de objetos gráficos.
Figura B.3: Diagrama de interacción para la creación de proyectos.
Figura B.4: Diagrama de interacción para la apertura de proyectos.
Figura B.5: Diagrama de interacción para la adición de un componente.
Figura B.6: Diagrama de interacción para la adición de un alambre.
Figura B.7: Diagrama de interacción para guardar un proyecto.

Figura B.8: Diagrama de interacción para el ajuste de parámetros de simulación.
Figura B.9: Diagrama de interacción del caso de uso Simular Circuito.
Figura B.10: Diagrama de interacción del caso de uso Adicionar Componentes.
Bibliografía


