
**Universidad Nacional de Educación a Distancia**

**Escuela Técnica Superior de Ingeniería Informática**

**Máster Universitario en Ingeniería Informática**

# Editor de circuitos digitales en lenguaje Modelica

**Trabajo de Fin de Máster**

**Presentado por:** Caponera De Cobellis, Romolo Rosario

**Dirigido por:** Urquía Moraleda, Alfonso

**Co-dirigido por:** Martín Villalba, Carla

Fecha: Septiembre de 2023

# Resumen

El modelado y la simulación de sistemas es una de las tareas más importantes durante la fase de diseño de productos de todo tipo, especialmente en el ámbito de la ingeniería. En general, cuando se diseña un producto, no se puede experimentar con él para determinar su eficiencia o su resistencia, puesto que no existe aún físicamente. O, en otros casos, sí se tiene un objeto sobre el que se quiere experimentar, pero resulta inviable hacerlo por el costo o el tiempo que esto implicaría. En estos casos, resulta interesante utilizar métodos de modelado para crear una representación digital del objeto a evaluar que se comporte del mismo modo que el original ante las pruebas a las que se va a someter y, a continuación realizar los experimentos sobre este modelo. Esto, si bien a día de hoy es posible, requiere frecuentemente de la utilización de lenguajes de modelado como Modelica, lo cual puede suponer un obstáculo para aquellos científicos o ingenieros que no estén familiarizados con este lenguaje o, incluso, que no tengan conocimientos de programación.

Este trabajo presenta una herramienta basada en interfaz de usuario que permite la construcción de modelos de circuitos digitales, y la posterior exportación de dichos modelos a código Modelica ejecutable y listo para ser utilizado en la experimentación. La aplicación basa su funcionamiento en la mecánica de arrastrar y soltar, permitiendo al usuario colocar, mover y conectar los distintos componentes sin necesidad de escribir código de ningún tipo. Además ofrece amplias opciones de personalización, permitiendo modificar los distintos parámetros de los componentes desde la propia interfaz, e incluye la mayoría de los modelos de electrónica digital contenidos en la librería estándar de Modelica.

El sofware propuesto, escrito en Java, será multiplataforma, portable, gratuito y de código abierto mediante el gestor de repositorios GitHub, y puede encontrarse su código fuente en (Caponera De Cobellis, 2023a), así como en el Anexo B de este mismo documento.

# Abstract

System modeling and simulation is one of the most important tasks when it comes to product design of any kind, especially in the field of engineering. When a product is being designed, experiments cannot be performed on it to measure its efficiency or resistance, for it has not yet been built. It is also possible that the system does exist in the physical world, but it is not possible to perform any experiment on it, either because it's too demanding in terms of required time, or it is too costly to be performed. It is in this situations where modeling and simulation methods become useful in order to create a digital copy of the object, which will behave as the original one would under the tests to perform, so that experiments can be carried out using this model rather than the physical object. While it is possible, as of today, to do this, it often requires the usage of modeling-oriented programming languages, such as Modelica, thus creating a barrier for these scientists or engineers who do not know this particular language or, even, do not possess any programming knowledge at all.

This work introduces a graphical-interface-based tool that allows for digital circuit models design and construction, as well as the conversion of these circuits into runnable, experiment-ready Modelica code. This application is based on the drag-and-drop approach, allowing the user to place, move and connect components in a simple manner, without having to write a single line of code. Moreover, customization options are available, allowing the user to edit the component parameters using the interface itself, and includes most of the digital electronics models contained in the Modelica Standard Library.

The proposed software, written in Java, shall be multiplatform, portable, free and open-source, with its source code being available on the GitHub repository manager at (Caponera De Cobellis, 2023a), as well as in the Annex B in this document.

# Índice general

# Índice de figuras

# Índice de tablas

# 1. Introducción, objetivos y estructura

## 1.1. Introducción

La construcción de sistemas electrónicos digitales es, hoy en día, una de las actividades más importantes dentro de la industria tecnológica, y es que la práctica totalidad de las herramientas utilizadas a diario por las personas emplean, en mayor o menor medida, algún tipo de circuitería digital, ya sea un sistema avanzado, como un ordenador, o uno más sencillo, como el mando del garaje. Esto hace del diseño de sistemas digitales una tarea primordial en multitud de ámbitos de la industria y, en consecuencia, se hace necesario hallar herramientas y métodos que faciliten las fases de diseño y pruebas de estos circuitos, sin necesidad de implementarlos físicamente con el fin de experimentar con ellos.

Para este propósito, existen distintos frameworks y lenguajes de programación orientados a la simulación y modelado de sistemas, como es Modelica (The Modelica Association, 1997). Estos lenguajes, sin embargo, si bien son extremadamente potentes y ofrecen una enorme cantidad de posibilidades, son también muy complejos y difíciles de dominar, lo que limita su uso a aquellos usuarios que, además de tener conocimiento de electrónica digital, tengan también una base sólida en la programación de ordenadores. Con el fin de eliminar esa barrera, este trabajo presenta una herramienta amigable y basada en interfaz gráfica de usuario — generalmente llamada GUI, como abreviación de Graphical User Interface — que permite la construcción de modelos de sistemas digitales ejecutables en entornos del lenguaje Modelica, sin la necesidad de escribir código de ningún tipo.

## 1.2. Objetivos

El objetivo principal de este trabajo es el desarrollo de una aplicación multiplataforma y sencilla de utilizar, capaz de generar código Modelica para sistemas digitales

basado en la mecánica Drag&drop (arrastrar y soltar). La aplicación contará con una zona de trabajo sobre la que se podrán construir visualmente los circuitos que se desea modelar, arrastrando y conectando los componentes mediante el uso del puntero. Además, permitirá guardar y cargar en otro momento los circuitos diseñados y, una vez finalizados, su exportación como código Modelica ejecutable desde cualquier entorno de este lenguaje. La aplicación contará con los principales componentes de la librería estándar de Modelica, permitiendo una gran variedad de posibilidades durante el diseño.

Sin embargo, parte de los objetivos de este proyecto es que la aplicación sea lo más escalable posible, de modo que aquellos usuarios que sean capaces de manipular el código fuente puedan añadir nuevos componentes de un modo sencillo. Por tanto, se busca que la aplicación cuente con patrones de diseño y buenas prácticas, además de documentación en el propio código, de manera que su modificación y adaptación por parte de los usuarios sea una tarea sencilla. Para facilitar este objetivo, la aplicación se distribuirá como software libre, y el código fuente estará disponible en repositorios públicos como GitHub[1].

## 1.3. Estructura

En primer lugar, se presenta una contextualización del problema y una breve discusión del estado del arte en el Capítulo 2, donde se introduce el modelado y simulación de sistemas digitales, así como algunos de los entornos de simulación más importantes para este tipo de tareas.

Posteriormente, se pasa a detallar la metodología de desarrollo seguida para este proyecto en el Capítulo 3, en el que se especifican los procesos de planificación ejecutados y sus resultantes cronogramas, así como un análisis formal de los requisitos de la aplicación a desarrollar.

A continuación, el Capítulo 4 describe la arquitectura de la aplicación, valiéndose de esquemas y diagramas UML generados durante el proceso de diseño. Además, se hace énfasis en los patrones de diseño y buenas prácticas utilizados, que contri-

[1]Puede verse el código en (Caponera De Cobellis, 2023a).

buyen a cumplir algunos de los objetivos mencionados en la sección anterior sobre mantenibilidad y escalabilidad de la aplicación.

El Capítulo 5 completa al Capítulo 4, mostrando cómo la arquitectura diseñada y los patrones seleccionados se traducen a la implementación del software objetivo, mostrándose las buenas prácticas de implementación empleadas.

Tras describir todos los pasos de diseño y desarrollo de la aplicación, en el Capítulo 6 se muestran las pruebas de calidad llevadas a cabo, junto con sus correspondientes resultados. De igual modo, se muestran también algunos sistemas de ejemplo modelados con la aplicación, comprobando su funcionalidad, y comparándolo con modelos desarrollados sin uso de la herramienta, para verificar que son análogos.

Finalmente, en el Capítulo 7 se presentan las conclusiones de este trabajo, destacando los puntos más importantes de los capítulos anteriores, y se proponen algunas posibles líneas de trabajo futuro basadas en el presente proyecto.

Por último, se presentan los Anexos A y B, que contienen un manual de usuario de la aplicación, y todo el código fuente de la aplicación generado, respectivamente.

# 2. Contexto y Estado del arte

## 2.1. Introducción

En este capítulo se realiza una contextualización del trabajo, en la que se describen, a alto nivel, qué son el modelado y simulación, por qué son importantes, y cómo se llevan a cabo actualmente, haciendo especial énfasis en el modelado de sistemas digitales, puesto que es el ámbito en el que se centra este trabajo. Se describe también el lenguaje Modelica, uno de los más importantes en el ámbito del modelado, y lenguaje en el que la aplicación exportará los resultados generados. Junto a este lenguaje, se citan también algunos de los entornos de desarrollo en Modelica más utilizados, como son Dymola u OpenModelica.

## 2.2. Conceptos fundamentales

Antes de adentrarse en la simulación de sistemas digitales, es conveniente comprender algunos conceptos fundamentales relativos al ámbito del modelado y la simulación, y hacer una breve clasificación de los mismos.

Tres de los conceptos más recurrentes e importantes en el ámbito del modelado son los de *sistema*, *experimento* y *modelo*. En general, cuando se realizan simulaciones y modelos, la finalidad es la de obtener conocimiento o información sobre un determinado objeto o conjunto de objetos. Son estos mismos objetos los que forman el sistema. Es decir, se denomina sistema a cualquier objeto cuyas propiedades se desean estudiar (Urquía Moraleda & Martín Villalba, 2017).

Dado, entonces, un sistema, obtener información sobre el mismo requiere la realización de distintas pruebas enfocadas a comprender el comportamiento del objeto en circunstancias o situaciones concretas que se deseen conocer, y extraer los resultados correspondientes. Esto es lo que se denomina experimento, es decir, el proceso de extraer datos de un sistema sobre el cual se ha ejercido una acción externa (Urquía Moraleda & Martín Villalba, 2017).

Dependiendo del sistema que se quiere estudiar, sin embargo, la experimenta-

ción puede ser muy costosa, difícil, o incluso imposible. Si, por ejemplo, se quieren realizar pruebas sobre un producto en fase de diseño, es decir, no aún ensamblado físicamente, la experimentación no es un método viable. Del mismo modo, existen experimentos que podrían llevar más tiempo del disponible debido a la velocidad a la que se producen ciertos cambios físicos, como en estudios geológicos o cosmológicos. Por lo tanto, es necesario encontrar otra metodología para la obtención de información sobre el sistema sin utilizar el propio sistema para la experimentación: el modelado. Se denomina como modelo a una representación de un sistema desarrollada para un propósito específico (Urquía Moraleda & Martín Villalba, 2017). Es importante matizar cómo, en esta definición, se hace referencia a un propósito específico. Esto se debe a que, en la práctica, es imposible modelar un sistema en todos sus aspectos, puesto que la cantidad de comportamientos a tener en cuenta es inmanejable, tanto para aquel que debe desarrollar el modelo, como para el ordenador que deba simularlo. Esto, sin embargo, no tiene por qué suponer un problema: si lo que se está desarrollando es, por ejemplo, un circuito digital, es suficiente con modelar las partes específicas relativas a la interacción eléctrica entre los distintos componentes, pero no es necesario contemplar las propiedades aerodinámicas de las piezas, puesto que queda fuera del objetivo del estudio. Así, al desarrollar un modelo, es importante definir qué información se desea conocer y, por tanto, qué propiedades deben ser modeladas y simuladas, y cuáles resultan superfluas.

Tal y como se muestra en la Figura 2.1, existen distintos tipos de modelos que se pueden utilizar para la obtención de información sobre un sistema. El modelo mental, el más básico de todos, se refiere a los razonamientos y pensamientos que, intrínsecamente, tienen los humanos sobre un determinado sistema, y que les permiten realizar experimentos mentales al respecto. Por ejemplo, al jugar al fútbol, el deportista posee un modelo mental, generado mediante la experiencia, de cómo se comporta el balón según la manera de golpearlo, y le permite hacer predicciones de dónde caerá y cuál será su trayectoria aproximada. Cuando se toma un modelo mental y se verbaliza, se convierte en un modelo verbal. Esta verbalización, que puede parecer inmediata, puede no resultar trivial en absoluto, puesto que definir con palabras y de manera precisa un comportamiento que se tiene interiorizado, como

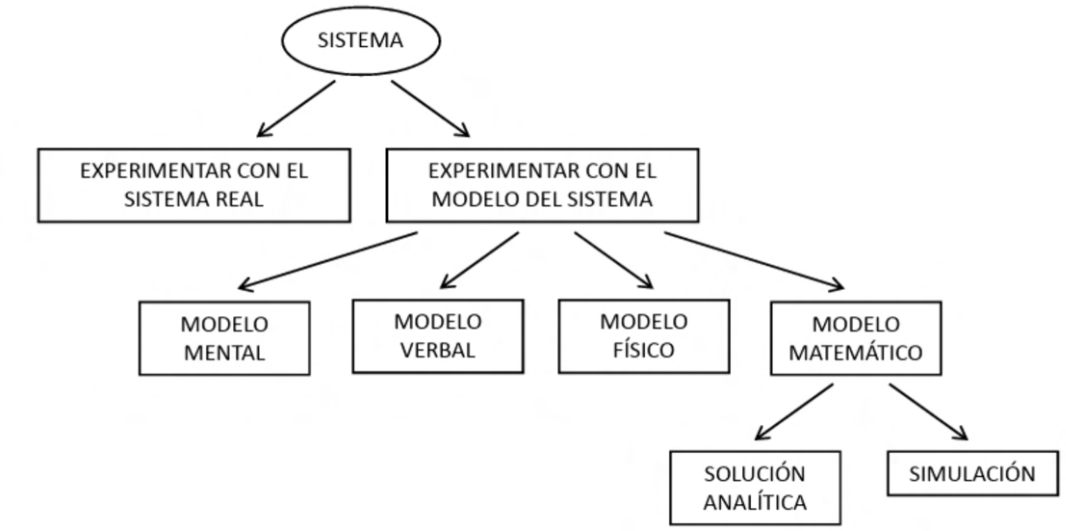



Figura 2.1: Formas de estudiar un sistema. Extraído de (Urquía Moraleda & Martín Villalba, 2017).

el del balón, puede llegar a ser muy complejo.

Tras ver estos modelos más "conceptuales", existen los modelos físicos, que traen al mundo físico un modelo del sistema. Por ejemplo, puede tratarse de una maqueta a escala, una versión reducida del producto, un prototipo enfatizado en determinadas propiedades, etc.

Por último, existe el modelado matemático, en el que se centra este trabajo. El modelado matemático puede verse, en cierto modo, como una ampliación del modelado verbal, en el sentido de que, en lugar de describir el sistemas con palabras, que pueden resultar vagas y poco precisas, se describe mediante ecuaciones, que no dan lugar a errores de interpretación, y que proporcionan resultados numéricos y fiables (al contrario que los modelos verbales, que ofrecen resultados más cercanos a lo cualitativo que a lo cuantitativo). En cierto modo, puede considerarse que la física en sí misma es, esencialmente, un modelo matemático del comportamiento del universo en su conjunto.

Estos modelos, generalmente, cuentan con soluciones analíticas que pueden ser calculadas matemáticamente. Sin embargo, es frecuente que estas soluciones sean muy complejas de calcular, debido a la complejidad del propio sistema y, por tanto,

a la de las ecuaciones, o incluso es posible que no exista un método de resolución analítica conocido, o se desconozca la existencia de una solución, como es el caso de las ecuaciones de Navier-Stokes, utilizadas para modelar el comportamiento de los fluidos, y para las cuales el determinar si existe siempre una solución analítica se considera, a día de hoy, uno de los 7 problemas del milenio aún sin resolver (Clay Mathematics Institute, 2000). Para estos problemas, pueden utilizarse métodos numéricos, es decir, valores muy cercanos a la solución analítica calculados mediante aproximaciones iterativas al valor real a través un ordenador, sin utilizar métodos analíticos para la resolución. Este tipo de resolución recibe el nombre de simulación, es decir, un experimento numérico realizado sobre el modelo matemático (Urquía Moraleda & Martín Villalba, 2017). Estos métodos, si bien no ofrecen soluciones exactas, pueden producir resultados con una precisión prácticamente arbitraria, haciendo que el error pueda ser despreciable.

Los modelos matemáticos pueden, a su vez, ser clasificados según la influencia del tiempo, en estáticos (aquellos en los que no interviene el tiempo) y dinámicos (aquellos en los que las propiedades se ven afectadas por el paso del tiempo). A su vez, los modelos dinámicos pueden ser clasificados según cómo se simule el paso del tiempo y la evolución de las propiedades, en:

- **Modelos de tiempo discreto.** En este tipo de modelo, el tiempo avanza de manera discreta, es decir, en intervalos regulares, y las propiedades del sistema pueden cambiar únicamente en esos instantes, permaneciendo constantes el resto del tiempo. Estos cambios de propiedades son denominados eventos.

- **Modelos de eventos discretos.** A diferencia de los modelos de tiempo discreto, en los modelos de eventos discretos el tiempo no avanza en intervalos regulares, sino que avanza por eventos. Es decir, se calcula un evento e, instantáneamente, se salta al siguiente, sin importar el tiempo que pase entre ambos eventos. Al igual que en el caso anterior, los cambios se producen de manera instantánea.

- **Modelos de tiempo continuo.** En este tipo de modelo, el tiempo trascurre de manera continua, de modo que la propiedad puede cambiar su valor en cual-

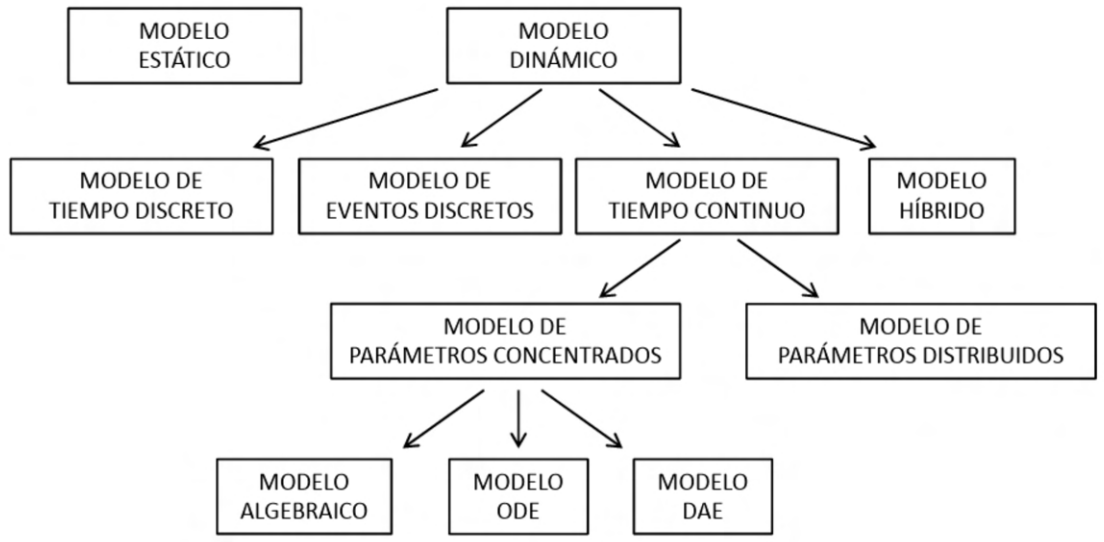



Figura 2.2: Algunas clasificaciones de los modelos matemáticos. Extraído de (Urquía Moraleda & Martín Villalba, 2017).

quier momento y de manera continuada en el tiempo, en lugar de producirse cambios instantáneos como en los casos anteriores. Es en este tipo de modelos en los que se centra este proyecto. Si bien existen clasificaciones de los modelos de tiempo continuo según el tipo de ecuaciones con los que se construyen, como se muestra en la Figura 2.2, tal nivel de profundidad escapa al alcance de este capítulo introductorio, por lo que no se hará mayor énfasis al respecto.

- **Modelos híbridos.** Cuando se combinan modelos de tiempo continuo con modelos de tiempo o eventos discretos, se obtiene un modelo híbrido. En estos modelos, algunas propiedades se tratarán mediante eventos o tiempo discreto, mientras que otras se tratarán mediante simulación de tiempo continuo.

## 2.3. Modelica

A principios de la década de los '90, empiezan a desarrollarse las primeras herramientas software y lenguajes de programación orientados a la simulación de sistemas modelados matemáticamente, con el objetivo de facilitar la descripción y construcción de modelos de todos los ámbitos, y no de un entorno concreto, es decir, lenguajes

de propósito general no ligados a ningún dominio específico. Puesto que distintos desarrolladores (generalmente ligados a instituciones académicas) desarrollaban distintos lenguajes, se llegó a un punto en el que la cantidad de herramientas distintas era enorme, y una misma librería se encontraba escrita en varios lenguajes debido a la imposibilidad de reutilizar código entre ellos, derrochando así tiempo y recursos de los desarrolladores. Además, al haber tantos lenguajes, la comunidad que utilizaba cada uno de ellos era muy reducida, minimizando así las ventajas derivadas de optimizaciones o mejoras en el lenguaje.

Para acabar con esta situación, en 1996 se establece un grupo de diseño cuyo fin es el de proponer un lenguaje de modelado estándar que permitiera la compatibilidad entre entornos de modelado y el intercambio de modelos entre desarrolladores, simplificando así la labor de todos los modeladores. El lenguaje propuesto, denominado Modelica, incorporó características de muchos de los lenguajes de modelado ya existentes, además de añadir funcionalidades propias como la composición de modelos. (Urquía Moraleda & Martín Villalba, 2017)

### 2.3.1. El lenguaje Modelica

Modelica es un lenguaje orientado al paradigma de modelado físico, una metodología para el modelado de sistemas físicos que se basa, en cierto modo, en la estrategia "divide y vencerás". La idea principal detrás del modelado físico es la de dividir el sistema en partes, cada una de ellas independiente de las demás, y definir el comportamiento de estas como si no existiera ningún otro componente. Posteriormente, se definen las interacciones entre las distintas partes, construyendo así el modelo completo. Por ejemplo, en el caso de los sistemas digitales, se puede definir por separado el comportamiento de un condensador y el de una fuente de tensión, mediante las ecuaciones que regulan el funcionamiento de cada uno y, a continuación, "conectarlos" y crear el circuito, definiendo las ecuaciones que regulen su interacción (por ejemplo, que la tensión de salida de la fuente deba ser igual a la tensión de entrada del condensador). Tras definir todas las ecuaciones que regulan el sistema, Modelica es capaz, si el modelo está correctamente definido, de determinar la causalidad computacional, es decir, el orden en el que debe calcular cada una de

las variables y desde cuál de las ecuaciones debe calcularlo.

Si bien describir todas las ecuaciones siguiendo este paradigma es viable, puede llegar a resultar confuso cuando los sistemas a modelar son complejos y el número de variables y ecuaciones crece, dando lugar a un código difícil de comprender y mantener. Para evitar esto, Modelica permite el modelado orientado a objetos que, en esencia, consiste en la composición de modelos, es decir, construir modelos a partir de otros modelos. Así, se puede crear un modelo para la fuente de tensión, otro modelo para el condensador, y un tercer modelo que represente al circuito, y que utilice los dos modelos anteriores. De este modo, no solo se facilita la reutilización del código, sino que se abre la puerta a la aplicación de principios y buenas prácticas provenientes de la programación orientada a objetos, que permiten crear modelos más robustos y con código de mayor calidad. Además, esto permite modularizar y jerarquizar los modelos y las librerías.

Una vez que los modelos "mínimos" (en el ejemplo, la fuente de tensión y el condensador) están definidos, Modelica ofrece el concepto de *conectores*, que permite realizar las conexiones entre varios modelos anidados. Esta conexión puede realizarse de dos modos diferentes: *across* o *through*. Cuando una conexión se realiza en modo *across*, el valor de todas las variables conectadas tiene que ser el mismo, por ejemplo, la fuerza ejercida por un muelle debe ser la misma que la "recibida" por el bloque al que está conectado; mientras que si la conexión se realiza como *through*, la suma de todas los valores de dicha conexión debe ser cero, por ejemplo en el nodo de un circuito electrónico, en el que la tensión entrante y la tensión saliente deben ser iguales en valor (y, al tener signo contrario, su suma vale 0).

Además, Modelica ofrece una serie de anotaciones que, escritas al lado de una variable, permiten documentar el modelo con comentarios, o especificar la representación gráfica del icono y del diagrama del modelo, haciendo posible la composición de modelos mediante un editor gráfico.

### 2.3.2. Entornos de desarrollo en Modelica

Como se ha mencionado con anterioridad, Modelica es un lenguaje estándar de modelado adoptado para facilitar el intercambio de modelos y reutilización de código

entre distintos desarrolladores. Sin embargo, no existe un único entorno de desarrollo estandarizado, sino que cada desarrollador es libre de utilizar el que prefiera de todos los que hay disponibles. Si bien existen un gran número de entornos de desarrollo, en este trabajo se estudian dos de los más importantes: *OpenModelica* y *Dymola*.

### OpenModelica

OpenModelica (Open Source Modelica Consortium, 2023) es uno de los entornos de desarrollo para Modelica más utilizados y más potentes hasta la fecha. Su principal característica es que es un proyecto gratuito y open source (puesto que está desarrollado por la OSMC — Open Source Modelica Consortium), lo que significa que el código fuente está disponible para los usuarios que deseen verlo o editarlo. Cuenta, por supuesto, con una interfaz gráfica de usuario intuitiva, que permite editar y componer de manera visual los modelos. Del mismo modo, incluye un compilador y simulador de modelos que permite ejecutar el código Modelica de manera sencilla directamente desde la aplicación, y visualizar los resultados de los experimentos llevados a cabo. Además de estas funcionalidades básicas, cuenta con una gran cantidad de funcionalidades avanzadas, como un debugger, que facilita enormemente la detección y corrección de errores en el código; un analizador de rendimiento que permite detectar posibles ineficiencias en el código; o soporte para el paralelismo, que permite una ejecución más rápida de los experimentos. Los desarrolladores, además, están comprometidos con el producto y la comunidad, por lo que ofrecen distintos notebooks, tutoriales y guías, además de la documentación base, para aprender a utilizar la herramienta, así como el propio lenguaje Modelica.

OpenModelica está formado por una serie de componentes que interactúan entre sí tal y como se muestra en la Figura 2.3. Sin estudiar en detalle todos los componentes, algunos de los más importantes y sus funcionalidades son:

- **OMC**. OpenModelica Compiler (OMC) es el compilador interactivo de OpenModelica. Este compilador es el que permite convertir el código Modelica a experimentos ejecutables y simulables, y es por ello que es el componente central de toda la aplicación. Junto a él trabajan estrechamente los componentes Simulation y Debugger que, como su nombre indica, son responsables de la

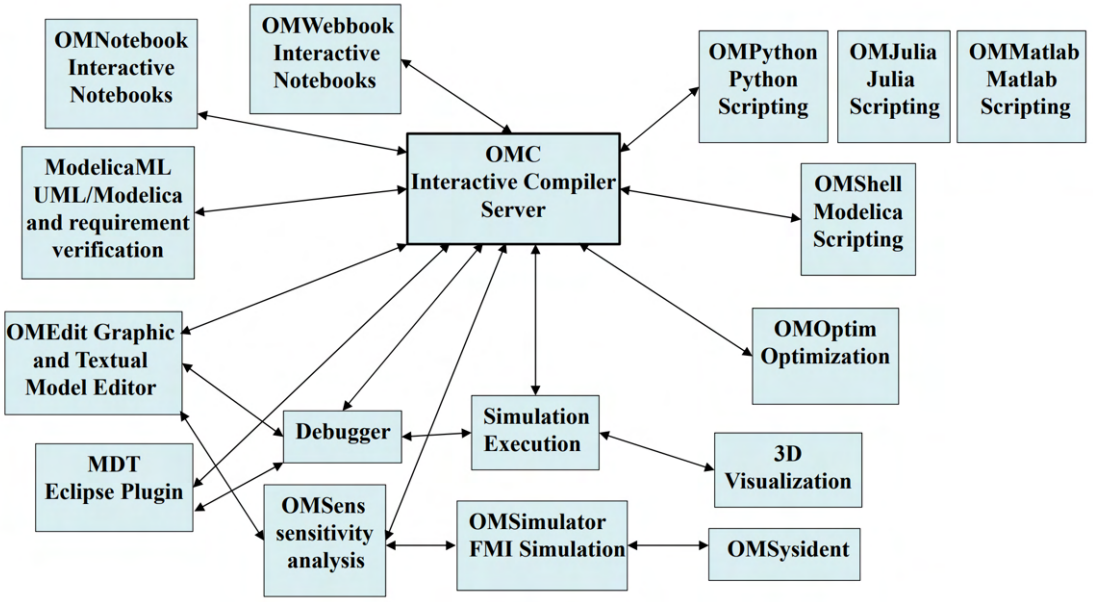



Figura 2.3: Componentes de OpenModelica y sus relaciones. Extraído de (Fritzson, 2018).

simulación y de ofrecer herramientas para la detección de errores, respectivamente.

- **OMEdit**. OMEdit es es un editor de modelos gráfico y textual, es decir, representa tanto la interfaz gráfica que permite la edición de modelos mediante Drag&drop, como la modificación del código Modelica dentro del propio programa.

- **OMShell**. Se trata de una consola de comandos que permite interactuar el modelo y/o el experimento mediante expresiones en lenguaje Modelica.

- **OMNotebook**. Es un entorno que permite utilizar el OMC y, por tanto, programar en lenguaje Modelica, desde notebooks al estilo de Jupyter Notebook. Estos Notebooks son, en esencia, ficheros que permiten intercalar texto e imágenes explicativos con celdas de código ejecutables. Algunos de los tutoriales interactivos de la OSMC se realizan mediante notebooks.

- **OMPython, OMJulia y OMMatlab** son extensiones que permiten programar funciones en los lenguajes Python, Julia y Matlab, respectivamente, e

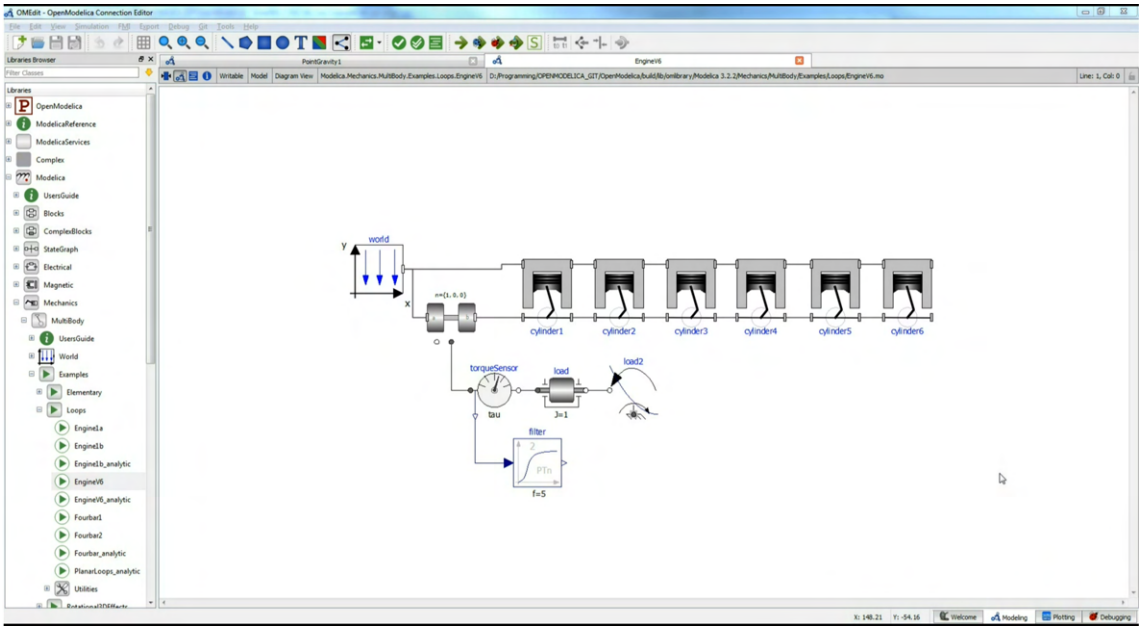



Figura 2.4: OpenModelica en funcionamiento. Extraído de (Fritzson, 2018).

invocarlas desde el código Modelica.

En la Figura 2.4 puede verse una captura de OpenModelica en funcionamiento, ejecutando una simulación de un motor de 6 cilindros.

### Dymola

Dymola (Elmqvist, 1978) es actualmente uno de los entornos de desarrollo más potentes y utilizados, si bien, a diferencia de OpenModelica, es un software propietario, es decir, de pago. Fue creado por Hilding Elmqvist como tesis doctoral en 1978, esto es, varios años antes de que existiera Modelica. Originalmente, Dymola (Dynamic Modeling Laboratory) estaba basado en uno de los muchos lenguajes entonces existentes, también llamado Dymola (Dynamic Modeling Language). Fue, de hecho, el propio Emqvist quien, en 1996, inició los esfuerzos para el desarrollo de un lenguaje común y estandarizado, Modelica, que se basa principalmente en el lenguaje Dymola, si bien se tienen en cuenta caracerísticas de otros lenguajes. Desde 2006, Dassault Systèmes es el propietario de Dymola, como parte de su suite de su suite de productos orientados a ingeniería de sistemas CATIA.

Las funcionalidades ofrecidas por Dymola son, a grandes rasgos, bastante simila-

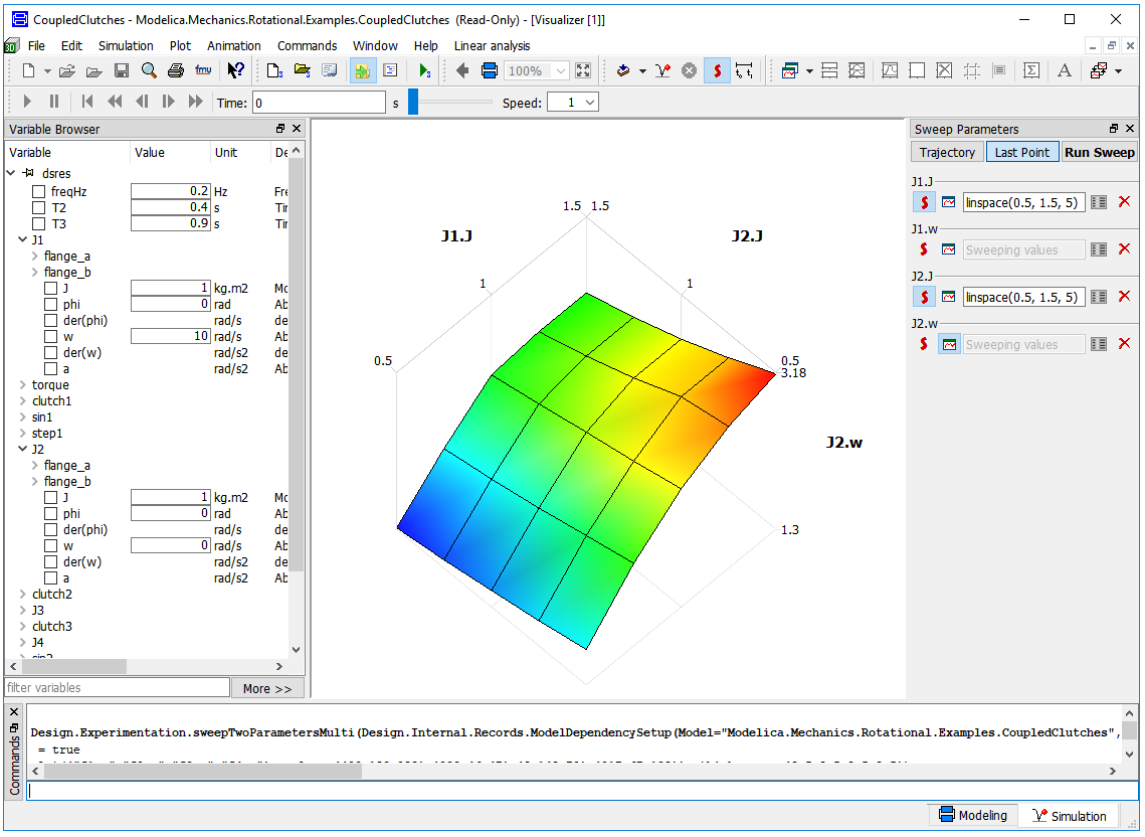



Figura 2.5: Dymola en funcionamiento. Extraído de (Dassault Systémes, 2023).

res a las que ofrece OpenModelica: ambos cuentan con un interfaz gráfico de usuario que permite construir y componer modelos de manera intuitiva y cómoda; cuentan con optimización de los modelos y características de debug, si bien lo hacen de maneras diferentes; soporte para la interoperabilidad con otros lenguajes como Python, etc. Dymola se centra en ofrecer un producto lo más pulido posible (cosa que, al ser un software propietario, es más sencillo de lograr que con un software de código abierto), y enormes librerías de modelos, tanto gratuitos como de pago, preparados para ser utilizados por los usuarios. Estas librerías incluyen una gran variedad de ámbitos, como la automoción, aeroespacial o robótica, entre otros.

Además, Dymola implementa ciertos algoritmos propietarios para optimizar la resolución de ecuaciones diferenciales, ofreciendo así un rendimiento superior para simulaciones en tiempo real en lo que Dassault Systémes denomina Hardware-in-the-Loop Simluations (HILS).

En la Figura 2.5 puede verse una captura de Dymola en funcionamiento.

## 2.4. Tecnologías utilizadas

En esta sección se estudian las tecnologías que se han utilizado para llevar a cabo la implementación del proyecto, y tiene la finalidad de introducir aquellas herramientas y frameworks más importantes para aquellos lectores que los desconozcan, con el objetivo de asegurar que secciones posteriores relativas a la implementación, como el Capítulo 5, en las que se profundizará en algunas de estas tecnologías, resulten más accesibles.

### 2.4.1. Java

Java (Oracle, 2021) es un lenguaje de programación de propósito general desarrollado durante los años '90 por James Gosling y posteriormente publicado como software propietario por Sun Microsystems, si bien actualmente se encuentra mantenido por Oracle. La principal característica que diferencia a Java de otros lenguajes de programación, como C/C++, es su orientación a la portabilidad: Java se diseñó con el objetivo Write Once, Run Anywhere (WORA), es decir, permitir que un código pueda ser ejecutado en distintas plataformas sin necesidad de modificarlo ni recompilarlo. Esto supone una enorme ventaja respecto a otros lenguajes como C, puesto que estos últimos requieren que el código sea compilado con un compilador específico según la arquitectura sobre la que se va a ejecutar el software y, en muchas ocasiones, requiere también modificaciones del código acordes a las características del dispositivo final. Java, sin embargo, hace uso de la Java Virtual Machine (JVM), una máquina virtual que es capaz de ejecutar el código Java independientemente de la arquitectura del computador sobre el que se ejecuta. De esta manera, el código Java se compila a un código binario comprensible por la máquina virtual y es esta última la que, a su vez, está compilada de modo diferente según la arquitectura, permitiendo que, una vez que el usuario tenga instalada la máquina virtual adecuada, todos los programas Java sean compatibles sin necesidad de modificación alguna. Gracias a esto, Java se ha convertido en uno de los lenguajes de programación más utilizados en software de todo tipo, habiendo sido durante muchos años el lenguaje

de programación principal para Android, hasta que en 2019 fue degradado a lenguaje secundario tras convertirse Kotlin (JetBrains, 2016) en el lenguaje principal. Es importante destacar, a este respecto, que Kotlin es un lenguaje que pretende agilizar la programación en comparación con Java, pero que es completamente intercambiable con el mismo, permite desarrollar software escrito parcialmente en Kotlin y parcialmente en Java, y puede ser ejecutado en la JVM sin dificultad.

Es por esta gran portabilidad que ofrece, junto con la versatilidad de las últimas características del lenguaje, por lo que se ha optado por utilizar Java para el desarrollo de este proyecto.

### 2.4.2. Maven

Maven (The Apache Software Foundation, 2002) es una herramienta de gestión de proyectos para Java, cuya finalidad es simplificar la documentación, el proceso de compilación y la utilización de librerías externas mediante un fichero denominado Project Object Model (POM). Este fichero, que utiliza un formato similar al XML para su definición, permite configurar distintas opciones de compilación, definir tests unitarios, y añadir dependencias, plugins y librerías a los proyectos. De este modo se evita la necesidad de descargar y configurar manualmente las librerías que se quieren utilizar y se reducen enormemente los problemas derivados de esto (errores de compilación o enlazado de las librerías, que el sistema no detecte correctamente las librerías por su ubicación,...). Maven cuenta con un repositorio que contiene una enorme cantidad de librerías y frameworks para Java, de modo que, para utilizar cualquiera de ellos, basta con especificar en el fichero POM la librería que se quiere utilizar para que Maven la descargue y configure. Además, puesto que es una tecnología ya instaurada, utilizada en infinidad de proyectos Java, la mayoría de entornos de desarrollo cuentan con soporte para Maven, facilitando aún más su utilización.

Dentro de este proyecto, librerías como Project Lombok o Derby, que se detallarán más adelante, han sido incluidas utilizando Maven.

### 2.4.3. Project Lombok

Project Lombok (The Project Lombok Authors, 2022) es una librería de Java que busca reducir el tiempo empleado por los desarrolladores para escribir código "vacío" y repetitivo, es decir, aquellas partes del código que, si bien son necesarias, son también triviales, puesto que no contienen lógica alguna, y que por tanto pueden ser automatizadas de manera sencilla y eficaz. El ejemplo más sencillo de esto son los Getters y Setters, es decir, fragmentos de código destinados a leer o escribir el valor de una variable en una clase. Estos fragmentos de código son, generalmente, muy sencillos, con un cuerpo de apenas una línea. Sin embargo, definir Getters y Setters para todos los atributos de una clase puede llegar a consumir un tiempo considerable.

De manera similar, las clases en Java cuentan con los métodos `equals` y `hashCode`, cuya función es muy importante: el método `equals` toma como parámetro otro objeto y devuelve un valor booleano que representa si ambos objetos son iguales o no. `hashCode`, por su parte, genera un código hash del objeto que debe ser único (o, al menos, con un bajo índice de colisiones) para cada objeto, en función de los valores de sus distintos parámetros. Si bien puede parecer que estos métodos deban ser escritos manualmente la realidad es que existen técnicas que permiten automatizar su generación, ahorrando tiempo y esfuerzo a los desarrolladores. Como último ejemplo, los constructores de las clases son, en muchas ocasiones, un simple conjunto de asignaciones, en los que se inicializa cada variable con el valor correspondiente pasado por parámetro. Si bien, generalmente, las clases cuentan con constructores "inteligentes", que realizan muchas otras funciones, es frecuente que las clases cuenten con, al menos, un constructor vacío o un constructor trivial que no realiza ninguna operación lógica. En este caso también, tiene sentido automatizar la generación de los constructores, y ahorrar tiempo y esfuerzo a los desarrolladores.

Por todos estos motivos, y otros, surge Project Lombok, con el fin de reducir el tiempo que los desarrolladores emplean en escribir código "trivial", para que puedan centrar sus esfuerzos en la lógica de negocio. Project Lombok cuenta con una serie de anotaciones que, como se verá más detenidamente en la sección 5.2, permiten

construir todos estos métodos, y más, con tan solo una línea de código. En Java, se denomina anotación a algunos modificadores del comportamiento de variables, métodos o clases, que se indican mediante el símbolo @. La anotación más frecuentemente utilizada es `@Override`, que se coloca sobre la cabecera de un método para indicar que sobrescribe a otro método homónimo de una clase a la que extiende o interfaz a la que implementa; sin embargo, existen anotaciones de todo tipo, y los desarrolladores pueden diseñar sus propias anotaciones y asignarles el comportamiento deseado. En Project Lombok, por ejemplo, una de las anotaciones más utilizadas es `@Getter`. Esta anotación, aplicada sobre una variable, genera de manera automática el Getter para la variable en cuestión. La misma anotación, si aplicada sobre una clase, genera métodos Getter para todas las variables que contiene. De manera análoga, existen anotaciones para una gran multitud de funciones, como `@Setter`, `@EqualsAndHashcode` o `@Data`.

### 2.4.4. JPA, EclipseLink y Derby

En general, la interacción con las bases de datos se hace mediante consultas en el lenguaje Structured Query Language (SQL), que permite extraer información de las distintas tablas almacenadas en el disco. Sin embargo, realizar las consultas y, posteriormente, traducir esa información a uno o varios objetos de Java, puede resultar una tarea tediosa si se realiza utilizando las herramientas estándar del lenguaje. De esta necesidad surge Jakarta Persistence API (JPA) [1], una API (Application Programming Interface o Interfaz de Programación de Aplicaciones, en español) diseñada específicamente para simplificar la interacción con las bases de datos mediante el paradigma de programación orientada a objetos de Java, sin necesidad de realizar consultas SQL básicas (como leer uno o todos los elementos, eliminar un elemento, actualizar un elemento,...) de manera directa, a no ser que se requieran consultas más complejas que no puedan ser construidas de manera automática. Incluso para

[1] JPA era conocido como Java Persistence API hasta 2019, cuando Oracle cedió los derechos a la Eclipse Foundation, y esta tuvo que cambiarle el nombre por ser Java una marca registrada. A pesar de ello, es frecuente que los programadores sigan refiriéndose a JPA como Java Persistence API.

estas últimas consultas, JPA define diversas funciones en Java que permiten construir la consulta sin conocer el lenguaje SQL[2]. Esto es conseguido, al igual que en el caso de Project Lombok, mediante anotaciones, que permiten identificar las clases como entidades de la base de datos, anotar las distintas relaciones entre objetos, o definir qué variables deben ser almacenadas y cuales no. Al igual que en el caso anterior, estas anotaciones se detallan en la sección 5.2.

Nótese, sin embargo, que, como el propio nombre indica, JPA es una API, es decir, una interfaz. Esto significa que define una serie de métodos y anotaciones estándar, pero no una implementación para los mismos. Es por ello que existen diversas implementaciones de JPA que, si bien cumplen la especificación, pueden contener diferencias relativas a la implementación que hagan variar el resultado ligeramente en ciertos casos concretos, sobre todo en términos de efectos secundarios. Dos de las implementaciones más utilizadas son Hibernate (Red Hat, 2001) y EclipseLink (The Eclipse Foundation, 2022), siendo esta última la utilizada en el desarrollo del presente proyecto. Estas implementaciones, frecuentemente denominadas Object-Relational Mapping (ORM), son las que realmente llevan a cabo las operaciones solicitadas por el cliente.

Una vez anotadas las distintas clases y variables de acuerdo con la interfaz definida por JPA, el ORM es capaz de construir, automáticamente, las tablas y relaciones necesarias, así como las distintas consultas para las operaciones básicas a realizar, ahorrando esfuerzo y tiempo a los desarrolladores. Sin embargo, un ORM no es un Sistema Gestor de Bases de Datos (SGDB), por lo que su función es la de crear las consultas, no la de ejecutarlas. Es necesario, por tanto, incluir también un SGDB en el proyecto que es, en esencia, el componente que ejecutará las consultas y persistirá los datos de la base de datos. Una vez más, existen multitud de implementaciones de SGDBs para Java, sin embargo se ha optado por utilizar Apache Derby (The Apache

[2]Cabe mencionar que, si bien es posible utilizar JPA para realizar consultas complejas, en general se considera que es menos eficiente que realizar las propias consultas SQL, puesto que las funciones ofrecidas no son lo suficientemente específicas como para resultar eficientes y no generar problemas en todos los casos. A pesar de ello, y puesto que en esta aplicación se realizan únicamente consultas simples, resulta práctico el uso de JPA.

Foundation, 2022), en primer lugar, por ser un SGDB ligero y sencillo, ideal para proyectos sin grandes demandas de rendimiento ni consultas complejas; y en segundo lugar, por permitir el uso de bases de datos embebidas. Esto significa que, a diferencia de otros SGDBs, que requieren que la propia base de datos esté configurada en el equipo, una base de datos embebida es gestionada enteramente por la aplicación y almacenada en un directorio local, de modo que se maximiza la portabilidad de la aplicación al no requerir la presencia de un SGDB preinstalado en el equipo, o tener que instalarlo al instalar la propia aplicación. De este modo, y como se verá en secciones posteriores, cuando se inicia la aplicación, esta crea un directorio en la misma ubicación en la que se encuentre, denominado "lizardClipsEmbeddedDatabase", que contendrá la base de datos necesaria para el funcionamiento del software.

Así, mediante los tres componentes descritos, es posible generar una base de datos e interactuar con ella de manera casi inmediata, sin necesidad de generarla ni actualizarla manualmente.

### 2.4.5. Swing

Java, a diferencia de otros lenguajes de programación anteriores, se diseñó teniendo en cuenta la necesidad de una interfaz gráfica, y se considera como el método principal de interacción con la aplicación. Es por esto que, dentro de los componentes estándar del lenguaje, se encuentra Swing, un framework gráfico que permite crear interfaces de usuario sencillas mediante código. Por supuesto, existen librerías externas más potentes y avanzadas, pero Swing es más que suficiente para llevar a cabo desarrollos con interfaces sencillas. Swing es parte, junto a Abstract Window Toolkit (AWT) y Java 2D, de las Java Foundation Classes (JFC), un conjunto de clases que forman un framework gráfico completo para la construcción de interfaces de usuario.

### 2.4.6. IntelliJ IDEA

Para cualquier desarrollo moderno, es necesario el uso de un entorno de desarrollo integrado, o Integrated Development Envoironment (IDE) en inglés. Un IDE es una

herramienta software que permite gestionar de manera unificada todos los ficheros de un proyecto, además de ofrecer funcionalidades de ayuda al desarrollador, como el autocompletado de código, sugerencias, detección de errores en tiempo real (es decir, mientras se está escribiendo el código, sin necesidad de compilarlo), y muchas otras. Sin el uso de un IDE, por ejemplo, compilar requeriría de conocer los comandos necesarios para utilizar el compilador, introducirlos en la consola y ejecutarlos. Sería necesario también, según el lenguaje, enumerar todos los ficheros o librerías que se quieren incluir en la compilación, puesto que de otro modo no se usarían, lo cual reduce en gran medida la viabilidad de este método para proyectos con un gran número de ficheros. Los IDEs, sin embargo, permiten compilar y ejecutar proyectos de enormes dimensiones pulsando un único botón. Además, los IDEs incluyen, generalmente, funcionalidades de debug, como los breakpoints, que permiten detener la ejecución del software en cualquier punto del código y examinar los valores de las variables, o avanzar sentencia a sentencia, para facilitar el debug y la búsqueda y corrección de errores.

Para el lenguaje Java existen multitud de IDEs con grandes capacidades, como NetBeans (The Apache Software Foundation & Oracle, 2000) o Eclipse (IBM & The Eclipse Foundation, 2001); sin embargo, uno de los más populares y preferidos de los desarrolladores, por su versatilidad y la potencia de su sistema para el autocompletado y sugerencia de código, es IntelliJ IDEA. Este software se encuentra disponible en dos versiones distintas: la Community Edition (JetBrains, 2023a), que está disponible de modo gratuito y que cuenta únicamente con la funcionalidad básica; y la versión Ultimate (JetBrains, 2023b), que cuenta con funcionalidades avanzadas como la generación de diagramas UML, herramientas para JPA y soporte para otras librerías externas no incluidas en la versión Community Edition. Esta versión, sin embargo, no es gratuita.

Para el desarrollo de este proyecto, se han utilizado ambas versiones: durante la mayor parte del proyecto se ha utilizado la versión Community Edition; sin embargo, durante las fases finales, se ha hecho uso de la prueba gratuita de la versión Ultimate, con el fin de aprovechar algunas de las funcionalidades exclusivas para mejorar la calidad del producto y generar algunos de los diagramas que se muestran en este

documento.

## 2.5. Conclusiones

Durante este capítulo se ha contextualizado la temática principal de este trabajo: la simulación y modelado. Tras describir, brevemente, los conceptos fundamentales de este ámbito, se ha tratado el lenguaje Modelica, el actual estándar para la definición de modelos para simulación. Además de describir el propio lenguaje, se han expuesto dos de los entornos de desarrollo más utilizados actualmente: OpenModelica y Dymola. Finalmente, se han enumerado y descrito las distintas tecnologías que se han utilizado para llevar a cabo este proyecto, que comprenden Java, como lenguaje de programación, junto con Maven para la gestión de paquetes y Project Lombok para agilizar el desarrollo; JPA como API para la base de datos; Swing como entorno gráfico; e IntelliJ IDEA como IDE.

# 3. Análisis y planificación

## 3.1. Introducción

Al igual que en cualquier proyecto de ingeniería, en el desarrollo de software es de gran importancia planificar con antelación y de manera precisa qué se va a llevar a cabo, cuándo se realizará cada tarea y cómo se hará. Del mismo modo que un albañil no debe empezar a colocar los ladrillos hasta tener los planos del proyecto, un ingeniero de software tampoco debería empezar a escribir código sin tener antes una visión global de lo que se debe hacer, y una planificación del proyecto. En el presente capítulo se detallan las fases de análisis y planificación seguidas para el desarrollo del proyecto, que han permitido una adecuada ejecución del mismo.

## 3.2. Análisis de requisitos

En primer lugar, se ha realizado un análisis de requisitos, estudiando y especificando qué es, exactamente, lo que se requiere de este proyecto. Tan solo plasmando por escrito y de la manera más concreta posible todos los requisitos es posible verificar que, efectivamente, el trabajo realizado se corresponde con el deseado y, no menos importante, con el acordado, puesto que al ser un documento firmado también por el cliente, protege al desarrollador ante posibles "cambios de opinión" por parte del cliente que pudieran afectar a la aceptación del mismo[1]. Los requisitos identificados se muestran listados a continuación, divididos en requisitos funcionales y no funcionales. Además, se ha asignado un código a cada requisito, de modo que resulte más sencillo su seguimiento y verificación.

[1] Al contarse con una lista de requisitos aprobada por el cliente, este no podrá rechazar la entrega de un producto que cumpla las especificaciones, ni podrá modificarlas durante el avance del proyecto. Generalmente, el desarrollador debe intentar adaptarse a las peticiones del cliente aunque estas se realicen fuera de la fase de análisis, sin embargo este documento le garantiza que no está obligado a hacerlo y que, si los cambios son demasiado grandes, puede rechazarlos (o renegociar los costes del desarrollo).

### 3.2.1. Requisitos funcionales

Los requisitos funcionales son aquellos que indican qué debe hacer el sistema, o cuáles son los objetivos que debe cumplir a nivel de funcionalidades. Por ejemplo, "El sistema debe permitir al usuario iniciar sesión mediante correo electrónico y contraseña" sería un requisito funcional, mientras que "La conexión con el servidor debe tener una latencia máxima de 15ms" no lo sería, puesto que no especifica una funcionalidad del sistema.

Se enumeran aquí los requisitos funcionales especificados, identificados mediante el prefijo RF (Requisito Funcional) para distinguirlos más fácilmente de los requisitos no funcionales.

**RF-1 Gestión de circuitos.** Debe ser posible la gestión de circuitos desde una base de datos interna a la aplicación.

**RF-1.1 Conexión a una base de datos.** La aplicación debe poder establecer una conexión con una base de datos que permita la lectura y escritura de los circuitos generados.

**RF-1.2 Guardado de circuitos.** La aplicación debe permitir al usuario guardar los circuitos en la base de datos, asignando a cada circuito un nombre.

**RF-1.3 Guardado de circuitos con distinto nombre.** La aplicación debe permitir guardar el circuito que se esté editando con un nombre distinto al que ya tenga asignado.

**RF-1.4 Carga de circuitos.** La aplicación debe permitir la carga de los circuitos guardados en la base de datos.

**RF-1.5 Edición de circuitos.** Tras cargar un circuito, el sistema debe permitir su modificación y posterior guardado, bien sobrescribiendo la versión anterior del modelo, o guardándolo con un nuevo nombre.

**RF-1.6 Validación de nombres.** Cuando se intente guardar un circuito, la aplicación debe verificar que no existe otro circuito con el mismo nombre y, en caso de haberlo, mostrar un mensaje de error.

**RF-2 Composición de circuitos.** La aplicación debe permitir componer circuitos digitales.

**RF-2.1 Área de trabajo.** La aplicación debe contar con un área de trabajo sobre la que se puedan componer los circuitos. Este área de trabajo deberá mostrar de manera esquemática el circuito que se está componiendo.

**RF-2.2 Paleta de componentes.** El sistema debe contar con una paleta de componentes extraídos de la librería estándar de Modelica, disponibles para ser utilizados por el usuario durante la composición de circuitos.

**RF-2.3 Colocación de componentes.** La aplicación debe permitir al usuario añadir componentes en el área de trabajo, y debe ser posible cambiar su posición una vez el componente ya esté colocado.

**RF-2.4 Rotación de componentes.** La aplicación debe permitir rotar los componentes colocados en el área de trabajo, en intervalos de 90°.

**RF-2.5 Conexión de componentes.** La aplicación debe permitir conectar entre sí los distintos componentes del área de trabajo, siempre que se trate de una conexión válida. En caso contrario, debe mostrar una alerta indicando por qué no puede realizarse la conexión solicitada. Se considera válida una conexión si se realiza entre un conector de salida y uno de entrada de componentes distintos, no estando el conector de entrada conectado a ningún otro componente.

**RF-2.6 Borrado de componentes.** La aplicación debe permitir al usuario eliminar componentes del área de trabajo. Si el componente está conectado a otros componentes, se deben eliminar también las conexiones que los unen.

**RF-2.7 Borrado de conexiones.** La aplicación debe permitir eliminar conexiones del área de trabajo.

**RF-2.8 Desplazamiento en el área de trabajo.** El sistema debe permitir al usuario desplazarse por el área de trabajo.

**RF-2.9 Acercamiento/alejamiento.** La aplicación debe permitir al usuario

acercar o alejar el área de trabajo, permitiendo así una visión general de todo el sistema, o una visión centrada en una zona concreta del mismo.

**RF-2.10 Modificación de las propiedades.** El sistema debe permitir al usuario modificar las propiedades o atributos de cada uno de los componentes de manera independiente. Estas propiedades incluyen el nombre del componente, que se utilizará posteriormente durante la generación de código; así como los atributos propios de cada componente, según establecido en la librería estándar de Modelica.

**RF-3 Generación de código.** El sistema debe ser capaz de generar código Modelica ejecutable.

**RF-3.1 Visualización de código.** La aplicación debe ofrecer la posibilidad de visualizar el código generado en una ventana interna a la misma. Durante la visualización, se debe permitir al usuario copiar el código mostrado. El código debe utilizar syntax highlighting, es decir, debe "colorear" el código, indicando visualmente cuáles son las palabras reservadas del lenguaje, para facilitar la lectura del código.

**RF-3.2 Exportación del código.** El sistema debe permitir la exportación del código generado a un fichero Modelica con extensión .mo, que podrá ser posteriormente abierto por editores de código Modelica.

**RF-3.3 Generación de diagramas.** Al generar el código, la aplicación debe incluir las anotaciones necesarias para que, al abrir el fichero en un editor con funcionalidades gráficas, se muestre un esquema del circuito lo más parecido posible al mostrado durante la composición del modelo.

**RF-3.4 Validación de nombres.** Al modificarse los nombres de los componentes, el sistema debe verificar que los nombres son válidos como nombres de variables, es decir, que no contienen espacios, no empiezan con un número ni una letra mayúscula, etc.

**RF-3.5 Integración con otras aplicaciones.** Tras exportar el código Mode-

lica, el sistema debe ofrecer al usuario la posibilidad de abrir el fichero generado y, en caso de aceptarse la petición, solicitar al sistema operativo la apertura del fichero con una aplicación adecuada, en caso de que haya alguna instalada en el equipo.

### 3.2.2. Requisitos no funcionales

A diferencia de los requisitos funcionales, los requisitos no funcionales son aquellos que describen cómo debe llevarse a cabo una acción, en lugar de describir la acción en sí. Por ejemplo, un requisito funcional podría ser "El sistema debe utilizar conexiones seguras", mientras que un requisito no funcional asociado puede ser "El sistema debe utilizar TLS para garantizar la seguridad de las comunicaciones".

Se enumeran aquí los requisitos no funcionales especificados, identificados mediante el prefijo RNF (Requisito No Funcional) para distinguirlos más fácilmente de los requisitos funcionales.

**RNF-1 Interacción mediante Drag&drop.** La interacción con el sistema y, especialmente, con el área de trabajo, debe hacerse mediante Drag&drop, de modo que los componentes sean arrastrados a sus respectivas posiciones para colocarlos o moverlos.

**RNF-2 Base de datos embebida.** Para gestionar la lectura/escritura de circuitos, la aplicación debe contar con una base de datos embebida, esto es, local en el equipo en el que se ejecuta.

**RNF-3 Validaciones del modelo.** Antes de proceder a la generación del código (ya sea para su visualización o para la exportación a un fichero), la aplicación debe comprobar que el modelo cumple ciertos requisitos mínimos de validez, e informar del error de lo contrario. Por ejemplo, un modelo en el que algún componente tiene conectores de entrada vacíos no será ejecutable, por lo que la aplicación informa de ello mediante un mensaje de warning, si bien permite al usuario proceder con la exportación.

**RNF-4 Disponibilidad multiplataforma.** La aplicación debe estar disponible para los principales sistemas operativos considerados, en este caso, Microsoft Windows, Linux y Apple MacOs. Para ello, debe desarrollarse en un lenguaje multiplataforma, como es Java.

**RNF-5 Internacionalización.** El sistema debe estar diseñado para poder ser utilizado en varios idiomas mediante herramientas de internacionalización. Durante el desarrollo del proyecto se implementarán únicamente los idiomas español e inglés, aunque debe permitirse la ampliación de estos idiomas de un modo sencillo.

**RNF-6 Herramientas de ayuda visual.** El sistema debe contar con herramientas de ayuda visual (o Tooltips en inglés), que permitan al usuario obtener más información sobre los distintos componentes, el significado de los campos que debe rellenar, etc.

## 3.3. Metodología de trabajo y planificación temporal

Para llevar a cabo este proyecto, se opta por utilizar una metodología ágil, que proporciona un desarrollo incremental y facilita la eventual modificación de requisitos, permitiendo adaptarse mejor a las necesidades en cualquier fase del desarrollo. Este tipo de metodologías se basan en realizar distintas iteraciones y, dentro de cada una de las iteraciones, ejecutar todas las fases del desarrollo: análisis de los requisitos asociados, diseño, implementación y pruebas. Al final de cada iteración se cuenta con un prototipo que debería implementar completamente algunas de las funcionalidades deseadas, es decir, las funcionalidades en las que se haya centrado dicha iteración deben estar terminadas, salvo cambios futuros en las especificaciones. Por supuesto, antes de realizar ninguna de las iteraciones, se ha realizado un análisis global del proyecto y definido, a rasgos generales, cuál podría ser una buena arquitectura para la aplicación final, sin hacer hincapié en los detalles.

Para el proyecto presente, se realizan las siguientes iteraciones o *sprints*:

**IT-1 Inicio del proyecto y área de trabajo.** Durante la primera iteración, se genera la estructura del proyecto, configurando las librerías y frameworks necesarios; y se crea una primera versión del área de trabajo. En esta versión se implementa la lógica relacionada con el Drag&drop, la interconexión de componentes, y el borrado de componentes y conexiones; así como el desplazamiento por el área de trabajo. En esta fase, no se trabaja con componentes reales de la librería Modelica, sino con mocks que sirven únicamente para probar la funcionalidad desarrollada.

La iteración puede dividirse en las siguientes tareas:

**T-1 Creación del proyecto.** Crear el proyecto, configurar los frameworks y librerías necesarias, y crear un repositorio para el control de versiones. También se implementarán los métodos necesarios para la internacionalización de la aplicación, de modo que todos los textos de la aplicación se añadan conforme a estos métodos, facilitando su posterior traducción a otros idiomas.

**T-2 Diseño de la estructura del circuito.** Definir cómo se representa internamente el circuito

**T-3 Creación del área de trabajo.** Se diseña e implementa el área de trabajo, de modo que permita las funcionalidades de Drag&drop, desplazamiento y acercamiento/alejamiento.

**T-4 Interacción con los componentes.** Se añade al área de trabajo la funcionalidad que permite interactuar con los componentes, esto es, moverlos, eliminarlos, rotarlos, o seleccionarlos (para, posteriormente realizar acciones con ellos, como editar las propiedades).

**T-5 Conexiones.** Se añade al área de trabajo la funcionalidad que permite conectar componentes entre sí, con las consiguientes validaciones requeridas.

Los requisitos satisfechos durante esta iteración son: **RF-2.1**, **RF-2.3**, **RF-2.4**, **RF-2.5**, **RF-2.6**, **RF-2.7**, **RF-2.8**, **RF-2.9**, **RNF-1** y **RNF-4**. Al final

de esta iteración, es posible crear un circuito completo (con los componentes ficticios utilizados).

**IT-2 Diseño e implementación de la base de datos.** Durante esta iteración los esfuerzos se centran en diseñar e implementar la base de datos que permita el guardado y carga de los circuitos modelados. Puede dividirse en las siguientes tareas:

**T-6 Diseño de la base de datos.** Diseñar la estructura necesaria para que la base de datos pueda almacenar correctamente los circuitos modelados.

**T-7 Implementación de la base de datos.** Implementar en el proyecto la base de datos, generando los métodos de interacción necesarios.

**T-8 Integración del circuito con la base de datos.** Modificar los componentes del circuito de modo que sean compatibles con la base de datos, y generar los métodos para el guardado y carga de los mismos.

Los requisitos satisfechos durante esta iteración son: **RF-1.1**, **RF-1.2**, **RF-1.3**, **RF-1.4**, **RF-1.5**, **RF-1.6** y **RNF-2**. Al final de esta iteración, es posible cargar, guardar y editar circuitos en la base de datos.

**IT-3 Paleta de componentes.** A partir de este punto, empiezan a añadirse componentes a la paleta de la aplicación. A diferencia del resto de iteraciones, la tarea principal contenida en esta iteración, **T-11**, no se realiza completamente en esta iteración, sino que se ejecuta en paralelo al resto de tareas de iteraciones posteriores, añadiendo los componentes de manera paulatina durante todo el proceso.

Esta iteración se divide en las siguientes tareas:

**T-9 Selección de componentes.** Se seleccionan qué componentes o paquetes, de aquellos presentes en la librería estándar de Modelica, se implementan en la aplicación.

**T-10 Implementación de la paleta.** Se diseña el elemento paleta y se añade al área de trabajo. Además, se diseña de modo que los componentes y

paquetes mostrados sean generados dinámicamente, de modo que no sea necesario modificar el componente. Se hace uso de Tooltips para facilitar su uso.

**T-11 Implementación de componentes.** Esta tarea, que se ejecuta a lo largo del resto de iteraciones (si bien se realizan dentro de esta iteración la mayoría de los cambios), consiste en la adición de los componentes de la librería estándar de Modelica a la paleta de la aplicación.

Los requisitos satisfechos durante esta iteración son: **RF-2.2**, **RNF-6**. Al final de esta iteración, es posible utilizar los componentes de la librería estándar de Modelica en la composición de circuitos.

**IT-4 Propiedades de los componentes.** En esta iteración, se desarrolla la lógica que permite que los distintos componentes tengan propiedades o atributos configurables por el usuario y que sean generados dinámicamente según el tipo de componente. Esto se realiza dividido en las siguientes tareas:

**T-12 Diseño de las propiedades.** Se modifica la arquitectura de la aplicación de modo que cada componente tenga las propiedades correspondientes a su clase.

**T-13 Implementación de las propiedades.** Se implementan los cambios diseñados en la tarea anterior, y se generan las ventanas necesarias para que el usuario pueda modificar las propiedades de manera sencilla. También se modifica la base de datos para que almacene las propiedades. Las propiedades implementan Tooltips para facilitar su uso.

**T-14 Implementación de las propiedades: casos especiales.** Dentro de la implementación, se encuentran dos casos especiales de propiedades: el nombre, que debe estar presente para todas las piezas y que regula el nombre de la variable Modelica, a diferencia del resto de propiedades, que producen un parámetro; y el número de conectores, puesto que algunos componentes pueden contar con un número variable de conectores de entrada o salida, y esto debe verse reflejado en el área de trabajo.

Los requisitos satisfechos durante esta iteración son: **RF-2.10**, **RNF-6**. Al final de esta iteración, es posible modificar y persistir los atributos o propiedades de los componentes.

**IT-5 Generación de Código Modelica.** Esta iteración se centra en desarrollar la lógica relacionada con la generación de código Modelica, tanto para su visualización como para su exportación. Para llevarlo a cabo, se realizan las siguientes tareas:

**T-15 Generalidades.** Se implementan las funciones básicas de generación de código, así como la generación de código "simple" e independiente del circuito, como los imports, la estructura general de los ficheros, etc.

**T-16 Generación de código para los componentes.** Se generan los fragmentos de código asociados con las declaraciones de los componentes del circuito, sus parámetros, o la asignación de nombres unívocos a las variables.

**T-17 Generación de código para las conexiones entre componentes.** Se generan los fragmentos de código asociados a las conexiones entre componentes.

**T-18 Generación de diagramas.** Se generan los fragmentos de código asociados con la representación visual de Modelica, haciendo que al abrir el código generado en un editor que lo soporte, se muestre un esquema del circuito tan similar como sea posible al mostrado durante la composición del mismo.

**T-19 Visualización del código.** Se añade una ventana dentro de la aplicación para la visualización del código generado, con las correspondientes validaciones previas; así como las funciones de syntax highlighting necesarias para que el código se muestra de manera visualmente más atractiva.

**T-20 Exportación de código.** Se añade la funcionalidad que permite la exportación de un fichero .mo con el código generado, realizándose las validaciones pertinentes antes de proceder. Se añade también la integración

con otras aplicaciones, de modo que sea posible abrir el código generado en un editor externo.

Los requisitos satisfechos durante esta iteración son: **RF-3.1**, **RF-3.2**, **RF-3.3**, **RF-3.4**, **RF-3.5**, **RNF-3**. Al final de esta iteración, es posible visualizar y exportar el código Modelica correspondiente al circuito modelado.

**IT-6** **Internacionalización.** En esta iteración, se traducen todos los textos de la aplicación a los idiomas especificados, cumpliéndose así con el requisito **RNF-5**. En realidad, esta tarea se encuentra distribuida a lo largo de las distintas iteraciones anteriores, de modo que los textos que se añaden deben añadirse, al menos, en español e inglés (que son los idiomas requeridos). En esta iteración, sin embargo, se verifica que no quedan textos sin traducir y que las traducciones son adecuadas.

Tras establecer el orden de las iteraciones y, consecuentemente, de las tareas, se diseña el diagrama de Gantt que puede verse en la Figura 3.1, en la que se puede apreciar mejor la distribución temporal del proyecto.

## 3.4. Conclusiones

En este capítulo se han presentado los requisitos funcionales y no funcionales del proyecto con la máxima claridad posible, de modo que no se dé lugar a error sobre qué debe realizar exactamente el producto. Para asegurar que los requisitos son lo más específicos posible, todos los requisitos cuentan, además de con una descripción detallada, de una jerarquización, en los que se muestran los requisitos que derivan de este (por ejemplo, se cuenta con el requisito **RF-1**, del cual se deriva el requisito **RF-1.1**). También se ha presentado la metodología de trabajo seguida, que consiste en una metodología ágil que se ha dividido en 6 iteraciones, al final de cada una de las cuales se debe tener una versión utilizable del producto, incluyendo una característica completa más respecto a la iteración anterior. Cada iteración se ha dividido a su vez en varias tareas, que aportan más detalle sobre qué cosas concretas han de realizarse en cada iteración. Además, se especifica de manera explícita los

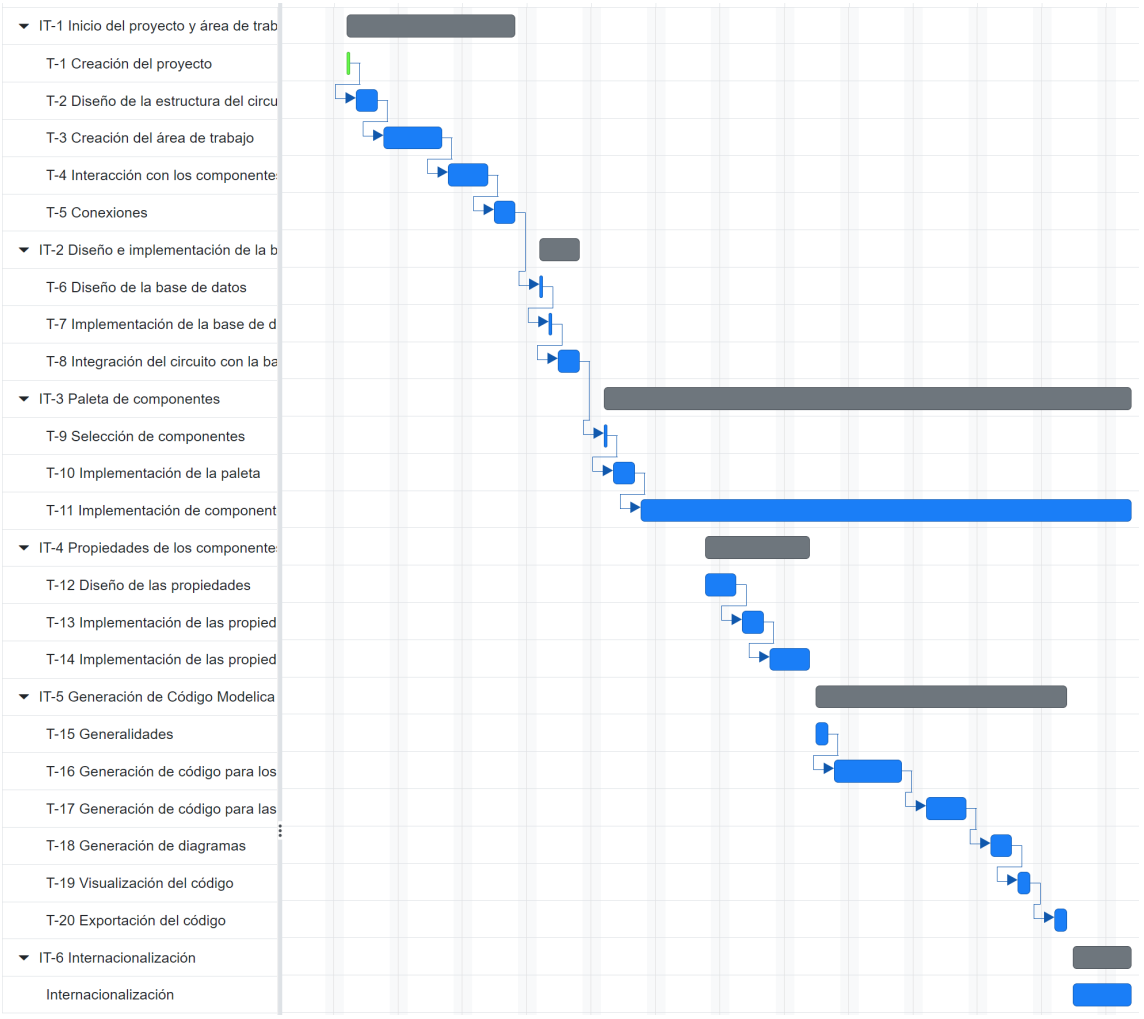



Figura 3.1: Representación de la planificación temporal del proyecto mediante diagrama de Gantt. Generado mediante el uso de (OnlineGantt, 2023).

requisitos funcionales y no funcionales que se satisfacen en cada iteración. Por último, se muestra un diagrama de Gantt que muestra las dependencias entre las distintas iteraciones y tareas, ofreciendo una visión más global de la planificación temporal seguida.

# 4. Arquitectura de la aplicación

## 4.1. Introducción

En el desarrollo de cualquier producto software, es esencial contar con una arquitectura bien definida que permita llevar a cabo un desarrollo medio-largo sin encontrar incongruencias, contradicciones o incompatibilidades dentro del código desarrollado. Para evitar estos errores, es común hacer uso de diagramas, generalmente siguiendo el estándar Unified Modeling Language (UML), que describan la estructura con la que contará la aplicación, las clases que se deberán implementar, y las relaciones entre ellas. Todo esto, por supuesto, debe ir acompañado de documentación que facilite la comprensión de los diagramas, puesto que, si bien son autoexplicativos, no siempre es sencillo interpretar qué se busca con un determinado diseño, cosa que resulta fundamental conocer cuando se deban generar cambios de diseño, puesto que los objetivos básicos serán, por lo general, los mismos. Además, es común que se genere un diagrama UML inicial, y que este se vaya actualizando para añadir los detalles necesarios según el desarrollo de la aplicación va avanzando, de modo que siempre pueda servir como referencia y documentación. En este capítulo se presenta la arquitectura de la aplicación desarrollada, mostrando su diagrama de clases UML, haciendo énfasis en los patrones de diseño seguidos, y justificando las decisiones tomadas.

## 4.2. Diagrama de clases

UML es un lenguaje de modelado que permite representar, entre otras cosas, aplicaciones software, de manera que estas queden descritas de manera unívoca y visual. En este framework se representan, principalmente, las distintas clases, encerradas en cuadrados junto con sus propiedades y, a veces, métodos; y las relaciones que hay entre ellas, mediante líneas, en las que el tipo de línea y la simbología al principio o al final de la línea denotan el tipo de relación. Si bien detallar el funcionamiento del lenguaje UML escapa al objetivo de este trabajo, sí se presentarán algunos

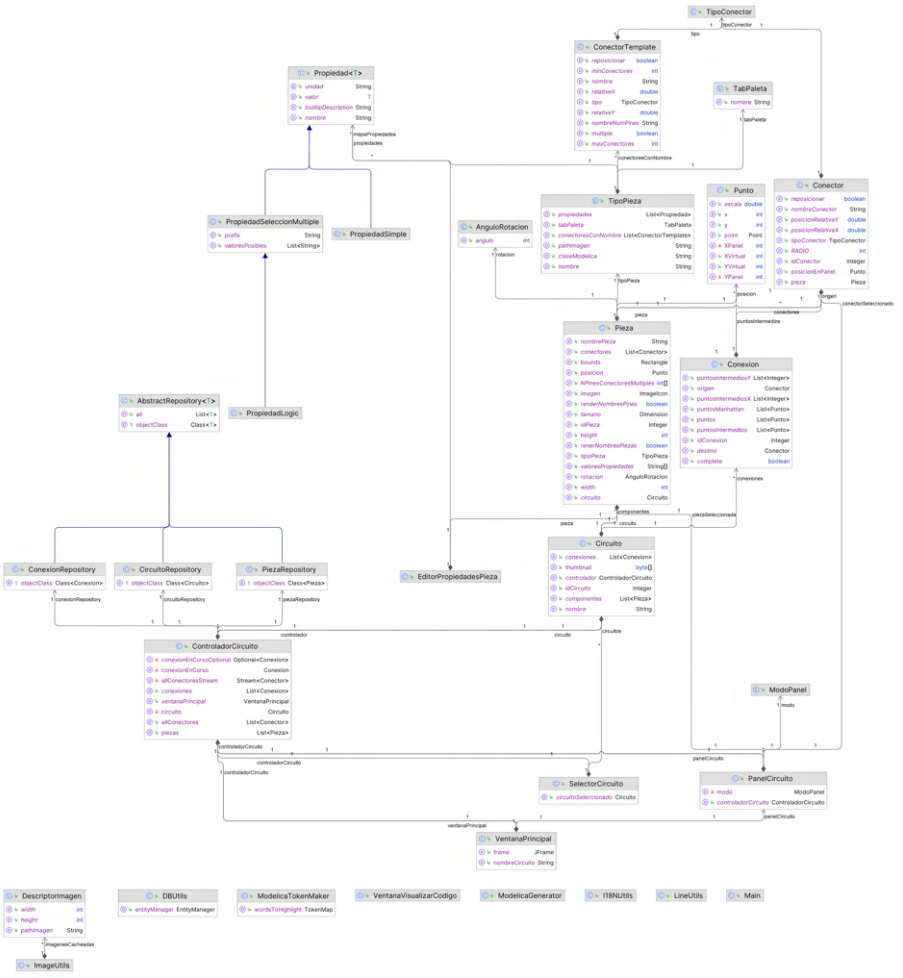



Figura 4.1: Diagrama UML de las clases del proyecto. Generado mediante (JetBrains, 2023b).

conceptos puntuales durante el desarrollo de este documento, cuando se considere imprescindible su conocimiento para la comprensión del tema tratado. En la Figura 4.1 se puede observar el diagrama completo del proyecto.

Puesto que, debido a su tamaño y a la gran cantidad de interconexiones, puede resultar compleja su interpretación, a continuación se detallan los puntos más destacables de la arquitectura.

### 4.2.1. Piezas y tipos de pieza

Las piezas y los tipos de pieza son una parte esencial de la composición de circuitos digitales puesto que, de manera muy simple, un circuito consta de componentes

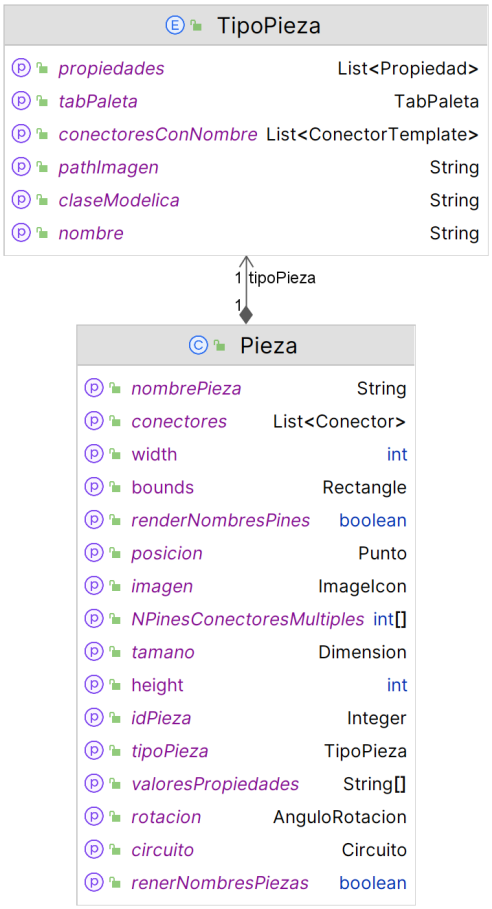



Figura 4.2: Representación UML de la relación entre las clases `Pieza` y `TipoPieza`. Generado mediante (JetBrains, 2023b).

(piezas) y conexiones entre ellas. Se muestra la sección del diagrama UML correspondiente a estos elementos en la Figura 4.2.

La clase `Pieza` representa cada uno de los componentes que se encuentran en el circuito, independientemente del tipo de pieza del que se trate. Sin embargo, esta clase contiene únicamente lógica genérica para funciones como el dibujado de una pieza, almacenar su posición y rotación, etc. Por otro lado, la lógica dependiente del tipo de pieza no se implementa aquí, sino que se proporciona toda la información necesaria, como la clase a la que corresponde en código Modelica o la imagen que se debe mostrar para representar la pieza, en el enumerado `TipoPieza`.

Un modo, tal vez más intuitivo, de implementar esto podría ser el de definir `Pieza` como una clase abstracta (o interfaz) de la que heredan (o es implementada por)

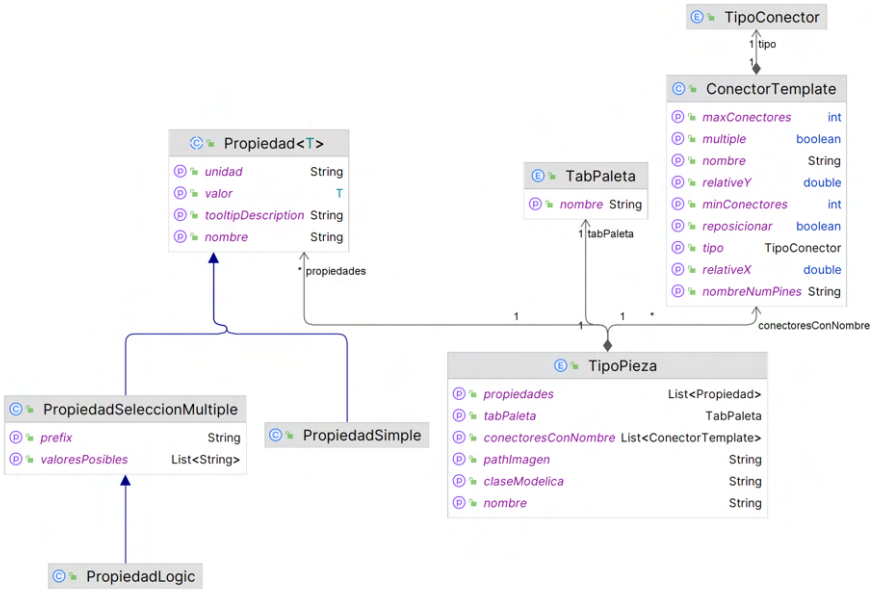



Figura 4.3: Representación UML de la clase `TipoPieza` y las clases auxiliares que utiliza. Generado mediante (JetBrains, 2023b).

las distintas clases que representen los distintos tipos de componente, como podrían ser las clases `PiezaPuertaAnd` y `PiezaPuertaOr`. Sin embargo, esto requeriría la creación de un gran número de clases casi vacías, complicando innecesariamente los diagramas UML, aumentando el número de ficheros del proyecto, etc. La alternativa propuesta, en cambio, hace que añadir un nuevo tipo de pieza sea tan simple como añadir un nuevo elemento al enumerado `TipoPieza`, con los atributos que este enumerado contiene. Estos atributos incluyen un nombre, que se utiliza como Tooltip en los menús de la aplicación; la pestaña de la paleta en la que debe mostrarse; la clase Modelica a la que pertenece; la imagen con la que debe mostrarse en el área de trabajo; un conjunto de los conectores que contiene; y un conjunto de las propiedades o atributos propios del componente. De este modo, el constructor de la clase `Pieza` utiliza la información contenida en el `TipoPieza` para determinar qué imagen debe utilizar, cuántos conectores contiene, etc. Algunas de estas propiedades de `TipoPieza`, como la lista de propiedades o de conectores, utilizan a su vez enumerados o clases externas para su definición, tal y como se muestra en la Figura 4.3. Esto se verá con más detalle en la sección 5.3, relativa a la implementación.

Se define una clase abstracta `Propiedad`, que representa los atributos de cada

tipo de pieza, de modo que `TipoPieza` contiene un conjunto de estas. Además, cuenta con dos subclases: `PropiedadSimple`, que representa aquellas propiedades que toman un valor arbitrario; y `PropiedadSeleccionMultiple`, que representa las propiedades que pueden tomar únicamente uno de entre un conjunto de valores predeterminados. Dentro de este tipo, se encuentran las propiedades de tipo *Logic*, es decir, aquellas que requieren un valor lógico, como puede ser el valor de salida de un componente de tipo Set. Puesto que el uso de `PropiedadSeleccionMultiple` requiere describir en el constructor la lista de todos los posibles valores, y dado que el uso de las propiedades de tipo Logic es muy frecuente, se extiende a `PropiedadSeleccionMultiple` con la subclase `PropiedadLogic`, que contiene estos valores por defecto.

Por otro lado, se define una clase `ConectorTemplate` que, en cierto modo, representa para `Conector` un análogo a lo que `TipoPieza` representa para `Pieza`. Es decir, `ConectorTemplate` contiene los atributos necesarios para que, al instanciarse la `Pieza`, se generen los `Conector`es apropiados.

Por último, el enumerado `TabPaleta` establece en cuál de las pestañas de la paleta deberá colocarse dicho `TipoPieza`.

La propia `Pieza`, por su parte, se relaciona principalmente con las clases mostradas en la Figura 4.4. Estas incluyen, además de la ya mencionada `TipoPieza`, algunas clases de gran importancia para el proyecto, como `Circuito` o `Conector`. `Circuito` encapsula cada uno de los circuitos, conteniendo por tanto los componentes y las conexiones, así como información adicional sobre el propio circuito, como el nombre o la imagen de previsualización. `Conector`, por su parte, representa un conector, y por tanto es el que permite conectar distintos componentes dentro del circuito. Cada pieza tiene, generalmente, varios conectores, que podrán ser de entrada o de salida. Se hará más énfasis en estas clases en futuras secciones.

La clase `Pieza` utiliza, además, dos clases auxiliares que permiten identificar su ubicación en el espacio: `AnguloRotacion`, que almacena en un enumerado la rotación de la pieza (en intervalos de 90°), y `Punto`, que almacena la posición de la pieza en el plano. Esta última clase, sin embargo, es más compleja que un simple par de coordenadas, puesto que contiene también variables relativas a la escala del área de trabajo (es decir, si se está más cerca o más lejos) y a la posición de referencia

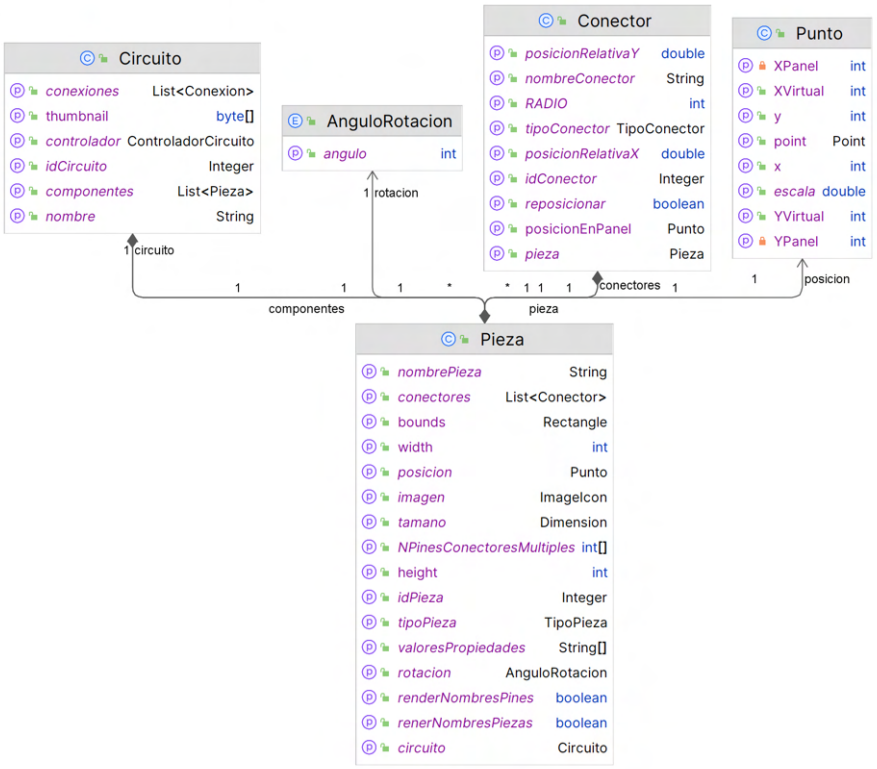



Figura 4.4: Representación UML de la clase `Pieza` y las clases con las que se relaciona. Generado mediante (JetBrains, 2023b).

(es decir, el desplazamiento que se ha efectuado sobre el área de trabajo). De este modo, la clase `Punto` contiene unas coordenadas (x,y) "virtuales", que representan la posición del punto en un hipotético plano infinito; y además proporciona otro par de coordenadas (x,y) relativas al espacio de trabajo visible por el usuario que, mediante uso de las coordenadas virtuales, la escala y el desplazamiento, calcula la posición en la que un elemento debe ser dibujado en la pantalla.

### 4.2.2. Conectores y conexiones

Tal y como se ha mencionado anteriormente, las dos partes esenciales de un circuito digital son los componentes, y las conexiones entre ellos. En este proyecto, y debido a una mayor similitud al lenguaje Modelica al que posteriormente el modelo será convertido, las conexiones no se realizan entre `Piezas`, sino entre `Conectores`, que "pertenecen" (mediante una relación de composición) a una `Pieza`. En la Figura 4.5 puede observarse la sección del diagrama relativa a las conexiones y los conectores.

Como puede verse, una `Conexion` viene representada, esencialmente, por dos `Conectores`, uno de origen y otro de destino[1], y una serie de `Puntos` intermedios, que son los puntos por los que el usuario decide que la conexión debe pasar obligatoriamente, con el fin de organizar mejor la representación visual del modelo.

Un `Conector`, por su parte, cuenta con un `TipoConector`, que representa si se trata de un conector de entrada o de salida; la `Pieza` a la que pertenece; una serie de atributos relacionados con el dibujo del conector, como el radio del círculo que lo representa gráficamente, o la posición relativa respecto a la posición de la `Pieza` que lo contiene; y un nombre, que se utiliza durante la generación de código Modelica.

[1]Si bien se han denominado origen y destino para denotar en cuál de los conectores el usuario inicia la conexión y en cuál la finaliza, lo cierto es que la conexión es bidireccional y ambos conectores son, por lo tanto, intercambiables.

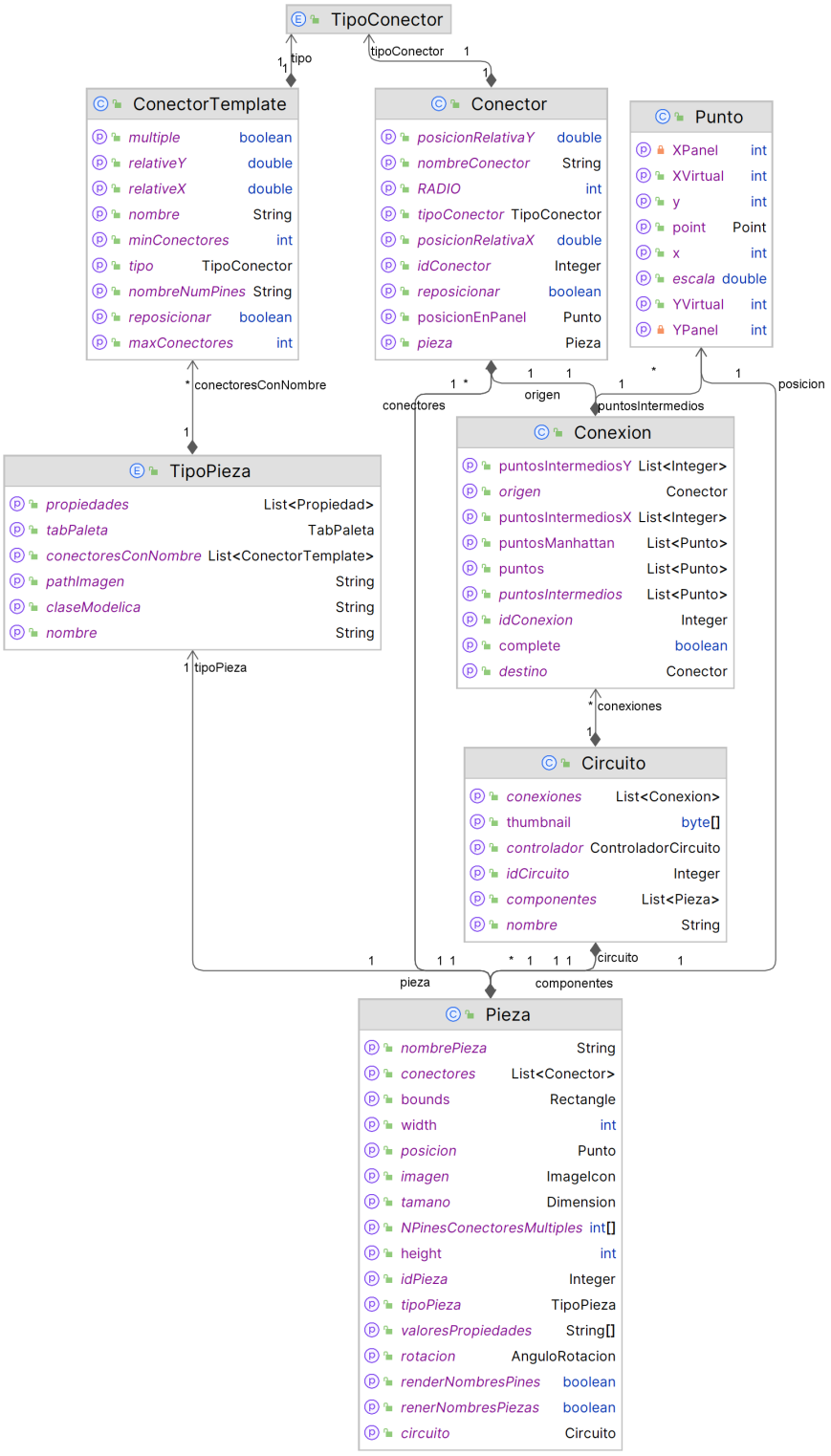



Figura 4.5: Representación UML de las clases `Conector` y `Conexion`, y las clases con las que se relacionan. Generado mediante (JetBrains, 2023b).

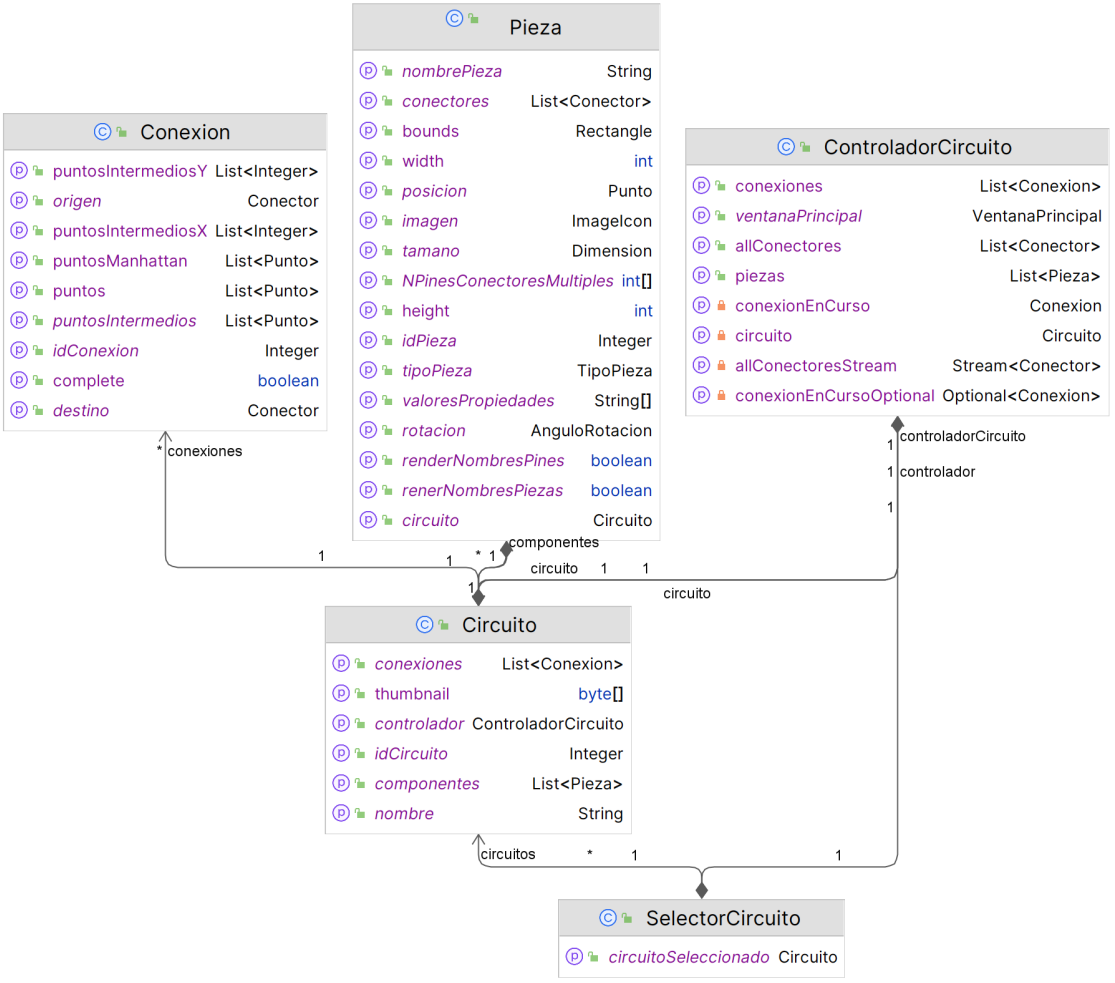



Figura 4.6: Representación UML de la clase `Circuito` y las clases con las que se relaciona. Generado mediante (JetBrains, 2023b).

### 4.2.3. Circuito

En base a los componentes anteriores, el circuito se modela tal y como se muestra en la Figura 4.6. Como se puede ver, el circuito cuenta con una serie de `Piezas`, y una serie de `Conexiones` que, en conjunto, proporcionan toda la información necesaria para modelar un circuito. El resto de los elementos de esta clase son relativos a la gestión de los propios circuitos, es decir, un nombre y un Thumbnail que permitan al usuario identificar los circuitos; y clases necesarias para la interacción de componentes, como `ControladorCircuito`, sobre los que se profundizará más adelante.

### 4.2.4. Controladores

Las clases mostradas hasta ahora componen, en conjunto, la parte de la aplicación denominada *modelo*, es decir, aquella que representa los objetos se quieren modelar (circuitos, piezas, conexiones,...). Sin embargo, en aplicaciones como la presente, que siguen patrones Modelo-Vista-Controlador (MVC)[2], se encuentran también componentes que hacen la función de controladores, es decir, son aquellos que contienen la lógica del software y son los que, como el nombre indica, "controlan" al modelo. Así, por ejemplo, el modelo define cómo podemos representar una pieza o una conexión, mientras que es el controlador el que se encarga de, a petición del usuario, colocar la pieza o generar la conexión.

La Figura 4.7 muestra el controlador principal de esta aplicación, `ControladorCircuito`, que regula la lógica de la mayoría de las funcionalidades de la aplicación.

Esta clase interactúa, como puede observarse, con el circuito al que "controla", pero también con clases de tipos muy diversos, como las de visualización (`VentanaPrincipal` y `SelectorCircuito`), que representan la vista del modelo MVC, y que realizan la función de mostrar al usuario por pantalla los distintos componentes de la aplicación, y le permite interactuar con ellos; o los repositorios, como `CircuitoRepository`, que extienden a la clase abstracta `AbstractRepository`, y que permiten la interacción con la base de datos. Así, podría afirmarse que, en cierto modo, el controlador hace de intermediario entre las distintas partes de la aplicación (vista, modelo, base de datos,...), y contiene la lógica que regula la interacción entre los mismos. Puede verse cómo la función de esta clase es principalmente lógica, más que representativa (como son, por ejemplo, los modelos), analizando los métodos ofrecidos por `ControladorCircuito`, que pueden verse en la Figura 4.8a. Estos métodos son, por lo general, de tipo funcional, es decir, realizan funciones (añadir/eliminar conexiones, cargar/guardar el circuito, calcular posiciones de los elementos,...), a diferencia de otro tipo de clases donde los métodos se utilizan principalmente para leer o escribir propiedades de los elementos, como puede verse en la Figura 4.8b, donde la mayoría de funciones son Getters y Setters.

[2]En la sección 4.3 se estudia con mayor detalle este patrón de diseño.

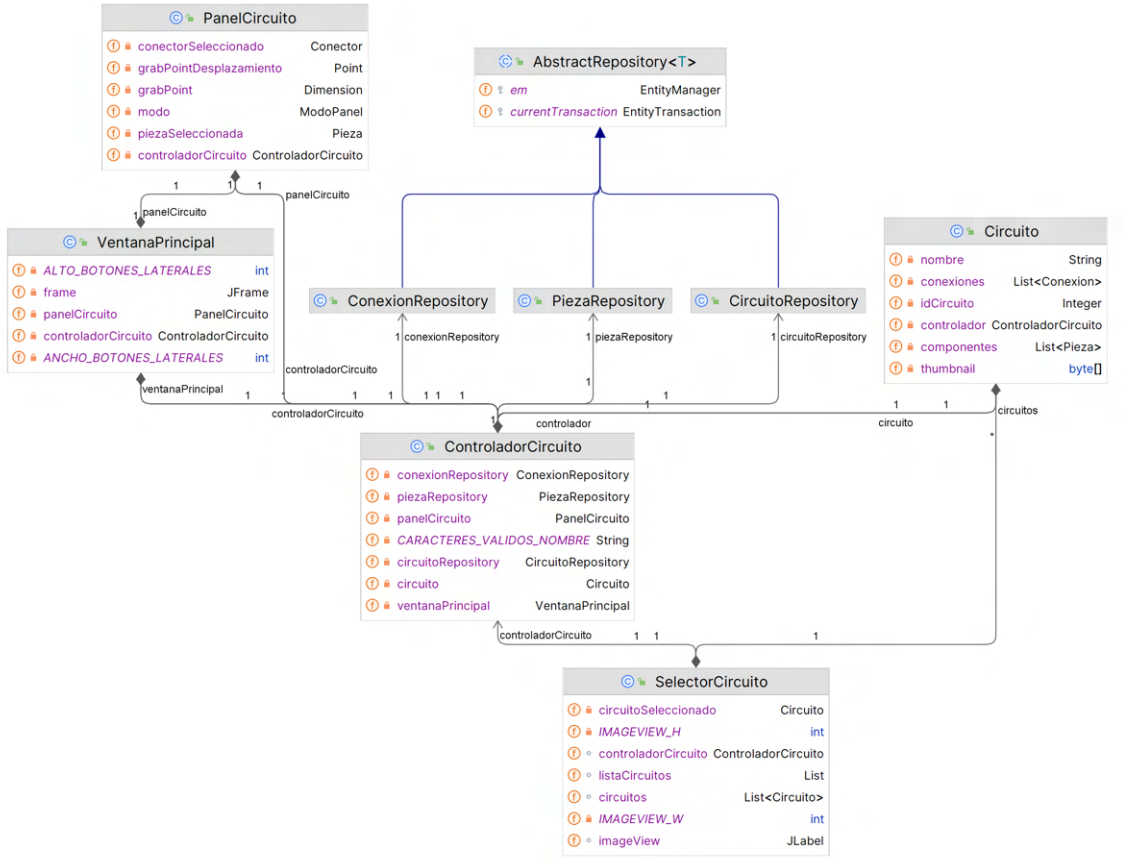
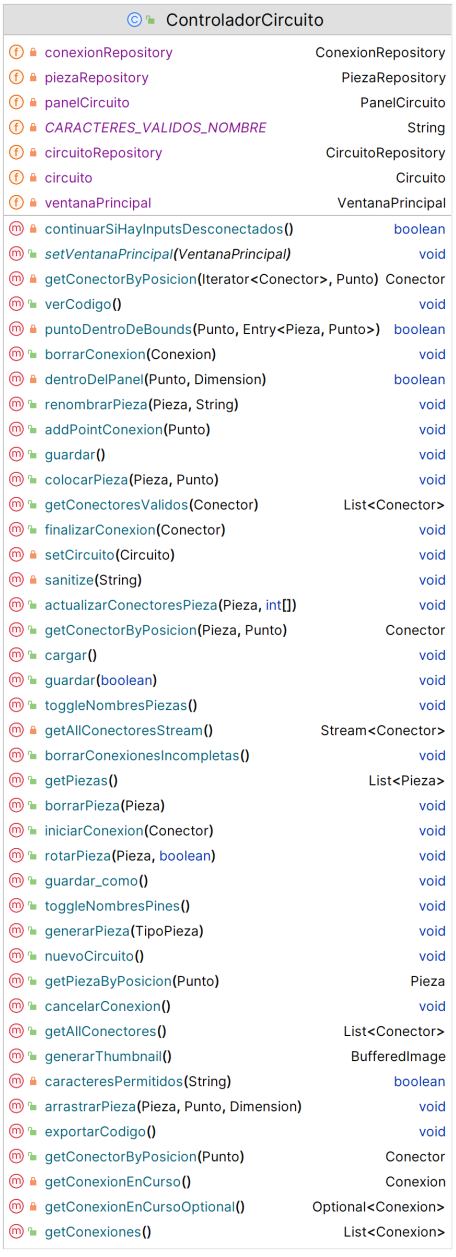



Figura 4.7: Representación UML de la clase `ControladorCircuito` y las clases con las que se relaciona. Generado mediante (JetBrains, 2023b).




(a) Representación UML de la clase `ControladorCircuito`, donde se indican los distintos métodos ofrecidos por la clase. Generado mediante (JetBrains, 2023b).

(b) Representación UML de la clase `Circuito`, donde se indican los distintos métodos ofrecidos por la clase. Generado mediante (JetBrains, 2023b).

Figura 4.8: Comparación entre los métodos ofrecidos por la clases `ControladorCircuito` y `Circuito`.

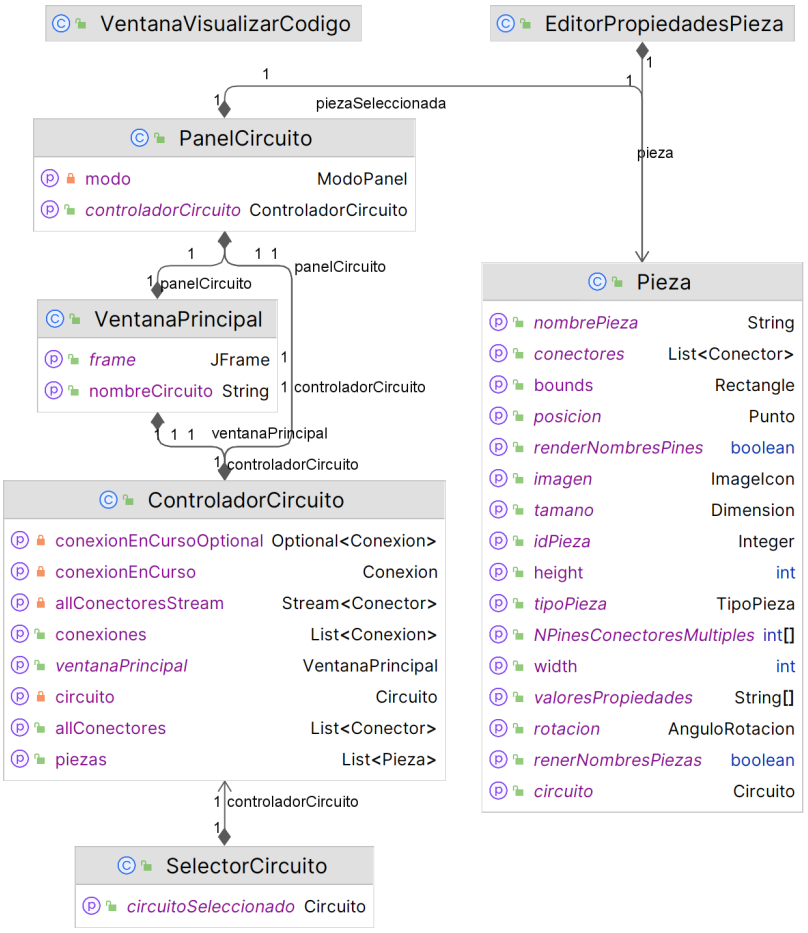



Figura 4.9: Representación UML de las vistas, y las clases con las que se relacionan. Generado mediante (JetBrains, 2023b).

### 4.2.5. Vistas

Tal y como se ha mencionado en secciones anteriores, existen algunas clases que están dedicadas a la visualización en pantalla de la aplicación, y en permitir al usuario interactuar con la misma. Estas son las vistas, que pueden verse en la Figura 4.9 junto con las clases con las que se relacionan de manera directa. Las vistas, en general, utilizan el framework de visualización gráfica Swing, ofrecido por Java, que permite el diseño de ventanas y distintos componentes.

La clase `VentanaPrincipal` representa, como su nombre indica, la ventana principal de la aplicación, en la que el usuario pasa la mayoría del tiempo. Esta ventana, además de los distintos menús o la paleta de componentes, contiene el `PanelCir-`

cuito, que representa el área de trabajo donde el usuario puede colocar o desplazar los componentes y, en definitiva, modelar el circuito. Si bien sería posible que el PanelCircuito se encontrara embebido en la VentanaPrincipal, se opta por utilizar una clase aparte para este elemento puesto que la lógica que contiene es significativa.

Existen, además, otras vistas o ventanas que se utilizan en momentos puntuales de la aplicación. SelectorCircuito, por ejemplo, es una ventana secundaria que permite al usuario seleccionar el circuito que desea cargar desde la base de datos, mostrando una previsualización. EditorPropiedadesPieza, por otra parte, representa la ventana emergente que surge al indicar que se quieren editar las propiedades o atributos de una Pieza. Finalmente, VentanaVisualizarCodigo representa la última ventana de la aplicación, que se utiliza para mostrar al usuario el código Modelica generado por el circuito, utilizando para ello coloreado de sintaxis.

### 4.2.6. Repositorios

La aplicación hace uso, para almacenar los distintos circuitos modelados, de una base de datos embebida. Con el fin de centralizar la interacción con la base de datos todo lo posible, y buscando simplificar al máximo su uso, el software implementa una serie de *repositorios* (que pueden verse en la Figura 4.10, esto es, clases dedicadas a encapsular la interacción con la base de datos.

Estos repositorios están organizados de modo que existe una superclase genérica, AbstractRepository, que contiene la lógica para la interacción con la base de datos, como el guardado o carga de información. De esta clase extienden los distintos repositorios, como ConexionRepository o PiezaRepository, que no son más que una subclase de AbstractRepository donde, en lugar de un tipo genérico, lo que se almacena con Conexiones o Piezas. Se observa cómo el ControladorCircuito contiene una instancia de cada uno de los repositorios.

Además, la clase DBUtils implementa algunas funciones de utilidad que son usadas por AbstractRepository.

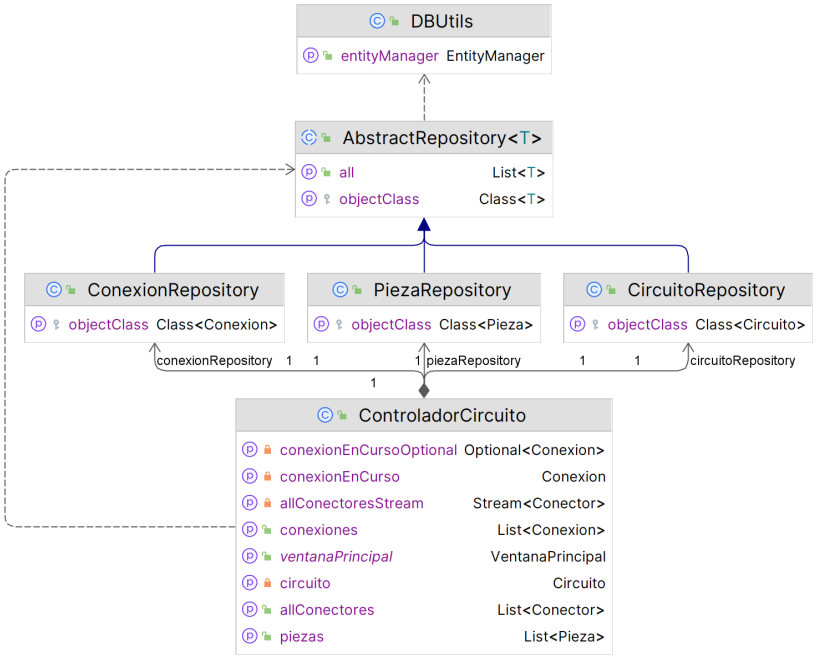



Figura 4.10: Representación UML de los repositorios, y las clases con las que se relacionan. Generado mediante (JetBrains, 2023b).

### 4.2.7. Utilidades

Al igual que en el apartado anterior se introduce el uso de la clase `DBUtils`, que contiene funciones de utilidad para la interacción con las bases de datos, la aplicación contiene varias clases de utilidades distintas que no se relacionan con el resto del software mediante las relaciones de composición, agregación, etc. con los que se relacionan otras clases, pero que son utilizadas en distintos puntos de toda la aplicación. En la Figura 4.11 se muestran estas clases.

Si bien algunas de las clases de utilidades ya han sido descritas antes, como `Punto` o `AnguloRotacion`, en este apartado se indica la funcionalidad de aquellas que no lo han sido. En general, y como se verá a continuación, todas estas clases cuentan con una funcionalidad muy específica, orientada a auxiliar a otros componentes de la aplicación.

- En primer lugar, se encuentra la clase `ImageUtils`, que ofrece una serie de métodos y funciones para simplificar el uso de imágenes en la aplicación, como son la carga de imágenes desde el disco, o el escalado y rotación de imágenes.

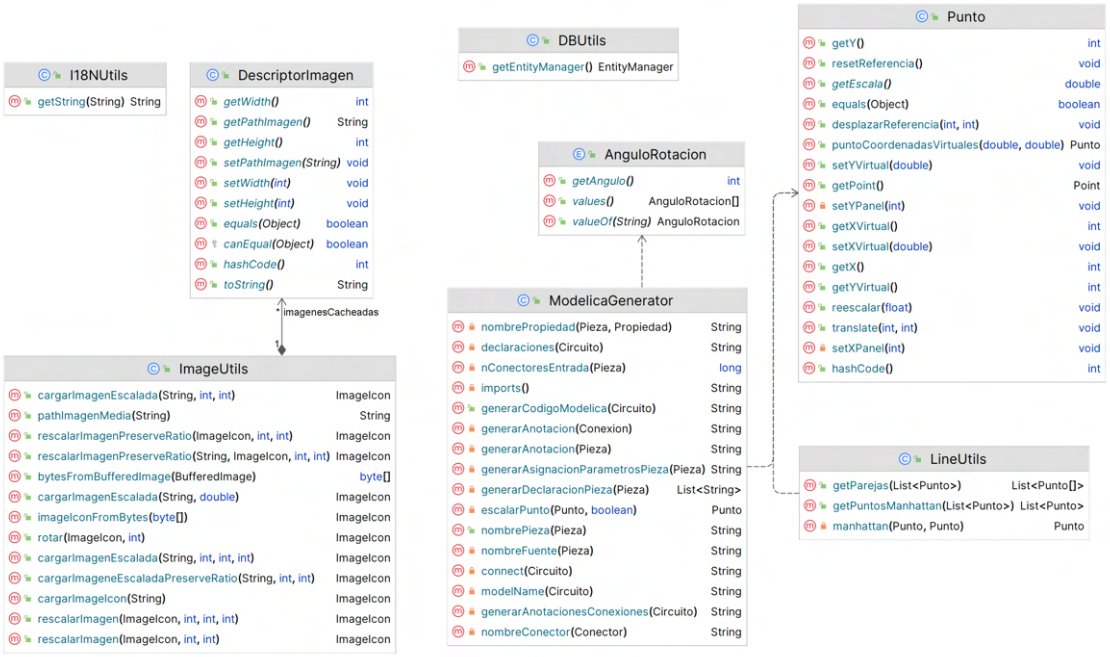



Figura 4.11: Representación UML de las clases de utilidad. Generado mediante (JetBrains, 2023b).

Además, con el fin de minimizar el uso del disco, se mantiene una caché de todas las imágenes cargadas y reescaladas en sus distintos tamaños, de modo que si vuelve a solicitarse una misma imagen con el mismo tamaño, esta no debe ser cargada ni reescalada de nuevo. Del mismo modo, si se solicita una imagen ya cargada anteriormente, pero en un tamaño diferente, entonces será necesario reescalarla, pero no volverá a cargarse del disco. Para que esta caché funcione correctamente, es necesario el uso de la clase `DescriptorImagen`, que permite identificar, de manera sencilla, una imagen, en función de su nombre y su tamaño. Así, `ImageUtils` utiliza `DescriptorImagenes` para inferir si una imagen está en la caché o no.

- Por otro lado, la clase `LineUtils` ofrece funciones para el cálculo de las líneas que deben ser mostradas en el área de trabajo, puesto que las conexiones se basan en dos o más puntos en el plano, sin embargo la unión directa de estos puntos generará, probablemente, líneas diagonales. Con el fin de evitar esto, y que todas las líneas sean horizontales o verticales, `LineUtils` ofrece una serie

de funciones que permiten convertir una secuencia de puntos no "alineados" a otra secuencia con puntos intermedios para evitar diagonales.

- `I18NUtils`, por su parte, ofrece utilidades para la internacionalización[3], esto es, ofrecer los textos en distintos idiomas. Mediante el uso de `I18NUtils`, se simplifica el proceso de carga de textos desde los correspondientes ficheros de configuración.

- Por último, la clase `ModelicaGenerator` contiene todas las funciones necesarias para, dado el modelo de un circuito, generar el código Modelica, de modo que pueda ser visualizado o exportado por el usuario.

## 4.3. Patrones de diseño

En el proceso de desarrollo de software, es común encontrar problemas que, a priori, no son sencillos de modelar de manera eficaz. Sin embargo, muchos de estos problemas se repiten en varios proyectos y, a pesar de que puedan ser proyectos de índole completamente distinta, las soluciones serán, generalmente, las mismas. Estas soluciones son los denominados *patrones de diseño*, es decir, arquitecturas estudiadas y probadas, diseñadas para resolver eficientemente un tipo de problema. Durante esta sección, se desarrollarán los patrones de diseño implementados en el proyecto.

### 4.3.1. Tipos de patrones de diseño

Por lo general, suelen subdividirse los patrones de diseño por tipos, según la clase de problema que buscan resolver. Los tipos principales son (RefactoringGuru, 2023a):

- **Patrones creacionales.** Este tipo de patrón ofrece mecanismos y herramientas de creación de objetos que aumentan la flexibilidad y la reutilización del código.

[3]Generalmente abreviado como I18N por el número de letras en la palabra internacionalización.

- **Patrones estructurales.** Estos patrones se centran en cómo combinar grandes grupos de objetos y clases en estructuras complejas, manteniendo la flexibilidad y eficiencia de las mismas.

- **Patrones de comportamiento.** Este tipo de patrón se utiliza para regular la interacción y comunicación entre componentes, y para la asignación de responsabilidades a los distintos objetos.

### 4.3.2. Patrón Modelo-Vista-Controlador

El patrón Modelo-Vista-Controlador (MVC) es un patrón de tipo estructural y es, en general, uno de los patrones de diseño más utilizados en las aplicaciones que cuentan con una interfaz gráfica. Este patrón consta, como se ha mencionado en secciones anteriores, de tres componentes principales:

- El Modelo representa los datos de la aplicación, pero no contiene ninguna lógica. Por ejemplo, clases como `Pieza`, `Conexion` o `Conector` son parte del modelo. Puede entenderse el modelo como una estructura de datos especialmente adaptada al problema.

- La Vista se encarga de la interacción con el usuario, es decir, le presenta la información del modelo mediante una interfaz gráfica, y recoge el la entrada del usuario. La vista no conoce la lógica intrínseca al problema, únicamente sabe cómo debe representar al modelo.

- El Controlador, finalmente, es el componente que contiene toda la lógica. Se encarga de recibir las acciones del usuario que la vista le proporciona, y actualizar el modelo en consecuencia. Además, es el controlador el que actualiza la vista cuando es necesario. En este patrón de diseño, la vista y el modelo no deben interactuar de manera directa, sino utilizando siempre al controlador como intermediario.

En la Figura 4.12 se muestra el diagrama UML relativo al patrón MVC.

Lo que este patrón de diseño permite, es modularizar el código, y facilitar la aplicación del principio de responsabilidad única o Single Responsibility Principle

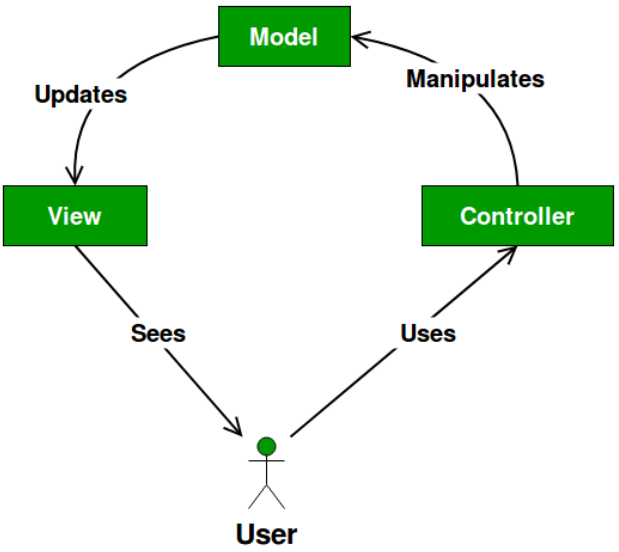



Figura 4.12: Representación esquemática de la estructura del patrón MVC. Extraído de (GeeksForGeeks, 2023).

(SRP), que dice que cada clase debe realizar una única función, y realizarla completa, de modo que el código de una funcionalidad quede centralizado y ordenado, y además separado del código de otras funcionalidades (Martin, 2003).

El presente proyecto utiliza tres paquetes diferentes para implementar las distintas clases: `modelo`, `gui` y `controlador`, que representan, respectivamente, los modelos, vistas y controladores de la aplicación. Como se ha mencionado con anterioridad, el contenido de cada uno de los paquetes es, respectivamente:

`modelo` contiene las clases:

- `Circuito`
- `Conector`
- `Conexion`
- `Pieza`
- `Propiedad`
- `PropiedadLogic`
- `PropiedadSeleccionMultiple`

- `PropiedadSimple`

`gui` contiene las clases:

- `EditorPropiedadesPieza`
- `PanelCircuito`
- `SelectorCircuito`
- `VentanaPrincipal`
- `VentanaVisualizarCodigo`

`controlador` contiene únicamente la clase `ControladorCircuito`

### 4.3.3. Patrón Singleton

El patrón singleton (RefactoringGuru, 2023b), también llamado instancia única, es un patrón de diseño creacional que garantiza que, en todo momento, solo se tiene una única instancia de una determinada clase. Esto puede ser interesante por varios motivos, incluyendo el ahorro de la memoria necesaria para almacenar varias instancias, el ahorro de los recursos computacionales necesarios para la creación de la instancia (que, según el tipo de objeto, puede ser un proceso costoso), o el mantenimiento de la consistencia de la lógica de negocio en ciertos casos. Este último caso es el más común, y se utiliza para, por ejemplo, gestionar el acceso a un recurso compartido, como puede ser la base de datos. También puede utilizarse como estrategia para "compartir" un objeto entre varias clases, sin necesidad de crear algún tipo de variable global, ni de transmitir el objeto entre clases para utilizarlo.

El funcionamiento del patrón singleton es simple: cuando se intenta instanciar un objeto, y ya se ha creado anteriormente otra instancia de la misma clase, entonces, en lugar de crear una nueva instancia, se devuelve la existente.

Este proyecto hace uso del patrón singleton en la clase `DBUtils`, que impide que haya varios gestores de bases de datos (`EntityManager`) simultáneamente, para evitar posibles inconsistencias. De modo similar, se hace uso de este patrón de diseño en `I18NUtils`, para mantener una única instancia del `ResourceBundle` que permite acceder a los textos traducidos a lo largo de toda la aplicación.

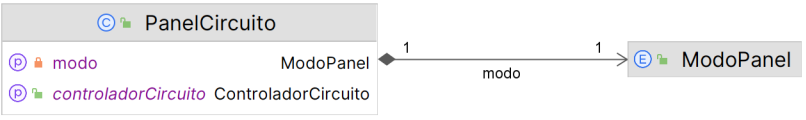



Figura 4.13: Representación esquemática de la estructura del patrón estado (denominado *modo* en la imagen). Generado mediante (JetBrains, 2023b).

### 4.3.4. Patrón Estado

El patrón estado (RefactoringGuru, 2023c) es una estrategia que permite simular una máquina de estados finitos dentro de un componente software, para controlar mejor su comportamiento. En una máquina de estados finitos, se definen una serie de estados y de transiciones entre estados, de manera que una misma entrada genera una salida diferente según el estado. Un ejemplo muy sencillo de esto es un reproductor de música en el que pulsar el botón principal en estado `reproduciendo` pone la música en pausa, mientras que pulsar el mismo botón en estado `pausado` reanuda la reproducción. El patrón estado se utiliza para reducir la complejidad de componentes que tengan una lógica muy densa e interconectada, en los casos en los que se pueda representar el problema mediante una máquina de estados.

Dentro de la aplicación, se utiliza el patrón estado para representar el estado actual del `PanelCircuito`, estando los estados representados por la acción que está siendo llevada a cabo por el usuario mediante el enumerado `ModoPanel`, tal y como se muestra en la Figura 4.13. Por ejemplo, si se está creando una conexión (estado `MODO_CONEXION`) y se hace click sobre un conector, se debe finalizar la conexión; mientras que si se hace click sobre una zona vacía del área de trabajo debe añadirse un punto intermedio de la conexión. Si, por otro lado, el estado es el estado "base" en el que no se está realizando ninguna acción (`MODO_NORMAL`), y se pulsa sobre un conector, debe iniciarse la conexión (y cambiar el estado de manera consecuente). Del mismo modo, si se pulsa sobre una pieza en modo borrado (estado `MODO_BORRADO`) esta deberá ser eliminada, mientras que si se pulsa en modo normal deberá mostrarse la ventana con las propiedades de la pieza.

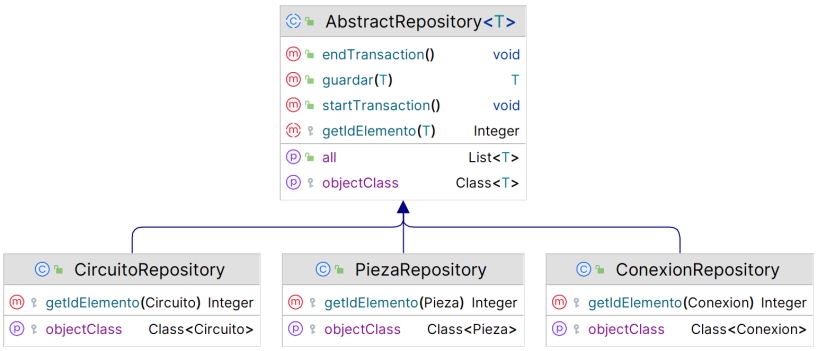



Figura 4.14: Diagrama UML de la aplicación del patrón estrategia en los repositorios. Generado mediante (JetBrains, 2023b).

### 4.3.5. Patrón Estrategia

El patrón estrategia es un patrón de comportamiento que permite definir una familia de algoritmos y hacer sus objetos intercambiables (RefactoringGuru, 2023d). En muchas ocasiones se encuentran algoritmos que, por lo general, tienen un funcionamiento y una estructura similares, sin embargo muestran diferencias en pasos concretos de la implementación. Por ejemplo, un algoritmo de búsqueda de caminos tiene, a rasgos generales, la misma estructura siempre. Sin embargo, según el caso de uso, se utilizarán funciones de coste diferentes. Es en este tipo de casos en los que el patrón estrategia puede resultar útil: es posible definir una clase abstracta que cuente con la estructura general del algoritmo de búsqueda, pero que defina un método abstracto para la función de coste, delegando así su implementación a una subclase. Por ejemplo, podrían definirse subclases como `BusquedaManhattan`, que implementa la distancia Manhattan como función de coste, o `BusquedaMahalonobis`, que implementa la distancia Mahalonobis o, incluso, distancias más complejas que puedan servir para buscar en el espacio de estados de un juego.

Este proyecto hace uso del patrón estrategia para la interacción de la base de datos, utilizando los repositorios, tal y como se muestra en la Figura 4.14.

En ella puede apreciarse cómo la clase `AbstractRepository` declara los métodos abstractos `getIdElemento` y `getObjectClass`, que son posteriormente implementados por las subclases `CircuitoRepository`, `PiezaRepository` y `ConexionReposi-`

tory. De este modo, AbstractRepository es capaz de implementar la estructura general del algoritmo, independientemente de cuál sea el modo de obtener el id o la clase de cada uno de los objetos concretos.

Se profundizará más en la implementación de los repositorios en la sección 5.6.

## 4.4. Estructura de la base de datos

La base de datos diseñada para esta aplicación almacena los 4 elementos principales del sistema: Conexion, Conector, Pieza y Circuito. En la Figura 4.15 puede verse un diagrama Entidad-Relación de la base de datos utilizada. Este diagrama utiliza puntas de flecha para representar relaciones múltiples, y líneas simples para representar conexiones únicas. Así, por ejemplo, la flecha que sale de Circuito y llega a Conexion indica que es una relación n:1, es decir, un Circuito tiene múltiples Conexiones, pero una Conexion tiene un único Circuito.

En este diagrama destaca el uso de una línea de color verde, entre Pieza y Punto, que en lugar de una punta de flecha utiliza el símbolo UML para la composición. Esto se debe a que Punto no existe como tal en una tabla, sino que es un elemento embebido. Esto significa que, si bien en la aplicación existen dos clases separadas para los objetos Pieza y los objetos Punto, cuando se hace la transformación a la base de datos los componentes de Punto son añadidos a la tabla Pieza. De este modo, se ahorran recursos al no ser necesario el uso de una tabla que cambiaría con mucha frecuencia, y cuyo manejo para evitar memory leaks sería muy complejo.

Por otro lado, puede observarse cómo algunos atributos relativos a listas de puntos, como Conexion.puntosIntermediosX, no hacen uso de la clase Punto en la base de datos. Esto se debe a la API ofrecida por JPA y Derby, y se profundizará en esto más adelante durante el capítulo 5, dedicado a la implementación del sistema.

## 4.5. Conclusiones

Durante este capítulo se ha presentado la arquitectura de la aplicación desarrollada. En primer lugar, se ha mostrado un diagrama UML de la aplicación completa,

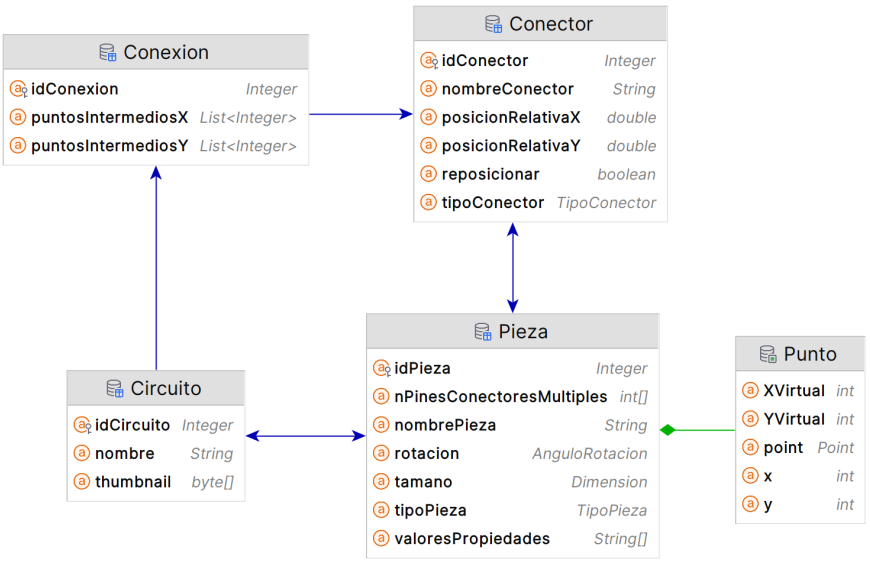



Figura 4.15: Diagrama Entidad-Relación de la base de datos. Generado mediante (JetBrains, 2023b).

para posteriormente hacer énfasis en aquellos componentes de mayor relevancia para el funcionamiento del sistema. También se han detallado las funcionalidades de todas las clases, sin profundizar en detalles de implementación.

A continuación se han analizado los principales patrones de diseño utilizados en la aplicación, tras describir brevemente qué es un patrón de diseño y cuál es su finalidad. Para cada uno de los patrones de diseño señalados, se ha descrito su funcionamiento, su utilidad, y se ha justificado su uso en la aplicación.

Por último, se ha ilustrado la estructura diseñada para la base de datos mediante un diagrama Entidad-Relación acompañado de una descripción textual en la que se señalan los puntos más importantes a considerar.

# 5. Implementación

## 5.1. Introducción

Tras describir a nivel conceptual la aplicación en el Capítulo 4, centrándose en la estructura de la aplicación mediante diagramas UML, este capítulo está dedicado a un análisis de los detalles relativos a la implementación realizada que puedan resultar de interés. Más allá de los fragmentos de código mostrados en este capítulo, puede encontrarse el código fuente completo en el Anexo B.

## 5.2. Anotaciones en Project Lombok y JPA

Tal y como se ha descrito en la sección 2.4.3, Project Lombok es una librería destinada a reducir la cantidad de código repetitivo que debe ser escrita por el programador, como son los métodos Getters o Setters, que, si bien son necesarios en la mayoría de clases, no tienen apenas complejidad y suponen un gasto de tiempo y recursos innecesario. En esta sección se describen las funciones principales de esta librería, de modo que sea posible reconocerlas y comprenderlas en el los fragmentos de código de las secciones posteriores.

La librería Project Lombok funciona mediante anotaciones o decoradores, es decir, elementos que se añaden en la cabecera de los métodos o clases, o encima de las variables, y que modifican su comportamiento. Un ejemplo de anotación nativa en Java es `@Override`, que indica que un método sobrescribe al homónimo de una superclase o interfaz implementada. En general, las anotaciones de Lombok pueden aplicarse tanto a clases como a variables, y aplicarlo a una clase es equivalente a aplicarlo sobre todas sus variables. Las principales anotaciones incluidas en Project Lombok son:

- `@Getter` genera un método Getter para la variable sobre la que se coloca.
- `@Setter` genera un método Setter para la variable sobre la que se coloca.

- `@EqualsAndHashcode` aplicado sobre una clase, genera los métodos `equals()` y `hashCode()` necesarios para ciertos métodos de comparación de objetos o el uso de estructuras de datos como los `HashMaps`

- `@ToString` aplicado sobre una clase, genera un método `toString()` utilizado para obtener una representación textual de un objeto. El método generado representa el objeto en un formato similar a JSON.

- `@ToString.Exclude` excluye a la variable sobre la que se aplica de la cadena de texto devuelta por el método generado mediante `@ToString`.

- `@Data` aplicado sobre una clase, es equivalente a utilizar los métodos `@Getter`, `@Setter`, `@EqualsAndHashcode` y `@RequiredArgsConstructor`.

- `@RequiredArgsConstructor` aplicado sobre una clase, genera un constructor que recibe como parámetros las variables requeridas para la clase.

- `@AllArgsConstructor` aplicado sobre una clase, genera un constructor que recibe como parámetros todas las variables de la clase.

- `@NoArgsConstructor` aplicado sobre una clase, genera un constructor que no toma ningún parámetro.

De manera similar a Project Lombok, JPA también hace uso de anotaciones para generar la estructura de la base de datos. Algunas de las anotaciones más utilizadas son:

- `@Entity` se aplica sobre una clase, e indica que la clase en cuestión es una entidad y que debe crearse/asociarse con una tabla de la base de datos.

- `@Id` indica que la variable sobre la que se aplica representa el la clave primaria de la tabla.

- `@GeneratedValue` indica que la clave marcada mediante `@Id` debe ser generada por la base de datos, y no por la aplicación.

- `@Transient` indica que la variable sobre la que se aplica no debe ser almacenada en la base de datos, sino que es necesaria únicamente en tiempo de ejecución. Por defecto, todas las variables que no estén marcadas mediante esta anotación serán almacenadas.

- `@OneToMany` y `@ManyToOne` indican las relaciones de claves foráneas en la base de datos. Se utilizan, generalmente, junto con la anotación `@JoinColumn`, que permite especificar el nombre de la columna a la que se hace referencia para la relación.

- `@PostLoad` se aplica sobre un método, y permite ejecutarlo justo después de cargar un objeto de esa clase desde la base de datos. Se utiliza generalmente para poblar las variables `@Transient`

- `@PreUpdate` se aplica sobre un método, y permite ejecutarlo justo antes de efectuar una acción `UPDATE` sobre la base de datos, y se utiliza generalmente para aplicar los efectos de las variables `@Transient` a las variables persistidas. De manera análoga, `@PrePersist` permite ejecutar un método justo antes de escribir un objeto en la base de datos.

- `@Enumerated` indica que la variable a almacenar es un enumerado, y permite especificar el modo en el que se almacenará en base de datos (como representación textual del nombre del enumerado, como número entero representando la posición en una lista ordenada de los valores del enumerado, o como una referencia a una tabla externa).

Como se verá más adelante, JPA permite almacenar de manera nativa tipos simples de Java (como `int`, `String`,...) y arrays de los mismos; sin embargo, no es inmediato almacenar otras clases, como un objeto de tipo `Punto`. Es por ello que, como se verá más adelante, se utilizan colecciones de `Puntos` anotadas como `@Transient`, y mediante los métodos `@PostLoad`, `@PreUpdate` y `@PrePersist` se convierten a arrays de tipos básicos (y viceversa) para permitir su persistencia.

Es importante aclarar que ambas librerías son compatibles entre sí, por lo que

```
final String nombre;
final String pathImagen;
final String claseModelica;
final TabPaleta tabPaleta;
final List<Propiedad> propiedades;
final List<ConectorTemplate> conectoresConNombre;
```

Figura 5.1: Captura de pantalla en la que se muestran las propiedades del enumerado `TipoPieza`.

no supone ninguna incongruencia anotar una variable o una clase con algunas anotaciones de Project Lombok y otras de JPA.

## 5.3. TipoPieza

Tal y como se ha enfatizado en la sección correspondiente del Capítulo 4, la relación entre `Pieza` y `TipoPieza` supone una estrategia para evitar la necesidad de crear subclases para cada uno de los distintos tipos de pieza que existen. Para ello, se define el "enumerado inteligente" `TipoPieza`, que cuenta con las propiedades que pueden verse en la Figura 5.1.

De este modo, es posible definir los distintos tipos de pieza en el enumerado haciendo uso del constructor de la clase, tal y como se muestra en la Figura 5.2. En esta figura, se muestra tan solo la definición del algunas puertas del paquete `Gates`, puesto que el fichero completo cuenta con varios cientos de líneas.

Como se puede ver, durante la definición del enumerado es donde se instancian las distintas clases de `Propiedad` o `ConectorTemplate`, que no son más que "generadores" de propiedades o de conectores, respectivamente, que serán utilizados durante la generación de una instancia de `Pieza`. De este modo, si bien la definición del enumerado puede resultar algo compleja, se reduce en gran medida la complejidad del código en otros ficheros de la aplicación.

Además, para facilitar la lectura de este enumerado, se utiliza la directiva `//-region`, que permite agrupar los distintos elementos del enumerado y comprimi-


Figura 5.2: Captura de pantalla en la que se muestra la definición de algunos elementos del enumerado `TipoPieza` correspondientes a componentes del paquete `Gates`.

r/expandir las distintas secciones del código, de modo que el fichero completo, al comprimir todas las regiones, tiene el aspecto mostrado en la Figura 5.3.

## 5.4. Propiedades

La gestión de las propiedades es, también, una parte fundamental de la aplicación, puesto que de otro modo el usuario se vería obligado a modificar las propiedades desde el editor de código Modelica, y este software dejaría de tener sentido.

Como ya se ha descrito en la fase de análisis de la arquitectura de la aplicación, una pieza cuenta con una serie de propiedades. Más concretamente, `TipoPieza` cuenta con una serie de `Propiedad` o, mejor dicho, de subclases de `Propiedad`, por ser esta una clase abstracta. La definición de esta clase, que puede verse en la Figura 5.4, es bastante simple.

En primer lugar, se definen dos constantes que representan dos posibles unidades de medida para la variable (las más utilizadas) y, posteriormente, se declaran las variables de la clase, que incluyen el nombre de la propiedad, la unidad en la

```java
@Getter
@AllArgsConstructor
public enum TipoPieza {
    Basic
    Gates
    Sources
    Tristates
    Multiplexers
    ;
    final String nombre;
    final String pathImagen;
    final String claseModelica;
    final TabPaleta tabPaleta;
    final List<Propiedad> propiedades;
    final List<ConectorTemplate> conectoresConNombre;
}
```

Figura 5.3: Captura de pantalla en la que se muestra el aspecto del enumerado `TipoPieza` al comprimir todas las regiones, que se encuentran resaltadas en verde.

que se mide, y la `tooltipDescription`, es decir, el texto que deberá aparecer en el Tooltip mostrado cuando el usuario mantenga su puntero sobre dicha variable. Además, la `Propiedad` tiene un `valor` de tipo genérico `T`. Esto significa que, potencialmente, podría almacenarse una propiedad de cualquier tipo (entero, cadena de texto, colección de valores,...). Sin embargo, como se ve en las implementaciones de esta clase abstracta, actualmente se utilizan únicamente propiedades de tipo `String`. Esto puede verse en la Figura 5.5, en la que se muestra la clase `PropiedadSeleccionMultiple`.

Este tipo de propiedad, que se representa mediante un tipo `String`, cuenta con una serie de valores predeterminados que serán mostrados al usuario, mediante un selector, para que elija el deseado. Estos valores, que son pasados como parámetros del constructor, pueden ser, en ocasiones, tediosos de construir. Es por ello, que se define una constante `SELECCION_MULTIPLE_LOGICAL`, que contiene los valores utilizados para las variables del tipo `Logic` de Modelica, por ser utilizado muy frecuentemente. Además, se ofrece un constructor que permite definir un prefijo, de manera que si los valores utilizados son, por ejemplo, del paquete `Logic` de Modelica (importado como `L`), no sea necesario definir los posibles valores como `L.'0'`, `L.'1'`,..., sino que baste con definirlos como `'0'`, `'1'`,... y añadir el prefijo `L.`, que

```java
@AllArgsConstructor
public abstract class Propiedad<T> {
    public static final String UNIDAD_REAL = "Real";
    public static final String UNIDAD_TIME = ModelicaGenerator.SI + ".Time";
    @Getter
    @Setter
    private T valor;

    @Getter
    private String nombre;

    @Getter
    @Setter
    private String unidad;

    @Getter
    @Setter
    private String tooltipDescription;


}
```

Figura 5.4: Captura de pantalla en la que se muestra la definición de la clase abstracta `Propiedad`.

será añadido a todos los valores.

Puesto que, como se ha mencionado, se hace un gran uso de la clase `PropiedadSeleccionMultiple` para definir propiedades de tipo `Logic`, se define también una subclase denominada `PropiedadLogic`.

Esta subclase, cuya implementación puede verse en la Figura 5.6, no hace más que generar un `PropiedadSeleccionMultiple` ofreciendo constructores que requieran de menos parámetros, puesto que gran parte de ellos (como los valore posibles) son conocidos. Tanto es así, que se ofrece un constructor en el que únicamente se requiere el nombre y la `tooltipDescription`.

Por otro lado, la clase `PropiedadSimple` permite crear una propiedad que será renderizada como una casilla de texto, en la que podrá introducirse cualquier valor.

La clase `Pieza`, por su parte, no almacena directamente las propiedades, sino que almacena únicamente un array con sus valores en forma de `String`, de modo que la asociación entre la propiedad y el valor se realiza por el orden de definición de las mismas, es decir, el primer valor del array corresponde a la primera propiedad, el

```java
public class PropiedadSeleccionMultiple extends Propiedad<String> {
    public static final List<String> SELECCION_MULTIPLE_LOGICAL =
            List.of("'U'", "'X'", "'0'", "'1'", "'Z'", "'W'", "'L'", "'H'", "'-'");
    public static final String PREFIX_SELECCION_MULTIPLE_LOCICAL = ModelicaGenerator.LOGIC + ".";

    @Getter
    @Setter
    private List<String> valoresPosibles;

    public PropiedadSeleccionMultiple(String valor, String nombre, List<String> valoresPosibles, String unidad,
                                      String tooltipDescription) {...}

    public PropiedadSeleccionMultiple(String valor, String nombre, List<String> valoresPosibles, String prefix,
                                      String unidad, String tooltipDescription) {...}
}
```

Figura 5.5: Captura de pantalla en la que se muestra la definición de la clase `PropiedadSeleccionMultiple`.

```java
public class PropiedadLogic extends PropiedadSeleccionMultiple {


    public PropiedadLogic(String nombre, String valor, String tooltipDescription) {
        super(valor, nombre, PropiedadSeleccionMultiple.SELECCION_MULTIPLE_LOGICAL,
                PropiedadSeleccionMultiple.PREFIX_SELECCION_MULTIPLE_LOCICAL,
                ModelicaGenerator.LOGIC, tooltipDescription);
    }

    public PropiedadLogic(String nombre, String tooltipDescription) {
        this(nombre,  valor: "'0'", tooltipDescription);
    }
}
```

Figura 5.6: Captura de pantalla en la que se muestra la definición de la clase `PropiedadLogic`.

segundo valor a la segunda propiedad, y así sucesivamente. Esto, si bien puede parecer más confuso, permite una gestión más sencilla de los valores de las propiedades, puesto que evita la necesidad de conectarlos de manera directa a una instancia de `Propiedad` que está relacionada con el `TipoPieza`, y no con la `Pieza`.

## 5.5. Punto

Como se ha comentado en secciones anteriores, el `Punto` supone una parte esencial de esta aplicación, no solo por lo que semánticamente representa, puesto que permite posicionar los distintos elementos en pantalla; sino también porque esta clase contiene la lógica necesaria para gestionar el correcto funcionamiento de ciertas

características de visualización, como el poder desplazarse por la zona de trabajo, o poder acercarla/alejarla.

En primer lugar, esta clase utiliza una anotación de JPA cuya funcionalidad no ha sido detallada anteriormente: `@Embeddable`. Esta anotación permite que las variables presentes en esta clase sean "embebidas" en la entidad que la utiliza, en este caso `Pieza`. Es decir, en lugar de existir una tabla con los distintos `Puntos`, y contar `Pieza` con una referencia a uno de los puntos, es la propia tabla `Pieza` que añade columnas para almacenar la información sobre el `Punto` (en este caso, las coordenadas `x` e `y`). Posteriormente, al ser leído de base de datos, JPA transforma la información de estas columnas a una instancia de la clase `Punto`. Todo esto se realiza de manera automática y completamente transparente para el programador.

```
@Embeddable
@NoArgsConstructor
public class Punto {
    @Transient
    private static final float ESCALA_MIN = 0.1f, ESCALA_MAX = 2f;

    //Coordenadas de referencia en las que se sitúa el panel
    @Transient
    private static int referenciaX = 0, referenciaY = 0;
    @Transient
    @Getter
    private static double escala = 1;

    //Coordenadas virtuales en un hipotético plano infinito
    private double x, y;
```

Figura 5.7: Captura de pantalla en la que se muestran las propiedades de la clase `Punto`.

Las propiedades con las que cuenta esta clase, que pueden verse en la Figura 5.7, son las que siguen:

- Las constantes `ESCALA_MIN` y `ESCALA_MAX` representan la escala mínima y máxima para acercar/alejar el circuito. Si se intenta acercar o alejar el circuito por encima o por debajo de dichos valores, la acción no surtirá efecto alguno.
- `referenciaX` y `referenciaY` son las coordenadas de la esquina superior iz-

quierda del área de trabajo dentro del plano infinito. Esto significa que, si las dimensiones del área de trabajo son `w` y `h`, lo que se mostrará en pantalla al usuario será el cuadrado definido por las esquinas (`referenciaX`, `referenciaY`), (`referenciaX`+`w`, `referenciaY`), (`referenciaX`, `referenciaY`+`h`) y (`referenciaX`+`w`, `referenciaY`+`h`). En definitiva, son estas variables las que permiten el desplazamiento por el área de trabajo. Estas variables están anotadas con `@Transient`, lo cual significa que no son almacenadas y que, al cargar un circuito, la referencia se inicia siempre en $(0, 0)$. Además son variables estáticas, lo cual significa que tienen el mismo valor para todas las instancias de `Punto`.

- De manera similar, la variable `escala` es la que permite al usuario acercarse y alejarse del circuito. Su funcionamiento se basa, sencillamente, en actuar como multiplicador para los valores de los puntos y las distintas dimensiones utilizadas en la aplicación (por ejemplo, para las imágenes de los componentes). Además de ser estática y estar anotada como `@Transient`, esta variable cuenta también con una anotación `@Getter`, puesto que se hace referencia a ella en otros puntos de la aplicación, como en la carga de imágenes.

- Finalmente, las variables `x` e `y` representan las coordenadas del punto dentro de un plano infinito y sin escala. Esto significa que las coordenadas son absolutas, y que serán escaladas o desplazadas posteriormente mediante otros métodos.

Para poder interactuar con estas variables, la clase `Punto` ofrece una serie de métodos, con distintas funcionalidades. En primer lugar, se utilizan métodos destinados a la conversión entre la clase propia `Punto`, utilizada en esta aplicación, y la clase estándar de Java `Point`, utilizada en los componentes de Swing. Estos métodos incluyen un constructor con un `Point` como parámetro, y un Getter `getPoint()`.

Además, existen métodos para modificar la escala y la referencia, mostrados en la Figura 5.8. En el primer caso, se incrementa la variable `escala` en una cantidad `dZ` (que podrá ser positiva o negativa), garantizando siempre que el resultado se encuentre dentro del rango de valores aceptados, dictado por las variables `ESCALA_MIN` y `ESCALA_MAX`. En el caso de esta aplicación, al girar la rueda del ratón se utilizan

```
public static void reescalar(float dZ) {
    escala = Math.min(ESCALA_MAX,
            Math.max(ESCALA_MIN, escala + dZ)); //Asegurar que escala está entre min y max
    System.out.println("Nueva escala: " + escala);
}

public static void desplazarReferencia(int dX, int dY) {
    referenciaX += dX;
    referenciaY += dY;
}
```

Figura 5.8: Captura de pantalla en la que se muestran las funciones de la clase `Punto` que permiten desplazar la referencia y modificar la escala.

valores de `dZ` de 0.1 o $-0.1$, según la dirección del giro.

Por otro lado, para desplazar la referencia se utilizan, de manera similar, los incrementos `dX` y `dY` para las variables `referenciaX` y `referenciaY`, respectivamente, esta vez sin ninguna restricción sobre los valores.

Finalmente, esta clase ofrece los métodos mostrados en la Figura 5.9, que permiten interactuar con las coordenadas `x`, `y` tanto de manera directa (accediendo y modificando las variables "virtuales", es decir, las coordenadas absolutas en el plano infinito) como definiendo las coordenadas relativas a la sección del plano mostrada en el área de trabajo[1]. De este modo, cuando el usuario arrastra un componente al área de trabajo, es posible configurar las coordenadas en base a esta referencia y que, de manera automática, `Punto` calcule las coordenadas absolutas del componente. Del mismo modo, cuando se almacenan los `Puntos` en la base de datos, se almacenan en sus coordenadas absolutas, y es el área de trabajo que, mediante los métodos `getX` y `getY` obtiene las coordenadas relativas a la sección mostrada. Es importante destacar que, para simplificar la interacción con esta clase, y para hacer todo este proceso más transparente a la clase cliente (aquella que "utiliza" a `Punto`), los métodos `getX` y `getY` devuelven, por defecto, las coordenadas relativas al área de trabajo. Del mismo modo, al crear un `Punto` indicando unas coordenadas `x` e `y`, estas se tratan como coordenadas relativas, mediante los métodos `setXPanel` y `setYPanel`.

[1]Estas últimas son aquellas cuyo nombre acaba en *-Panel*, debido a que la clase que representa el área de trabajo es `PanelCircuito`.

```
public int getX() { return (int) (x * escala + referenciaX); }

private void setXPanel(int xPanel) { x = (xPanel - referenciaX) / escala; }

public int getY() { return (int) ((y * escala + referenciaY)); }

private void setYPanel(int yPanel) { y = (yPanel - referenciaY) / escala; }

public int getXVirtual() { return (int) x; }

public int getYVirtual() { return (int) y; }

public void setXVirtual(double x) { this.x = x; }

public void setYVirtual(double y) { this.y = y; }
```

Figura 5.9: Captura de pantalla en la que se muestran las funciones de la clase `Punto` que permiten trabajar con la referencia de las coordenadas en el plano infinito (virtual) y en el área de trabajo (panel).

## 5.6. Repositorios

Para la interacción con la base de datos, la aplicación utiliza una serie de *repositorios*, que extienden de una clase abstracta genérica `AbstractRepository`. Esta clase ofrece una serie de métodos genéricos para escribir, leer o modificar de la base de datos, tal y como se muestra en la Figura 5.10.

Estos métodos incluyen iniciar y finalizar una transacción que, en el framework JPA, representa un conjunto de acciones sobre la base de datos, de manera que cuando se va a realizar una serie de cambios relacionados, estos deberían ser parte de la misma transacción para, entre otras cosas, permitir el rollback (anulación de los cambios y vuelta a la versión anterior) en caso de error. De este modo, el cliente del repositorio es quien inicia la transacción, realiza todos los cambios, y finalmente la finaliza.

Además, `AbstractRepository` cuenta con dos métodos abstractos (es decir, deben ser implementados por la subclase) `getIdElemento` y `getObjectClass`. Estos se utilizan porque, para algunas de las consultas, es necesario conocer el id o la clase

```
public abstract class AbstractRepository<T> {
    9 usages
    protected static EntityManager em;
    4 usages
    protected static EntityTransaction currentTransaction;
    3 usages  elKuston
    public AbstractRepository() {...}
    1 usage  elKuston
    public static void startTransaction() {...}
    1 usage  elKuston
    public static void endTransaction() {...}
    1 usage  elKuston
    public List<T> getAll() {...}
    3 usages  elKuston
    public T guardar(T t) {...}
    1 usage  3 implementations  new *
    protected abstract Integer getIdElemento(T elemento);
    2 usages  3 implementations  new *
    protected abstract Class<T> getObjectClass();
}
```

Figura 5.10: Captura de pantalla en la que se muestra el código de la clase `AbstractRepository`.

del objeto con el que se está trabajando y la manera de obtener esta información variará según la clase del objeto en cuestión. Por ello, se deja la implementación de los mismos a los repositorios específicos. Por ejemplo, en la Figura 5.11 se muestra la implementación de estos dos métodos realizada por la clase `PiezaRepository`.

Esto representa, tal y como se describe en la sección 4.3.5, una implementación del patrón estrategia.

## 5.7. Generación de código Modelica

La parte más importante de este proyecto es, probablemente, la generación del código Modelica, es decir, la conversión del circuito diseñado por el usuario en un código ejecutable en los editores Modelica. Esto se hace mediante la clase `ModelicaGenerator`, que contiene toda la lógica necesaria para esta conversión.

En primer lugar, este fichero define una serie de constantes, que se muestran en la Figura 5.12.

Estas constantes son de dos tipos: aquellas cuyo nombre finaliza con `_IMPORT` representan el paquete de Modelica que debe ser importado (por ejemplo, `TRIS-`

```java
public class PiezaRepository extends AbstractRepository<Pieza> {
    1 usage   elKuston
    @Override
    protected Integer getIdElemento(Pieza elemento) {
        return elemento.getIdPieza();
    }

    2 usages   elKuston
    @Override
    protected Class<Pieza> getObjectClass() {
        return Pieza.class;
    }
}
```

Figura 5.11: Captura de pantalla en la que se muestra la implementación de `AbstractRepository` llevada a cabo por `PiezaRepository`.

```java
public static final String BASIC = "B";
public static final String LOGIC = "L";
public static final String DIGITAL = "D";
public static final String SOURCES = "S";
public static final String GATES = "G";
public static final String TRISTATES = "TS";
public static final String MULTIPLEXERS = "MP";
public static final String SI = "SI";
private static final String DIGITAL_IMPORT = "Modelica.Electrical.Digital";
private static final String BASIC_IMPORT = "Modelica.Electrical.Digital.Basic";
private static final String LOGIC_IMPORT = "Modelica.Electrical.Digital.Interfaces.Logic";
private static final String SOURCES_IMPORT = "Modelica.Electrical.Digital.Sources";
private static final String GATES_IMPORT = "Modelica.Electrical.Digital.Gates";
private static final String TRISTATES_IMPORT = "Modelica.Electrical.Digital.Tristates";
private static final String MULTIPLEXERS_IMPORT = "Modelica.Electrical.Digital.Multiplexers";
private static final String SI_IMPORT = "Modelica.Units.SI";
```

Figura 5.12: Captura de pantalla en la que se muestran las constantes definidas en `GeneradorModelica`.

TATES_IMPORT, mientras que las homónimas que no terminan en _IMPORT (como TRISTATES) representan el alias con el que se importan. Esto se puede ver de manera más clara en la Figura 5.13, en el que se muestran las sentencias que se generan para la importación de las librerías.

```
private static String imports() {
    StringJoiner sj = new StringJoiner( delimiter: ";\n\t");
    sj.add("import " + DIGITAL + " = " + DIGITAL_IMPORT);
    sj.add("import " + BASIC + " = " + BASIC_IMPORT);
    sj.add("import " + LOGIC + " = " + LOGIC_IMPORT);
    sj.add("import " + SOURCES + " = " + SOURCES_IMPORT);
    sj.add("import " + GATES + " = " + GATES_IMPORT);
    sj.add("import " + TRISTATES + " = " + TRISTATES_IMPORT);
    sj.add("import " + MULTIPLEXERS + " = " + MULTIPLEXERS_IMPORT);

    sj.add("import " + SI + " = " + SI_IMPORT);
    sj.add("\n");
    return sj.toString();
}
```

Figura 5.13: Captura de pantalla en la que se muestra la generación de código relativa a la importación de paquetes.

La generación del código se lleva a cabo mediante el método `generarCodigoModelica`, que puede verse en la Figura 5.14. Este código invoca los distintos métodos que generan las distintas secciones del código Modelica, utilizando un `StringBuilder` para unir las cadenas de texto devueltas. Así, el código se encuentra modularizado de manera que cada método se ocupa de generar una sección del código concreta, y devolver una `String` que la contenga.

```
public static String generarCodigoModelica(Circuito circuito) {
    StringBuilder codigoModelica = new StringBuilder();
    codigoModelica.append("model ").append(modelName(circuito)).append("\n\t");
    codigoModelica.append(imports()).append("\n\t");
    codigoModelica.append(declaraciones(circuito));
    codigoModelica.append("equation\n\t").append(connect(circuito)).append("\n\t");
    codigoModelica.append(generarAnotacionesConexiones(circuito)).append(";\n");
    codigoModelica.append("end ").append(modelName(circuito)).append(";");

    return codigoModelica.toString();
}
```

Figura 5.14: Captura de pantalla en la que se muestran las constantes definidas en `GeneradorModelica`.

La primera función que se utiliza, `modelName()`, extrae el nombre del circuito en caso de que este tenga (realizando las pertinentes comprobaciones de que el nombre

no esté vacío, no contenga espacios, etc.) y, en otro caso devuelve el nombre genérico "CircuitoAutogenerado", para aquellos circuitos que, por ejemplo, aún no hayan sido guardados en base de datos (y, por tanto, no tengan nombre).

A continuación se incluyen los `imports` mediante el método ya ilustrado en la Figura 5.13 para, posteriormente, generar las declaraciones de las variables mediante el método `declaraciones()`, que puede verse en la Figura 5.15.

```
private static String declaraciones(Circuito circuito) {
    StringJoiner sj = new StringJoiner( delimiter: ";\n\t");
    for (Pieza p : circuito.getComponentes()) {
        List<String> lineasDeclaracion = generarDeclaracionPieza(p);
        for (String s : lineasDeclaracion) {
            sj.add(s);
        }
    }

    sj.add("\n");
    return sj.toString();
}
```

Figura 5.15: Captura de pantalla en la que se muestra el método `declaraciones`.

El método `declaraciones()` utiliza un `StringJoiner` para unir las distintas declaraciones de las piezas del circuito, sobre las que itera. Para generar la declaración de cada una de las piezas, sin embargo, hace uso del método auxiliar `generarDeclaracionPieza`, visible en la Figura 5.16, sobre el que delega parte de su funcionalidad.

```
private static List<String> generarDeclaracionPieza(Pieza p) {
    List<String> declaracion = new ArrayList<>();
    List<Propiedad> propiedadesPieza = p.getTipoPieza().getPropiedades();
    for (int i = 0; i < propiedadesPieza.size(); i++) {
        Propiedad prop = propiedadesPieza.get(i);
        declaracion.add(String.format("parameter %s %s = %s", prop.getUnidad(),
                nombrePropiedad(p, prop), p.getValoresPropiedades()[i]));
    }

    declaracion.add(
            String.format("%s %s %s %s", p.getTipoPieza().getClaseModelica(), nombrePieza(p),
                    generarAsignacionParametrosPieza(p), generarAnotacion(p)));
    return declaracion;
}
```

Figura 5.16: Captura de pantalla en la que se muestra el método `generarDeclaracionPieza`.

Este método genera, en primer lugar, un parámetro para cada una de las propiedades de la `Pieza`, asignándoles el valor especificado por el usuario; y, posteriormente genera la declaración de la propia pieza, utilizando el método `generarAsignacionParametrosPieza` para listar las distintas propiedades de la variable y asociarlas a los distintos parámetros declarados (Figura 5.17); y `generarAnotacion` para generar la anotación de la `Pieza` (Figura 5.18).

```
private static String generarAsignacionParametrosPieza(Pieza p) {
    List<ConectorTemplate> conectoresMultiples =
            p.getTipoPieza().getConectoresConNombre().stream().filter(ConectorTemplate::isMultiple).toList();

    List<Propiedad> propiedadesPieza = p.getTipoPieza().getPropiedades();
    String res = "";
    if (propiedadesPieza.size() > 0 || conectoresMultiples.size() > 0) {
        StringJoiner sj = new StringJoiner( delimiter: ", ");
        for (int i = 0; i < propiedadesPieza.size(); i++) {
            Propiedad prop = propiedadesPieza.get(i);
            sj.add(prop.getNombre() + " = " + nombrePropiedad(p, prop));
        }

        for (int i = 0; i < conectoresMultiples.size(); i++) {
            String nombreAtributoNumPines = conectoresMultiples.get(i).getNombreNumPines();
            int numPines = p.getNPinesConectorMultiple(i);
            sj.add(nombreAtributoNumPines + " = " + numPines);
        }

        res = "(" + sj + ")";
    }
```

Figura 5.17: Captura de pantalla en la que se muestra el método `generarAsignacionParametrosPieza`.

Este método recorre las distintas propiedades de la pieza y genera los parámetros de "constructor" en Modelica, es decir, una lista en la que se asigna a cada propiedaed del componente un parámetro del modelo, siendo el resultado de la forma `(prop1 = param_prop1, prop2 = param_prop2,...)`.

```
private static String generarAnotacion(Pieza p) {
    Punto pos = p.getPosicion();
    Punto bottom_left = new Punto(pos.getX(),  y: pos.getY() + p.getHeight());
    Punto top_right = new Punto( x: pos.getX() + p.getWidth(), pos.getY());
    Punto bottom_left_scaled = escalarPunto(bottom_left,  invertirY: true);
    Punto top_right_scaled = escalarPunto(top_right,  invertirY: true);
    double offset_x = 0, offset_y = 0;

    switch (p.getRotacion()) {
        case ROT_0 -> {

        }
        case ROT_90 -> {
            offset_x = -p.getHeight() * ESCALA_ANNOTATION;
        }
        case ROT_180 -> {
            offset_y = -p.getHeight() * ESCALA_ANNOTATION;
            offset_x = -p.getWidth() * ESCALA_ANNOTATION;
        }
        case ROT_270 -> {
            offset_y = -p.getHeight() * ESCALA_ANNOTATION;
        }
    }
    return String.format(
            "annotation (Placement(transformation(extent={{%d,%d},{%d,%d}}, rotation=%d, origin={%d,%d})))",
            (int) (0 + offset_x), //bottom left x
            (int) (-p.getHeight() * ESCALA_ANNOTATION - offset_y), //bottom left y
            (int) (p.getWidth() * ESCALA_ANNOTATION + offset_x), //top right x
            (int) (0 - offset_y), //top right y
            p.getRotacion().getAngulo(), //rotation
            (int) (pos.getX() * ESCALA_ANNOTATION),//origin x
            (int) (-pos.getY() * ESCALA_ANNOTATION));//origin y
}
```

Figura 5.18: Captura de pantalla en la que se muestra el método `generarAnotacion`.

`generarAnotacion`, por su parte, genera la annotation de Modelica que permite representar visualmente los componentes del circuito en los editores Modelica. Para ello, calcula la posición y rotación de la pieza, utilizando un factor de escala `ESCALA_ANNOTATION` para redimensionar el circuito, y devuelve una cadena de texto que contiene la annotation generada.

Una vez declaradas las variables, se escriben las ecuaciones que las relacionan que, en el caso de esta aplicación, son en su totalidad conexiones entre los distintos componentes. Para ello, se utiliza el método `connect()`, que puede verse en la Figura 5.19.

```
private static String connect(Circuito circuito) {
    StringJoiner sj = new StringJoiner( delimiter: ";\n\t");
    for (Conexion c : circuito.getConexiones()) {
        Conector salida =
                c.getOrigen().getTipoConector().equals(TipoConector.SALIDA) ? c.getOrigen() :
                        c.getDestino();
        Conector entrada =
                c.getOrigen().getTipoConector().equals(TipoConector.ENTRADA) ? c.getOrigen() :
                        c.getDestino();

        sj.add(String.format("connect(%s,%s)", nombreConector(salida), nombreConector(entrada)));
    }
    sj.add("\n");
    return sj.toString();
}
```

Figura 5.19: Captura de pantalla en la que se muestra el método `connect`.

Puede verse que este método, con el fin de mejorar la legibilidad del código generado, crea las conexiones de modo que el primer parámetro del método `connect` de Modelica es siempre la salida, y el segundo es la entrada.

De manera análoga a lo que se ha hecho con las `Piezas`, también se generan las anotaciones correspondientes a las `Conexiones`, mediante el método `generarAnotacionesConexiones`, que puede verse detallado en la Figura 5.20, y que de igual manera se apoya sobre el método `generarAnotacion` (Figura 5.21). Es importante destacar que, si bien este último método tiene el mismo nombre que el utilizado al generar las anotaciones de las `Piezas`, toman argumentos distintos (una `Conexion` y una `Pieza`, respectivamente) y son, por tanto, métodos distintos.

```
private static String generarAnotacionesConexiones(Circuito circuito) {
    StringJoiner sj = new StringJoiner( delimiter: ",\n\t\t");
    for (Conexion c : circuito.getConexiones()) {
        sj.add(generarAnotacion(c));
    }

    return String.format("annotation (Diagram(graphics={%s}))", sj);
}
```

Figura 5.20: Captura de pantalla en la que se muestra el método `generarAnotacionesConexiones`.

```
private static String generarAnotacion(Conexion c) {
    StringJoiner sj = new StringJoiner( delimiter: ",");
    c.getPuntosManhattan().stream().map(p -> escalarPunto(p, invertirY: true))
            .map(p -> String.format("{%d,%d}", p.getX(), p.getY())).forEachOrdered(sj::add);
    return String.format("Line(points={%s}, color={0,0,0})", sj);
}
```

Figura 5.21: Captura de pantalla en la que se muestra el método `generarAnotacion` utilizado para `Conexione`s.

En comparación a `generarAnotacion` para piezas, este método es considerablemente más sencillo, puesto que lo único que se hace es declarar un elemento de tipo `Line` con los puntos intermedios pertenecientes a la `Conexion`. Del mismo modo, `generarAnotacionesConexiones` no hace más que concatenar las distintas líneas generadas en una única cadena de texto.

Por último, se añade al código generado la sentencia `end`, utilizando el nombre del modelo, y se devuelve una única cadena de texto con todos los componentes generados hasta el momento. Esta cadena de texto podrá ser posteriormente utilizada para su exportación a un fichero de código Modelica, así como para su previsualización por parte del usuario en la ventana específicamente diseñada para ello.

## 5.8. Imágenes

Con el fin de mostrar gráficamente los distintos componentes del circuito, es necesario hacer uso de imágenes que se muestren en el área de trabajo en las posiciones en las que el usuario haya colocado el componente en cuestión. Para ello, cada `Pieza` contiene una imagen, que coincide con la utilizada por la librería estándar Modelica para dicho componente. Estas imágenes se encuentran almacenadas como recursos y, por lo tanto, se encuentran dentro del directorio Resources/media del proyecto. Cada imagen, además, está nombrada según el nombre del componente que representa.

Como ya se ha mencionado en capítulos anteriores, se ha diseñado una clase `ImageUtils` que centraliza todas las operaciones de carga, escalado y rotación de las imágenes, mediante los métodos apropiados.

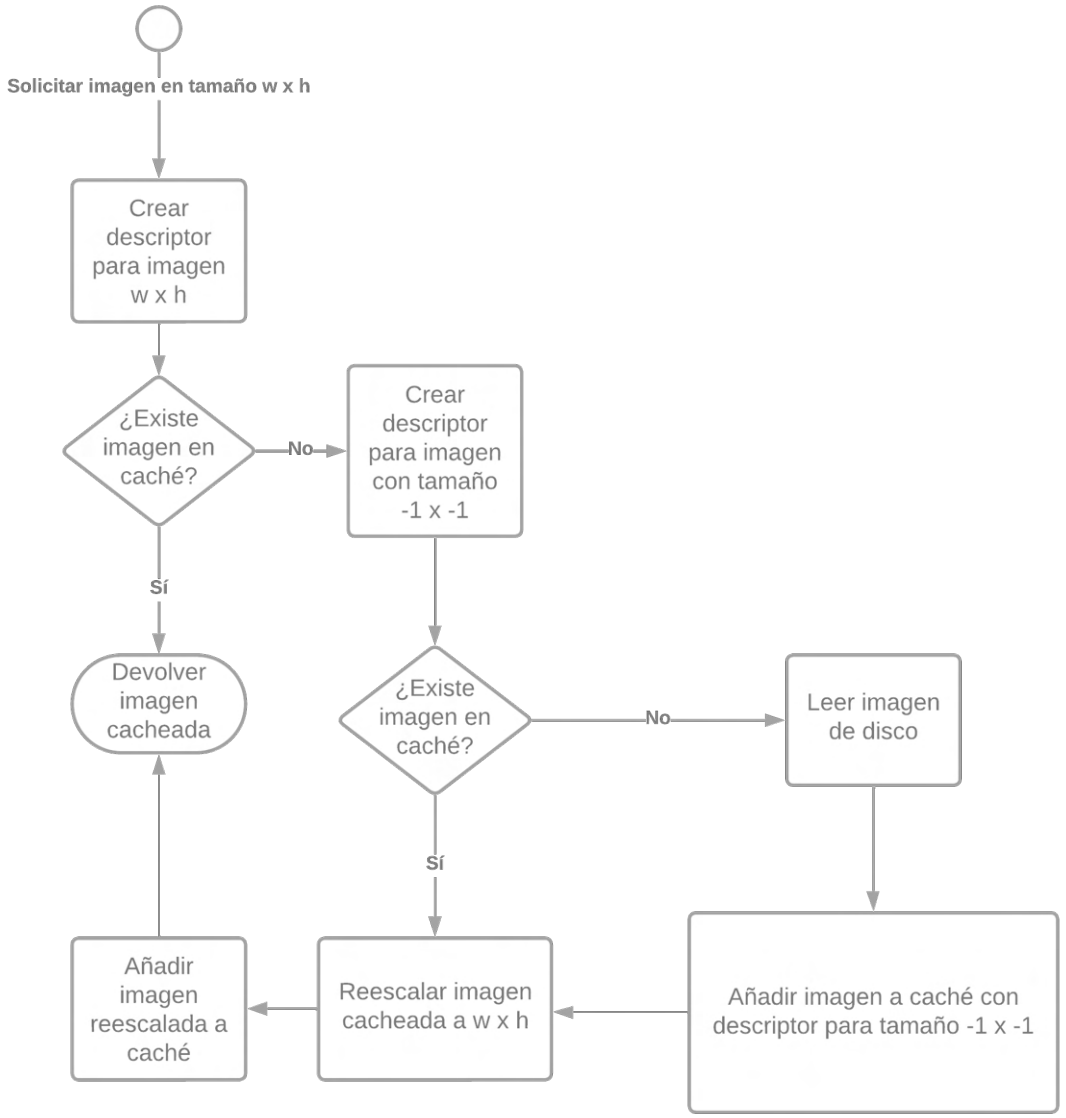



Figura 5.22: Diagrama de flujo para la carga de imágenes. Realizado mediante (Lucidchart, 2023).

Además de las funciones básicas de procesamiento de imagen ya mencionadas, esta clase cuenta con una "caché" que permite reducir la cantidad de operaciones de lectura de disco necesarias. Esta caché viene representada por una estructura de datos clave-valor, donde la clave viene representada por un objeto de tipo `DescriptorImagen`, y el valor es la propia imagen en caché. De este modo, cada vez que el cliente solicita la carga de una imagen, se comprueba si existe ya en caché y, de hacerlo, se devuelve sin necesidad de cargarla del disco. En la Figura 5.22 puede

verse un diagrama de flujo del proceso seguido al solicitarse la carga de una imagen, mientras que la Figura 5.23 muestra la implementación en Java de este proceso[2].

```java
private static Map<DescriptorImagen, ImageIcon> imagenesCacheadas = new HashMap<>();

public static ImageIcon cargarImageIcon(String pathImagen) {
    DescriptorImagen desc = new DescriptorImagen(pathImagen);
    if (!imagenesCacheadas.containsKey(desc)) {
        ClassLoader classLoader = Thread.currentThread().getContextClassLoader();
        URL resource = classLoader.getResource(pathImagen);
        System.out.println("Cargando " + pathImagen);
        imagenesCacheadas.put(desc, new ImageIcon(resource));
    }
    return imagenesCacheadas.get(desc);
}

public static ImageIcon cargarImagenEscalada(String pathImagen, int ancho, int alto, int modo) {
    DescriptorImagen desc = new DescriptorImagen(pathImagen, ancho, alto);
    if (!imagenesCacheadas.containsKey(desc)) {
        imagenesCacheadas.put(desc,
                rescalarImagen(cargarImageIcon(pathImagen), ancho, alto, modo));
    }
    return imagenesCacheadas.get(desc);
}
```

(a) Captura de pantalla en la que se muestra un fragmento de `ImageUtils` en el que se aprecia el funcionamiento de la caché

```java
@Data
@AllArgsConstructor
public class DescriptorImagen {
    private String pathImagen;
    private int width, height;

    public DescriptorImagen(String pathImagen) {
        this.pathImagen = pathImagen;
        this.width = -1;
        this.height = -1;
    }
}
```

(b) Captura de pantalla en la que se muestra la implementación de `DescriptorImagen`, componente fundamental de la caché de imágenes.

Figura 5.23: Capturas de pantalla en las que se muestran los métodos necesarios para el funcionamiento de la caché de imágenes.

Como se observa en la Figura 5.23a, cuando se solicita la carga de una imagen

[2]Se muestran tan solo dos de los métodos implicados ya que, si bien existen muchos otros en la clase, son suficientemente representantivos como para ilustrar el funcionamiento básico de la caché.

mediante `cargarImagenEscalada`, en primer lugar se comprueba si esta imagen existe en caché y, si no existe, se genera y se añade. Durante la generación, se invoca a `cargarImageIcon` que, a su vez, realiza un proceso similar, generando un `DescriptorImagen` basado únicamente el el path de la imagen, independientemente del tamaño. Como puede verse en la Figura 5.23b, además del `@AllArgsConstructor` existe un constructor que, recibiendo como parámetro únicamente el path de la imagen, asigna el ancho y el alto a -1. Así, `ImageUtils` vuelve a buscar en caché la imagen y, si no la encuentra, la carga del disco y la añade.

Es importante notar, en la definición de `DescriptorImagen`, que utiliza la anotación `@Data`, lo que incluye los métodos `equals` y `hashsCode`. Estos métodos son de vital importancia, puesto que esta clase se utiliza como clave en un diccionario, y por tanto es esencial poder comparar distintos `DescriptorImagen` entre sí, especialmente teniendo en cuenta que la estructura de datos utilizada es un `HashMap`, cuyo funcionamiento depende del resultado de `hashCode`.

## 5.9. Internacionalización

Para la internacionalización, la aplicación hace uso de los denominados Resource Bundle (paquete de recursos) de Java. Estos bundles son paquetes de recursos (textos, imágenes,...) agrupados según su nombre, y que se activan según distintas características. Un ejemplo de estos bundles se puede ver en el desarrollo de aplicaciones Android, donde para cada imagen que se desea utilizar, se crea en realidad todo un Resource Bundle, con el nombre de la imagen, que contienen la misma imagen en distintos tamaños, de manera que, según el tamaño de la pantalla del dispositivo, se selecciona una u otra. De modo similar, se pueden definir layouts (disposición de los elementos en pantalla) diferentes según el tamaño de la pantalla.

Entre todas las posibles propiedades según las cuales seleccionar uno u otro recurso, se encuentra el idioma del sistema. De este modo, se define un bundle `TextosAplicacion`, visible en la Figura 5.24, que contiene, actualmente, dos recursos: `TextosAplicacion.properties` y `TextosAplicacion_es_ES.properties`.

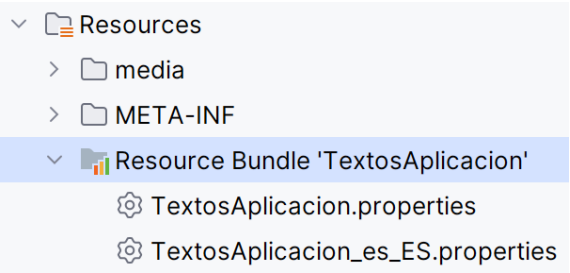



Figura 5.24: Captura de pantalla en la que se muestra el Resource Bundle utilizado para la internacionalización.

Dentro de este bundle, el segundo recurso, `TextosAplicacion_es_ES.properties` está nombrado con el sufijo `_es_ES`, que indica que es el recurso a utilizar cuando el idioma del sistema es el español. De igual modo, podrían añadirse otros ficheros para otros idiomas, sencillamente utilizando el sufijo de idioma adecuado. El primer recurso, `TextosAplicacion.properties`, no tiene ningún prefijo, por lo que es el utilizado por defecto en caso de que no exista un recurso más específico (en este caso, para cualquier idioma excepto el español). Así, tal y como se muestra en la Figura 5.25, `TextosAplicacion_es_ES.properties` contiene los textos en español, mientras que `TextosAplicacion.properties` contiene los textos en inglés.

```
menu_circuito_title=Circuito
menuitem_new_circuit=Nuevo circuito
menuitem_load_circuit=Cargar circuito
menuitem_save_circuit=Guardar circuito
menuitem_save_circuit_as=Guardar circuito como
menu_modelica_title=Modelica
menuitem_export_modelica=Exportar código Modelica
menu_view_title=Vista
menuitem_show_hide_piece_names=Mostrar/ocultar nombres de piezas
app_name=Lizard Clips
untitled_circuit=Circuito sin nombre
no_current_connection=No hay ninguna conexión en curso
```

(a) Captura de pantalla en la que se muestra un fragmento del recurso `TextosAplicacion_es_ES.properties`.

```
menu_circuito_title=Circuit
menuitem_new_circuit=New circuit
menuitem_load_circuit=Load circuit
menuitem_save_circuit=Save circuit
menuitem_save_circuit_as=Save circuit as
menu_modelica_title=Modelica
menuitem_export_modelica=Export Modelica code
menu_view_title=View
menuitem_show_hide_piece_names=Show/hide component names
app_name=Lizard Clips
untitled_circuit=Untitled circuit
no_current_connection=No connection is currently being made
```

(b) Captura de pantalla en la que se muestra un fragmento del recurso `TextosAplicacion.properties`.

Figura 5.25: Capturas de pantalla en las que se muestra el contenido de los ficheros de recursos para la internacionalización.

En estos ficheros, los textos de la aplicación están definidos con una estructura clave-valor, de modo que a cada texto le corresponde una clave que se utiliza como

identificador único. Como puede verse en la Figura 5.25, estas claves deben ser las mismas en ambos ficheros para que el texto sea obtenido correctamente.

Una vez definidos los textos en los ficheros, pueden extraerse mediante la clase `I18NUtils`, cuya implementación se muestra en la Figura 5.26, y que ofrece el método `getString`, que busca la cadena de texto indicada, según su nombre, en los recursos, y devuelve su valor. Además, como ya se ha señalado en la sección 4.3.3, esta clase utiliza el patrón singleton para asegurar que, en todo momento, existe, a lo sumo, una única instancia de la clase `ResourceBundle`, clase que se ocupa de leer el fichero de recursos adecuados para, después, devolver las cadenas de texto solicitadas.

```
public class I18NUtils {
    private static final String RESOURCE_BUNDLE_NAME = "TextosAplicacion";

    private static ResourceBundle textosAplicacion;

    public static String getString(String key) {
        if (textosAplicacion == null) {
            textosAplicacion = ResourceBundle.getBundle(RESOURCE_BUNDLE_NAME);
        }
        return textosAplicacion.getString(key);
    }
}
```

Figura 5.26: Captura de pantalla en la que se muestra el código fuente para la clase `I18NUtils`.

## 5.10. Conclusiones

En este capítulo se han detallado los aspectos más importantes de la implementación llevada a cabo para esta aplicación. Algunos de estos aspectos incluyen la implementación de `TipoPieza` y su relación con `Pieza`; la implementación de las propiedades, y el almacenamiento de sus valores en `Pieza`; la lógica contenida en `Punto`, que permite el desplazamiento y acercamiento/alejamiento por el área de trabajo; detalles relativos a la gestión de la base de datos; gestión de las imágenes y estrategias utilizadas para minimizar las operaciones de lectura; o todo lo relativo al uso de Resource Bundles para la internacionalización; haciendo especial énfasis en el que es el objetivo principal de esta aplicación: la generación de código Modelica.

A pesar de haberse tratado una gran cantidad de temas en este capítulo, es innegable que existen aún muchos detalles de implementación que, por no ser de especial interés, no se han tratado. Es por esto que, por si fuera de interés su consulta, se puede encontrar el código fuente completo en el Anexo B.

# 6. Pruebas

## 6.1. Introducción

En este capítulo, se validará el software desarrollado con los requisitos establecidos durante la sección 3.2, comprobando que, efectivamente, todos se cumplen; y se validará, además, el código generado por la aplicación, comparándolo con algunos ejemplos de referencia de la librería estándar de Modelica. Concretamente, se replicarán en la aplicación algunos de los ejemplos del paquete `Modelica.Electrical.Digital.Examples` (en adelante, simplemente `Examples`) y se compararán tanto el circuito desarrollado, de manera visual, como el código generado. Las imágenes y el código de referencia están extraídos de (MapleSoft, 2009).

## 6.2. Validación de requisitos

En esta sección, se muestran en funcionamiento las distintas características del software, y se correlaciona con el análisis de requisitos realizado previamente, verificando que se cumplen todos.

En primer lugar, la Figura 6.1 muestra la ventana principal de la aplicación. En ella, se puede observar cómo la pantalla está dividida en dos partes bien diferenciadas: la paleta de componentes, a la izquierda; y el área de trabajo, a la derecha. En el área de trabajo, se puede apreciar cómo hay un pequeño circuito construido, utilizando componentes de varios de los paquetes disponibles, como `Basic`, `Sources` o `Multiplexers`. El circuito se ha construido mediante Drag&drop. Además, se puede observar cómo el componente `not_3` se encuentra girado 90°.

En la paleta, puede verse cómo está dividida en dos secciones: a la izquierda, la distintas tabs con los distintos paquetes de componentes incluidos en la aplicación; y a la derecha, los componentes que pertenecen al paquete seleccionado.

Así, se cumple con los requisitos **RF-2**, **RF-2.1**, **RF-2.2**, **RF-2.3**, **RF-2.4**, **RF-2.5** y **RNF-1**.

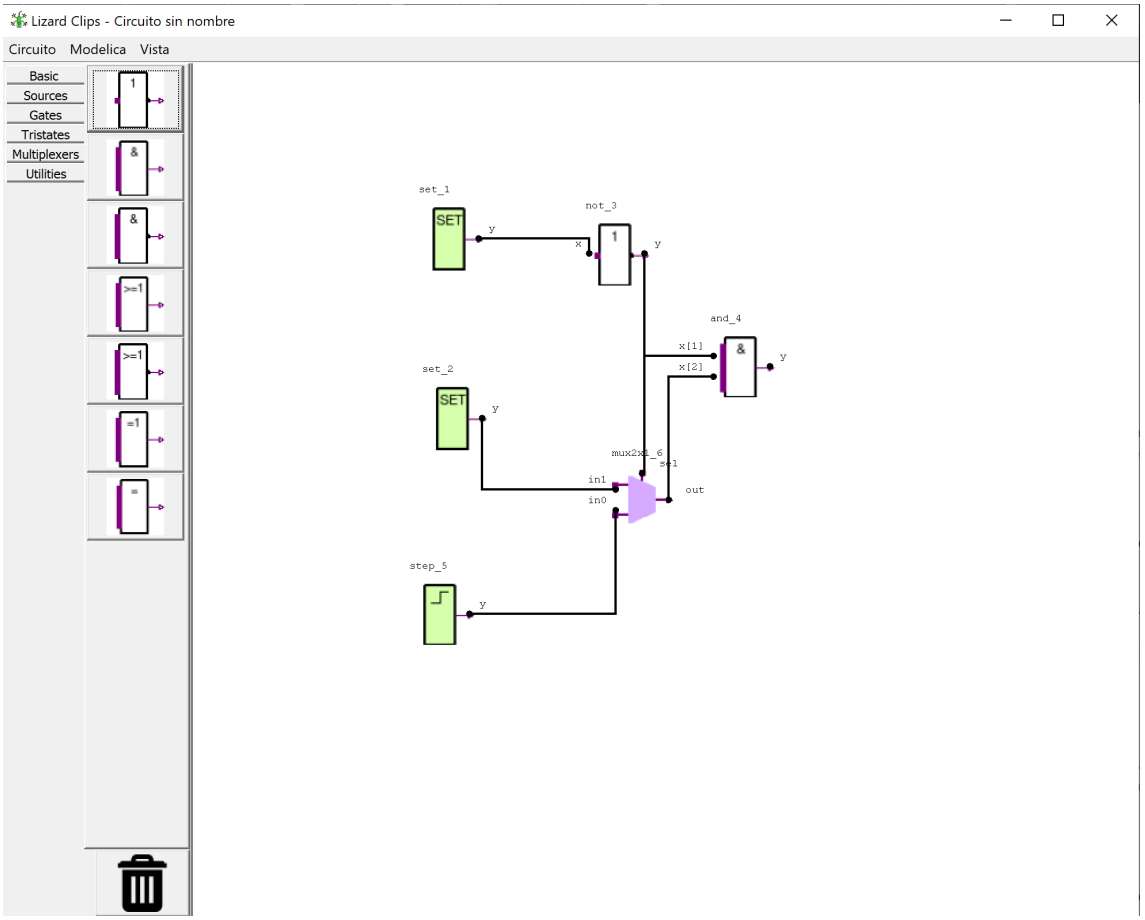



Figura 6.1: Ventana principal de la aplicación.

Observando la Figura 6.2, se puede ver la ventana de propiedades que emerge de hacer doble click sobre el componente `mux2x1_6`, en la que pueden editarse sus propiedades, así como el nombre de la pieza. De nuevo, al mantenerse el puntero sobre uno de las propiedades, se muestra el tooltip correspondiente.

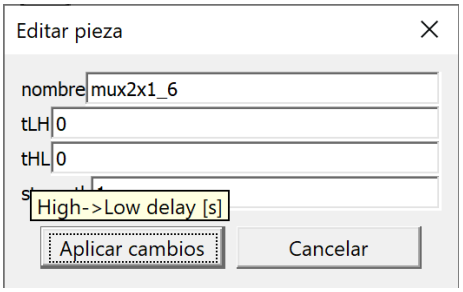



Figura 6.2: Ventana de edición de propiedades para la pieza `mux2x1_6`.

Así, se cumplen los requisitos **RF-2.10** y **RNF-6**. Además, como se observa en

la Figura 6.3, al introducir un nombre no válido para la pieza, la aplicación muestra un mensaje de error. Un error similar se muestra al asignar un nombre no válido al propio circuito, cumpliéndose así los requisitos **RF-1.6** y **RF-3.4**.

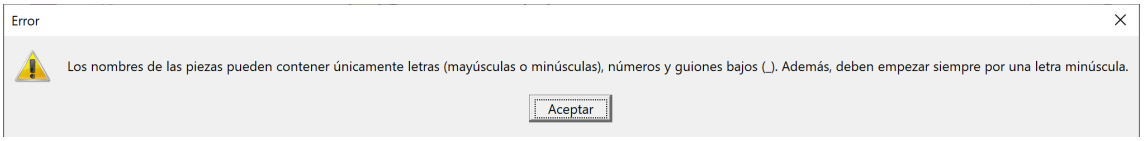
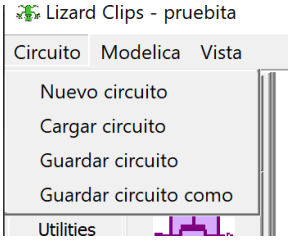
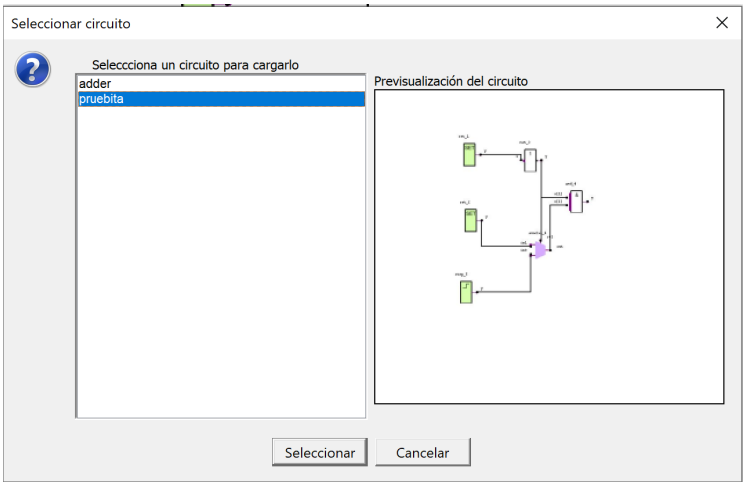



Figura 6.3: Mensaje de error que se muestra al introducir un nombre no válido para la pieza.

En la parte superior de la ventana principal, se muestran una serie de menús con distintas opciones. Dentro del menú *circuito*, se encuentran las opciones relativas a la gestión de circuitos en la base de datos (nuevo circuito, cargar, guardar y guardar como). En la Figura 6.4a se muestra este menú con sus opciones, mientras que la Figura 6.4b muestra la ventana se selección de circuitos utilizada para cargar un circuito de la base de datos.




(a) Menú Circuito.



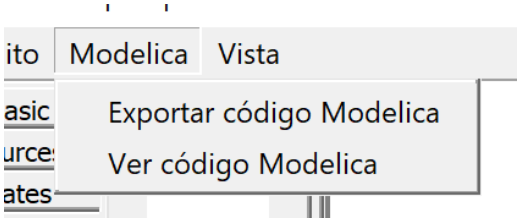

(b) Ventana de selección de circuitos desde la base de datos.

Figura 6.4: Gestión de circuitos desde la base de datos.

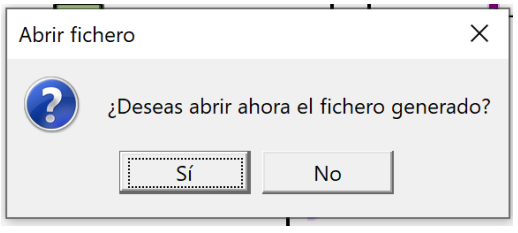
Así, se cumple con los requisitos **RF-1**, **RF-1.1**, **RF-1.2**, **RF-1.3**, **RF-1.4** y **RF-1.5**.

A la derecha de este menú, se encuentra el menú Modelica, en el que se recogen las funcionalidades relativas a la generación de código. Las opciones incluidas son,

como se muestra en la Figura 6.5a, exportar el código Modelica a un fichero `.mo`, o visualizarlo desde la propia aplicación. Al exportarlo, y tras seleccionar la ubicación y el nombre con el que quiere guardarse el fichero, se mostrará el mensaje que se puede ver en la Figura 6.5b, de modo que si se decide abrir el fichero en el momento, el sistema abrirá automáticamente un entorno de desarrollo Modelica si está instalado o, de lo contrario, preguntará al usuario con qué aplicación desea abrir el fichero. Además, al abrir el fichero generado con un programa como Dymola, se puede ver que el diagrama generado y representado por este software es análogo al de la aplicación, tal y como se muestra en la Figura 6.6.




(a) Menú Modelica.




(b) Mensaje solicitando si se desea abrir el fichero generado.

Figura 6.5: Opciones de generación de código.

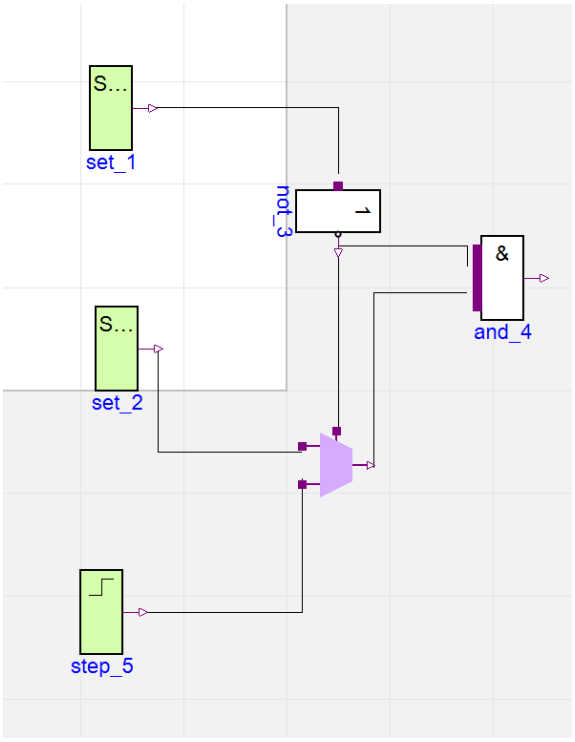



Figura 6.6: Representación esquemática del circuito generado en Dymola.

La visualización de código, por otra parte, se realiza dentro de la propia aplicación en una nueva ventana emergente que muestra y colorea el código generado, tal y como se observa en a Figura 6.7.

Visualizador de código Modelica

```
model pruebita
      import D = Modelica.Electrical.Digital;
      import B = Modelica.Electrical.Digital.Basic;
      import L = Modelica.Electrical.Digital.Interfaces.Logic;
      import S = Modelica.Electrical.Digital.Sources;
      import G = Modelica.Electrical.Digital.Gates;
      import TS = Modelica.Electrical.Digital.Tristates;
      import MP = Modelica.Electrical.Digital.Multiplexers;
      import U = Modelica.Electrical.Digital.Examples.Utilities;
      import SI = Modelica.Units.SI;


      parameter L set_1_x = L.'0';
      S.Set set_1 (x = set_1_x) annotation (Placement(transformation(extent={{0,-16},{16,0}}, rotation=0, origin={58,-37})));
      parameter L set_2_x = L.'0';
      S.Set set_2 (x = set_2_x) annotation (Placement(transformation(extent={{0,-16},{16,0}}, rotation=0, origin={59,-84})));
      B.Not not_3  annotation (Placement(transformation(extent={{0,0},{16,16}}, rotation=270, origin={102,-57})));
      B.And and_4 (n = 2) annotation (Placement(transformation(extent={{0,-16},{16,0}}, rotation=0, origin={134,-70})));
      parameter L step_5_before = L.'0';
      parameter L step_5_after = L.'1';
      parameter Real step_5_stepTime = 0;
      S.Step step_5 (before = step_5_before, after = step_5_after, stepTime = step_5_stepTime) annotation (Placement(transformation(extent={{0,-16},{16,0}}, rotation=0, origin={56,-135})));
      parameter SI.Time mux2x1_6_tLH = 0;
      parameter SI.Time mux2x1_6_tHL = 0;
      parameter Real mux2x1_6_strength = 1;
      MP.MUX2x1 mux2x1_6 (tLH = mux2x1_6_tLH, tHL = mux2x1_6_tHL, strength = mux2x1_6_strength) annotation (Placement(transformation(extent={{0,-15},{15,0}}, rotation=0, origin={102,-107})));

equation
      connect(set_1.y,not_3.x);
      connect(not_3.y,and_4.x[1]);
      connect(not_3.y,mux2x1_6.sel);
      connect(mux2x1_6.out,and_4.x[2]);
      connect(set_2.y,mux2x1_6.in1);
      connect(step_5.y,mux2x1_6.in0);


      annotation (Diagram(graphics={Line(points={{74,-45},{110,-45},{110,-58}}, color={0,0,0}),
            Line(points={{110,-72},{135,-72},{135,-76}}, color={0,0,0}),
            Line(points={{110,-72},{110,-108},{110,-108}}, color={0,0,0}),
            Line(points={{117,-115},{117,-81},{135,-81}}, color={0,0,0}),
            Line(points={{103,-112},{75,-112},{75,-92}}, color={0,0,0}),
            Line(points={{72,-143},{103,-143},{103,-117}}, color={0,0,0})}));
end pruebita;
```

Figura 6.7: Visualización del código generado.

Si, al intentar visualizar o exportar el código, el sistema detecta algún error grave en la composición del circuito, mostrará una alerta, como la que aparece en la Figura 6.8, informando de la situación.

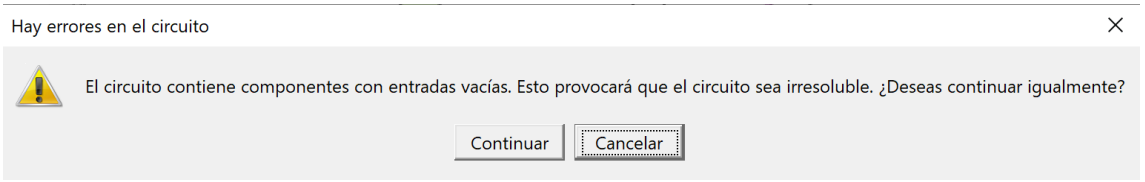



Figura 6.8: Alerta mostrada al intentar generar el código de un circuito con errores.

De este modo, se cumple con los requisitos **RF-3.1**, **RF-3.2**, **RF-3.3**, **RF-3.5** y **RNF-3**.

En los apartados anteriores, se ha verificado que se cumplen la mayoría de los requisitos establecidos. Sin embargo, existen requisitos, como el **RNF-1**, relativo a la funcionalidad Drag&drop que no pueden ser demostrados mediante un documento escrito. A pesar de ello, pueden ser comprobados por el lector utilizando el propio software, o verificando que se encuentran implementadas y documentadas en el Capítulo 5, relativo a la implementación, o en el Anexo B, en el que se encuentra el código fuente completo.

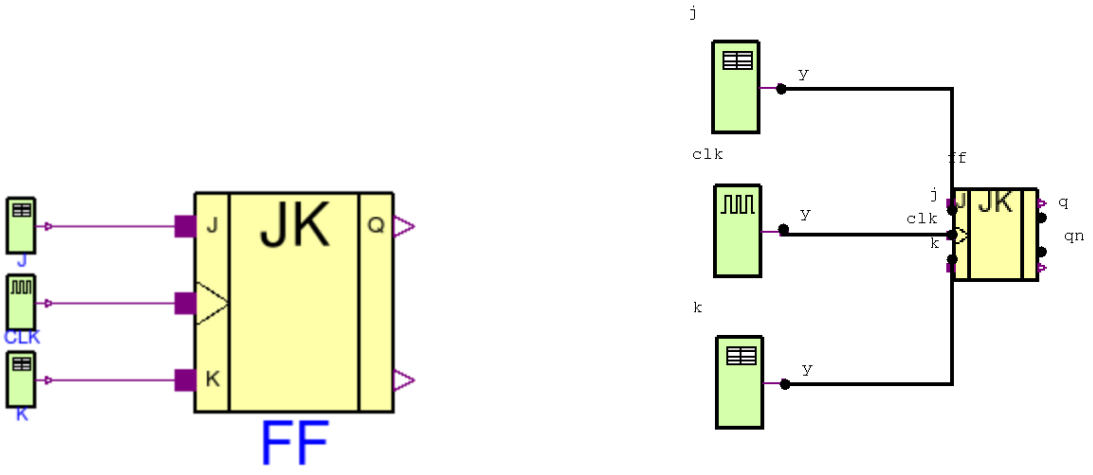
## 6.3. Ejemplos y pruebas

A continuación, se comparan una serie de ejemplos del paquete `Examples` con los mismos circuitos reconstruidos en la aplicación desarrollada. La estructura general para todos los ejemplos es la de mostrar la comparativa visual de los circuitos, y posteriormente comparar el código generado con el código de referencia. En general, se puede observar cómo ambos códigos son análogos, sin embargo, el código generado por la aplicación es considerablemente más extenso, debido, en primer lugar, a que todos los atributos de los componentes son declarados como parámetros del modelo (ocupando, por lo tanto, más líneas); y en segundo lugar, a que se generan las anotaciones para la representación gráfica. Si se observan, sin embargo, las sentencias

de código relativas al modelo en sí, así como la sección `equation`, se puede concluir como ambos códigos son, en efecto, equivalentes.

### 6.3.1. Flip-Flop

En esta sección se compara el software desarrollado con el ejemplo FlipFlop del paquete `Examples`, que prueba un Flip-Flop JK activado por un pulso. En la Figura 6.9a se muestra el circuito de referencia, mientras que la Figura 6.9b muestra el circuito creado mediante la aplicación.




(a) Circuito FlipFlop de referencia. Extraído de (MapleSoft, 2009).

(b) Circuito FlipFlop generado mediante la aplicación.

Figura 6.9: Comparación del circuito de referencia con el generado por la aplicación.

El código asociado al circuito generado se muestra en la Figura 6.10, mientras que el código de referencia puede verse en la Figura 6.11.

Visualizador de código Modelica

```
model example_flipflop
     import D = Modelica.Electrical.Digital;
     import B = Modelica.Electrical.Digital.Basic;
     import L = Modelica.Electrical.Digital.Interfaces.Logic;
     import S = Modelica.Electrical.Digital.Sources;
     import G = Modelica.Electrical.Digital.Gates;
     import TS = Modelica.Electrical.Digital.Tristates;
     import MP = Modelica.Electrical.Digital.Multiplexers;
     import U = Modelica.Electrical.Digital.Examples.Utilities;
     import SI = Modelica.Units.SI;


     parameter SI.Time jkff_1_delayTime = 0.01;
     parameter L jkff_1_q0 = L.'0';
     U.JKFF jkff_1 (delayTime = jkff_1_delayTime, q0 = jkff_1_q0)
          annotation (Placement(transformation(extent={{0,-15},{15,0}}, rotation=0, origin={86,-75})));
     parameter L j_x[:] = {L.'1',L.'0',L.'1',L.'0'};
     parameter Real j_t[:] = {50,100,145,200};
     parameter L j_y0 = L.'0';
     S.Table j (x = j_x, t = j_t, y0 = j_y0)
          annotation (Placement(transformation(extent={{0,-15},{15,0}}, rotation=0, origin={55,-53})));
     parameter L k_x[:] = {L.'1',L.'0',L.'1',L.'0'};
     parameter Real k_t[:] = {22,140,150,180};
     parameter L k_y0 = L.'0';
     S.Table k (x = k_x, t = k_t, y0 = k_y0)
          annotation (Placement(transformation(extent={{0,-15},{15,0}}, rotation=0, origin={55,-96})));
     parameter SI.Time clk_startTime = 0;
     parameter SI.Time clk_period = 10;
     parameter Real clk_width = 50;
     S.DigitalClock clk (startTime = clk_startTime, period = clk_period, width = clk_width)
          annotation (Placement(transformation(extent={{0,-15},{15,0}}, rotation=0, origin={56,-75})));

equation
     connect(j.y,jkff_1.j);
     connect(clk.y,jkff_1.clk);
     connect(k.y,jkff_1.k);


     annotation (Diagram(graphics={Line(points={{70,-61},{70,-79},{87,-79}}, color={0,0,0}),
          Line(points={{87,-83},{70,-83},{70,-83}}, color={0,0,0}),
          Line(points={{87,-87},{70,-87},{70,-104}}, color={0,0,0})}));
end example_flipflop;
```

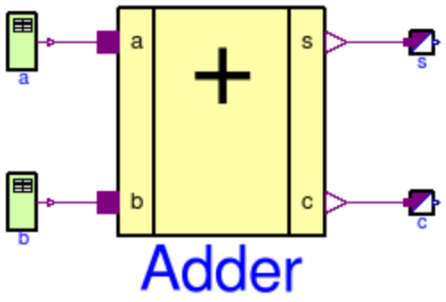
Figura 6.10: Código Modelica generado para el circuito FlipFlop.

```
model FlipFlop "Pulse Triggered Master Slave Flip-Flop"
  import D = Modelica.Electrical.Digital;
  import L = Modelica.Electrical.Digital.Interfaces.Logic;
  D.Examples.Utilities.JKFF FF;
  D.Sources.Clock CLK(period=10);
  D.Sources.Table J(
    y0=3,
    x={4,3,4,3},
    t={50,100,145,200});
  D.Sources.Table K(
    y0=3,
    x={4,3,4,3},
    t={22,140,150,180});
equation
  connect(J.y, FF.j);
  connect(CLK.y, FF.clk);
  connect(K.y, FF.k);
end FlipFlop;
```

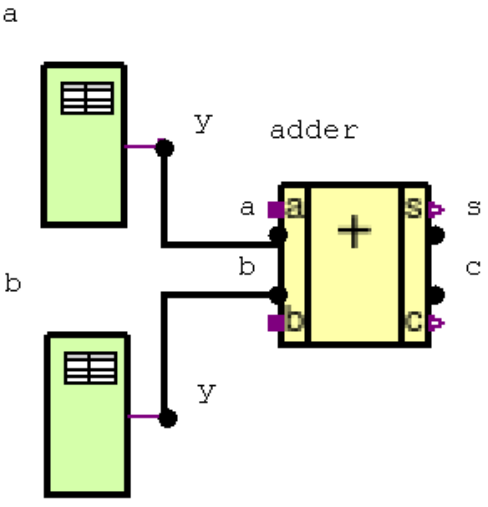
Figura 6.11: Código Modelica de referencia para el circuito FlipFlop. Extraído de (MapleSoft, 2009).

### 6.3.2. HalfAdder

En esta sección se compara el software desarrollado con el ejemplo HalfAdder del paquete `Examples`, que suma dos números binarios (sin entrada carry). En la Figura 6.12a se muestra el circuito de referencia, mientras que la Figura 6.12b muestra el circuito creado mediante la aplicación.




(a) Circuito HalfAdder de referencia. Extraído de (MapleSoft, 2009).




(b) Circuito HalfAdder generado mediante la aplicación.

Figura 6.12: Comparación del circuito de referencia con el generado por la aplicación.

El código asociado al circuito generado se muestra en la Figura 6.16, mientras que el código de referencia puede verse en la Figura 6.14.

Visualizador de código Modelica

```
model adder
     import D = Modelica.Electrical.Digital;
     import B = Modelica.Electrical.Digital.Basic;
     import L = Modelica.Electrical.Digital.Interfaces.Logic;
     import S = Modelica.Electrical.Digital.Sources;
     import G = Modelica.Electrical.Digital.Gates;
     import TS = Modelica.Electrical.Digital.Tristates;
     import MP = Modelica.Electrical.Digital.Multiplexers;
     import U = Modelica.Electrical.Digital.Examples.Utilities;
     import SI = Modelica.Units.SI;


     parameter L a_x[:] = {L.'1',L.'0',L.'1',L.'0'};
     parameter Real a_t[:] = {1,2,3,4};
     parameter L a_y0 = L.'0';
     S.Table a (x = a_x, t = a_t, y0 = a_y0)
          annotation (Placement(transformation(extent={{0,-15},{15,0}}, rotation=0, origin={41,-86})));
     parameter L b_x[:] = {L.'1',L.'0'};
     parameter Real b_t[:] = {2,4};
     parameter L b_y0 = L.'0';
     S.Table b (x = b_x, t = b_t, y0 = b_y0)
          annotation (Placement(transformation(extent={{0,-15},{15,0}}, rotation=0, origin={41,-104})));
     parameter SI.Time adder_delayTime = 0.3;
     U.HalfAdder adder (delayTime = adder_delayTime)
          annotation (Placement(transformation(extent={{0,-15},{15,0}}, rotation=0, origin={61,-96})));

equation
     connect(b.y,adder.a);
     connect(a.y,adder.b);


     annotation (Diagram(graphics={Line(points={{56,-112},{62,-112},{62,-107}}, color={0,0,0}),
          Line(points={{56,-94},{56,-102},{62,-102}}, color={0,0,0})}));
end adder;
```

Figura 6.13: Código Modelica generado para el circuito HalfAdder.

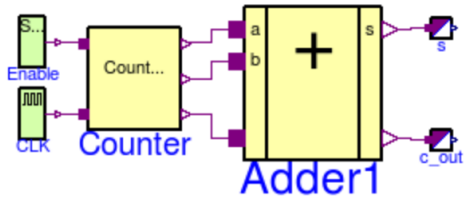
```
model HalfAdder
 "adding circuit for binary numbers without input carry bit"
  import Modelica.Electrical.Digital;

  Sources.Table a(
    t={1,2,3,4},
    x={4,3,4,3},
    y0=3);
  Sources.Table b(
    x={4,3},
    t={2,4},
    y0=3);
  Digital.Examples.Utilities.HalfAdder Adder(delayTime=0.3);
  Digital.Converters.LogicToReal s;
  Digital.Converters.LogicToReal c;
equation
  connect(b.y,Adder. b);
  connect(a.y,Adder. a);
  connect(Adder.s, s.x[1]);
  connect(Adder.c, c.x[1]);
end HalfAdder;
```

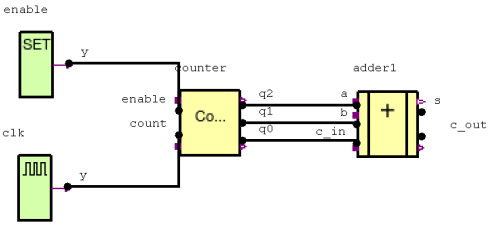
Figura 6.14: Código Modelica de referencia para el circuito HalfAdder. Extraído de (MapleSoft, 2009).

### 6.3.3. FullAdder

En esta sección se compara el software desarrollado con el ejemplo FullAdder del paquete `Examples`, que suma dos números binarios dando un resultado completo (con carry). En la Figura 6.15a se muestra el circuito de referencia, mientras que la Figura 6.15b muestra el circuito creado mediante la aplicación.




(a) Circuito FullAdder de referencia. Extraído de (MapleSoft, 2009).




(b) Circuito FullAdder generado mediante la aplicación.

Figura 6.15: Comparación del circuito de referencia con el generado por la aplicación.

El código asociado al circuito generado se muestra en la Figura **??**, mientras que el código de referencia puede verse en la Figura 6.17.

Visualizador de código Modelica

```
model example_fulladder
     import D = Modelica.Electrical.Digital;
     import B = Modelica.Electrical.Digital.Basic;
     import L = Modelica.Electrical.Digital.Interfaces.Logic;
     import S = Modelica.Electrical.Digital.Sources;
     import G = Modelica.Electrical.Digital.Gates;
     import TS = Modelica.Electrical.Digital.Tristates;
     import MP = Modelica.Electrical.Digital.Multiplexers;
     import U = Modelica.Electrical.Digital.Examples.Utilities;
     import SI = Modelica.Units.SI;


     parameter SI.Time adder1_delayTime = 0;
     U.FullAdder adder1 (delayTime = adder1_delayTime)
          annotation (Placement(transformation(extent={{0,-16},{16,0}}, rotation=0, origin={99,-80})));
     parameter SI.Time clk_startTime = 0;
     parameter SI.Time clk_period = 1;
     parameter Real clk_width = 50;
     S.DigitalClock clk (startTime = clk_startTime, period = clk_period, width = clk_width)
          annotation (Placement(transformation(extent={{0,-15},{15,0}}, rotation=0, origin={21,-94})));
     parameter L enable_x = L.'1';
     S.Set enable (x = enable_x)
          annotation (Placement(transformation(extent={{0,-15},{15,0}}, rotation=0, origin={22,-67})));
     U.Counter3 counter
          annotation (Placement(transformation(extent={{0,-15},{15,0}}, rotation=0, origin={60,-80})));

equation
     connect(counter.q2,adder1.a);
     connect(enable.y,counter.enable);
     connect(counter.q0,adder1.c_in);
     connect(counter.q1,adder1.b);
     connect(clk.y,counter.count);


     annotation (Diagram(graphics={Line(points={{75,-84},{100,-84}}, color={0,0,0}),
          Line(points={{36,-74},{60,-74},{60,-85}}, color={0,0,0}),
          Line(points={{75,-91},{100,-91},{100,-92}}, color={0,0,0}),
          Line(points={{75,-87},{100,-87},{100,-88}}, color={0,0,0}),
          Line(points={{36,-102},{60,-102},{60,-90}}, color={0,0,0})}));
end example_fulladder;
```

Figura 6.16: Código Modelica generado para el circuito FullAdder.

```
model FullAdder "Full 1 Bit Adder Example"
  import D = Modelica.Electrical.Digital;
  import L = Modelica.Electrical.Digital.Interfaces.Logic;

  Digital.Examples.Utilities.FullAdder Adder1;
  Digital.Converters.LogicToReal s;
  Digital.Converters.LogicToReal c_out;
  Digital.Examples.Utilities.Counter3 Counter;
  Digital.Sources.Set Enable(x=L.'1');
  Digital.Sources.Clock CLK;
equation
  connect(Adder1.s, s.x[1]);
  connect(Adder1.c_out, c_out.x[1]);
  connect(CLK.y, Counter.count);
  connect(Enable.y, Counter.enable);
  connect(Counter.q2, Adder1.a);
  connect(Counter.q1, Adder1.b);
  connect(Counter.q0, Adder1.c_in);
end FullAdder;
```

Figura 6.17: Código Modelica de referencia para el circuito FullAdder. Extraído de (MapleSoft, 2009).

## 6.4. Conclusiones

En este capítulo se ha realizado una verificación del software desarrollado, con el fin de comprobar que cumple con los requisitos solicitados. Para ello, se han comprobado, uno a uno, los requisitos verificables (si bien, algunos de estos no pueden ser demostrados por escrito, como la funcionalidad Drag&drop, en cuyo caso se remite al lector a apartados relativos a la implementación, o a probar el propio software). Además, para verificar que el código generado es válido, ejecutable, y que cumple con unos requisitos mínimos de calidad, se han replicado con la aplicación desarrollada algunos de los ejemplos presentes en el paquete `Examples` de la librería estándar de Modelica, y se ha comparado el resultado, observándose que el código generado es análogo y equivalente al código de referencia.

# 7. Conclusiones y trabajos futuros

## 7.1. Introducción

Tras mostrar todo el proceso de desarrollo del software y validar su funcionalidad, se presentan en este capítulo las conclusiones finales del trabajo, realizando un resumen del trabajo realizado y planteando algunas reflexiones al respecto. Finalmente, se concluye con algunas posibles líneas de trabajo futuras a partir del proyecto descrito en este documento.

## 7.2. Conclusiones

El presente trabajo ha propuesto un software diseñado con la finalidad de simplificar el uso de ciertos entornos de modelado basados en el lenguaje Modelica, para permitir que usuarios sin conocimientos técnicos de programación o de este lenguaje en concreto, puedan aprovechar todas sus funcionalidades y llevar a cabo simulaciones y experimentos del modo más cómodo posible. Para ello, en primer lugar se ha realizado, en el Capítulo 2, una introducción a la temática del modelado y la simulación, definiendo algunos conceptos fundamentales como los de sistema, experimento o modelo. Se ha hecho también una revisión de dos de los entornos de desarrollo más populares y potentes actualmente en el mercado: OpenModelica y Dymola.

Tras sentar las bases de este ámbito, se ha descrito durante el Capítulo 3 la fase de análisis y planificación llevada a cabo. Durante esta fase, se han recogido todos los requisitos, funcionales y no funcionales, con los que el sistema final debe contar, y se han detallado con un nivel de precisión suficiente para evitar posibles ambigüedades. Además, se ha descrito la metodología seguida para el desarrollo: una metodología ágil, basada en distintas iteraciones, al final de cada una de las cuales se cuenta con una versión incompleta pero utilizable del producto.

Tras describir la fase de análisis, en el Capítulo 4 se ha detallado la arquitectura seguida para desarrollar la aplicación. Esta arquitectura se basa en ciertos patrones

de diseño y buenas prácticas de desarrollo que han sido enumeradas en el capítulo correspondiente, como son el patrón MVC, el patrón singleton o el patrón estado, entre otros. Cada patrón de diseño ha sido descrito, mostrando su funcionamiento, para después enfatizar qué sección de la aplicación hace uso de este patrón; todo esto no sin antes realizar una breve descripción del concepto de patrones de diseño y clasificación de los mismos. Al detallar la arquitectura de la aplicación, se han mostrado diagramas de clase de todas las clases presentes en el sistema, así como se ha detallado su funcionalidad, haciendo especial énfasis en aquellas con mayor importancia o que pudieran resultar confusas.

Este análisis se ha realizado de forma teórica, es decir, se ha analizado la arquitectura de la aplicación a partir de los diagramas. En el Capítulo 5, se ha realizado un análisis similar, atendiendo esta vez a los detalles de la implementación y a cómo se han materializado los distintos diagramas examinados en el capítulo anterior. Una vez más, aquí se han enfatizado especialmente aquellas clases de mayor importancia, o que pudieran resultar más complejas de interpretar basándose únicamente en el código. Si bien no se ha descrito la totalidad del código fuente, este queda disponible para su consulta en el Anexo B.

Por último, tras describir todo el proceso de desarrollo, el Capítulo 6 ha mostrado parte del software final desarrollado, con el objetivo de contrastar el resultado final con los requisitos obtenidos en el Capítulo 3, enumerando así todos los requisitos cumplidos. Si bien no todos los requisitos han podido demostrarse de manera escrita, como las funcionalidades relativas al Drag&drop, todos pueden ser verificados aludiendo al código fuente o utilizando el propio software. Tras validar los requisitos, se han replicado algunos de los ejemplos presentes en el paquete `Modelica.Electrical.Digital.Examples` y se ha comparado el diagrama y el código generados con los presentes en la librería concluyéndose así que, si bien son diferentes, son análogos y equivalentes.

Con este último capítulo, se concluye el proceso de desarrollo de manera exitosa, habiéndose construido un producto que cumple con todos los requisitos establecidos durante la fase de análisis, y que resulta completamente funcional.

## 7.3. Trabajos futuros

Si bien el desarrollo de este proyecto ha concluido, existen aún muchas mejoras y ampliaciones que puedan llevarse a cabo. En primer lugar, el software incluye muchos, pero no todos, los componentes de la librería estándar de Modelica. Así, una potencial y útil ampliación es la de implementar los componentes faltantes e, incluso, añadir componentes de librerías ajenas a la estándar.

Además, este software está centrado en el ámbito de la electrónica digital, sin embargo, como ya se ha expuesto en el Capítulo 2, Modelica es un lenguaje válido para prácticamente cualquier campo de la ingeniería e, incluso, campos fuera de la misma. Por ello, una posible mejora que puede llevarse a cabo en el futuro, es la extensión de este software a otros dominios, como pueden ser la industria automovilística o aeroespacial. En este sentido, la aplicación permite la implementación de nuevos componentes de modo sencillo, por lo que podrían integrarse modelos de otras librerías sin que esto suponga un esfuerzo inasumible. Sin embargo, la lógica de negocio está diseñada para funcionar con circuitos digitales por lo que, según la librería que desee implementarse, es posible que algunas de las peculiaridades con las que cuente sean incompatibles con el funcionamiento intrínseco de la aplicación.

Si bien estas son solo algunas de las posibles ampliaciones o líneas de trabajo futuro basadas en este proyecto, la aplicación se distribuye como software libre en la plataforma GitHub[1], de modo que está disponible para cualquier usuario o desarrollador que desee modificarla y ampliarla para sus propias necesidades o las de otros, abriendo así miles de posibilidades, limitadas tan solo por la creatividad de la comunidad.

[1]Disponible en (Caponera De Cobellis, 2023a).

# Bibliografía


Open Source Modelica Consortium. (2023). *OpenModelica* (Ver. 1.21.0). https://openmodelica.org/

Caponera De Cobellis, R. R. (2023a). LizardClips. https://github.com/elKuston/LizardClips

Caponera De Cobellis, R. R. (2023b). LizardClips Release. https://github.com/elKuston/LizardClips/releases/tag/Release

Clay Mathematics Institute. (2000). The Millenium Prize Problems. https://www.claymath.org/millennium-problems/

Dassault Systémes. (2023). Multi-Engineering Modeling and Simulation - Dymola product line. https://www.3ds.com/products-services/catia/products/dymola/model-design-tools/

Elmqvist, H. (1978). *Dymola*. https://www.3ds.com/es/productos-y-servicios/catia/productos/dymola/

Fritzson, P. Modeling, Simulation and Development of Cyber-Physical Systems with OpenModelica and FMI. En: 2018, octubre. https://openmodelica.org/images/M_images/Modelica-OpenModelica-slides.pdf

GeeksForGeeks. (2023, marzo). MVC design pattern. https://www.geeksforgeeks.org/mvc-design-pattern/

IBM & The Eclipse Foundation. (2001). *Eclipse*. https://www.eclipse.org/

JetBrains. (2016). *Kotlin*. https://kotlinlang.org/

JetBrains. (2023a). *IntelliJ IDEA Community Edition 2023* (Ver. 2023.1.3). https://www.jetbrains.com/idea/

JetBrains. (2023b). *IntelliJ IDEA Ultimate 2023* (Ver. 2023.2). https://www.jetbrains.com/idea/

Lucidchart. (2023). *Lucidchart*. https://lucidchart.com

MapleSoft. (2009). Modelica.Electrical.Digital examples. https://www.maplesoft.com/documentation_center/online_manuals/modelica/Modelica_Electrical_Digital_Examples.html

Martin, R. C. (2003). *Agile Software Development: Principles, patterns, and practices*. Prentice Hall.

OnlineGantt. (2023). *OnlineGantt*. https://www.onlinegantt.com/#/gantt

Oracle. (2021). *Java* (Ver. 17). https://www.java.com/es/

Red Hat. (2001). *Hibernate*. https://hibernate.org/

RefactoringGuru. (2023a). Classification of Patterns. https://refactoring.guru/design-patterns/classification

RefactoringGuru. (2023b). Singleton. https://refactoring.guru/es/design-patterns/singleton

RefactoringGuru. (2023c). State. https://refactoring.guru/es/design-patterns/state

RefactoringGuru. (2023d). Strategy. https://refactoring.guru/es/design-patterns/strategy

The Apache Foundation. (2022). *Apache Derby* (Ver. 10.16.1.1). https://db.apache.org/derby/

The Apache Software Foundation. (2002). *Maven*. https://maven.apache.org/

The Apache Software Foundation & Oracle. (2000). *NetBeans*. https://netbeans.apache.org/

The Eclipse Foundation. (2022). *EclipseLink* (Ver. 4.0.0). https://eclipse.dev/eclipselink/

The Modelica Association. (1997). *Modelica*. https://modelica.org/

The Project Lombok Authors. (2022, abril). *Project Lombok* (Ver. 1.18.24). https://projectlombok.org/

Urquía Moraleda, A., & Martín Villalba, C. (2017). *Métodos de simulación y modelado*. Universidad Nacional de Educación a Distancia.

# Glosario

**A**

**API** Application Programming Interface. 18, 19, 22, 105

**AWT** Abstract Window Toolkit. 20

**D**

**Drag&drop** "Arrastrar y soltar" en español. Es una técnica de interacción con aplicaciones software que permite arrastrar los elementos con el puntero y, al soltar el puntero, "soltar" los elementos en la posición en la que se encuentran. 2, 12, 27, 29, 86, 91, 98, 100, 108

**G**

**Getter** En programación, los getters son aquellos métodos que se utilizan para leer una determinada propiedad de un objeto. Generalmente, y por convención, siguen la nomenclatura *get_()*, de modo que si se quiere obtener una propiedad llamada `propiedad`, se utiliza un método `getPropiedad()`. 17, 18, 45, 60, 69

**GUI** Graphical User Interface. 1

**H**

**HILS** Hardware-in-the-Loop Simluations. 14

**I**

**IDE** Integrated Development Envoironment. 20–22

**J**

**JFC** Java Foundation Classes. 20

**JPA** Jakarta Persistence API. 18, 19, 21, 22, 61–63, 68, 71

**JVM** Java Virtual Machine. 15, 16

**M**

**MVC** Modelo-Vista-Controlador. 45, 53, 100

**O**

**OMC** OpenModelica Compiler. 11, 12

**ORM** Object-Relational Mapping. 19

**OSMC** Open Source Modelica Consortium. 11, 12, 102

**P**

**POM** Project Object Model. 16

**R**

**RF** Requisito Funcional. 24

**RNF** Requisito No Funcional. 27

**S**

**Setter** En programación, los setters son aquellos métodos que se utilizan para escribir el valor de una determinada propiedad de un objeto. Generalmente, y por convención, siguen la nomenclatura *set_()*, de modo que si se quiere modificar una propiedad llamada `propiedad`, se utiliza un método `setPropiedad()`. 17, 45, 60

**SGDB** Sistema Gestor de Bases de Datos. 19, 20

**SQL** Structured Query Language. 18, 19

**SRP** Single Responsibility Principle. 53

**T**

**Thumbnail** Imagen a tamaño reducido que se utiliza cuando la imagen real es demasiado grande para mostrarse como vista previa. 44

**TLS** Transport Layer Security. 27

**Tooltip** Herramienta de ayuda visual que permite al usuario conocer más sobre la funcionalidad de un determinado botón, significado de campo de texto, etc. Generalmente se muestra como un pequeño globo flotante al mantener el ratón sobre el elemento en cuestión. 28, 31, 39, 65

**U**

**UML** Unified Modeling Language. 21, 36, 38, 58, 60

**W**

**WORA** Write Once, Run Anywhere. 15

# A. Manual de usuario

## A.1. Manual de instalación

Puesto que dentro de los objetivos de este software se encuentran la facilidad de uso y la portabilidad, la instalación de la aplicación es simple. Basta con descargar el fichero `.jar`, disponible en (Caponera De Cobellis, 2023b), y ejecutarlo haciendo doble click sobre el mismo. Esto generará, en la carpeta en la que se encuentre el fichero, otro directorio que contendrá la base de datos, tal y como se muestra en la Figura A.1. Por lo tanto, resulta conveniente que, antes de ejecutar por primera vez la aplicación, se coloque el fichero en la ubicación más conveniente. Este proceso se realiza únicamente cuando el programa se inicia por primera vez, y podría tomar unos segundos.

A partir de este punto, es posible utilizar el software con normalidad.

## A.2. Composición de circuitos

Una vez abierta la aplicación, se visualizará la ventana principal mostrada en la Figura A.2. Esta ventana está dividida en dos secciones bien diferenciadas: a la izquierda, la paleta de componentes. A la derecha, el área de trabajo, donde se construirá el circuito.

Para componer el circuito, basta con pulsar en alguno de los componentes de la paleta, y arrastrar el componente a la ubicación deseada, haciendo click para colocarlo. A partir de este punto, los componentes pueden moverse utilizando una dinámica Drag&drop. En la mitad izquierda de la paleta, puede seleccionarse cuál

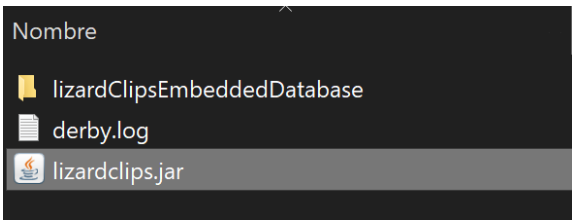



Figura A.1: Ficheros generados por el software al iniciarse por primera vez.

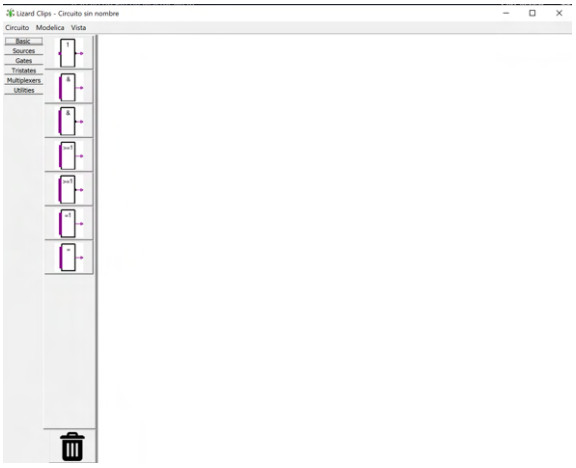



Figura A.2: Ventana principal de la aplicación.

de los paquetes se desea visualizar, de modo que al pulsar en uno de ellos cambiarán los elementos disponibles para colocar en el circuito.

Además, en la parte inferior de la paleta se encuentra el icono de borrado, que permite iniciar o finalizar el modo borrado. Cuando se inicia el modo borrado, hacer click sobre una pieza o conexión la eliminará del área de trabajo.

Una vez colocados los componentes, es posible realizar conexiones entre ellos mediante los distintos conectores. En la Figura A.3, se puede observar como cada componente cuenta con una serie de conectores (puntos negros), que pueden ser utilizados para crear una conexión. Para ello, basta con pulsar en un conector de un componente y, posteriormente, en otro conector de otro componente. Si la conexión es válida, se creará; si no, se mostrará un mensaje de error indicando por qué no puede realizarse dicha conexión. Además, si, mientras se está realizando la conexión, se pulsa sobre una zona vacía del área de trabajo, se creará un punto intermedio por el que el "cable" de la conexión pasará. De este modo, es posible crear un circuito tan complejo como se quiera.

Para desplazarse por el área de trabajo, basta con pulsar la rueda del ratón y mover el puntero, moviendo así el área de trabajo. Para acercar o alejar el área de trabajo, y tener así una visión más completa del circuito o una visión centrada en una zona específica, basta con girar la rueda del ratón hacia arriba y hacia abajo, respectivamente.

Además, si se pulsa con el botón derecho del ratón sobre cualquiera de los com-

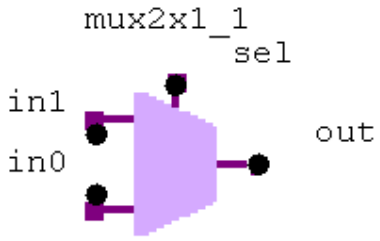



Figura A.3: Detalle de una pieza en el área de trabajo.

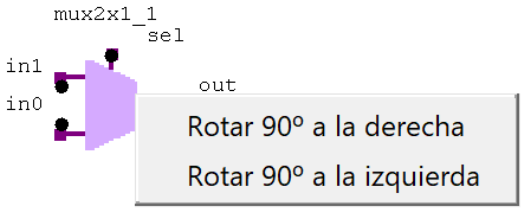



Figura A.4: Opciones de rotación de un componente.

ponentes, se mostrarán las opciones visibles en la Figura A.4, que permiten rotar el componente 90° a la derecha o a la izquierda, respectivamente.

Por último, hacer doble click sobre cualquiera de los componentes mostrará una ventana como la de la Figura A.5, en la que será posible modificar las propiedades del componente, incluyendo el nombre del mismo. En esta ventana, es posible mantener el puntero sobre cualquiera de las propiedades para obtener más información sobre la misma.

## A.3. Generación de código

Una vez ensamblado un circuito, es posible generar el código Modelica correspondiente a través del menú Modelica que se encuentra en la parte superior de la ventana, y cuyas opciones se muestran en la Figura A.6.

Si se pulsa en la opción de exportar código Modelica, la aplicación permitirá

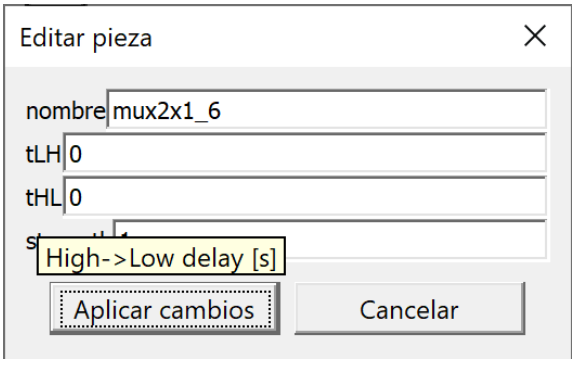



Figura A.5: Ventana de edición de propiedades de un componente.

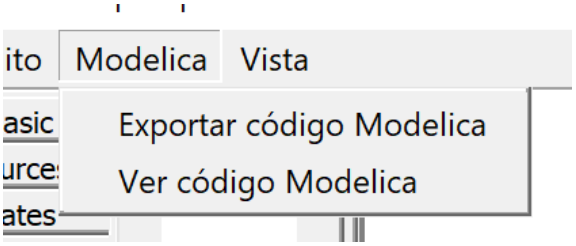



Figura A.6: Menú Modelica.

generar un fichero .mo con el código generado y, a continuación, preguntará al usuario si desea abrirlo en el momento. Si es así, abrirá el fichero en el editor Modelica instalado en el dispositivo o, en su defecto, preguntará al usuario con qué aplicación desea abrirlo.

La opción *Ver código Modelica*, por otro lado, abre una ventana emergente que contiene todo el código generado, como la que se muestra en la Figura A.7

En esta ventana es posible visualizar y copiar el código Modelica, pero no editarlo. Además, las palabras reservadas del lenguaje se muestran coloreadas en azul.

## A.4. Gestión de Circuitos

Mediante el menú *Circuito*, cuyas opciones se muestran en la Figura A.8, es posible gestionar los distintos circuitos presentes en la aplicación.

En primer lugar, la opción *Nuevo circuito* crea un nuevo circuito, limpiando completamente el área de trabajo. Si hay cambios no guardados en el circuito actual, estos se perderán.

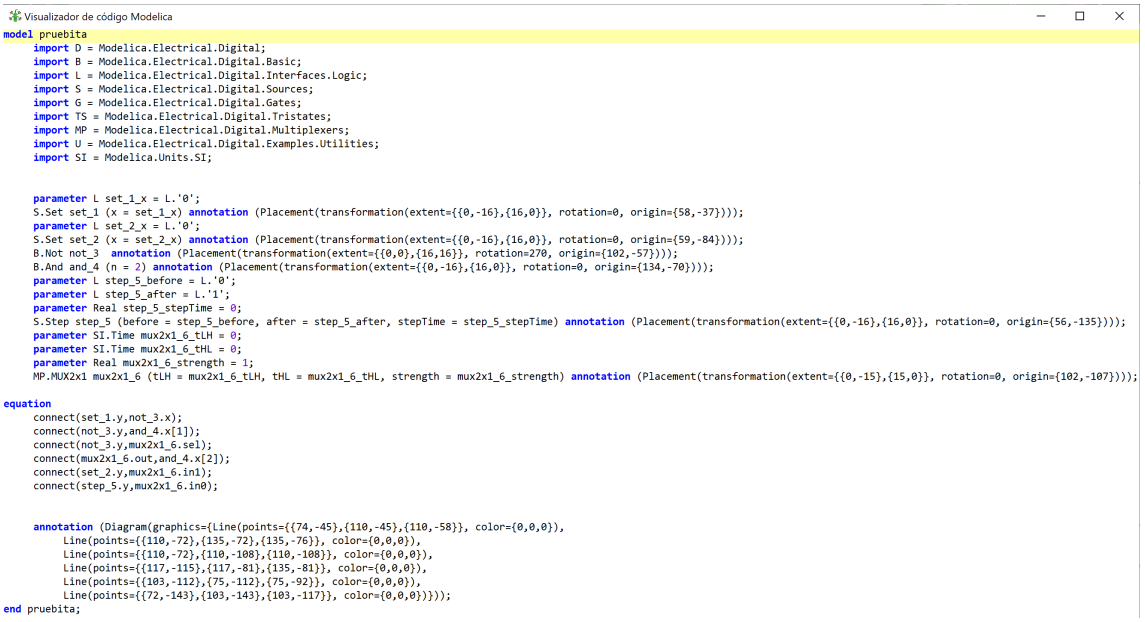



Figura A.7: Ventana de visualización de código Modelica.

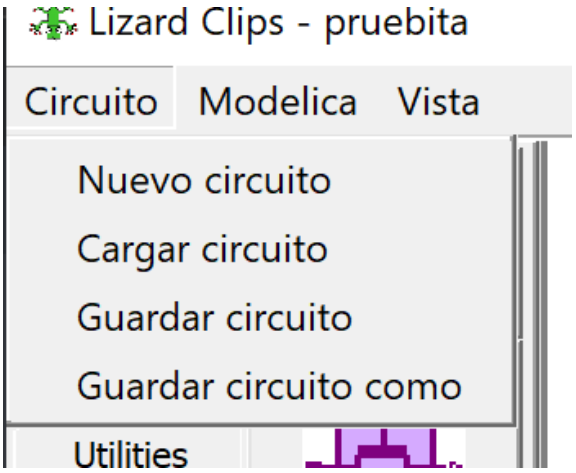



Figura A.8: Menú Circuito.

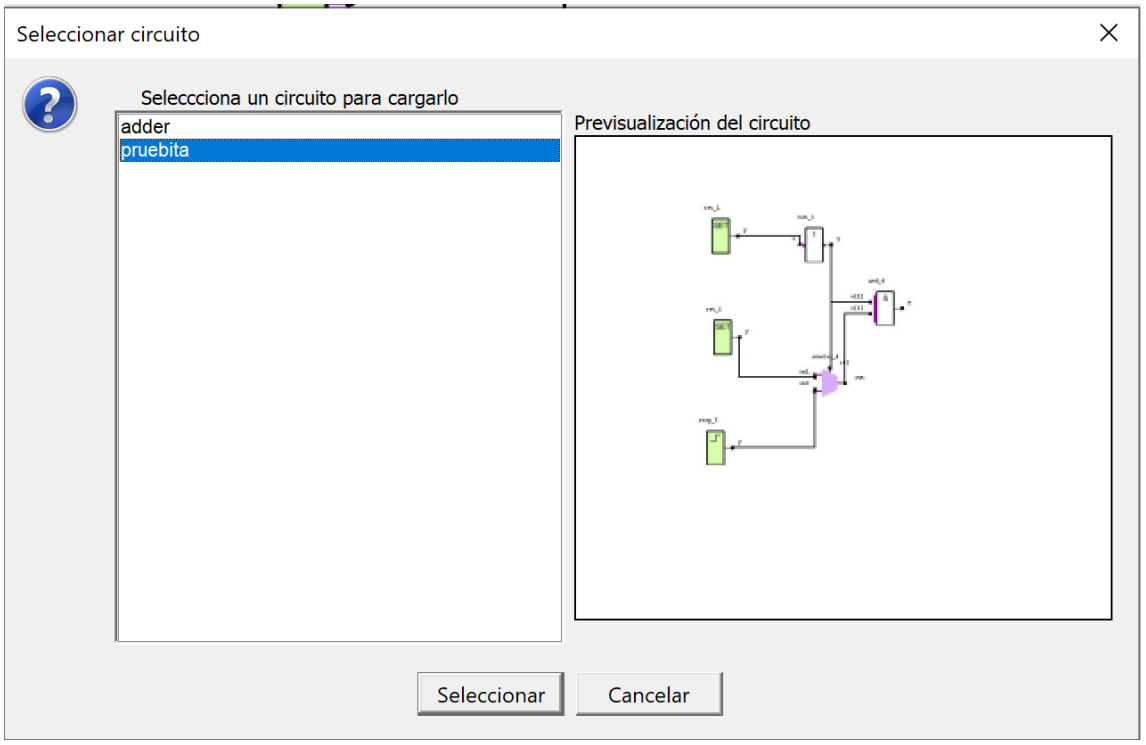



Figura A.9: Ventana de selección de circuito.

Una vez que se ha creado un circuito con el que se está satisfecho, y se desea guardar en la base de datos, es posible hacerlo utilizando la opción *Guardar circuito*. Al pulsarla, el sistema preguntará al usuario con qué nombre desea guardar el circuito y, si el nombre es válido, lo guardará; en caso contrario, mostrará un mensaje de error indicando por qué el nombre no es válido.

Una vez que se tiene un circuito guardado, es posible volver a cargarlo en la aplicación mediante la opción *Cargar circuito*. Esta opción presentará al usuario una ventana como la que se muestra en la Figura A.9, en la que, a la izquierda, se encuentra una lista de los distintos circuitos presentes en la base de datos, mientas que a la derecha se muestra una previsualización del circuito. Al seleccionar cualquiera de los circuitos, pasará al área de trabajo, y será posible editarlo o exportar el código.

Tras realizar las modificaciones deseadas, es posible volver a guardar el circuito, sobrescribiendo la versión anterior, mediante la opción *Guardar circuito*. Si, por el contrario, se desea guardar el circuito sin sobrescribir la versión previa, puede guardarse el circuito con otro nombre pulsando en la opción *Guardar circuito como*,

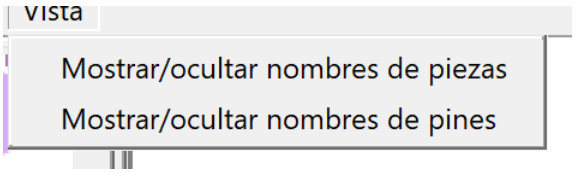



Figura A.10: Menú Vista.

en cuyo caso la aplicación volverá a preguntar cuál es el nuevo nombre con el que se quiere guardar el circuito y, si es válido, lo guardará.

## A.5. Menú Vista

Finalmente, el tercer menú de la aplicación es el menú *vista*, cuyas opciones pueden verse en la Figura A.10.

Este menú ofrece opciones para personalizar la visualización del circuito. Como se ha visto en la Figura A.3, junto a cada componente se muestra, por defecto, su nombre, en la parte superior, y el nombre de cada uno de los conectores[1], al lado de los mismos. Las opciones del menú vista *Mostrar/ocultar nombres de piezas* y *Mostrar/ocultar nombres de pines* permiten, respectivamente, ocultar estos textos, para evitar el ruido visual que puede producir cuando se muestran muchos componentes simultáneamente; o, por el contrario, mostrar los textos cuando se debe realizar una conexión y no se conoce exactamente el significado de cada uno de los conectores, por ejemplo.

[1]En la imagen se hace referencia a pines, y no a conectores, porque existen conectores que pueden tener varios pines (es decir, conectores cuya entrada es un array de valores). Los pines de estos conectores serán de la forma `x[0]`, `x[1]`,...

# B. Código fuente

En este anexo se muestra el código fuente de todos los ficheros del proyecto. Puesto que algunas sentencias pueden ser especialmente largas, se han indicado con una flecha roja aquellos saltos de línea que no están en el fichero original, sino que se han añadido para que el texto no sobresalga de la página del documento.

## B.1. Fichero `Main.java`

```
package caponera.uned.tfm.lizardclips;

import caponera.uned.tfm.lizardclips.controlador.
    ↪ ControladorCircuito;
import caponera.uned.tfm.lizardclips.gui.PanelCircuito;
import caponera.uned.tfm.lizardclips.gui.VentanaPrincipal;
import caponera.uned.tfm.lizardclips.modelo.Circuito;

import javax.swing.UIManager;
import javax.swing.UnsupportedLookAndFeelException;
import java.util.Arrays;


public class Main {
    private static final String LOOK_AND_FEEL_PREFERIDO =
            "com.sun.java.swing.plaf.windows.
                ↪ WindowsClassicLookAndFeel";

    public static void main(String[] args) {
```

```
        setLAF();

        Circuito circuito = new Circuito();
        PanelCircuito panelCircuito = new PanelCircuito();
        ControladorCircuito controlador = new
            ↪ ControladorCircuito(circuito, panelCircuito);

        VentanaPrincipal ventanaPrincipal =
                new VentanaPrincipal(1000, 800, controlador,
                    ↪  panelCircuito);
        ventanaPrincipal.mostrar();

    }

    private static void setLAF() {
        try {
            if (Arrays.stream(UIManager.
                ↪ getInstalledLookAndFeels())
                        .map(UIManager.LookAndFeelInfo::
                            ↪ getClassName)
                        .anyMatch(s -> s.equals(
                            ↪ LOOK_AND_FEEL_PREFERIDO))) {
                UIManager.setLookAndFeel(
                    ↪ LOOK_AND_FEEL_PREFERIDO);
            } else {
                UIManager.setLookAndFeel(UIManager.
                    ↪ getSystemLookAndFeelClassName());
            }
```

```
        } catch (ClassNotFoundException |
            ↪ InstantiationException |
            ↪ IllegalAccessException |
            ↪ UnsupportedLookAndFeelException ex) {
             ex.printStackTrace();
        }
    }
}
```

## B.2. Fichero constant/ConectorTemplate.java

```
package caponera.uned.tfm.lizardclips.constant;

import caponera.uned.tfm.lizardclips.modelo.Conector;
import lombok.AllArgsConstructor;
import lombok.Data;

@Data
@AllArgsConstructor
public class ConectorTemplate {
    private TipoConector tipo;
    private String nombre;
    private boolean multiple;
    private int minConectores, maxConectores;
    private String nombreNumPines;
    private double relativeX, relativeY;
    private boolean reposicionar = true;

    public ConectorTemplate(TipoConector tipo, String nombre
        ↪ , boolean multiple) {
        this(tipo, nombre, multiple, 1, Integer.MAX_VALUE, "
```

```
            ↪ n", tipo.equals(TipoConector.ENTRADA) ? 0 : 1,
            ↪  0, true);
    }

    public ConectorTemplate(TipoConector tipo, String nombre
       ↪ ) {
        this(tipo, nombre, false);
    }

    public ConectorTemplate(TipoConector tipo, String nombre
       ↪ , int minConectores) {
        this(tipo, nombre, true, minConectores, Integer.
           ↪ MAX_VALUE, "n", tipo.equals(TipoConector.
           ↪ ENTRADA) ? 0 : 1, 0,
                true);
    }

    public ConectorTemplate(TipoConector tipo, String nombre
       ↪ , double relativeX, double relativeY) {
        this(tipo, nombre, false);
        this.relativeX = relativeX;
        this.relativeY = relativeY;
        this.reposicionar = false;
    }
}
```

## B.3. Fichero `constant/ModoPanel.java`

```
package caponera.uned.tfm.lizardclips.constant;

public enum ModoPanel {
```

```
    MODO_NORMAL, MODO_ARRASTRANDO, MODO_BORRADO,
        ↪ MODO_CONEXION, MODO_DESPLAZANDO;
}
```

## B.4. Fichero constant/TabPaleta.java

```
package caponera.uned.tfm.lizardclips.constant;

import lombok.Getter;

public enum TabPaleta {
    BASIC("Basic"), SOURCES("Sources"), GATES("Gates"),
        ↪ TRISTATES("Tristates"),
    MULTIPLEXERS("Multiplexers"), UTILITIES("Utilities");
    @Getter
    private final String nombre;

    TabPaleta(String nombre) {
        this.nombre = nombre;
    }
}
```

## B.5. Fichero constant/TipoConector.java

```
package caponera.uned.tfm.lizardclips.constant;

public enum TipoConector {
    ENTRADA, SALIDA;
}
```

## B.6. Fichero constant/TipoPieza.java

```
package caponera.uned.tfm.lizardclips.constant;

import caponera.uned.tfm.lizardclips.modelica.
    ↪ ModelicaGenerator;
import caponera.uned.tfm.lizardclips.modelo.Propiedad;
import caponera.uned.tfm.lizardclips.modelo.PropiedadLogic;
import caponera.uned.tfm.lizardclips.modelo.PropiedadSimple;
import caponera.uned.tfm.lizardclips.utils.ImageUtils;
import lombok.Getter;

import java.util.List;

@Getter
public enum TipoPieza {
    //region Basic
    NOT("NOT", ImageUtils.pathImagenMedia("not.png"),
        ↪ ModelicaGenerator.BASIC + ".Not",
            TabPaleta.BASIC, List.of(), List.of(new
                ↪ ConectorTemplate(TipoConector.ENTRADA, "x"
                ↪ ),
            new ConectorTemplate(TipoConector.SALIDA, "y")))
                ↪ ,
    AND("AND", ImageUtils.pathImagenMedia("and.png"),
        ↪ ModelicaGenerator.BASIC + ".And",
            TabPaleta.BASIC, List.of(), List.of(new
                ↪ ConectorTemplate(TipoConector.ENTRADA, "x"
                ↪ , 2),
            new ConectorTemplate(TipoConector.SALIDA, "y")))
                ↪ ,
    NAND("NAND", ImageUtils.pathImagenMedia("nand.png"),
        ↪ ModelicaGenerator.BASIC + ".Nand",
```

```
            TabPaleta.BASIC, List.of(), List.of(new
                ↪ ConectorTemplate(TipoConector.ENTRADA, "x"
                ↪ , 2),
            new ConectorTemplate(TipoConector.SALIDA, "y")))
                ↪ ,
    OR("OR", ImageUtils.pathImagenMedia("or.png"),
        ↪ ModelicaGenerator.BASIC + ".Or", TabPaleta.BASIC,
            List.of(), List.of(new ConectorTemplate(
                ↪ TipoConector.ENTRADA, "x", 2),
            new ConectorTemplate(TipoConector.SALIDA, "y")))
                ↪ ,
    NOR("NOR", ImageUtils.pathImagenMedia("nor.png"),
        ↪ ModelicaGenerator.BASIC + ".Nor",
            TabPaleta.BASIC, List.of(), List.of(new
                ↪ ConectorTemplate(TipoConector.ENTRADA, "x"
                ↪ , 2),
            new ConectorTemplate(TipoConector.SALIDA, "y")))
                ↪ ,
    XOR("XOR", ImageUtils.pathImagenMedia("xor.png"),
        ↪ ModelicaGenerator.BASIC + ".Xor",
            TabPaleta.BASIC, List.of(), List.of(new
                ↪ ConectorTemplate(TipoConector.ENTRADA, "x"
                ↪ , 2),
            new ConectorTemplate(TipoConector.SALIDA, "y")))
                ↪ ,
    XNOR("XNOR", ImageUtils.pathImagenMedia("xnor.png"),
        ↪ ModelicaGenerator.BASIC + ".Xnor",
            TabPaleta.BASIC, List.of(), List.of(new
                ↪ ConectorTemplate(TipoConector.ENTRADA, "x"
                ↪ , 2),
            new ConectorTemplate(TipoConector.SALIDA, "y")))
```

```
            ↪ , //endregion
//region Gates
INVGATE("INVGATE", ImageUtils.pathImagenMedia("invgate.
    ↪ png"),
        ModelicaGenerator.GATES + ".InvGate", TabPaleta.
            ↪ GATES,
        List.of(new PropiedadSimple("0", "tLH",
            ↪ Propiedad.UNIDAD_TIME,
                    "rise inertial delay [s]"),
                new PropiedadSimple("0", "tHL",
                    ↪ Propiedad.UNIDAD_TIME,
                        "fall inertial delay [s]"),
                new PropiedadLogic("y0", "'U'", "initial
                    ↪ value of output")),
        List.of(new ConectorTemplate(TipoConector.
            ↪ ENTRADA, "x"),
                new ConectorTemplate(TipoConector.SALIDA
                    ↪ , "y"))),
ANDGATE("ANDGATE", ImageUtils.pathImagenMedia("andgate.
    ↪ png"),
        ModelicaGenerator.GATES + ".AndGate", TabPaleta.
            ↪ GATES,
        List.of(new PropiedadSimple("0", "tLH",
            ↪ Propiedad.UNIDAD_TIME,
                    "rise inertial delay [s]"),
                new PropiedadSimple("0", "tHL",
                    ↪ Propiedad.UNIDAD_TIME,
                        "fall inertial delay [s]"),
                new PropiedadLogic("y0", "'U'", "initial
                    ↪ value of output")),
        List.of(new ConectorTemplate(TipoConector.
```

```
        ↪ ENTRADA, "x", 2),
                new ConectorTemplate(TipoConector.SALIDA
                    ↪ , "y"))),
NANDGATE("ANDGATE", ImageUtils.pathImagenMedia("nandgate
    ↪ .png"),
        ModelicaGenerator.GATES + ".NandGate", TabPaleta
            ↪ .GATES,
        List.of(new PropiedadSimple("0", "tLH",
            ↪ Propiedad.UNIDAD_TIME,
                    "rise-inertial-delay-[s]"),
                new PropiedadSimple("0", "tHL",
                    ↪ Propiedad.UNIDAD_TIME,
                        "fall-inertial-delay-[s]"),
                new PropiedadLogic("y0", "'U'", "initial
                    ↪ -value-of-output")),
        List.of(new ConectorTemplate(TipoConector.
            ↪ ENTRADA, "x", 2),
                new ConectorTemplate(TipoConector.SALIDA
                    ↪ , "y"))),
ORGATE("ORGATE", ImageUtils.pathImagenMedia("orgate.png"
    ↪ ), ModelicaGenerator.GATES + ".OrGate",
        TabPaleta.GATES, List.of(new PropiedadSimple("0"
            ↪ , "tLH", Propiedad.UNIDAD_TIME,
                "rise-inertial-delay-[s]"),
        new PropiedadSimple("0", "tHL", Propiedad.
            ↪ UNIDAD_TIME, "fall-inertial-delay-[s]"),
        new PropiedadLogic("y0", "'U'", "initial-value-
            ↪ of-output")),
        List.of(new ConectorTemplate(TipoConector.
            ↪ ENTRADA, "x", 2),
                new ConectorTemplate(TipoConector.SALIDA
```

```
                    ↪ , "y"))),
NORGATE("NORGATE", ImageUtils.pathImagenMedia("norgate.
    ↪ png"),
        ModelicaGenerator.GATES + ".NorGate", TabPaleta.
            ↪ GATES,
        List.of(new PropiedadSimple("0", "tLH",
            ↪ Propiedad.UNIDAD_TIME,
                    "rise-inertial-delay-[s]"),
                new PropiedadSimple("0", "tHL",
                    ↪ Propiedad.UNIDAD_TIME,
                        "fall-inertial-delay-[s]"),
                new PropiedadLogic("y0", "'U'", "initial
                    ↪ -value-of-output")),
        List.of(new ConectorTemplate(TipoConector.
            ↪ ENTRADA, "x", 2),
                new ConectorTemplate(TipoConector.SALIDA
                    ↪ , "y"))),
XORGATE("XORGATE", ImageUtils.pathImagenMedia("xorgate.
    ↪ png"),
        ModelicaGenerator.GATES + ".XorGate", TabPaleta.
            ↪ GATES,
        List.of(new PropiedadSimple("0", "tLH",
            ↪ Propiedad.UNIDAD_TIME,
                    "rise-inertial-delay-[s]"),
                new PropiedadSimple("0", "tHL",
                    ↪ Propiedad.UNIDAD_TIME,
                        "fall-inertial-delay-[s]"),
                new PropiedadLogic("y0", "'U'", "initial
                    ↪ -value-of-output")),
        List.of(new ConectorTemplate(TipoConector.
            ↪ ENTRADA, "x", 2),
```

```
                new ConectorTemplate(TipoConector.SALIDA
                    ↪ , "y"))),
XNORGATE("XNORGATE", ImageUtils.pathImagenMedia("
    ↪ xnorgate.png"),
        ModelicaGenerator.GATES + ".XnorGate", TabPaleta
            ↪ .GATES,
        List.of(new PropiedadSimple("0", "tLH",
            ↪ Propiedad.UNIDAD_TIME,
                    "rise-inertial-delay-[s]"),
                new PropiedadSimple("0", "tHL",
                    ↪ Propiedad.UNIDAD_TIME,
                        "fall-inertial-delay-[s]"),
                new PropiedadLogic("y0", "'U'", "initial
                    ↪ -value-of-output")),
        List.of(new ConectorTemplate(TipoConector.
            ↪ ENTRADA, "x", 2),
                new ConectorTemplate(TipoConector.SALIDA
                    ↪ , "y"))),
BUFGATE("BUFGATE", ImageUtils.pathImagenMedia("bufgate.
    ↪ png"),
        ModelicaGenerator.GATES + ".BufGate", TabPaleta.
            ↪ GATES,
        List.of(new PropiedadSimple("0", "tLH",
            ↪ Propiedad.UNIDAD_TIME,
                    "rise-inertial-delay-[s]"),
                new PropiedadSimple("0", "tHL",
                    ↪ Propiedad.UNIDAD_TIME,
                        "fall-inertial-delay-[s]"),
                new PropiedadLogic("y0", "'U'", "initial
                    ↪ -value-of-output")),
        List.of(new ConectorTemplate(TipoConector.
```

```
            ↪ ENTRADA, "x"),
                new ConectorTemplate(TipoConector.SALIDA
                    ↪ , "y"))), //endregion
//region Sources
SET("SET", ImageUtils.pathImagenMedia("set.png"),
    ↪ ModelicaGenerator.SOURCES + ".Set",
        TabPaleta.SOURCES, List.of(new PropiedadLogic("x
            ↪ ", "")),
        List.of(new ConectorTemplate(TipoConector.SALIDA
            ↪ , "y"))),
STEP("STEP", ImageUtils.pathImagenMedia("step.png"),
    ↪ ModelicaGenerator.SOURCES + ".Step",
        TabPaleta.SOURCES, List.of(new PropiedadLogic("
            ↪ before", "Logic value before step"),
        new PropiedadLogic("after", "'1'", "Logic value
            ↪ after step"),
        new PropiedadSimple("0", "stepTime", Propiedad.
            ↪ UNIDAD_REAL, "step time")),
        List.of(new ConectorTemplate(TipoConector.SALIDA
            ↪ , "y"))),
DIGITAL_CLOCK("DIGITAL_CLOCK", ImageUtils.
    ↪ pathImagenMedia("clock.png"),
        ModelicaGenerator.SOURCES + ".DigitalClock",
            ↪ TabPaleta.SOURCES,
        List.of(new PropiedadSimple("0", "startTime",
            ↪ Propiedad.UNIDAD_TIME,
                    "Output = offset for time <
                        ↪ startTime [s]"),
                new PropiedadSimple("1", "period",
                    ↪ Propiedad.UNIDAD_TIME,
                        "Time for one period [s]"),
```

```
                    new PropiedadSimple("50", "width",
                        ↪ Propiedad.UNIDAD_REAL,
                            "Width of pulses in % of period"
                                ↪ )),
            List.of(new ConectorTemplate(TipoConector.SALIDA
                ↪ , "y"))),
    TABLE("TABLE", ImageUtils.pathImagenMedia("table.png"),
        ↪ ModelicaGenerator.SOURCES + ".Table",
            TabPaleta.SOURCES,
            List.of(new PropiedadSimple("{L.'U'}", "x[:]",
                ↪ ModelicaGenerator.LOGIC,
                            "vector of values"), new
                                ↪ PropiedadSimple("{1}", "t
                                ↪ [:]", Propiedad.
                                ↪ UNIDAD_REAL,
                            "vector of corresponding time
                                ↪ points"),
                    new PropiedadLogic("y0", "'U'", "initial
                        ↪ output value")),
            List.of(new ConectorTemplate(TipoConector.SALIDA
                ↪ , "y"))), //endregion
    //region Tristates
    NXFERGATE("NXFERGATE", ImageUtils.pathImagenMedia("
        ↪ NXFERGATE.png"),
            ModelicaGenerator.TRISTATES + ".NXFERGATE",
                ↪ TabPaleta.TRISTATES,
            List.of(new PropiedadSimple("0", "tLH",
                ↪ Propiedad.UNIDAD_TIME, "Low->High delay [s
                ↪ ]"),
                    new PropiedadSimple("0", "tHL",
                        ↪ Propiedad.UNIDAD_TIME, "High->Low
```

```
                    ↪ delay [s]")),
            List.of(new ConectorTemplate(TipoConector.
                ↪ ENTRADA, "enable"),
                    new ConectorTemplate(TipoConector.
                        ↪ ENTRADA, "x"),
                    new ConectorTemplate(TipoConector.SALIDA
                        ↪ , "y", 1, .7))),
NRXFERGATE("NRXFERGATE", ImageUtils.pathImagenMedia("
    ↪ NRXFERGATE.png"),
            ModelicaGenerator.TRISTATES + ".NRXFERGATE",
                ↪ TabPaleta.TRISTATES,
            List.of(new PropiedadSimple("0", "tLH",
                ↪ Propiedad.UNIDAD_TIME, "Low->High delay [s
                ↪ ]"),
                    new PropiedadSimple("0", "tHL",
                        ↪ Propiedad.UNIDAD_TIME, "High->Low
                        ↪ delay [s]")),
            List.of(new ConectorTemplate(TipoConector.
                ↪ ENTRADA, "enable"),
                    new ConectorTemplate(TipoConector.
                        ↪ ENTRADA, "x"),
                    new ConectorTemplate(TipoConector.SALIDA
                        ↪ , "y", 1, .7))),
PXFERGATE("PXFERGATE", ImageUtils.pathImagenMedia("
    ↪ PXFERGATE.png"),
            ModelicaGenerator.TRISTATES + ".PXFERGATE",
                ↪ TabPaleta.TRISTATES,
            List.of(new PropiedadSimple("0", "tLH",
                ↪ Propiedad.UNIDAD_TIME, "Low->High delay [s
                ↪ ]"),
                    new PropiedadSimple("0", "tHL",
```

```
                    ↪ Propiedad.UNIDAD_TIME, "High->Low
                    ↪ delay [s]")),
            List.of(new ConectorTemplate(TipoConector.
                ↪ ENTRADA, "enable"),
                    new ConectorTemplate(TipoConector.
                        ↪ ENTRADA, "x"),
                    new ConectorTemplate(TipoConector.SALIDA
                        ↪ , "y", 1, .7))),
PRXFERGATE("PRXFERGATE", ImageUtils.pathImagenMedia("
    ↪ PRXFERGATE.png"),
            ModelicaGenerator.TRISTATES + ".PRXFERGATE",
                ↪ TabPaleta.TRISTATES,
            List.of(new PropiedadSimple("0", "tLH",
                ↪ Propiedad.UNIDAD_TIME, "Low->High delay [s
                ↪ ]"),
                    new PropiedadSimple("0", "tHL",
                        ↪ Propiedad.UNIDAD_TIME, "High->Low
                        ↪ delay [s]")),
            List.of(new ConectorTemplate(TipoConector.
                ↪ ENTRADA, "enable"),
                    new ConectorTemplate(TipoConector.
                        ↪ ENTRADA, "x"),
                    new ConectorTemplate(TipoConector.SALIDA
                        ↪ , "y", 1, .7))),
BUF3S("BUF3S", ImageUtils.pathImagenMedia("BUF3S.png"),
    ↪ ModelicaGenerator.TRISTATES + ".BUF3S",
            TabPaleta.TRISTATES,
            List.of(new PropiedadSimple("0", "tLH",
                ↪ Propiedad.UNIDAD_TIME, "Low->High delay [s
                ↪ ]"),
                    new PropiedadSimple("0", "tHL",
```

```
                    ↪ Propiedad.UNIDAD_TIME, "High->Low
                    ↪ delay [s]"),
                new PropiedadSimple("1", "strength",
                    ↪ Propiedad.UNIDAD_REAL, "Output
                    ↪ strength")),
            List.of(new ConectorTemplate(TipoConector.
                ↪ ENTRADA, "enable", 0, 0),
                    new ConectorTemplate(TipoConector.
                        ↪ ENTRADA, "x", 0, 0.5),
                    new ConectorTemplate(TipoConector.SALIDA
                        ↪ , "y"))),
    BUF3SL("BUF3SL", ImageUtils.pathImagenMedia("BUF3S.png")
        ↪ ,//las imagenes son iguales
            ModelicaGenerator.TRISTATES + ".BUF3SL",
                ↪ TabPaleta.TRISTATES,
            List.of(new PropiedadSimple("0", "tLH",
                ↪ Propiedad.UNIDAD_TIME, "Low->High delay [s
                ↪ ]"),
                    new PropiedadSimple("0", "tHL",
                        ↪ Propiedad.UNIDAD_TIME, "High->Low
                        ↪ delay [s]"),
                    new PropiedadSimple("1", "strength",
                        ↪ Propiedad.UNIDAD_REAL, "Output
                        ↪ strength")),
            List.of(new ConectorTemplate(TipoConector.
                ↪ ENTRADA, "enable", 0, 0),
                    new ConectorTemplate(TipoConector.
                        ↪ ENTRADA, "x", 0, .5),
                    new ConectorTemplate(TipoConector.SALIDA
                        ↪ , "y"))),
    INV3S("INV3S", ImageUtils.pathImagenMedia("INV3S.png"),
```

```
↪ ModelicaGenerator.TRISTATES + ".INV3S",
        TabPaleta.TRISTATES,
        List.of(new PropiedadSimple("0", "tLH",
            ↪ Propiedad.UNIDAD_TIME, "Low->High delay [s
            ↪ ]"),
                new PropiedadSimple("0", "tHL",
                    ↪ Propiedad.UNIDAD_TIME, "High->Low
                    ↪ delay [s]"),
                new PropiedadSimple("1", "strength",
                    ↪ Propiedad.UNIDAD_REAL, "Output
                    ↪ strength")),
        List.of(new ConectorTemplate(TipoConector.
            ↪ ENTRADA, "enable", 0, 0),
                new ConectorTemplate(TipoConector.
                    ↪ ENTRADA, "x", 0, .5),
                new ConectorTemplate(TipoConector.SALIDA
                    ↪ , "y"))),
INV3SL("INV3SL", ImageUtils.pathImagenMedia("INV3SL.png"
    ↪ ),
        ModelicaGenerator.TRISTATES + ".INV3SL",
            ↪ TabPaleta.TRISTATES,
        List.of(new PropiedadSimple("0", "tLH",
            ↪ Propiedad.UNIDAD_TIME, "Low->High delay [s
            ↪ ]"),
                new PropiedadSimple("0", "tHL",
                    ↪ Propiedad.UNIDAD_TIME, "High->Low
                    ↪ delay [s]"),
                new PropiedadSimple("1", "strength",
                    ↪ Propiedad.UNIDAD_REAL, "Output
                    ↪ strength")),
        List.of(new ConectorTemplate(TipoConector.
```

```
            ↪ ENTRADA, "enable", 0, 0),
                new ConectorTemplate(TipoConector.
                    ↪ ENTRADA, "x", 0, .5),
                new ConectorTemplate(TipoConector.SALIDA
                    ↪ , "y"))),
WIRED_X("WIRED_X", ImageUtils.pathImagenMedia("WiredX.
    ↪ png"),
        ModelicaGenerator.TRISTATES + ".WiredX",
            ↪ TabPaleta.TRISTATES, List.of(),
        List.of(new ConectorTemplate(TipoConector.
            ↪ ENTRADA, "x", 1),
                new ConectorTemplate(TipoConector.SALIDA
                    ↪ , "y"))), //endregion
//region Multiplexers
MUX2x1("MUX2x1", ImageUtils.pathImagenMedia("MUX2x1.png"
    ↪ ),
        ModelicaGenerator.MULTIPLEXERS + ".MUX2x1",
            ↪ TabPaleta.MULTIPLEXERS,
        List.of(new PropiedadSimple("0", "tLH",
            ↪ Propiedad.UNIDAD_TIME, "Low->High delay [s
            ↪ ]"),
                new PropiedadSimple("0", "tHL",
                    ↪ Propiedad.UNIDAD_TIME, "High->Low
                    ↪ delay [s]"),
                new PropiedadSimple("1", "strength",
                    ↪ Propiedad.UNIDAD_REAL, "Output
                    ↪ strength")),
        List.of(new ConectorTemplate(TipoConector.
            ↪ ENTRADA, "in1"),
                new ConectorTemplate(TipoConector.
                    ↪ ENTRADA, "in0"),
```

```
                    new ConectorTemplate(TipoConector.
                        ↪ ENTRADA, "sel", 0.5, 0),
                    new ConectorTemplate(TipoConector.SALIDA
                        ↪ , "out"))), //endregion
//region Examples.Utilities
MUX4("MUX4", ImageUtils.pathImagenMedia("MUX4.png"),
    ↪ ModelicaGenerator.UTILITIES + ".MUX4",
        TabPaleta.UTILITIES,
        List.of(new PropiedadSimple("0", "delayTime",
            ↪ Propiedad.UNIDAD_TIME, "delay time [s]"),
                new PropiedadLogic("q0", "'0'", "High->
                    ↪ Low delay [s]")),
        List.of(new ConectorTemplate(TipoConector.
            ↪ ENTRADA, "d0"),
                new ConectorTemplate(TipoConector.
                    ↪ ENTRADA, "d1"),
                new ConectorTemplate(TipoConector.
                    ↪ ENTRADA, "d2"),
                new ConectorTemplate(TipoConector.
                    ↪ ENTRADA, "d3"),
                new ConectorTemplate(TipoConector.
                    ↪ ENTRADA, "a0"),
                new ConectorTemplate(TipoConector.
                    ↪ ENTRADA, "a1"),
                new ConectorTemplate(TipoConector.SALIDA
                    ↪ , "d"))),
RS("RS", ImageUtils.pathImagenMedia("RS.png"),
    ↪ ModelicaGenerator.UTILITIES + ".RS",
        TabPaleta.UTILITIES,
        List.of(new PropiedadSimple("0", "delayTime",
            ↪ Propiedad.UNIDAD_TIME, "delay time [s]"),
```

```
                new PropiedadLogic("q0", "'0'", "initial
                    ↪ -value")),
            List.of(new ConectorTemplate(TipoConector.
                ↪ ENTRADA, "s"),
                    new ConectorTemplate(TipoConector.
                        ↪ ENTRADA, "r"),
                    new ConectorTemplate(TipoConector.SALIDA
                        ↪ , "q"),
                    new ConectorTemplate(TipoConector.SALIDA
                        ↪ , "qn"))),
RSFF("RSFF", ImageUtils.pathImagenMedia("RSFF.png"),
    ↪ ModelicaGenerator.UTILITIES + ".RSFF",
            TabPaleta.UTILITIES,
            List.of(new PropiedadSimple("0.01", "delayTime",
                ↪  Propiedad.UNIDAD_TIME,
                    "delay-time-[s]"), new PropiedadLogic("
                        ↪ q0", "'0'", "initial-value")),
            List.of(new ConectorTemplate(TipoConector.
                ↪ ENTRADA, "s"),
                    new ConectorTemplate(TipoConector.
                        ↪ ENTRADA, "r"),
                    new ConectorTemplate(TipoConector.SALIDA
                        ↪ , "q"),
                    new ConectorTemplate(TipoConector.SALIDA
                        ↪ , "qn"),
                    new ConectorTemplate(TipoConector.
                        ↪ ENTRADA, "clk"))),
DFF("DFF", ImageUtils.pathImagenMedia("DFF.png"),
    ↪ ModelicaGenerator.UTILITIES + ".DFF",
            TabPaleta.UTILITIES,
            List.of(new PropiedadSimple("0.01", "delayTime",
```

```
        ↪  Propiedad.UNIDAD_TIME,
            "delay-time-[s]"), new PropiedadLogic("
                ↪ q0", "'0'", "initial-value")),
        List.of(new ConectorTemplate(TipoConector.
            ↪ ENTRADA, "d"),
                new ConectorTemplate(TipoConector.SALIDA
                    ↪ , "q"),
                new ConectorTemplate(TipoConector.SALIDA
                    ↪ , "qn"),
                new ConectorTemplate(TipoConector.
                    ↪ ENTRADA, "clk"))),
JKFF("JKFF", ImageUtils.pathImagenMedia("JKFF.png"),
    ↪ ModelicaGenerator.UTILITIES + ".JKFF",
        TabPaleta.UTILITIES,
        List.of(new PropiedadSimple("0.01", "delayTime",
            ↪  Propiedad.UNIDAD_TIME,
            "delay-time-[s]"), new PropiedadLogic("
                ↪ q0", "'0'", "initial-value")),
        List.of(new ConectorTemplate(TipoConector.
            ↪ ENTRADA, "j"),
                new ConectorTemplate(TipoConector.SALIDA
                    ↪ , "q"),
                new ConectorTemplate(TipoConector.SALIDA
                    ↪ , "qn"),
                new ConectorTemplate(TipoConector.
                    ↪ ENTRADA, "clk"),
                new ConectorTemplate(TipoConector.
                    ↪ ENTRADA, "k"))),
HALFADDER("HalfAdder", ImageUtils.pathImagenMedia("
    ↪ HalfAdder.png"),
        ModelicaGenerator.UTILITIES + ".HalfAdder",
```

```
            ↪ TabPaleta.UTILITIES,
        List.of(new PropiedadSimple("0", "delayTime",
            ↪ Propiedad.UNIDAD_TIME, "delay-time-[s]")),
        List.of(new ConectorTemplate(TipoConector.
            ↪ ENTRADA, "a"),
                new ConectorTemplate(TipoConector.SALIDA
                    ↪ , "s"),
                new ConectorTemplate(TipoConector.
                    ↪ ENTRADA, "b"),
                new ConectorTemplate(TipoConector.SALIDA
                    ↪ , "c"))),
FULLADDER("FullAdder", ImageUtils.pathImagenMedia("
    ↪ FullAdder.png"),
        ModelicaGenerator.UTILITIES + ".FullAdder",
            ↪ TabPaleta.UTILITIES,
        List.of(new PropiedadSimple("0", "delayTime",
            ↪ Propiedad.UNIDAD_TIME, "delay-time-[s]")),
        List.of(new ConectorTemplate(TipoConector.
            ↪ ENTRADA, "a"),
                new ConectorTemplate(TipoConector.
                    ↪ ENTRADA, "b"),
                new ConectorTemplate(TipoConector.
                    ↪ ENTRADA, "c_in"),
                new ConectorTemplate(TipoConector.SALIDA
                    ↪ , "s"),
                new ConectorTemplate(TipoConector.SALIDA
                    ↪ , "c_out"))),
COUNTER3("Counter3", ImageUtils.pathImagenMedia("
    ↪ Counter3.png"),
        ModelicaGenerator.UTILITIES + ".Counter3",
            ↪ TabPaleta.UTILITIES, List.of(),
```

```
            List.of(new ConectorTemplate(TipoConector.
                ↪ ENTRADA, "enable"),
                    new ConectorTemplate(TipoConector.
                        ↪ ENTRADA, "count"),
                    new ConectorTemplate(TipoConector.SALIDA
                        ↪ , "q2"),
                    new ConectorTemplate(TipoConector.SALIDA
                        ↪ , "q1"),
                    new ConectorTemplate(TipoConector.SALIDA
                        ↪ , "q0"))),
    COUNTER("Counter", ImageUtils.pathImagenMedia("Counter3.
        ↪ png"),
            ModelicaGenerator.UTILITIES + ".Counter",
                ↪ TabPaleta.UTILITIES,
            List.of(new PropiedadSimple("0.001", "delayTime"
                ↪ , Propiedad.UNIDAD_TIME,
                    "delay-of-each-JKFF-[s]"), new
                        ↪ PropiedadLogic("q0", "initial-
                        ↪ value")),
            List.of(new ConectorTemplate(TipoConector.
                ↪ ENTRADA, "enable"),
                    new ConectorTemplate(TipoConector.
                        ↪ ENTRADA, "count"),
                    new ConectorTemplate(TipoConector.SALIDA
                        ↪ , "q", 1))),
    //endregion
    ;
    final String nombre;
    final String pathImagen;
    final String claseModelica;
    final TabPaleta tabPaleta;
```

```
    final List<Propiedad> propiedades;
    final List<ConectorTemplate> conectoresConNombre;

    TipoPieza(String nombre, String pathImagen, String
        ↪ claseModelica, TabPaleta tabPaleta, List<Propiedad
        ↪ > propiedades, List<ConectorTemplate>
        ↪ conectoresConNombre) {
        this.nombre = nombre;
        this.pathImagen = pathImagen;
        this.claseModelica = claseModelica;
        this.tabPaleta = tabPaleta;
        this.propiedades = propiedades;
        this.conectoresConNombre = conectoresConNombre;
    }
}
```

## B.7. Fichero `controlador/ControladorCircuito.java`

```
package caponera.uned.tfm.lizardclips.controlador;

import caponera.uned.tfm.lizardclips.constant.TipoConector;
import caponera.uned.tfm.lizardclips.constant.TipoPieza;
import caponera.uned.tfm.lizardclips.db.AbstractRepository;
import caponera.uned.tfm.lizardclips.db.CircuitoRepository;
import caponera.uned.tfm.lizardclips.db.ConexionRepository;
import caponera.uned.tfm.lizardclips.db.PiezaRepository;
import caponera.uned.tfm.lizardclips.gui.PanelCircuito;
import caponera.uned.tfm.lizardclips.gui.SelectorCircuito;
import caponera.uned.tfm.lizardclips.gui.VentanaPrincipal;
import caponera.uned.tfm.lizardclips.gui.
    ↪ VentanaVisualizarCodigo;
```

```
import caponera.uned.tfm.lizardclips.modelica.
    ↪ ModelicaGenerator;
import caponera.uned.tfm.lizardclips.modelo.Circuito;
import caponera.uned.tfm.lizardclips.modelo.Conector;
import caponera.uned.tfm.lizardclips.modelo.Conexion;
import caponera.uned.tfm.lizardclips.modelo.Pieza;
import caponera.uned.tfm.lizardclips.utils.I18NUtils;
import caponera.uned.tfm.lizardclips.utils.ImageUtils;
import caponera.uned.tfm.lizardclips.utils.Punto;
import lombok.Setter;

import javax.swing.JFileChooser;
import javax.swing.JOptionPane;
import javax.swing.SwingUtilities;
import javax.swing.filechooser.FileFilter;
import java.awt.Desktop;
import java.awt.Dimension;
import java.awt.Graphics2D;
import java.awt.Rectangle;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Iterator;
import java.util.List;
import java.util.Locale;
import java.util.Map;
import java.util.Optional;
import java.util.stream.Collectors;
```

```
import java.util.stream.Stream;

public class ControladorCircuito {
    private static final String CARACTERES_VALIDOS_NOMBRE =
            "
                ↪ abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ_123
                ↪ ";
    private final PanelCircuito panelCircuito;
    private final CircuitoRepository circuitoRepository;
    private final PiezaRepository piezaRepository;
    private final ConexionRepository conexionRepository;
    private Circuito circuito;

    @Setter
    private VentanaPrincipal ventanaPrincipal;

    public ControladorCircuito(Circuito circuito,
       ↪ PanelCircuito panelCircuito) {
        this.panelCircuito = panelCircuito;
        panelCircuito.setControladorCircuito(this);
        setCircuito(circuito);
        circuitoRepository = new CircuitoRepository();
        piezaRepository = new PiezaRepository();
        conexionRepository = new ConexionRepository();
    }


    private void setCircuito(Circuito circuito) {
        if (this.circuito != null) {
            for (Pieza p : circuito.getComponentes()) {
                p.setCircuito(circuito);
```

```
            }
        }
        this.circuito = circuito;
        this.circuito.setControlador(this);

        panelCircuito.cambiarCircuito();
        if (ventanaPrincipal != null && circuito.getNombre()
            ↪ != null &&
                !circuito.getNombre().isBlank()) {
            ventanaPrincipal.setNombreCircuito(circuito.
                ↪ getNombre());
        }

    }

    public void colocarPieza(Pieza pieza, Punto posicion) {
        circuito.colocarPieza(pieza, posicion);
    }

    public void borrarPieza(Pieza pieza) {
        circuito.borrarPieza(pieza);
    }

    public void arrastrarPieza(Pieza pieza, Punto posicion,
        ↪ Dimension grabPoint) {
        Punto posicionReal = posicion;
        posicionReal.translate((int) (-grabPoint.getWidth()
            ↪ / Punto.getEscala()),
                (int) (-grabPoint.getHeight() / Punto.
                    ↪ getEscala()));
        if (dentroDelPanel(posicionReal, pieza.getTamano()))
```

```
        ↪ {
            circuito.moverPieza(pieza, posicionReal);
        }
    }

    private boolean dentroDelPanel(Punto posicion, Dimension
        ↪ tamanoPieza) {
        Rectangle tamanoPanel = panelCircuito.getBounds();
        tamanoPanel.setLocation(0,
                0);//Si no, se tiene un offset igual al
                    ↪ tama o de los componentes a la
                    ↪ izquierda en X y arriba en Y
        Punto esquinaInferiorDerecha = new Punto((int) (
            ↪ posicion.getX() + tamanoPieza.getWidth()),
                (int) (posicion.getY() + tamanoPieza.
                    ↪ getHeight()));
        return tamanoPanel.contains(posicion.getPoint()) &&
                tamanoPanel.contains(esquinaInferiorDerecha.
                    ↪ getPoint());
    }

    public Pieza getPiezaByPosicion(Punto posicionRaton) {
        return circuito.getComponentes().stream()
                .filter(p -> p.getBounds().contains(
                    ↪ posicionRaton.getPoint())).findFirst()
                .orElse(null);
    }

    public Conector getConectorByPosicion(Punto posicion) {
        return getConectorByPosicion(getAllConectoresStream
            ↪ ().iterator(), posicion);
```

```
}

public List<Pieza> getPiezas() {
    return circuito.getComponentes();
}

private Conector getConectorByPosicion(Iterator<Conector
    ↪ > candidatos, Punto posicion) {
    Conector conector = null;
    while (conector == null && candidatos.hasNext()) {
        Conector c = candidatos.next();
        Punto posicionConector =
                c.getPieza().getPosicionConectorEnPanel(
                    ↪ c, c.getPieza().getPosicion());
        double d = Math.sqrt(Math.pow(posicionConector.
            ↪ getX() - posicion.getX(), 2) +
                Math.pow(posicionConector.getY() -
                    ↪ posicion.getY(), 2));
        System.out.println(d);
        if (d <= Conector.getRadio()) {
            System.out.println("clicking on conector");
            conector = c;
        }
    }
    return conector;
}

private Stream<Conector> getAllConectoresStream() {
    return circuito.getComponentes().stream().map(Pieza
        ↪ ::getConectores).flatMap(List::stream);
```

```
}

public List<Conector> getAllConectores() {
    return getAllConectoresStream().collect(Collectors.
        ↪ toList());
}

public List<Conector> getConectoresValidos(Conector
    ↪ conectorSeleccionado) {
    //Solo puede haber una conexion por conector de
        ↪ entrada
    List<Conector> conectoresEntradaOcupados =
            circuito.getConexiones().stream().filter(
                ↪ Conexion::isComplete)
                    .flatMap(conexion -> Stream.of(
                        ↪ conexion.getOrigen(), conexion
                        ↪ .getDestino()))
                    .filter(conector -> conector.
                        ↪ getTipoConector().equals(
                        ↪ TipoConector.ENTRADA))
                    .toList();
    if (conectoresEntradaOcupados.contains(
        ↪ conectorSeleccionado)) {
        return new ArrayList<>();// Si se selecciona un
            ↪ conector de entrada que y est  ocupado no
            ↪  se puede conectar a ning n sitio
    }
    return getAllConectoresStream().filter(
            con -> !con.getPieza().equals(
                ↪ conectorSeleccionado.getPieza()) &&
                    !con.getTipoConector().equals(
```

```
                        ↪ conectorSeleccionado.
                        ↪ getTipoConector()) &&
                    !conectoresEntradaOcupados.contains(
                        ↪ con)).collect(Collectors.
                        ↪ toList());
    }

    public Conector getConectorByPosicion(Pieza pieza, Punto
        ↪ posicion) {
        Iterator<Conector> it = pieza.getConectores().
            ↪ iterator();
        return getConectorByPosicion(it, posicion);
    }

    private boolean puntoDentroDeBounds(Punto punto, Map.
        ↪ Entry<Pieza, Punto> par) {
        Rectangle bounds = par.getKey().getBounds();
        return bounds.contains(punto.getPoint());
    }

    public void generarPieza(TipoPieza tipoPieza) {
        panelCircuito.addPiezaByDragging(new Pieza(circuito,
            ↪ tipoPieza));
    }

    private Optional<Conexion> getConexionEnCursoOptional()
        ↪ {
        return circuito.getConexiones().stream().filter(
            ↪ Conexion::enCurso).findFirst();
    }
```

```
private Conexion getConexionEnCurso() {
    Optional<Conexion> enCurso =
       ↪ getConexionEnCursoOptional();
    if (!enCurso.isPresent()) {
        throw new RuntimeException(I18NUtils.getString("
           ↪ no_current_connection"));
    }
    return enCurso.get();
}

public void finalizarConexion(Conector destino) {
    getConexionEnCurso().cerrar(destino);
}


public void iniciarConexion(Conector origen) {
    Conexion conexionEnCurso = new Conexion(origen);
    circuito.addConexion(conexionEnCurso);
}

public void addPointConexion(Punto punto) {
    System.out.println("adding point");
    getConexionEnCurso().addPoint(punto);
    System.out.println(getConexionEnCurso());
}

public List<Conexion> getConexiones() {
    return circuito.getConexiones();
}

public void borrarConexionesIncompletas() {
```

```
        Optional<Conexion> enCurso =
            ↪ getConexionEnCursoOptional();
        enCurso.ifPresent(circuito::borrarConexion);
    }

    public void borrarConexion(Conexion clicada) {
        circuito.borrarConexion(clicada);
    }

    public void guardar(boolean duplicar) {
        String nombre;
        if (duplicar) {
            /*circuito.setIdCircuito(null);
            circuito.setNombre("");
            circuito.getComponentes().forEach(pieza -> pieza
                ↪ .setIdPieza(null));
            circuito.getConexiones().forEach(conexion ->
                ↪ conexion.setIdConexion(null));
            circuito.getConexiones().forEach(conexion ->
                ↪ conexion.getOrigen().setIdConector(null));
            circuito.getConexiones().forEach(conexion ->
                ↪ conexion.getDestino().setIdConector(null))
                ↪ ;*/
            setCircuito(new Circuito(circuito));
        }

        if (circuito.getNombre() == null || circuito.
            ↪ getNombre().isBlank()) {
            nombre = JOptionPane.showInputDialog(I18NUtils.
                ↪ getString("save_circuit_as"));
            circuito.setNombre(nombre);
```

```
        }
        if (circuito.getNombre() != null &&
                !circuito.getNombre().isBlank()) { //No
                    ↪ guardar si se hace cancel
            circuito.setThumbnail(ImageUtils.
                ↪ bytesFromBufferedImage(generarThumbnail())
                ↪ );
            AbstractRepository.startTransaction();
            circuito.getComponentes().forEach(
                ↪ piezaRepository::guardar);
            circuito.getConexiones().forEach(
                ↪ conexionRepository::guardar);
            Circuito guardado = circuitoRepository.guardar(
                ↪ circuito);
            AbstractRepository.endTransaction();
            setCircuito(guardado);
        }
    }

    public void guardar() {
        guardar(false);
    }

    public void guardar_como() {
        guardar(true);
    }

    public void cargar() {
        List<Circuito> circuitos = circuitoRepository.getAll
            ↪ ();
        SelectorCircuito sc = new SelectorCircuito(this,
```

```
            ↪ circuitos);
        String[] opciones = {I18NUtils.getString("select"),
            ↪ I18NUtils.getString("cancel")};
        int res = JOptionPane.showOptionDialog(null, sc,
            ↪ I18NUtils.getString("select_circuit"),
                JOptionPane.YES_NO_OPTION, JOptionPane.
                    ↪ QUESTION_MESSAGE, null, opciones,
                opciones[0]);
        if (res == 0) {
            Circuito seleccionado = sc.
                ↪ getCircuitoSeleccionado();
            if (seleccionado == null) {
                throw new RuntimeException(I18NUtils.
                    ↪ getString("must_select_circuit"));
            }
            setCircuito(seleccionado);
            System.out.println("cargado circuito" + circuito
                ↪ );
            //Punto.resetReferencia();
        }

    }

    public void nuevoCircuito() {
        setCircuito(new Circuito());
        panelCircuito.cambiarCircuito();
        ventanaPrincipal.setNombreCircuito(I18NUtils.
            ↪ getString("untitled_circuit"));
    }

    public BufferedImage generarThumbnail() {
```

```
        BufferedImage image = new BufferedImage(
            ↪ panelCircuito.getWidth(), panelCircuito.
            ↪ getHeight(),
                    BufferedImage.TYPE_INT_RGB);
        Graphics2D g = image.createGraphics();
        panelCircuito.printAll(g);
        g.dispose();
        return image;
    }

    private boolean continuarSiHayInputsDesconectados() {
        boolean continuar = true;
        List<Conexion> conexiones = circuito.getConexiones()
            ↪ ;
        List<Conector> conectoresInput =
                circuito.getComponentes().stream().flatMap(p
                    ↪ -> p.getConectores().stream())
                        .filter(c -> c.getTipoConector().
                            ↪ equals(TipoConector.ENTRADA)).
                            ↪ toList();
        //Si, para un determinado conector, no existe
            ↪ ninguna conexi n que lo contenga
        if (conectoresInput.stream().anyMatch(conector ->
            ↪ conexiones.stream().noneMatch(
                conexion -> conexion.getOrigen().equals(
                    ↪ conector) || conexion.getDestino().
                    ↪ equals(conector)))) {
            String[] opciones = {I18NUtils.getString("
                ↪ continue"), I18NUtils.getString("cancel")
                ↪ };
            int confirmacion =
```

```
                    JOptionPane.showOptionDialog(
                        ↪ ventanaPrincipal.getFrame(),
                        ↪ I18NUtils.getString("empty_inputs"
                        ↪ ),
                            I18NUtils.getString("
                                ↪ errors_in_the_circuit"),
                                ↪ JOptionPane.YES_NO_OPTION,
                            JOptionPane.WARNING_MESSAGE,
                                ↪ null, opciones, opciones
                                ↪ [1]);
            continuar = confirmacion == 0;
        }

        return continuar;
    }

    public void exportarCodigo() {
        if (continuarSiHayInputsDesconectados()) {
            String codigoModelica = ModelicaGenerator.
                ↪ generarCodigoModelica(circuito);
            System.out.println(codigoModelica);

            JFileChooser fileChooser = new JFileChooser();
            fileChooser.setDialogTitle(I18NUtils.getString("
                ↪ export_modelica_as"));
            fileChooser.setAcceptAllFileFilterUsed(false);
            fileChooser.setFileFilter(new FileFilter() {
                @Override
                public boolean accept(File f) {
                    return f.getName().endsWith(".mo");
                }
```

```
        @Override
        public String getDescription() {
            return "Modelica files";
        }
    });

    int userSelection = fileChooser.showSaveDialog(
        ↪ ventanaPrincipal.getFrame());

    if (userSelection == JFileChooser.APPROVE_OPTION
        ↪ ) {
        File fileToSave = fileChooser.
            ↪ getSelectedFile();
        if (!fileToSave.getName().toLowerCase(Locale
            ↪ .ROOT).endsWith(".mo")) {
            fileToSave = new File(fileToSave.
                ↪ getParentFile(), fileToSave.
                ↪ getName() + ".mo");
        }
        try {
            FileWriter fw = new FileWriter(
                ↪ fileToSave);
            fw.write(codigoModelica);
            fw.close();
            int result = JOptionPane.
                ↪ showConfirmDialog(ventanaPrincipal
                ↪ .getFrame(),
                    I18NUtils.getString("
                        ↪ open_file_now_prompt"),
                    I18NUtils.getString("open_file")
```

```
                                    ↪ , JOptionPane.
                                    ↪ YES_NO_OPTION,
                                JOptionPane.QUESTION_MESSAGE);
                        if (result == JOptionPane.YES_OPTION) {
                            Desktop.getDesktop().open(fileToSave
                                ↪ );
                        }
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                }
            }
        }

        public void verCodigo() {
            if (continuarSiHayInputsDesconectados()) {
                String codigoModelica = ModelicaGenerator.
                    ↪ generarCodigoModelica(circuito);
                SwingUtilities.invokeLater(
                        () -> new VentanaVisualizarCodigo(
                            ↪ codigoModelica, 500, 500).
                            ↪ setVisible(true));
            }
        }

        /*public void addConectorToPieza(Pieza p) {
            p.addConectorEntrada();
            panelCircuito.repaint();
        }

        public void removeConectorFromPieza(Pieza p) {
```

```
        p.removeConectorEntrada();
        panelCircuito.repaint();
    }*/

    public void cancelarConexion() {
        circuito.cancelarConexion();
    }

    public void toggleNombresPiezas() {
        Pieza.setRenerNombresPiezas(!Pieza.
            ↪ isRenerNombresPiezas());
        panelCircuito.repaint();
    }

    public void toggleNombresPines() {
        Pieza.setRenderNombresPines(!Pieza.
            ↪ isRenderNombresPines());
        panelCircuito.repaint();
    }

    public void renombrarPieza(Pieza pieza, String
       ↪ nuevoNombre) {
        sanitize(nuevoNombre);
        if (circuito.getComponentes().stream()
                .anyMatch(p -> ModelicaGenerator.nombrePieza
                    ↪ (p).equals(nuevoNombre))) {
            throw new RuntimeException(I18NUtils.getString("
                ↪ duplicate_component_name"));
        }
        pieza.setNombrePieza(nuevoNombre);
        panelCircuito.repaint();
```

```
    }

    private void sanitize(String nombre) {
        char char0 = nombre.charAt(0);
        if (!caracteresPermitidos(nombre) || Character.
            ↪ isUpperCase(char0) ||
                !Character.isAlphabetic(char0)) {
            throw new RuntimeException(I18NUtils.getString("
                ↪ invalid_component_name"));
        }
    }

    private boolean caracteresPermitidos(String nombre) {
        return Arrays.stream(nombre.split("")).allMatch(
            ↪ CARACTERES_VALIDOS_NOMBRE::contains);
    }

    public void rotarPieza(Pieza pieza, boolean derecha) {
        pieza.rotar(derecha);
        panelCircuito.repaint();
    }


    public void actualizarConectoresPieza(Pieza p, int[]
        ↪ newNPinesConectoresMultiples) {
        p.actualizarConectores(newNPinesConectoresMultiples)
            ↪ ;
        panelCircuito.repaint();
    }
}
```

## B.8. Fichero db/AbstractRepository.java

```
package caponera.uned.tfm.lizardclips.db;

import jakarta.persistence.EntityManager;
import jakarta.persistence.EntityTransaction;
import jakarta.persistence.TypedQuery;
import jakarta.persistence.criteria.CriteriaBuilder;
import jakarta.persistence.criteria.CriteriaQuery;
import jakarta.persistence.criteria.Root;

import java.util.List;

public abstract class AbstractRepository<T> {
    protected static EntityManager em;
    protected static EntityTransaction currentTransaction;


    public AbstractRepository() {
        if (em == null) {
            em = DBUtils.getEntityManager();
        }
    }

    public static void startTransaction() {
        currentTransaction = em.getTransaction();
        currentTransaction.begin();
        System.out.println("Transaction started");
    }
```

```
public static void endTransaction() {
    currentTransaction.commit();
    System.out.println("Transaction ended");
}

protected abstract Integer getIdElemento(T elemento);

protected abstract Class<T> getObjectClass();

public List<T> getAll() {
    //Query q = em.createQuery(getAllQuery());
    CriteriaBuilder cb = em.getCriteriaBuilder();
    CriteriaQuery<T> cq = cb.createQuery(getObjectClass
    ↪ ());
    Root<T> rootEntry = cq.from(getObjectClass());
    CriteriaQuery<T> all = cq.select(rootEntry);
    TypedQuery<T> allQuery = em.createQuery(all);
    List<T> result = allQuery.getResultList();
    for (T t : result) {
        em.detach(t);
    }
    return result;
    //return (List<T>) q.getResultList();
}

public T guardar(T t) {
    System.out.println("Guardando " + t.getClass().
    ↪ getSimpleName() + ": " + t);
    //startTransaction();
    try {
        if (getIdElemento(t) == null) {
```

```
                em.persist(t);
            } else {
                em.merge(t);
                em.flush();
            }
            //endTransaction();
            System.out.println(t.getClass().getSimpleName()
               ↪ + "-guardado");
        } catch (Exception e) {
            e.printStackTrace();
            currentTransaction.rollback();
        }
        return t;

    }
}
```

## B.9. Fichero db/CircuitoRepository.java

```
package caponera.uned.tfm.lizardclips.db;

import caponera.uned.tfm.lizardclips.modelo.Circuito;

public class CircuitoRepository extends AbstractRepository<
   ↪ Circuito> {

    @Override
    protected Integer getIdElemento(Circuito elemento) {
        return elemento.getIdCircuito();
    }
```

```
    @Override
    protected Class<Circuito> getObjectClass() {
        return Circuito.class;
    }
}
```

## B.10. Fichero db/ConexionRepository.java

```
package caponera.uned.tfm.lizardclips.db;

import caponera.uned.tfm.lizardclips.modelo.Conexion;

public class ConexionRepository extends AbstractRepository<
    ↪ Conexion> {

    @Override
    protected Integer getIdElemento(Conexion elemento) {
        return elemento.getIdConexion();
    }

    @Override
    protected Class<Conexion> getObjectClass() {
        return Conexion.class;
    }
}
```

## B.11. Fichero db/DBUtils.java

```
package caponera.uned.tfm.lizardclips.db;

import jakarta.persistence.EntityManager;
```

```
import jakarta.persistence.EntityManagerFactory;
import jakarta.persistence.Persistence;

public class DBUtils {
    public static final String PERSISTENCE_UNIT_NAME = "
        ↪ lizardClipsPersistenceUnit";

    private static EntityManager em;

    /**
     * EntityManagerSingleton
     *
     * @return The entity manager
     */
    public static EntityManager getEntityManager() {
        if (em == null) {
            EntityManagerFactory factory =
                    Persistence.createEntityManagerFactory(
                        ↪ DBUtils.PERSISTENCE_UNIT_NAME);
            em = factory.createEntityManager();
        }

        return em;
    }
}
```

## B.12. Fichero db/PiezaRepository.java

```
package caponera.uned.tfm.lizardclips.db;

import caponera.uned.tfm.lizardclips.modelo.Pieza;
```

```
public class PiezaRepository extends AbstractRepository<
    ↪ Pieza> {
    @Override
    protected Integer getIdElemento(Pieza elemento) {
        return elemento.getIdPieza();
    }

    @Override
    protected Class<Pieza> getObjectClass() {
        return Pieza.class;
    }
}
```

## B.13. Fichero gui/EditorPropiedadesPieza.java

```
package caponera.uned.tfm.lizardclips.gui;

import caponera.uned.tfm.lizardclips.constant.
    ↪ ConectorTemplate;
import caponera.uned.tfm.lizardclips.modelica.
    ↪ ModelicaGenerator;
import caponera.uned.tfm.lizardclips.modelo.Pieza;
import caponera.uned.tfm.lizardclips.modelo.Propiedad;
import caponera.uned.tfm.lizardclips.modelo.
    ↪ PropiedadSeleccionMultiple;
import caponera.uned.tfm.lizardclips.modelo.PropiedadSimple;

import javax.swing.*;
import java.util.ArrayList;
import java.util.HashMap;
```

```
import java.util.List;
import java.util.Map;

public class EditorPropiedadesPieza extends JComponent {
    private Pieza pieza;
    private Map<Propiedad, JComponent> mapaPropiedades;

    private JTextField tfNombrePieza;
    private List<JSpinner> tfNumeroPinesConectoresMultiples;

    public EditorPropiedadesPieza(Pieza pieza) {
        this.pieza = pieza;
        mapaPropiedades = new HashMap<>();

        setLayout(new BoxLayout(this, BoxLayout.Y_AXIS));

        tfNombrePieza = new JTextField(ModelicaGenerator.
            ↪ nombrePieza(pieza));
        addPropiedad("nombre", null, tfNombrePieza);

        tfNumeroPinesConectoresMultiples = new ArrayList<>()
            ↪ ;
        List<ConectorTemplate> conectoresMultiples =
                pieza.getTipoPieza().getConectoresConNombre
                    ↪ ().stream().filter(ConectorTemplate::
                    ↪ isMultiple).toList();
        for (int i = 0; i < conectoresMultiples.size(); i++)
            ↪ {
            ConectorTemplate ct = conectoresMultiples.get(i)
                ↪ ;
            JSpinner selector =
```

```
                new JSpinner(new SpinnerNumberModel(
                    ↪ pieza.getNPinesConectorMultiple(i)
                    ↪ , ct.getMinConectores(),
                        ct.getMaxConectores(), 1));
        addPropiedad("n_" + ct.getNombre(), "n mero de
            ↪ pines para el conector " + ct.getNombre(),
            ↪  selector);
        tfNumeroPinesConectoresMultiples.add(selector);
    }


    List<Propiedad> propiedades = pieza.getTipoPieza().
        ↪ getPropiedades();
    for (int i = 0; i < propiedades.size(); i++) {
        Propiedad p = propiedades.get(i);
        JComponent rightSide;
        if (p instanceof PropiedadSeleccionMultiple) {
            List<String> valoresPosibles =
                    ((PropiedadSeleccionMultiple) p).
                        ↪ getValoresPosibles();
            rightSide = new JComboBox<String>(
                ↪ valoresPosibles.toArray(new String[0])
                ↪ );
            ((JComboBox) rightSide).setSelectedIndex(
                    valoresPosibles.indexOf(pieza.
                        ↪ getValoresPropiedades()[i]));

        } else if (p instanceof PropiedadSimple) {
            rightSide = new JTextField(pieza.
                ↪ getValoresPropiedades()[i]);
        } else {
```

```
                throw new RuntimeException("Propiedad de
                    ↪ tipo desconocido: " + p);
            }

            mapaPropiedades.put(p, rightSide);

            addPropiedad(p.getNombre(), p.
                ↪ getTooltipDescription(), rightSide);
        }

    }

    private void addPropiedad(String name, String
        ↪ tooltipDescription, JComponent rightSide) {
        JPanel row = new JPanel();
        row.setLayout(new BoxLayout(row, BoxLayout.X_AXIS));
        JLabel nameLabel = new JLabel(name);
        if (tooltipDescription != null) {
            nameLabel.setToolTipText(tooltipDescription);
        }
        row.add(nameLabel);
        row.add(rightSide);
        add(row);
    }

    public void actualizarValorPropiedades() {
        String nuevoNombre = tfNombrePieza.getText();
        if (nuevoNombre != null && !nuevoNombre.isEmpty() &&
                !nuevoNombre.equals(ModelicaGenerator.
                    ↪ nombrePieza(pieza))) {
            pieza.getCircuito().getControlador().
```

```
            ↪ renombrarPieza(pieza, nuevoNombre);
    }

    int[] newNPinesConectoresMultiples =
        ↪ tfNumeroPinesConectoresMultiples.stream()
            .mapToInt(sp -> ((SpinnerNumberModel) sp.
                ↪ getModel()).getNumber().intValue()).
                ↪ toArray();
    pieza.getCircuito().getControlador().
        ↪ actualizarConectoresPieza(pieza,
        ↪ newNPinesConectoresMultiples);

    List<Propiedad> propiedades = pieza.getTipoPieza().
        ↪ getPropiedades();
    for (int i = 0; i < propiedades.size(); i++) {
        Propiedad p = propiedades.get(i);
        String nuevoValor;
        if (p instanceof PropiedadSeleccionMultiple) {
            nuevoValor = ((PropiedadSeleccionMultiple) p
                ↪ ).getValoresPosibles()
                    .get(((JComboBox) mapaPropiedades.
                        ↪ get(
                            p)).getSelectedIndex());
        } else if (p instanceof PropiedadSimple) {
            nuevoValor = ((JTextField) mapaPropiedades.
                ↪ get(p)).getText();
        } else {
            throw new RuntimeException("Propiedad de
                ↪ tipo desconocido: " + p);
        }
        pieza.setValorPropiedad(i, nuevoValor);
```

```
        }
    }
}
```

## B.14. Fichero gui/PanelCircuito.java

```
package caponera.uned.tfm.lizardclips.gui;

import caponera.uned.tfm.lizardclips.constant.ModoPanel;
import caponera.uned.tfm.lizardclips.controlador.
    ↪ ControladorCircuito;
import caponera.uned.tfm.lizardclips.modelo.Conector;
import caponera.uned.tfm.lizardclips.modelo.Conexion;
import caponera.uned.tfm.lizardclips.modelo.Pieza;
import caponera.uned.tfm.lizardclips.utils.I18NUtils;
import caponera.uned.tfm.lizardclips.utils.ImageUtils;
import caponera.uned.tfm.lizardclips.utils.LineUtils;
import caponera.uned.tfm.lizardclips.utils.Punto;
import lombok.Getter;

import javax.swing.AbstractAction;
import javax.swing.JComponent;
import javax.swing.JMenuItem;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JPopupMenu;
import javax.swing.KeyStroke;
import javax.swing.SwingUtilities;
import java.awt.BasicStroke;
import java.awt.Color;
import java.awt.Dimension;
```

```
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.MouseInfo;
import java.awt.Point;
import java.awt.Rectangle;
import java.awt.event.ActionEvent;
import java.awt.event.KeyEvent;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;
import java.awt.event.MouseWheelEvent;
import java.awt.event.MouseWheelListener;
import java.awt.geom.Line2D;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class PanelCircuito extends JPanel implements
    ↪ MouseListener, MouseMotionListener, MouseWheelListener
    ↪ {
    private ModoPanel modo;
    private Dimension grabPoint;
    private Pieza piezaSeleccionada;
    private Conector conectorSeleccionado;
    private Point grabPointDesplazamiento;
    @Getter
    private ControladorCircuito controladorCircuito;

    public PanelCircuito() {
        addMouseListener(this);
```

```
        addMouseMotionListener(this);
        addMouseWheelListener(this);
        setModo(ModoPanel.MODO_NORMAL);
        setupEscKey();
        setBackground(Color.WHITE);
    }

    private void setupEscKey() {
        getInputMap(JComponent.WHEN_IN_FOCUSED_WINDOW).put(
                KeyStroke.getKeyStroke(KeyEvent.VK_ESCAPE,
                    ↪ 0), "LizardClips:canelConexion");
        getActionMap().put("LizardClips:canelConexion", new
            ↪ AbstractAction() {
            @Override
            public void actionPerformed(ActionEvent e) {
                if (isModo(ModoPanel.MODO_CONEXION)) {
                    controladorCircuito.cancelarConexion();
                    setModo(ModoPanel.MODO_NORMAL);
                }
            }
        });
    }

    public void cambiarCircuito() {
        setModo(ModoPanel.MODO_NORMAL);
        grabPoint = null;
        piezaSeleccionada = null;
        conectorSeleccionado = null;
        repaint();
    }
```

```
public void setControladorCircuito(ControladorCircuito
    ↪ controladorCircuito) {
    this.controladorCircuito = controladorCircuito;
    repaint();
}

public void addPieza(Punto posicion, Pieza pieza) {
    controladorCircuito.colocarPieza(pieza, posicion);
    repaint();
}

public void addPiezaByDragging(Pieza pieza) {
    Point raton = MouseInfo.getPointerInfo().getLocation
        ↪ ();
    raton.translate((int) -getLocationOnScreen().getX(),
        ↪  (int) -getLocationOnScreen().getY());
    addPieza(new Punto((int) Math.max(0, raton.getX()),
        ↪ (int) Math.max(0, raton.getY())),
            pieza);
    startDragging(pieza, new Dimension(pieza.getWidth()
        ↪ / 2, pieza.getHeight() / 2));
}

public void toggleDeleteMode() {
    if (isModo(ModoPanel.MODO_BORRADO)) {
        setModo(ModoPanel.MODO_NORMAL);
    } else {
        setModo(ModoPanel.MODO_BORRADO);
    }
}
```

```
private void startDragging(Pieza pieza, Dimension
    ↪ grabPoint) {
    if (!isModo(ModoPanel.MODO_BORRADO)) {
        setModo(ModoPanel.MODO_ARRASTRANDO);
        this.grabPoint = grabPoint;
    }
    this.piezaSeleccionada = pieza;
}


@Override
public void mousePressed(MouseEvent e) {
    if (SwingUtilities.isMiddleMouseButton(e)) {
        setModo(ModoPanel.MODO_DESPLAZANDO);
        grabPointDesplazamiento = e.getPoint();
    } else {
        Pieza piezaSeleccionado =
                controladorCircuito.getPiezaByPosicion(
                    ↪ new Punto(e.getPoint()));
        if (piezaSeleccionado == null) {//No ha hecho
            ↪ click sobre una pieza ni conector
            if (isModo(ModoPanel.MODO_BORRADO)) {
                Conexion clicada = detectarClickLineas(e
                    ↪ .getPoint());
                if (clicada != null) { //Ha pulsado una
                    ↪ conexion
                    controladorCircuito.borrarConexion(
                        ↪ clicada);
                    repaint();
                }
            } else if (isModo(ModoPanel.MODO_CONEXION))
```

```
                    ↪ {
                        controladorCircuito.addPointConexion(new
                            ↪ Punto(e.getPoint()));
                        repaint();
                    }
                } else { //Ha pulsado una pieza o un conector
                    Conector conector = controladorCircuito.
                        ↪ getConectorByPosicion(
                        ↪ piezaSeleccionado,
                            new Punto(e.getPoint()));
                    if (conector == null) { //Ha pulsado una
                        ↪ pieza
                        if (SwingUtilities.isRightMouseButton(e)
                            ↪ ) {//Click derecho sobre la pieza
                            clickDerechoPieza(piezaSeleccionado,
                                ↪ e);
                        } else {//Click normal sobre la pieza
                            piezaSeleccionada(piezaSeleccionado,
                                ↪ e);
                        }
                    } else {//Ha pulsado un conector
                        conectorSeleccionado(conector);
                    }
                }
            }
        }

        private void clickDerechoPieza(Pieza piezaSeleccionado,
            ↪ MouseEvent e) {
            JPopupMenu menuPieza = new JPopupMenu("Modificar
                ↪ pieza");
```

```
        JMenuItem rotarDerecha = new JMenuItem(I18NUtils.
            ↪ getString("rotate_right"));
        rotarDerecha.addActionListener(
                click -> controladorCircuito.rotarPieza(
                    ↪ piezaSeleccionado, true));
        menuPieza.add(rotarDerecha);

        JMenuItem rotarIzquierda = new JMenuItem(I18NUtils.
            ↪ getString("rotate_left"));
        rotarIzquierda.addActionListener(
                click -> controladorCircuito.rotarPieza(
                    ↪ piezaSeleccionado, false));
        menuPieza.add(rotarIzquierda);
        menuPieza.show(this, e.getX(), e.getY());
    }

    private Conexion detectarClickLineas(Point point) {
        return controladorCircuito.getConexiones().stream()
                .filter(con -> LineUtils.getParejas(con.
                    ↪ getPuntosManhattan())
                        .stream().anyMatch(
                                pareja -> new Line2D.Double(
                                    ↪ pareja[0].getPoint(),
                                        pareja[1].getPoint()
                                            ↪ ).intersects(
                                        new Rectangle(point,
                                                new
                                                    ↪ Dimension
                                                    ↪ (10,
                                                    ↪ 10))))
```

```
                                                    ↪ )
                .findFirst().orElse(null);
    }

    private void conectorSeleccionado(Conector conector) {
        if (isModo(ModoPanel.MODO_CONEXION)) {
            if (controladorCircuito.getConectoresValidos(
                ↪ conectorSeleccionado).contains(conector))
                ↪ {
                controladorCircuito.finalizarConexion(
                    ↪ conector);
                setModo(ModoPanel.MODO_NORMAL);
            } else {
                throw new RuntimeException(I18NUtils.
                    ↪ getString("invalid-connection"));
            }
        } else {
            this.conectorSeleccionado = conector;
            controladorCircuito.iniciarConexion(conector);
            setModo(ModoPanel.MODO_CONEXION);
        }

        repaint();
    }

    private void piezaSeleccionada(Pieza piezaSeleccionado,
        ↪ MouseEvent e) {
        if (isModo(ModoPanel.MODO_BORRADO)) {
            borrarComponente(piezaSeleccionado);
        } else {
            Dimension grabPoint =
```

```
                    new Dimension((int) (e.getX() -
                        ↪ piezaSeleccionado.getPosicion().
                        ↪ getX()),
                            (int) (e.getY() -
                                ↪ piezaSeleccionado.
                                ↪ getPosicion().getY()));
            startDragging(piezaSeleccionado, grabPoint);
        }

    }


    private void borrarComponente(Pieza componente) {
        System.out.println("Borrando");
        controladorCircuito.borrarPieza(componente);
        repaint();
    }

    @Override
    public void mouseReleased(MouseEvent e) {
        if (isModo(ModoPanel.MODO_ARRASTRANDO) || isModo(
            ↪ ModoPanel.MODO_DESPLAZANDO)) {
            setModo(ModoPanel.MODO_NORMAL);
        }
    }


    @Override
    public void mouseDragged(MouseEvent e) {
        if (isModo(ModoPanel.MODO_ARRASTRANDO)) {
            controladorCircuito.arrastrarPieza(
```

```
                ↪ piezaSeleccionada, new Punto(e.getPoint())
                ↪ ,
                        grabPoint);
                repaint();
            } else if (isModo(ModoPanel.MODO_DESPLAZANDO)) {
                Punto.desplazarReferencia((int) (e.getX() -
                ↪ grabPointDesplazamiento.getX()),
                        (int) (e.getY() -
                        ↪ grabPointDesplazamiento.getY()));
                grabPointDesplazamiento = e.getPoint();
                repaint();
            }
        }

        @Override
        public void mouseMoved(MouseEvent e) {
            mouseDragged(e);
            if (isModo(ModoPanel.MODO_CONEXION)) {
                repaint();
            }
        }


        @Override
        public void mouseClicked(MouseEvent e) {
            if (e.getClickCount() == 2 && SwingUtilities.
            ↪ isLeftMouseButton(e)) {
                Pieza pieza = controladorCircuito.
                ↪ getPiezaByPosicion(new Punto(e.getPoint())
                ↪ );
                if (pieza != null) {
```

```
/*String nombreOriginal = pieza.
    ↪ getNombrePieza() != null ? pieza.
    ↪ getNombrePieza() :
            ModelicaGenerator.nombrePieza(pieza)
                ↪ ;
String nuevoNombre =
            JOptionPane.showInputDialog(
                ↪ I18NUtils.getString("
                ↪ rename_component_prompt"),
                    nombreOriginal);
System.out.println(nuevoNombre);
if (nuevoNombre != null && !nuevoNombre.
    ↪ isEmpty() &&
            !nuevoNombre.equals(nombreOriginal))
                ↪ {
      controladorCircuito.renombrarPieza(pieza
          ↪ , nuevoNombre);
}*/
EditorPropiedadesPieza epp = new
    ↪ EditorPropiedadesPieza(pieza);
String[] opciones =
            {I18NUtils.getString("apply_changes"
                ↪ ), I18NUtils.getString("cancel
                ↪ ")};
int res = JOptionPane.showOptionDialog(null,
    ↪ epp,
            I18NUtils.getString("edit_component"
                ↪ ), JOptionPane.YES_NO_OPTION,
            JOptionPane.PLAIN_MESSAGE, null,
                ↪ opciones, opciones[0]);
if (res == 0) {//Aceptar cambios
```

```
                    epp.actualizarValorPropiedades();
                }
            }
        }
    }

    @Override
    public void mouseWheelMoved(MouseWheelEvent e) {
        System.out.println("Mouse wheel moved!");
        reescalar(e.getWheelRotation() * .1f);
    }

    private void reescalar(float dZ) {
        Punto.reescalar(dZ);
        for (Pieza p : controladorCircuito.getPiezas()) {
            p.setImagen(ImageUtils.rotar(
                    ImageUtils.cargarImagenEscalada(p.
                        ↪ getTipoPieza().getPathImagen(),
                            Punto.getEscala()), -p.
                                ↪ getRotacion().getAngulo())
                                ↪ );
        }
        repaint();
    }


    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        Map<Conector, Color> coloresConectores = null;
```

```
//Dibujar piezas
if (isModo(ModoPanel.MODO_CONEXION)) {
    coloresConectores = controladorCircuito.
        ↪ getAllConectores().stream().collect(
            Collectors.toMap(con -> con,
                    con -> controladorCircuito.
                        ↪ getConectoresValidos(
                        ↪ conectorSeleccionado)
                            .contains(con) ?
                                ↪ Conector.
                                ↪ colorConector :
                            Color.GRAY));
}
//for (Map.Entry<Pieza, Punto> entry :
    ↪ controladorCircuito.getPiezasPosicionEntrySet
    ↪ ()) {
for (Pieza p : controladorCircuito.getPiezas()) {
    p.dibujar(this, g, p.getPosicion(), false,
        ↪ coloresConectores);
}

//Dibujar conexiones
g.setColor(Color.BLACK);
Graphics2D g2 = (Graphics2D) g;
for (Conexion c : controladorCircuito.getConexiones
    ↪ ()) {
    List<Punto> puntos = new ArrayList<>(c.getPuntos
        ↪ ());
    if (isModo(ModoPanel.MODO_CONEXION) && c.enCurso
        ↪ () && getMousePosition() != null) {
        puntos.add(new Punto(getMousePosition()));
```

```
            }
            List<Punto> puntosManhattan = LineUtils.
                ↪ getPuntosManhattan(puntos);
            int[] x = puntosManhattan.stream().mapToInt(pt
                ↪ -> (int) pt.getX()).toArray();
            int[] y = puntosManhattan.stream().mapToInt(pt
                ↪ -> (int) pt.getY()).toArray();
            g2.setStroke(new BasicStroke(2));
            g2.drawPolyline(x, y, x.length);
        }
    }

    private void setModo(ModoPanel nuevoModo) {
        if (modo != null && isModo(ModoPanel.MODO_CONEXION))
            ↪ {
            //Al salir del modo conexion borramos las
                ↪ conexiones incompletas
            controladorCircuito.borrarConexionesIncompletas
                ↪ ();
        }
        this.modo = nuevoModo;
        repaint();
    }

    private boolean isModo(ModoPanel target) {
        return this.modo.equals(target);
    }

    //region m todos overriden que no queremos para nada
    @Override
    public void mouseEntered(MouseEvent e) {
```

```
    }

    @Override
    public void mouseExited(MouseEvent e) {
    }
    //endregion
}
```

## B.15. Fichero gui/SelectorCircuito.java

```
package caponera.uned.tfm.lizardclips.gui;

import caponera.uned.tfm.lizardclips.controlador.
    ↪ ControladorCircuito;
import caponera.uned.tfm.lizardclips.modelo.Circuito;
import caponera.uned.tfm.lizardclips.utils.I18NUtils;
import caponera.uned.tfm.lizardclips.utils.ImageUtils;
import lombok.Getter;

import javax.swing.BoxLayout;
import javax.swing.JComponent;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.border.MatteBorder;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.FlowLayout;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
import java.util.List;
```

```
public class SelectorCircuito extends JComponent implements
    ↪ ItemListener {
    private static final int IMAGEVIEW_W = 300, IMAGEVIEW_H
        ↪ = 300;
    ControladorCircuito controladorCircuito;
    List<Circuito> circuitos;
    java.awt.List listaCircuitos;
    JLabel imageView;

    @Getter
    private Circuito circuitoSeleccionado;

    public SelectorCircuito(ControladorCircuito
        ↪ controladorCircuito, List<Circuito> circuitos) {
        this.controladorCircuito = controladorCircuito;
        this.circuitos = circuitos;
        setLayout(new FlowLayout());
        //Lista circuitos
        JPanel contenedorListaCircuitos = new JPanel();
        contenedorListaCircuitos.setLayout(
                new BoxLayout(contenedorListaCircuitos,
                    ↪ BoxLayout.Y_AXIS));
        contenedorListaCircuitos.add(new JLabel(I18NUtils.
            ↪ getString("select_circuit_to_load_it")));
        listaCircuitos = new java.awt.List(circuitos.size(),
            ↪  false);
        for (Circuito c : circuitos) {
            listaCircuitos.add(c.getNombre());
        }
        listaCircuitos.addItemListener(this);
```

```
        JScrollPane scrollListaCircuitos = new JScrollPane(
            ↪ listaCircuitos);
        scrollListaCircuitos.setPreferredSize(new Dimension(
            ↪ IMAGEVIEW_W / 2, IMAGEVIEW_H));
        contenedorListaCircuitos.add(scrollListaCircuitos);
        this.add(contenedorListaCircuitos);
        JPanel contenedorImageView = new JPanel();
        contenedorImageView.setLayout(new BoxLayout(
            ↪ contenedorImageView, BoxLayout.Y_AXIS));
        contenedorImageView.add(new JLabel(I18NUtils.
            ↪ getString("circuit_preview")));
        imageView = new JLabel();
        imageView.setIcon(
                ImageUtils.cargarImagenEscalada(ImageUtils.
                    ↪ MEDIA_BASE_FOLDER + "/lizardclips.png"
                    ↪ ,
                            IMAGEVIEW_W, IMAGEVIEW_H));
        imageView.setBorder(new MatteBorder(1, 1, 1, 1,
            ↪ Color.BLACK));
        contenedorImageView.add(imageView);
        this.add(contenedorImageView);
    }


    @Override
    public void itemStateChanged(ItemEvent e) {
        circuitoSeleccionado = circuitos.get(listaCircuitos.
            ↪ getSelectedIndex());
        imageView.setIcon(ImageUtils.
            ↪ rescalarImagenPreserveRatio(
                ImageUtils.imageIconFromBytes(
```

```
                    ↪ circuitoSeleccionado.getThumbnail()),
                    ↪ IMAGEVIEW_W,
                IMAGEVIEW_H));
        repaint();
    }
}
```

## B.16. Fichero gui/VentanaPrincipal.java

```
package caponera.uned.tfm.lizardclips.gui;

import caponera.uned.tfm.lizardclips.constant.TabPaleta;
import caponera.uned.tfm.lizardclips.constant.TipoPieza;
import caponera.uned.tfm.lizardclips.controlador.
    ↪ ControladorCircuito;
import caponera.uned.tfm.lizardclips.utils.I18NUtils;
import caponera.uned.tfm.lizardclips.utils.ImageUtils;
import lombok.Getter;

import javax.swing.BoxLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JTabbedPane;
import javax.swing.JToggleButton;
import javax.swing.WindowConstants;
import javax.swing.border.MatteBorder;
```

```
import java.awt.BorderLayout;
import java.awt.Color;

public class VentanaPrincipal {
    private static final int ANCHO_BOTONES_LATERALES = 50;
    private static final int ALTO_BOTONES_LATERALES = 50;

    @Getter
    private JFrame frame;
    private ControladorCircuito controladorCircuito;
    private PanelCircuito panelCircuito;

    public VentanaPrincipal(int width, int height,
    ↪ ControladorCircuito controlador, PanelCircuito
    ↪ panelCircuito) {
        this.controladorCircuito = controlador;
        controladorCircuito.setVentanaPrincipal(this);
        this.panelCircuito = panelCircuito;
        setupLayout(width, height);

        Thread.setDefaultUncaughtExceptionHandler((Thread t,
        ↪  Throwable e) -> {
            JOptionPane.showMessageDialog(panelCircuito, e.
            ↪ getMessage(), "Error",
                    JOptionPane.WARNING_MESSAGE);
            e.printStackTrace();
        });
    }

    public void setNombreCircuito(String nombreCircuito) {
        frame.setTitle(I18NUtils.getString("app_name") + "-
```

```
        ↪ -" + nombreCircuito);
}

private void setupLayout(int width, int height) {
    //Frame
    frame = new JFrame();
    frame.setIconImage(
            ImageUtils.cargarImageIcon(ImageUtils.
                ↪ MEDIA_BASE_FOLDER + "/lizardclips.png"
                ↪ )
                        .getImage());
    frame.setSize(width, height);
    setNombreCircuito(I18NUtils.getString("
        ↪ untitled_circuit"));
    frame.setDefaultCloseOperation(WindowConstants.
        ↪ EXIT_ON_CLOSE);
    frame.setLayout(new BorderLayout());

    //Menu superior
    JMenuBar barraSuperior = new JMenuBar();
    //    Menu circuito
    JMenu menuCircuito = new JMenu(I18NUtils.getString("
        ↪ menu_circuito_title"));

    JMenuItem menuItemCircuitoNuevo =
            new JMenuItem(I18NUtils.getString("
                ↪ menuitem_new_circuit"));
    menuItemCircuitoNuevo.addActionListener(e ->
        ↪ controladorCircuito.nuevoCircuito());
    menuCircuito.add(menuItemCircuitoNuevo);
```

```
JMenuItem menuItemCircuitoCargar =
        new JMenuItem(I18NUtils.getString("
            ↪ menuitem_load_circuit"));
menuItemCircuitoCargar.addActionListener(e ->
    ↪ controladorCircuito.cargar());
menuCircuito.add(menuItemCircuitoCargar);

JMenuItem menuItemCircuitoGuardar =
        new JMenuItem(I18NUtils.getString("
            ↪ menuitem_save_circuit"));
menuItemCircuitoGuardar.addActionListener(e ->
    ↪ controladorCircuito.guardar());
menuCircuito.add(menuItemCircuitoGuardar);

JMenuItem menuItemCircuitoGuardarComo =
        new JMenuItem(I18NUtils.getString("
            ↪ menuitem_save_circuit_as"));
menuItemCircuitoGuardarComo.addActionListener(e ->
    ↪ controladorCircuito.guardar_como());
menuCircuito.add(menuItemCircuitoGuardarComo);

barraSuperior.add(menuCircuito);
//    Menu modelica
JMenu menuModelica = new JMenu(I18NUtils.getString("
    ↪ menu_modelica_title"));
JMenuItem menuItemModelicaExportar =
        new JMenuItem(I18NUtils.getString("
            ↪ menuitem_export_modelica"));
menuItemModelicaExportar.addActionListener(e ->
    ↪ controladorCircuito.exportarCodigo());
menuModelica.add(menuItemModelicaExportar);
```

```
JMenuItem menuItemModelicaVisualizar =
        new JMenuItem(I18NUtils.getString("
            ↪ menuitem_view_modelica"));
menuItemModelicaVisualizar.addActionListener(e ->
    ↪ controladorCircuito.verCodigo());
menuModelica.add(menuItemModelicaVisualizar);

barraSuperior.add(menuModelica);

//    Menu vista
JMenu menuVista = new JMenu(I18NUtils.getString("
    ↪ menu_view_title"));
JMenuItem menuItemVistaToggleNombres =
        new JMenuItem(I18NUtils.getString("
            ↪ menuitem_show_hide_piece_names"));
menuItemVistaToggleNombres.addActionListener(
        e -> controladorCircuito.toggleNombresPiezas
            ↪ ());
menuVista.add(menuItemVistaToggleNombres);

JMenuItem menuItemVistaToggleNombresPines =
        new JMenuItem(I18NUtils.getString("
            ↪ menuitem_show_hide_pin_names"));
menuItemVistaToggleNombresPines.addActionListener(
        e -> controladorCircuito.toggleNombresPines
            ↪ ());
menuVista.add(menuItemVistaToggleNombresPines);

barraSuperior.add(menuVista);
```

```
frame.setJMenuBar(barraSuperior);


//Panel
frame.add(panelCircuito, BorderLayout.CENTER);


//Sidebar
JPanel lateral = new JPanel();
lateral.setLayout(new BoxLayout(lateral, BoxLayout.
    ↪ Y_AXIS));
lateral.setBorder(new MatteBorder(0, 0, 0, 2, Color.
    ↪ GRAY));

JTabbedPane tabs = new JTabbedPane(JTabbedPane.LEFT,
    ↪  JTabbedPane.SCROLL_TAB_LAYOUT);
for (TabPaleta tab : TabPaleta.values()) {
    JPanel tabContent = new JPanel();
    tabContent.setLayout(new BoxLayout(tabContent,
        ↪ BoxLayout.Y_AXIS));
    for (TipoPieza tp : TipoPieza.values()) {
        if (tp.getTabPaleta().equals(tab)) {
            JButton b = new JButton("",
                    ImageUtils.
                        ↪ cargarImageneEscaladaPreserveRatio
                        ↪ (tp.getPathImagen(),
                            ANCHO_BOTONES_LATERALES,
                                ↪
                                ↪ ALTO_BOTONES_LATERALES
                                ↪ ));
            b.setToolTipText(tp.getNombre());
```

```
                    b.addActionListener(e ->
                        ↪ controladorCircuito.generarPieza(
                        ↪ tp));
                    tabContent.add(b);
                }
            }

            tabs.add(tab.getNombre(), tabContent);
        }

        lateral.add(tabs);

        JToggleButton borrar = new JToggleButton("",
            ↪ ImageUtils.cargarImageneEscaladaPreserveRatio(
                ImageUtils.MEDIA_BASE_FOLDER + "/trash.png",
                    ↪ ANCHO_BOTONES_LATERALES,
                ALTO_BOTONES_LATERALES));
        borrar.addActionListener(e -> panelCircuito.
            ↪ toggleDeleteMode());
        lateral.add(borrar);


        frame.add(lateral, BorderLayout.WEST);
    }

    public void mostrar() {
        frame.setVisible(true);
    }
}
```

## B.17. Fichero gui/VentanaVisualizarCodigo.java

```
package caponera.uned.tfm.lizardclips.gui;

import caponera.uned.tfm.lizardclips.utils.I18NUtils;
import caponera.uned.tfm.lizardclips.utils.ImageUtils;
import org.fife.ui.rsyntaxtextarea.AbstractTokenMakerFactory
    ↪ ;
import org.fife.ui.rsyntaxtextarea.RSyntaxTextArea;
import org.fife.ui.rsyntaxtextarea.TokenMakerFactory;

import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.WindowConstants;
import java.awt.BorderLayout;

public class VentanaVisualizarCodigo extends JFrame {
    public VentanaVisualizarCodigo(String codigo, int width,
        ↪ int height) {
        JPanel panel = new JPanel(new BorderLayout());
        setSize(width, height);
        setTitle(I18NUtils.getString("code_viewer"));
        setIconImage(ImageUtils.cargarImageIcon(ImageUtils.
            ↪ MEDIA_BASE_FOLDER + "/lizardclips.png")
                                .getImage());
        setDefaultCloseOperation(WindowConstants.
            ↪ DISPOSE_ON_CLOSE);

        AbstractTokenMakerFactory atmf =
                (AbstractTokenMakerFactory)
```

```
                    ↪ TokenMakerFactory.getDefaultInstance()
                    ↪ ;
        atmf.putMapping("text/modelica",
                "caponera.uned.tfm.lizardclips.modelica.
                    ↪ ModelicaTokenMaker");
        RSyntaxTextArea textArea = new RSyntaxTextArea(20,
            ↪ 60);

        textArea.setSyntaxEditingStyle("text/modelica");
        textArea.setEditable(false);
        textArea.setCodeFoldingEnabled(true);
        textArea.setText(codigo);
        panel.add(textArea);
        setContentPane(panel);
        pack();
        setLocationRelativeTo(null);

    }
}
```

## B.18. Fichero `modelica/ModelicaGenerator.java`

```
package caponera.uned.tfm.lizardclips.modelica;

import caponera.uned.tfm.lizardclips.constant.
    ↪ ConectorTemplate;
import caponera.uned.tfm.lizardclips.constant.TipoConector;
import caponera.uned.tfm.lizardclips.modelo.Circuito;
import caponera.uned.tfm.lizardclips.modelo.Conector;
import caponera.uned.tfm.lizardclips.modelo.Conexion;
import caponera.uned.tfm.lizardclips.modelo.Pieza;
```

```
import caponera.uned.tfm.lizardclips.modelo.Propiedad;
import caponera.uned.tfm.lizardclips.utils.Punto;

import java.util.ArrayList;
import java.util.List;
import java.util.Locale;
import java.util.StringJoiner;

public class ModelicaGenerator {
    public static final String BASIC = "B";
    public static final String LOGIC = "L";
    public static final String DIGITAL = "D";
    public static final String SOURCES = "S";
    public static final String GATES = "G";
    public static final String TRISTATES = "TS";
    public static final String MULTIPLEXERS = "MP";
    public static final String UTILITIES = "U";
    public static final String SI = "SI";
    private static final String DIGITAL_IMPORT = "Modelica.
        ↪ Electrical.Digital";
    private static final String BASIC_IMPORT = "Modelica.
        ↪ Electrical.Digital.Basic";
    private static final String LOGIC_IMPORT = "Modelica.
        ↪ Electrical.Digital.Interfaces.Logic";
    private static final String SOURCES_IMPORT = "Modelica.
        ↪ Electrical.Digital.Sources";
    private static final String GATES_IMPORT = "Modelica.
        ↪ Electrical.Digital.Gates";
    private static final String TRISTATES_IMPORT = "Modelica
        ↪ .Electrical.Digital.Tristates";
    private static final String MULTIPLEXERS_IMPORT = "
```

```
    ↪ Modelica.Electrical.Digital.Multiplexers";
private static final String UTILITIES_IMPORT = "Modelica
    ↪ .Electrical.Digital.Examples.Utilities";
private static final String SI_IMPORT = "Modelica.Units.
    ↪ SI";

private static final double ESCALA_ANNOTATION = 0.3;

public static String generarCodigoModelica(Circuito
    ↪ circuito) {
    StringBuilder codigoModelica = new StringBuilder();
    codigoModelica.append("model ").append(modelName(
        ↪ circuito)).append("\n\t");
    codigoModelica.append(imports()).append("\n\t");
    codigoModelica.append(declaraciones(circuito));
    codigoModelica.append("equation\n\t").append(connect
        ↪ (circuito)).append("\n\t");
    codigoModelica.append(generarAnotacionesConexiones(
        ↪ circuito)).append(";\n");
    codigoModelica.append("end ").append(modelName(
        ↪ circuito)).append(";");

    return codigoModelica.toString();
}

private static String imports() {
    StringJoiner sj = new StringJoiner(";\n\t");
    sj.add("import " + DIGITAL + " = " + DIGITAL_IMPORT)
        ↪ ;
    sj.add("import " + BASIC + " = " + BASIC_IMPORT);
    sj.add("import " + LOGIC + " = " + LOGIC_IMPORT);
```

```
        sj.add("import " + SOURCES + " = " + SOURCES_IMPORT)
            ↪ ;
        sj.add("import " + GATES + " = " + GATES_IMPORT);
        sj.add("import " + TRISTATES + " = " +
            ↪ TRISTATES_IMPORT);
        sj.add("import " + MULTIPLEXERS + " = " +
            ↪ MULTIPLEXERS_IMPORT);
        sj.add("import " + UTILITIES + " = " +
            ↪ UTILITIES_IMPORT);

        sj.add("import " + SI + " = " + SI_IMPORT);
        sj.add("\n");
        return sj.toString();
    }

    private static String declaraciones(Circuito circuito) {
        StringJoiner sj = new StringJoiner(";\n\t");
        for (Pieza p : circuito.getComponentes()) {
            List<String> lineasDeclaracion =
                ↪ generarDeclaracionPieza(p);
            for (String s : lineasDeclaracion) {
                sj.add(s);
            }
        }

        sj.add("\n");
        return sj.toString();
    }

    private static String generarAnotacionesConexiones(
        ↪ Circuito circuito) {
```

```
        StringJoiner sj = new StringJoiner(",\n\t\t");
        for (Conexion c : circuito.getConexiones()) {
            sj.add(generarAnotacion(c));
        }

        return String.format("annotation (Diagram(graphics
            ↪ ={%s}))", sj);
    }

    private static String generarAnotacion(Conexion c) {
        StringJoiner sj = new StringJoiner(",");
        c.getPuntosManhattan().stream().map(p ->
            ↪ escalarPunto(p, true))
         .map(p -> String.format("{%d,%d}", p.getX(), p.getY
            ↪ ())).forEachOrdered(sj::add);
        return String.format("Line(points={%s}, color
            ↪ ={0,0,0})", sj);
    }

    private static Punto escalarPunto(Punto punto, boolean
        ↪ invertirY) {
        int multiplicadorY = invertirY ? -1 : 1;
        return new Punto((int) (punto.getX() *
            ↪ ESCALA_ANNOTATION),
                (int) (punto.getY() * ESCALA_ANNOTATION *
                    ↪ multiplicadorY));
    }

    private static String generarAnotacion(Pieza p) {
        Punto pos = p.getPosicion();
        Punto bottom_left = new Punto(pos.getX(), pos.getY()
```

```
    ↪  + p.getHeight());
Punto top_right = new Punto(pos.getX() + p.getWidth
    ↪ (), pos.getY());
Punto bottom_left_scaled = escalarPunto(bottom_left,
    ↪  true);
Punto top_right_scaled = escalarPunto(top_right,
    ↪ true);
double offset_x = 0, offset_y = 0;

switch (p.getRotacion()) {
    case ROT_0 -> {

    }
    case ROT_90 -> {
        offset_x = -p.getHeight() *
            ↪ ESCALA_ANNOTATION;
        //offset_x = -p.getWidth() *
            ↪ ESCALA_ANNOTATION;
    }
    case ROT_180 -> {
        offset_y = -p.getHeight() *
            ↪ ESCALA_ANNOTATION;
        offset_x = -p.getWidth() * ESCALA_ANNOTATION
            ↪ ;
    }
    case ROT_270 -> {
        offset_y = -p.getHeight() *
            ↪ ESCALA_ANNOTATION;
    }
}
return String.format(
```

```
                "annotation(Placement(transformation(extent
                    ↪ ={{%d,%d},{%d,%d}},rotation=%d,
                    ↪ origin={%d,%d})))",
                (int) (0 + offset_x), //bottom left x
                (int) (-p.getHeight() * ESCALA_ANNOTATION -
                    ↪ offset_y), //bottom left y
                (int) (p.getWidth() * ESCALA_ANNOTATION +
                    ↪ offset_x), //top right x
                (int) (0 - offset_y), //top right y
                p.getRotacion().getAngulo(), //rotation
                (int) (pos.getX() * ESCALA_ANNOTATION),//
                    ↪ origin x
                (int) (-pos.getY() * ESCALA_ANNOTATION));//
                    ↪ origin y
    }

    private static String nombrePropiedad(Pieza p, Propiedad
        ↪  prop) {
        return nombrePieza(p) + "_" + prop.getNombre();
    }

    private static String generarAsignacionParametrosPieza(
        ↪ Pieza p) {
        List<ConectorTemplate> conectoresMultiples =
                p.getTipoPieza().getConectoresConNombre().
                    ↪ stream()
                 .filter(ConectorTemplate::isMultiple).
                     ↪ toList();

        List<Propiedad> propiedadesPieza = p.getTipoPieza().
            ↪ getPropiedades();
```

```
String res = "";
if (propiedadesPieza.size() > 0 ||
    ↪ conectoresMultiples.size() > 0) {
    StringJoiner sj = new StringJoiner(",");
    for (int i = 0; i < propiedadesPieza.size(); i
        ↪ ++) {
        Propiedad prop = propiedadesPieza.get(i);
        String nombreProp = prop.getNombre();
        String nombrePropiedad = nombrePropiedad(p,
            ↪ prop);
        if (nombreProp.endsWith("[:]")) {
            nombreProp = nombreProp.substring(0,
                ↪ nombreProp.length() - 3);
            nombrePropiedad = nombrePropiedad.
                ↪ substring(0, nombrePropiedad.
                ↪ length() - 3);
        }
        sj.add(nombreProp + "=" + nombrePropiedad)
            ↪ ;
    }

    for (int i = 0; i < conectoresMultiples.size();
        ↪ i++) {
        String nombreAtributoNumPines =
            ↪ conectoresMultiples.get(i).
            ↪ getNombreNumPines();
        int numPines = p.getNPinesConectorMultiple(i
            ↪ );
        sj.add(nombreAtributoNumPines + "=" +
            ↪ numPines);
    }
```

```
            res = "(" + sj + ")";
        }
        return res;
    }

    private static List<String> generarDeclaracionPieza(
        ↪ Pieza p) {
        List<String> declaracion = new ArrayList<>();
        List<Propiedad> propiedadesPieza = p.getTipoPieza().
            ↪ getPropiedades();
        for (int i = 0; i < propiedadesPieza.size(); i++) {
            Propiedad prop = propiedadesPieza.get(i);
            declaracion.add(String.format("parameter␣%s␣%s␣=
                ↪ ␣%s", prop.getUnidad(),
                    nombrePropiedad(p, prop), p.
                        ↪ getValoresPropiedades()[i]));
        }

        declaracion.add(String.format("%s␣%s␣%s␣\n\t\t%s", p
            ↪ .getTipoPieza().getClaseModelica(),
                nombrePieza(p),
                    ↪ generarAsignacionParametrosPieza(p),
                    ↪ generarAnotacion(p)));
        return declaracion;
    }

    private static String nombreFuente(Pieza p) {
        return "src_" + nombrePieza(p);
    }
```

```
private static long nConectoresEntrada(Pieza p) {
    return p.getConectores().stream()
            .filter(c -> c.getTipoConector().equals(
              ↪ TipoConector.ENTRADA)).count();
}

public static String nombrePieza(Pieza p) {
    if (p.getNombrePieza() != null) {
        return p.getNombrePieza();
    }
    String clase = p.getTipoPieza().getClaseModelica().
      ↪ split("\\.")[1];
    int posicion = p.getCircuito().getComponentes().
      ↪ indexOf(p) + 1;
    return clase.toLowerCase(Locale.ROOT) + "_" +
      ↪ posicion;
}

private static String connect(Circuito circuito) {
    StringJoiner sj = new StringJoiner(";\n\t");
    for (Conexion c : circuito.getConexiones()) {
        Conector salida =
                c.getOrigen().getTipoConector().equals(
                  ↪ TipoConector.SALIDA) ? c.getOrigen
                  ↪ () :
                      c.getDestino();
        Conector entrada =
                c.getOrigen().getTipoConector().equals(
                  ↪ TipoConector.ENTRADA) ? c.
                  ↪ getOrigen() :
                      c.getDestino();
```

```
                sj.add(String.format("connect(%s,%s)",
                    ↪ nombreConector(salida),
                        nombreConector(entrada)));
            }
            sj.add("\n");
            return sj.toString();
        }

        private static String nombreConector(Conector conector)
            ↪ {
            return nombrePieza(conector.getPieza()) + "." +
                ↪ conector.getNombreConector();
        }

        private static String modelName(Circuito circuito) {
            String modelName = "CircuitoAutogenerado";
            if (circuito.getNombre() != null && !circuito.
                ↪ getNombre().isBlank()) {
                modelName = circuito.getNombre();
            }

            return modelName;
        }
    }
```

## B.19. Fichero `modelica/ModelicaTokenMaker.java`

```
package caponera.uned.tfm.lizardclips.modelica;

import org.fife.ui.rsyntaxtextarea.AbstractTokenMaker;
```

```
import org.fife.ui.rsyntaxtextarea.RSyntaxUtilities;
import org.fife.ui.rsyntaxtextarea.Token;
import org.fife.ui.rsyntaxtextarea.TokenMap;
import org.fife.ui.rsyntaxtextarea.TokenTypes;

import javax.swing.text.Segment;
import java.util.List;

public class ModelicaTokenMaker extends AbstractTokenMaker {
    public static final List<String> reserved_words =
            List.of("equation", "annotation", "extends", "
               ↪ encapsulated", "flow", "for", "if",
                    "import", "input", "output", "partial",
                       ↪ "stream", "time", "when", "while",
                    "block", "class", "connector", "function
                       ↪ ", "model", "package", "record", "
                       ↪ type",
                    "end", "else", "parameter");
    public static final List<String> operators =
            List.of("abs", "acos", "actualStream", "array",
               ↪ "asin", "assert", "atan", "atan2",
                    "cardinality", "cat", "ceil", "change",
                       ↪ "connect", "Connections.root",
                    "Connections.branch", "Connections.
                       ↪ potentialRoot", "Connections.
                       ↪ isRoot",
                    "Connections.rooted", "cos", "cosh", "
                       ↪ cross", "delay", "der", "diagonal"
                       ↪ , "div",
                    "edge", "exp", "fill", "floor", "
                       ↪ homotopy", "identity", "initial",
```

```
            ↪ "inStream",
        "Integer", "integer", "linspace", "log",
            ↪  "log10", "matrix", "max", "min",
            ↪ "mod",
        "ndims", "noEvent", "ones", "
            ↪ outerProduct", "pre", "product", "
            ↪ reinit", "rem",
        "rooted", "sample", "scalar", "
            ↪ semiLinear", "sign", "sin", "sinh"
            ↪ , "size",
        "skew", "smooth", "sqrt", "String", "sum
            ↪ ", "symmetric", "tan", "tanh",
        "terminal", "terminate", "transpose", "
            ↪ vector", "zeros");

@Override
public TokenMap getWordsToHighlight() {
    TokenMap tm = new TokenMap();
    reserved_words.forEach(word -> tm.put(word,
        ↪ TokenTypes.RESERVED_WORD));
    operators.forEach(op -> tm.put(op, TokenTypes.
        ↪ FUNCTION));
    tm.put("true", TokenTypes.LITERAL_BOOLEAN);
    tm.put("false", TokenTypes.LITERAL_BOOLEAN);
    return tm;
}

@Override
public void addToken(Segment segment, int start, int end
    ↪ , int tokenType, int startOffset) {
    if (tokenType == TokenTypes.IDENTIFIER) {
```

```
            int value = wordsToHighlight.get(segment, start,
               ↪ end);
            if (value != -1) {
                tokenType = value;
            }
        }

        super.addToken(segment, start, end, tokenType,
           ↪ startOffset);
    }

    /**
     * Returns a list of tokens representing the given text.
     *
     * @param text          The text to break into tokens.
     * @param startTokenType The token with which to start
        ↪ tokenizing.
     * @param startOffset    The offset at which the line of
        ↪  tokens begins.
     * @return A linked list of tokens representing <code>
        ↪ text</code>.
     */
    public Token getTokenList(Segment text, int
       ↪ startTokenType, int startOffset) {

        resetTokenList();

        char[] array = text.array;
        int offset = text.offset;
        int count = text.count;
        int end = offset + count;
```

```
// Token starting offsets are always of the form:
// 'startOffset + (currentTokenStart-offset)', but
   ↪ since startOffset and
// offset are constant, tokens' starting positions
   ↪ become:
// 'newStartOffset+currentTokenStart'.
int newStartOffset = startOffset - offset;

int currentTokenStart = offset;
int currentTokenType = startTokenType;

for (int i = offset; i < end; i++) {

    char c = array[i];

    switch (currentTokenType) {

        case Token.NULL:

            currentTokenStart = i;    // Starting a
               ↪ new token here.

            switch (c) {

                case ' ':
                case '\t':
                    currentTokenType = Token.
                       ↪ WHITESPACE;
                    break;
```

```
case '"':
    currentTokenType = Token.
    ↪ LITERAL_STRING_DOUBLE_QUOTE
    ↪ ;
    break;

case '#':
    currentTokenType = Token.
    ↪ COMMENT_EOL;
    break;

default:
    if (RSyntaxUtilities.isDigit(c))
    ↪  {
      currentTokenType = Token.
      ↪ LITERAL_NUMBER_DECIMAL_INT
      ↪ ;
      break;
    } else if (RSyntaxUtilities.
    ↪ isLetter(c) || c == '/' ||
    ↪  c == '_') {
      currentTokenType = Token.
      ↪ IDENTIFIER;
      break;
    }

    // Anything not currently
    ↪ handled - mark as an
    ↪ identifier
    currentTokenType = Token.
    ↪ IDENTIFIER;
```

```
                    break;

                } // End of switch (c).

                break;

            case Token.WHITESPACE:

                switch (c) {

                    case '\r':
                    case '\t':
                        break;    // Still whitespace.

                    case '"':
                        addToken(text, currentTokenStart
                            ↪ , i - 1, Token.WHITESPACE,
                                newStartOffset +
                                    ↪ currentTokenStart)
                                    ↪ ;
                        currentTokenStart = i;
                        currentTokenType = Token.
                            ↪ LITERAL_STRING_DOUBLE_QUOTE
                            ↪ ;
                        break;

                    case '#':
                        addToken(text, currentTokenStart
                            ↪ , i - 1, Token.WHITESPACE,
                                newStartOffset +
                                    ↪ currentTokenStart)
```

```
                    ↪ ;
            currentTokenStart = i;
            currentTokenType = Token.
               ↪ COMMENT_EOL;
            break;

        default:    // Add the whitespace
           ↪ token and start anew.

            addToken(text, currentTokenStart
               ↪ , i - 1, Token.WHITESPACE,
                     newStartOffset +
                        ↪ currentTokenStart)
                        ↪ ;
            currentTokenStart = i;

            if (RSyntaxUtilities.isDigit(c))
               ↪ {
               currentTokenType = Token.
                  ↪ LITERAL_NUMBER_DECIMAL_INT
                  ↪ ;
               break;
            } else if (RSyntaxUtilities.
               ↪ isLetter(c) || c == '/' ||
               ↪  c == '_') {
               currentTokenType = Token.
                  ↪ IDENTIFIER;
               break;
            }

            // Anything not currently
```

```
                    ↪ handled - mark as
                    ↪ identifier
                currentTokenType = Token.
                    ↪ IDENTIFIER;

        } // End of switch (c).

        break;

    default: // Should never happen
    case Token.IDENTIFIER:

        switch (c) {

            case '-':
            case '\t':
                addToken(text, currentTokenStart
                    ↪ , i - 1, Token.IDENTIFIER,
                        newStartOffset +
                            ↪ currentTokenStart)
                            ↪ ;
                currentTokenStart = i;
                currentTokenType = Token.
                    ↪ WHITESPACE;
                break;

            case '"':
                addToken(text, currentTokenStart
                    ↪ , i - 1, Token.IDENTIFIER,
                        newStartOffset +
                            ↪ currentTokenStart)
```

```
                        ↪ ;
                currentTokenStart = i;
                currentTokenType = Token.
                    ↪ LITERAL_STRING_DOUBLE_QUOTE
                    ↪ ;
                break;

            default:
                if (RSyntaxUtilities.
                    ↪ isLetterOrDigit(c) || c ==
                    ↪  '/' || c == '_') {
                  break;    // Still an
                      ↪ identifier of some
                      ↪ type.
                }
                // Otherwise, we're still an
                    ↪ identifier (?).

        } // End of switch (c).

        break;

    case Token.LITERAL_NUMBER_DECIMAL_INT:

        switch (c) {

            case '-':
            case '\t':
                addToken(text, currentTokenStart
                    ↪ , i - 1,
                        Token.
```

```
                        ↪ LITERAL_NUMBER_DECIMAL_INT
                        ↪ ,
                    newStartOffset +
                        ↪ currentTokenStart)
                        ↪ ;
            currentTokenStart = i;
            currentTokenType = Token.
                ↪ WHITESPACE;
            break;

        case '"':
            addToken(text, currentTokenStart
                ↪ , i - 1,
                    Token.
                        ↪ LITERAL_NUMBER_DECIMAL_INT
                        ↪ ,
                    newStartOffset +
                        ↪ currentTokenStart)
                        ↪ ;
            currentTokenStart = i;
            currentTokenType = Token.
                ↪ LITERAL_STRING_DOUBLE_QUOTE
                ↪ ;
            break;

        default:

            if (RSyntaxUtilities.isDigit(c))
                ↪ {
                break;    // Still a literal
                    ↪ number.
```

```
                }

                // Otherwise, remember this was
                    ↪ a number and start over.
                addToken(text, currentTokenStart
                    ↪ , i - 1,
                        Token.
                            ↪ LITERAL_NUMBER_DECIMAL_INT
                            ↪ ,
                        newStartOffset +
                            ↪ currentTokenStart)
                            ↪ ;
                i--;
                currentTokenType = Token.NULL;

            } // End of switch (c).

            break;

        case Token.COMMENT_EOL:
            i = end - 1;
            addToken(text, currentTokenStart, i,
                ↪ currentTokenType,
                    newStartOffset +
                        ↪ currentTokenStart);
            // We need to set token type to null so
                ↪ at the bottom we don't add one
                ↪ more token.
            currentTokenType = Token.NULL;
            break;
```

```
                case Token.LITERAL_STRING_DOUBLE_QUOTE:
                    if (c == '"') {
                        addToken(text, currentTokenStart, i,
                            ↪ Token.
                            ↪ LITERAL_STRING_DOUBLE_QUOTE,
                                newStartOffset +
                                    ↪ currentTokenStart);
                        currentTokenType = Token.NULL;
                    }
                    break;

            } // End of switch (currentTokenType).

        } // End of for (int i=offset; i<end; i++).

        switch (currentTokenType) {

            // Remember what token type to begin the next
                ↪ line with.
            case Token.LITERAL_STRING_DOUBLE_QUOTE:
                addToken(text, currentTokenStart, end - 1,
                    ↪ currentTokenType,
                        newStartOffset + currentTokenStart);
                break;

            // Do nothing if everything was okay.
            case Token.NULL:
                addNullToken();
                break;

            // All other token types don't continue to the
```

```
                ↪ next line...
            default:
                addToken(text, currentTokenStart, end - 1,
                    ↪ currentTokenType,
                            newStartOffset + currentTokenStart);
                addNullToken();

        }

        // Return the first token in our linked list.
        return firstToken;

    }
}
```

## B.20. Fichero `modelo/Circuito.java`

```
package caponera.uned.tfm.lizardclips.modelo;

import caponera.uned.tfm.lizardclips.controlador.
    ↪ ControladorCircuito;
import caponera.uned.tfm.lizardclips.utils.Punto;
import jakarta.persistence.Basic;
import jakarta.persistence.CascadeType;
import jakarta.persistence.Entity;
import jakarta.persistence.FetchType;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.Lob;
import jakarta.persistence.OneToMany;
```

```java
import jakarta.persistence.Transient;
import lombok.Getter;
import lombok.Setter;
import lombok.ToString;
import org.hibernate.Hibernate;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;
import java.util.Objects;
import java.util.Optional;

@Getter
@Setter
@ToString
@Entity
public class Circuito implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer idCircuito;

    /*@Transient
    private Map<Pieza, Punto> componentes;*/

    @Transient
    private ControladorCircuito controlador;


    @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.
        ↪ EAGER)
    private List<Conexion> conexiones;
```

```
@OneToMany(cascade = CascadeType.ALL, fetch = FetchType.
    ↪ EAGER, mappedBy = "circuito")
private List<Pieza> componentes;

@Lob
@Basic(fetch = FetchType.EAGER)
@ToString.Exclude
private byte[] thumbnail;

private String nombre = "";

public Circuito() {
    componentes = new ArrayList<>();
    conexiones = new ArrayList<>();
}

public Circuito(Circuito otro) {
    componentes = new ArrayList<>(
            otro.getComponentes().stream().map(comp ->
                ↪ new Pieza(this, comp)).toList());
    conexiones = new ArrayList<>(
            otro.getConexiones().stream().map(con -> new
                ↪  Conexion(con, componentes)).toList())
                ↪ ;
    controlador = otro.getControlador();
}

public byte[] getThumbnail() {
    return thumbnail;
}
```

```
public void moverPieza(Pieza pieza, Punto posicion) {
    pieza.setPosicion(posicion);
}

public void borrarPieza(Pieza pieza) {
    borrarConexionesPieza(pieza);
    componentes.remove(pieza);
}

public void borrarConexionesPieza(Pieza pieza) {
    conexiones.removeIf(con -> (con.getDestino().
       ↪ getPieza().equals(pieza) ||
            con.getOrigen().getPieza().equals(pieza)));
}

public void addConexion(Conexion c) {
    this.conexiones.add(c);
}

public void borrarConexion(Conexion c) {
    this.conexiones.remove(c);
}

@Override
public boolean equals(Object o) {
    if (this == o)
        return true;
    if (o == null || Hibernate.getClass(this) !=
       ↪ Hibernate.getClass(o))
        return false;
```

```
        Circuito circuito = (Circuito) o;
        return idCircuito != null && Objects.equals(
            ↪ idCircuito, circuito.idCircuito);
    }

    @Override
    public int hashCode() {
        return getClass().hashCode();
    }

    public void colocarPieza(Pieza pieza, Punto posicion) {
        componentes.add(pieza);
        pieza.setPosicion(posicion);
    }

    public void borrarConexionesConector(Conector c) {
        conexiones.removeIf(con -> con.getOrigen().equals(c)
            ↪ || con.getDestino().equals(c));
    }

    public void cancelarConexion() {
        Optional<Conexion> enCurso = conexiones.stream().
            ↪ filter(Conexion::enCurso).findFirst();
        enCurso.ifPresent(this::borrarConexion);
    }
}
```

## B.21. Fichero `modelo/Conector.java`

```
package caponera.uned.tfm.lizardclips.modelo;
```

```
import caponera.uned.tfm.lizardclips.constant.
    ↪ ConectorTemplate;
import caponera.uned.tfm.lizardclips.constant.TipoConector;
import caponera.uned.tfm.lizardclips.utils.Punto;
import jakarta.persistence.CascadeType;
import jakarta.persistence.Entity;
import jakarta.persistence.EnumType;
import jakarta.persistence.Enumerated;
import jakarta.persistence.FetchType;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.ManyToOne;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;
import lombok.ToString;
import org.hibernate.Hibernate;

import java.awt.Color;
import java.io.Serializable;
import java.util.Objects;

@Getter
@Setter
@ToString
@Entity
@NoArgsConstructor
public class Conector implements Serializable {
    private static final int RADIO = 3;
```

```
public static final Color colorConector = Color.BLACK;
    ↪ //new Color(119, 0, 200);

@ToString.Exclude
private double posicionRelativaX;

@ToString.Exclude
private double posicionRelativaY;

@Enumerated(EnumType.STRING)
private TipoConector tipoConector;

@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Integer idConector;

@ManyToOne(cascade = CascadeType.ALL, fetch = FetchType.
    ↪ EAGER)
@JoinColumn(name = "id_pieza")
@ToString.Exclude
@Setter
private Pieza pieza;

private String nombreConector;

private boolean reposicionar = true;

public Conector(double posicionRelativaX, double
    ↪ posicionRelativaY, TipoConector tipoConector,
    ↪ String nombre) {
     this.posicionRelativaX = posicionRelativaX;
```

```
        this.posicionRelativaY = posicionRelativaY;
        this.tipoConector = tipoConector;
        this.nombreConector = nombre;
    }

    public Conector(ConectorTemplate template, int index) {
        this(template.getRelativeX(), template.getRelativeY
            ↪ (), template.getTipo(),
                template.getNombre() + "[" + index + "]");
        this.reposicionar = template.isReposicionar();
    }

    public Conector(ConectorTemplate template) {
        this(template.getRelativeX(), template.getRelativeY
            ↪ (), template.getTipo(),
                template.getNombre());
        this.reposicionar = template.isReposicionar();

    }

    public static int getRadio() {
        return (int) (RADIO * (1 + Math.max((Punto.getEscala
            ↪ () - 1), 0)));//se agranda pero no se reduce
    }

    public Punto getPosicionEnPanel() {
        return pieza.getPosicionConectorEnPanel(this);
    }

    @Override public boolean equals(Object o) {
        if (this == o) return true;
```

```
        if (o == null || getClass() != o.getClass()) return
            ↪ false;
        Conector conector = (Conector) o;
        return getTipoConector() == conector.getTipoConector
            ↪ () &&
                Objects.equals(getPieza(), conector.getPieza
                    ↪ ()) &&
                Objects.equals(getNombreConector(), conector
                    ↪ .getNombreConector());
    }

    @Override public int hashCode() {
        return Objects.hash(getTipoConector(), getPieza(),
            ↪ getNombreConector());
    }
}
```

## B.22. Fichero `modelo/Conexion.java`

```
package caponera.uned.tfm.lizardclips.modelo;

import caponera.uned.tfm.lizardclips.utils.LineUtils;
import caponera.uned.tfm.lizardclips.utils.Punto;
import jakarta.persistence.CascadeType;
import jakarta.persistence.Entity;
import jakarta.persistence.FetchType;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.ManyToOne;
```

```
import jakarta.persistence.PostLoad;
import jakarta.persistence.PrePersist;
import jakarta.persistence.PreUpdate;
import jakarta.persistence.Transient;
import lombok.Getter;
import lombok.Setter;
import lombok.ToString;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

@Getter
@Setter
@Entity
public class Conexion implements Serializable {
    @ManyToOne(cascade = CascadeType.ALL, fetch = FetchType.
        ↪ EAGER)
    @JoinColumn(name = "id_conector_origen")
    private Conector origen;
    @ManyToOne(cascade = CascadeType.ALL, fetch = FetchType.
        ↪ EAGER)
    @JoinColumn(name = "id_conector_destino")
    private Conector destino;
    @Transient
    @ToString.Exclude
    private List<Punto> puntosIntermedios;

    @ToString.Exclude
    private List<Integer> puntosIntermediosX;
    @ToString.Exclude
```

```
private List<Integer> puntosIntermediosY;
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Integer idConexion;


public Conexion(Conector origen) {
    this.origen = origen;
    this.puntosIntermedios = new ArrayList<>();
}

public Conexion(Conexion otra, List<Pieza> componentes)
   ↪ {
    this.puntosIntermedios =
            new ArrayList<>(otra.puntosIntermedios.
               ↪ stream().map(Punto::new).toList());
    this.puntosIntermediosX = new ArrayList<>();
    this.puntosIntermediosY = new ArrayList<>();

    Pieza piezaOrigen = componentes.get(
            otra.getOrigen().getPieza().getCircuito().
               ↪ getComponentes()
                  .indexOf(otra.getOrigen().getPieza()
                     ↪ ));
    Pieza piezaDestino = componentes.get(
            otra.getDestino().getPieza().getCircuito().
               ↪ getComponentes()
                  .indexOf(otra.getDestino().getPieza
                     ↪ ()));
    this.origen = piezaOrigen.getConectores().get(
       ↪ piezaOrigen.getConectores().indexOf(otra.
```

```
        ↪ getOrigen()));
    this.destino = piezaDestino.getConectores().get(
        ↪ piezaDestino.getConectores().indexOf(otra.
        ↪ getDestino()));
}

public Conexion() {
    this.puntosIntermedios = new ArrayList<>();
    puntosIntermediosX = new ArrayList<>();
    puntosIntermediosY = new ArrayList<>();
}

@PostLoad
protected void postLoad() {
    //Cargar los puntos intermedios
    if (puntosIntermediosX != null && puntosIntermediosY
        ↪ != null) {
        for (int i = 0; i < puntosIntermediosX.size(); i
            ↪ ++) {
            puntosIntermedios.add(Punto.
                ↪ puntoCoordenadasVirtuales(
                ↪ puntosIntermediosX.get(i),
                    puntosIntermediosY.get(i)));
        }
    }
}

@PreUpdate
@PrePersist
protected void preUpdatePersist() {
    this.puntosIntermediosX = puntosIntermedios.stream()
```

```
        ↪ .map(Punto::getXVirtual).toList();
    this.puntosIntermediosY = puntosIntermedios.stream()
        ↪ .map(Punto::getYVirtual).toList();
}

public List<Integer> getPuntosIntermediosX() {
    return puntosIntermedios.stream().map(Punto::getX).
        ↪ toList();
}

public List<Integer> getPuntosIntermediosY() {
    return puntosIntermedios.stream().map(Punto::getY).
        ↪ toList();
}

public void addPoint(Punto punto) {
    puntosIntermedios.add(punto);
}

public void cerrar(Conector destino) {
    this.destino = destino;
}

public List<Punto> getPuntos() {
    List<Punto> puntos = new ArrayList<>(List.of(origen.
        ↪ getPosicionEnPanel()));
    puntos.addAll(puntosIntermedios);
    if (isComplete()) {
        puntos.add(destino.getPosicionEnPanel());
    }
    return puntos;
```

```
}

public List<Punto> getPuntosManhattan() {
    return LineUtils.getPuntosManhattan(getPuntos());
}

public boolean isComplete() {
    return destino != null;
}

public boolean enCurso() {
    return !isComplete();
}

public String toString() {
    return getPuntos().toString();
}

@Override
public boolean equals(Object o) {
    if (this == o)
        return true;
    if (o == null || getClass() != o.getClass())
        return false;

    Conexion conexion = (Conexion) o;

    if (origen != null ? !origen.equals(conexion.origen)
      ↪ : conexion.origen != null)
        return false;
    if (destino != null ? !destino.equals(conexion.
```

```
            ↪ destino) : conexion.destino != null)
                return false;
        if (puntosIntermedios != null ? !puntosIntermedios.
            ↪ equals(conexion.puntosIntermedios) :
                    conexion.puntosIntermedios != null)
                return false;
        if (puntosIntermediosX != null ? !puntosIntermediosX
            ↪ .equals(conexion.puntosIntermediosX) :
                    conexion.puntosIntermediosX != null)
                return false;
        if (puntosIntermediosY != null ? !puntosIntermediosY
            ↪ .equals(conexion.puntosIntermediosY) :
                    conexion.puntosIntermediosY != null)
                return false;
        return idConexion != null ? idConexion.equals(
            ↪ conexion.idConexion) :
                    conexion.idConexion == null;
    }

    @Override
    public int hashCode() {
        int result = origen != null ? origen.hashCode() : 0;
        result = 31 * result + (destino != null ? destino.
            ↪ hashCode() : 0);
        result = 31 * result + (puntosIntermedios != null ?
            ↪ puntosIntermedios.hashCode() : 0);
        result = 31 * result + (puntosIntermediosX != null ?
            ↪  puntosIntermediosX.hashCode() : 0);
        result = 31 * result + (puntosIntermediosY != null ?
            ↪  puntosIntermediosY.hashCode() : 0);
        result = 31 * result + (idConexion != null ?
```

```
            ↪ idConexion.hashCode() : 0);
        return result;
    }
}
```

## B.23. Fichero `modelo/Pieza.java`

```
package caponera.uned.tfm.lizardclips.modelo;

import caponera.uned.tfm.lizardclips.constant.
    ↪ ConectorTemplate;
import caponera.uned.tfm.lizardclips.constant.TipoConector;
import caponera.uned.tfm.lizardclips.constant.TipoPieza;
import caponera.uned.tfm.lizardclips.gui.PanelCircuito;
import caponera.uned.tfm.lizardclips.modelica.
    ↪ ModelicaGenerator;
import caponera.uned.tfm.lizardclips.utils.AnguloRotacion;
import caponera.uned.tfm.lizardclips.utils.ImageUtils;
import caponera.uned.tfm.lizardclips.utils.Punto;
import jakarta.persistence.CascadeType;
import jakarta.persistence.Entity;
import jakarta.persistence.EnumType;
import jakarta.persistence.Enumerated;
import jakarta.persistence.FetchType;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.ManyToOne;
import jakarta.persistence.OneToMany;
import jakarta.persistence.PostLoad;
```

```
import jakarta.persistence.Transient;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;
import lombok.ToString;

import javax.swing.ImageIcon;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.Graphics;
import java.awt.Rectangle;
import java.io.Serializable;
import java.util.*;

@Getter
@Setter
@ToString
@Entity
@NoArgsConstructor
public class Pieza implements Serializable {
    @Transient
    @Getter
    @Setter
    private static boolean renerNombresPiezas = true;

    @Transient
    @Getter
    @Setter
    private static boolean renderNombresPines = true;
```

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Integer idPieza;

private String nombrePieza;

private Punto posicion;

@Enumerated(EnumType.ORDINAL)
private AnguloRotacion rotacion = AnguloRotacion.ROT_0;

@Transient
@ToString.Exclude
private ImageIcon imagen;

@OneToMany(mappedBy = "pieza", cascade = CascadeType.ALL
    ↪ , fetch = FetchType.EAGER)
private List<Conector> conectores;

private Dimension tamano;

@ManyToOne(fetch = FetchType.EAGER)
@JoinColumn(name = "id_circuito")
@ToString.Exclude
private Circuito circuito;

@Enumerated(EnumType.ORDINAL)
private TipoPieza tipoPieza;

private String[] valoresPropiedades;
```

```
private int[] nPinesConectoresMultiples;

public Pieza(Circuito circuito, TipoPieza tipoPieza) {
    commonSetup(circuito, tipoPieza);
    setupConectores();
    this.valoresPropiedades =
            tipoPieza.getPropiedades().stream().map(prop
                ↪ -> prop.getValor().toString())
                    .toArray(String[]::new);

}

private void commonSetup(Circuito circuito, TipoPieza
    ↪ tipoPieza) {
    this.circuito = circuito;
    this.tipoPieza = tipoPieza;
    setImagen(ImageUtils.cargarImagenEscalada(tipoPieza.
        ↪ getPathImagen(), Punto.getEscala()));
}

private void setupConectores() {
    setupConectores(
            tipoPieza.getConectoresConNombre().stream().
                ↪ filter(ConectorTemplate::isMultiple)
                    .mapToInt(ConectorTemplate::
                        ↪ getMinConectores).toArray());

}

private void setupConectores(int[]
    ↪ nPinesConectoresMultiples) {
```

```
        this.nPinesConectoresMultiples =
            ↪ nPinesConectoresMultiples.clone();
        this.conectores = generarConectores(tipoPieza);
        this.conectores.forEach(con -> con.setPieza(this));
        reposicionarConectores();

    }

    public Pieza(Circuito circ, Pieza otra) {
        commonSetup(circ, otra.getTipoPieza());
        this.setNombrePieza(otra.getNombrePieza());
        this.setPosicion(new Punto(otra.getPosicion()));
        this.setRotacion(otra.getRotacion());
        this.setValoresPropiedades(otra.valoresPropiedades.
            ↪ clone());
        setupConectores(otra.getNPinesConectoresMultiples())
            ↪ ;

    }

    public void setValorPropiedad(int nPropiedad, String
        ↪ valorPropiedad) {
        valoresPropiedades[nPropiedad] = valorPropiedad;
    }

    public int getNPinesConectorMultiple(int
        ↪ indiceConectorMultiple) {
        return nPinesConectoresMultiples[
            ↪ indiceConectorMultiple];
    }
```

```
public void setNPinesConectorMultiple(int
    ↪ indiceConectorMultiple, int nPines) {
    nPinesConectoresMultiples[indiceConectorMultiple] =
        ↪ nPines;
}

private List<Conector> generarConectores(TipoPieza
    ↪ tipoPieza) {
    List<Conector> lista = new ArrayList<>();

    //Conectores con nombre

    List<ConectorTemplate> conectoresMultiples =
            tipoPieza.getConectoresConNombre().stream().
                ↪ filter(ConectorTemplate::isMultiple)
                    .toList();
    List<ConectorTemplate> conectoresIndividuales =
            tipoPieza.getConectoresConNombre().stream().
                ↪ filter(con -> !con.isMultiple())
                    .toList();

    for (int i = 0; i < conectoresMultiples.size(); i++)
        ↪ {
        ConectorTemplate cm = conectoresMultiples.get(i)
            ↪ ;
        for (int c = 0; c < nPinesConectoresMultiples[i
            ↪ ]; c++) {
            lista.add(new Conector(cm, c + 1));
        }
    }
```

```
        for (ConectorTemplate ci : conectoresIndividuales) {
            lista.add(new Conector(ci));
        }
        return lista;
    }

    @PostLoad
    protected void postLoad() {
        setImagen(ImageUtils.rotar(
                ImageUtils.cargarImagenEscalada(tipoPieza.
                    ↪ getPathImagen(), tamano.width,
                        tamano.height), -rotacion.getAngulo
                            ↪ ()));
    }

    public Punto getPosicionConectorEnPanel(Conector
       ↪ conector, Punto posicionPieza) {
        double posicionRelativaX = conector.
            ↪ getPosicionRelativaX();
        double posicionRelativaY = conector.
            ↪ getPosicionRelativaY();
        for (int i = 0; i < getRotacion().ordinal(); i++) {
            double temp = posicionRelativaX;
            posicionRelativaX = posicionRelativaY;
            posicionRelativaY = 1 - temp;

        }

        Punto posicionConector =
                new Punto((int) (posicionPieza.getX() +
                    ↪ posicionRelativaX * getWidth()),
```

```
                    (int) (posicionPieza.getY() +
                        ↪ posicionRelativaY * getHeight
                        ↪ ()));

        //Mantener posici n dentro de los limites de la
            ↪ pieza
        if (posicionConector.getX() - Conector.getRadio() <
            ↪ posicionPieza.getX()) {
            posicionConector.translate(Conector.getRadio(),
                ↪ 0);
        }
        if (posicionConector.getX() + Conector.getRadio() >
            ↪ posicionPieza.getX() + getWidth()) {
            posicionConector.translate(-Conector.getRadio(),
                ↪  0);
        }
        if (posicionConector.getY() - Conector.getRadio() <
            ↪ posicionPieza.getY()) {
            posicionConector.translate(0, Conector.getRadio
                ↪ ());
        }
        if (posicionConector.getY() + Conector.getRadio() >
            ↪ posicionPieza.getY() + getHeight()) {
            posicionConector.translate(0, -Conector.getRadio
                ↪ ());
        }
        return posicionConector;
    }

    public void rotar(boolean derecha) {
        int incr = derecha ? -1 : 1;
```

```
        rotacion = AnguloRotacion.values()[
                (rotacion.ordinal() + incr + AnguloRotacion.
                    ↪ values().length) %
                        AnguloRotacion.values().length];
        setImagen(ImageUtils.rotar(imagen, 90 * -incr));
    }

    public Punto getPosicionConectorEnPanel(Conector
        ↪ conector) {
        return getPosicionConectorEnPanel(conector, posicion
            ↪ );
    }


    public void dibujar(PanelCircuito panelCircuito,
        ↪ Graphics g, Punto posicion, boolean
        ↪ dibujarContorno,
                        Map<Conector, Color>
                            ↪ coloresConectores) {
        if (getImagen() != null) {
            getImagen().paintIcon(panelCircuito, g, (int)
                ↪ posicion.getX(), (int) posicion.getY());
        }
        if (dibujarContorno) {
            g.drawRect((int) posicion.getX(), (int) posicion
                ↪ .getY(), getWidth(), getHeight());
        }

        g.setFont(new Font(Font.MONOSPACED, Font.PLAIN, 3 *
            ↪ Conector.getRadio()));
```

```
for (Conector c : conectores) {
    Color color;
    if (coloresConectores == null) {
        //color = c.getTipoConector().equals(
            ↪ TipoConector.ENTRADA) ? Color.BLUE :
            ↪ Color.green;
        color = Conector.colorConector;
    } else {
        color = coloresConectores.get(c);
    }
    g.setColor(color);
    Punto pos = getPosicionConectorEnPanel(c,
        ↪ posicion);
    g.fillOval((int) (pos.getX() - Conector.getRadio
        ↪ ()),
            (int) (pos.getY() - Conector.getRadio())
                ↪ , (2 * Conector.getRadio()),
            (2 * Conector.getRadio()));
    g.setColor(Color.BLACK);

    if (isRenderNombresPines()) {
        int desplazamientoX, desplazamientoY;
        if (getRotacion().equals(AnguloRotacion.
            ↪ ROT_0) ||
                getRotacion().equals(AnguloRotacion.
                    ↪ ROT_180)) {
            desplazamientoY = -2 * Conector.getRadio
                ↪ ();
            desplazamientoX = (c.getNombreConector()
                ↪ .length() + 2) * Conector.getRadio
                ↪ ();
```

```
                double centro = getPosicion().getX() +
                    ↪ getWidth() / 2;
                if (pos.getX() < centro) {
                    desplazamientoX *= -1;
                    desplazamientoX -= c.
                        ↪ getNombreConector().length() *
                        ↪  Conector.getRadio();
                }
            } else {
                desplazamientoX = 0;
                desplazamientoY = 3 * Conector.getRadio
                    ↪ ();
                double centro = getPosicion().getY() +
                    ↪ getHeight() / 2;
                if (pos.getY() < centro) {
                    desplazamientoY *= -1;
                    desplazamientoY += Conector.getRadio
                        ↪ ();
                }
            }
            g.drawString(c.getNombreConector(), pos.getX
                ↪ () + desplazamientoX,
                    pos.getY() + desplazamientoY);
        }
    }
    if (isRenerNombresPiezas()) {
        g.drawString(ModelicaGenerator.nombrePieza(this)
            ↪ , posicion.getX(),
                posicion.getY() - 4 * Conector.getRadio
                    ↪ ());
    }
```

```
}

public void reposicionarConectores() {
    List<Conector> entradas =
            conectores.stream().filter(c -> c.
               ↪ getTipoConector().equals(TipoConector.
               ↪ ENTRADA) && c.isReposicionar())
                    .toList();
    List<Conector> salidas =
            conectores.stream().filter(c -> c.
               ↪ getTipoConector().equals(TipoConector.
               ↪ SALIDA) && c.isReposicionar())
                    .toList();

    for (int i = 0; i < entradas.size(); i++) {
        float pos_y = (i + 1) / ((entradas.size() + 1) *
           ↪  1f);
        entradas.get(i).setPosicionRelativaY(pos_y);
    }

    for (int i = 0; i < salidas.size(); i++) {
        float pos_y = (i + 1) / ((salidas.size() + 1) *
           ↪ 1f);
        salidas.get(i).setPosicionRelativaY(pos_y);
    }
}


public void actualizarConectores(int[]
   ↪ newNPinesConectoresMultiples) {
    int counter = 0;
```

```
        for (int i = 0; i < nPinesConectoresMultiples.length
            ↪ ; i++) {
            ConectorTemplate ct = tipoPieza.
                ↪ getConectoresConNombre().get(i);
            int n = nPinesConectoresMultiples[i];
            int newN = newNPinesConectoresMultiples[i];
            counter += Math.min(newN, n);
            int diff = newN - n;
            for (int d = 0; d < Math.abs(diff); d++) {
                if (diff > 0) {
                    Conector c = new Conector(ct, n + d + 1)
                        ↪ ;
                    c.setPieza(this);
                    conectores.add(counter, c);
                    counter++;
                } else {
                    Conector c = conectores.get(counter);
                    circuito.borrarConexionesConector(c);
                    conectores.remove(c);
                }
            }

        }
        this.nPinesConectoresMultiples =
            ↪ newNPinesConectoresMultiples.clone();

        reposicionarConectores();
    }

    public int getWidth() {
        return (int) tamano.getWidth();
```

```
}

public int getHeight() {
    return (int) tamano.getHeight();
}

public Rectangle getBounds() {
    return new Rectangle(posicion.getPoint(), getTamano
        ↪ ());
}

public void setImagen(ImageIcon imagen) {
    this.imagen = imagen;
    this.tamano = new Dimension(imagen.getIconWidth(),
        ↪ imagen.getIconHeight());
}

@Override public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return
        ↪ false;
    Pieza pieza = (Pieza) o;
    return Objects.equals(getNombrePieza(), pieza.
        ↪ getNombrePieza()) &&
            Objects.equals(getPosicion(), pieza.
                ↪ getPosicion()) && getRotacion() ==
                ↪ pieza.getRotacion() &&
            getTipoPieza().equals(pieza.getTipoPieza())
                ↪ &&
            Arrays.equals(getValoresPropiedades(), pieza
                ↪ .getValoresPropiedades()) &&
```

```
                Arrays.equals(nPinesConectoresMultiples,
                    ↪ pieza.nPinesConectoresMultiples);
    }

    @Override public int hashCode() {
        int result = Objects.hash(getNombrePieza(),
            ↪ getPosicion(), getRotacion(), getTipoPieza());
        result = 31 * result + Arrays.hashCode(
            ↪ getValoresPropiedades());
        result = 31 * result + Arrays.hashCode(
            ↪ nPinesConectoresMultiples);
        return result;
    }
}
```

## B.24. Fichero `modelo/Propiedad.java`

```
package caponera.uned.tfm.lizardclips.modelo;

import caponera.uned.tfm.lizardclips.modelica.
    ↪ ModelicaGenerator;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.Setter;

@AllArgsConstructor
public abstract class Propiedad<T> {
    public static final String UNIDAD_REAL = "Real";
    public static final String UNIDAD_TIME =
        ↪ ModelicaGenerator.SI + ".Time";
    @Getter
```

```
    @Setter
    private T valor;

    @Getter
    private String nombre;

    @Getter
    @Setter
    private String unidad;

    @Getter
    @Setter
    private String tooltipDescription;


}
```

## B.25. Fichero `modelo/PropiedadLogic.java`

```
package caponera.uned.tfm.lizardclips.modelo;

import caponera.uned.tfm.lizardclips.modelica.
    ↪ ModelicaGenerator;

public class PropiedadLogic extends
    ↪ PropiedadSeleccionMultiple {


    public PropiedadLogic(String nombre, String valor,
        ↪ String tooltipDescription) {
        super(valor, nombre, PropiedadSeleccionMultiple.
```

```
            ↪ SELECCION_MULTIPLE_LOGICAL,
                PropiedadSeleccionMultiple.
                    ↪ PREFIX_SELECCION_MULTIPLE_LOCICAL,
                ModelicaGenerator.LOGIC, tooltipDescription)
                    ↪ ;
    }

    public PropiedadLogic(String nombre, String
        ↪ tooltipDescription) {
        this(nombre, "'0'", tooltipDescription);
    }
}
```

## B.26. Fichero `modelo/PropiedadSeleccionMultiple.java`

```
package caponera.uned.tfm.lizardclips.modelo;

import lombok.Getter;
import lombok.Setter;

import java.util.List;

public class PropiedadSeleccionMultiple extends Propiedad<
    ↪ String> {
    public static final List<String>
        ↪ SELECCION_MULTIPLE_LOGICAL =
            List.of("'U'", "'X'", "'0'", "'1'", "'Z'", "'W'"
                ↪ , "'L'", "'H'", "'-'");
    public static final String
        ↪ PREFIX_SELECCION_MULTIPLE_LOCICAL = "L.";
```

```
    @Getter
    @Setter
    private List<String> valoresPosibles;
    @Getter
    private String prefix;

    public PropiedadSeleccionMultiple(String valor, String
        ↪ nombre, List<String> valoresPosibles, String
        ↪ unidad, String tooltipDescription) {
        super(valor, nombre, unidad, tooltipDescription);
        this.valoresPosibles = valoresPosibles;
    }

    public PropiedadSeleccionMultiple(String valor, String
        ↪ nombre, List<String> valoresPosibles, String
        ↪ prefix, String unidad, String tooltipDescription)
        ↪ {
        this(prefix + valor, nombre, valoresPosibles.stream
            ↪ ().map(vp -> prefix + vp).toList(),
                unidad, tooltipDescription);
    }
}
```

## B.27. Fichero `modelo/PropiedadSimple.java`

```
package caponera.uned.tfm.lizardclips.modelo;

public class PropiedadSimple extends Propiedad<String> {
    public PropiedadSimple(String valor, String nombre,
        ↪ String unidad, String tooltipDescription) {
        super(valor, nombre, unidad, tooltipDescription);
```

```
    }
}
```

## B.28. Fichero utils/AnguloRotacion.java

```
package caponera.uned.tfm.lizardclips.utils;

import lombok.AllArgsConstructor;
import lombok.Getter;

@AllArgsConstructor
@Getter
public enum AnguloRotacion {
    ROT_0(0), ROT_90(90), ROT_180(180), ROT_270(270);

    private final int angulo;
}
```

## B.29. Fichero utils/DescriptorImagen.java

```
package caponera.uned.tfm.lizardclips.utils;

import lombok.AllArgsConstructor;
import lombok.Data;

@Data
@AllArgsConstructor
public class DescriptorImagen {
    private String pathImagen;
    private int width, height;
```

```
}
```

## B.30. Fichero utils/I18NUtils.java

```
package caponera.uned.tfm.lizardclips.utils;

import java.util.ResourceBundle;

public class I18NUtils {
    private static final String RESOURCE_BUNDLE_NAME = "
        ↪ TextosAplicacion";

    private static ResourceBundle textosAplicacion;

    public static String getString(String key) {
        if (textosAplicacion == null) {
            textosAplicacion = ResourceBundle.getBundle(
                ↪ RESOURCE_BUNDLE_NAME);
        }
        return textosAplicacion.getString(key);
    }
}
```

## B.31. Fichero utils/ImageUtils.java

```
package caponera.uned.tfm.lizardclips.utils;

import javax.imageio.ImageIO;
import javax.swing.ImageIcon;
import java.awt.Graphics;
import java.awt.Graphics2D;
```

```
import java.awt.Image;
import java.awt.geom.AffineTransform;
import java.awt.image.BufferedImage;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.net.URL;
import java.util.HashMap;
import java.util.Map;

public class ImageUtils {
    public static final String MEDIA_BASE_FOLDER = "media";
    public static final int DEFAULT_IMAGE_WIDTH = 65;
    public static final int DEFAULT_IMAGE_HEIGHT = 65;
    private static Map<DescriptorImagen, ImageIcon>
        ↪ imagenesCacheadas = new HashMap<>();

    public static final String pathImagenMedia(String
        ↪ nombreImagen) {
        return ImageUtils.MEDIA_BASE_FOLDER + "/" +
            ↪ nombreImagen;
    }

    public static ImageIcon cargarImageIcon(String
        ↪ pathImagen) {
        ClassLoader classLoader = Thread.currentThread().
            ↪ getContextClassLoader();
        //ClassLoader classLoader = ImageUtils.class.
            ↪ getClassLoader();
        URL resource = classLoader.getResource(pathImagen);
        System.out.println("Cargando " + pathImagen);
        return new ImageIcon(resource);
```

```
}

public static ImageIcon cargarImagenEscalada(String
    ↪ pathImagen, int ancho, int alto, int modo) {
    DescriptorImagen desc = new DescriptorImagen(
        ↪ pathImagen, ancho, alto);
    if (!imagenesCacheadas.containsKey(desc)) {
        imagenesCacheadas.put(desc,
                rescalarImagen(cargarImageIcon(
                    ↪ pathImagen), ancho, alto, modo));
    }
    return imagenesCacheadas.get(desc);
}

public static ImageIcon rescalarImagen(ImageIcon imagen,
    ↪  int ancho, int alto, int modo) {
    return new ImageIcon(imagen.getImage().
        ↪ getScaledInstance(ancho, alto, modo));
}

public static ImageIcon rescalarImagen(ImageIcon imagen,
    ↪  int ancho, int alto) {
    return rescalarImagen(imagen, ancho, alto, Image.
        ↪ SCALE_SMOOTH);
}

public static ImageIcon
    ↪ cargarImageneEscaladaPreserveRatio(String
    ↪ pathImagen, int ancho, int alto) {
    ImageIcon raw = cargarImageIcon(pathImagen);
    return rescalarImagenPreserveRatio(pathImagen, raw,
```

```
        ↪ ancho, alto);
}

public static ImageIcon cargarImagenEscalada(String
    ↪ pathImagen, double ratio) {
    ImageIcon raw = cargarImageIcon(pathImagen);
    return rescalarImagenPreserveRatio(pathImagen, raw,
        ↪ (int) (raw.getIconWidth() * ratio),
            (int) (raw.getIconHeight() * ratio));
}

public static ImageIcon rescalarImagenPreserveRatio(
    ↪ String pathImagen, ImageIcon raw, int ancho, int
    ↪ alto) {
    double imageRatio = 1.0 * raw.getIconWidth() / raw.
        ↪ getIconHeight();
    double containerRatio = 1.0 * ancho / alto;
    if (imageRatio > containerRatio) {
        alto = (int) (ancho / imageRatio);
    } else {
        ancho = (int) (imageRatio * alto);
    }
    DescriptorImagen desc = new DescriptorImagen(
        ↪ pathImagen, ancho, alto);
    if (!imagenesCacheadas.containsKey(desc)) {
        imagenesCacheadas.put(desc, rescalarImagen(raw,
            ↪ ancho, alto, Image.SCALE_SMOOTH));
    }

    return imagenesCacheadas.get(desc);
}
```

```
public static ImageIcon rescalarImagenPreserveRatio(
    ↪ ImageIcon raw, int ancho, int alto) {
     return rescalarImagenPreserveRatio(String.valueOf(
        ↪ raw.hashCode()), raw, ancho, alto);
}

public static ImageIcon cargarImagenEscalada(String
    ↪ pathImagen, int ancho, int alto) {
     return cargarImagenEscalada(pathImagen, ancho, alto,
        ↪  Image.SCALE_SMOOTH);
}

public static byte[] bytesFromBufferedImage(
    ↪ BufferedImage bi) {
     ByteArrayOutputStream baos = new
        ↪ ByteArrayOutputStream();
     try {
         ImageIO.write(bi, "jpg", baos);
     } catch (IOException e) {
         e.printStackTrace();
     }
     return baos.toByteArray();
}

public static ImageIcon imageIconFromBytes(byte[] bytes)
    ↪ {
     return new ImageIcon(bytes);
}

public static ImageIcon rotar(ImageIcon original, int
```

```
↪ grados) {
 BufferedImage image = new BufferedImage(original.
    ↪ getIconWidth(), original.getIconHeight(),
        BufferedImage.TYPE_INT_RGB);
 Graphics g = image.createGraphics();
 original.paintIcon(null, g, 0, 0);
 g.dispose();
 // Calculate the new size of the image based on the
    ↪ angle of rotaion
 double radians = Math.toRadians(grados);
 double sin = Math.abs(Math.sin(radians));
 double cos = Math.abs(Math.cos(radians));
 int newWidth = (int) Math.round(image.getWidth() *
    ↪ cos + image.getHeight() * sin);
 int newHeight = (int) Math.round(image.getWidth() *
    ↪ sin + image.getHeight() * cos);

 // Create a new image
 BufferedImage rotate = new BufferedImage(newWidth,
    ↪ newHeight, BufferedImage.TYPE_INT_ARGB);
 Graphics2D g2d = rotate.createGraphics();
 // Calculate the "anchor" point around which the
    ↪ image will be rotated
 int x = (newWidth - image.getWidth()) / 2;
 int y = (newHeight - image.getHeight()) / 2;
 // Transform the origin point around the anchor
    ↪ point
 AffineTransform at = new AffineTransform();
 at.setToRotation(radians, x + (image.getWidth() /
    ↪ 2.0), y + (image.getHeight() / 2.0));
 at.translate(x, y);
```

```
        g2d.setTransform(at);
        // Paint the orginl image
        g2d.drawImage(image, 0, 0, null);
        g2d.dispose();
        return new ImageIcon(rotate);
    }
}
```

## B.32. Fichero utils/LineUtils.java

```
package caponera.uned.tfm.lizardclips.utils;

import java.util.ArrayList;
import java.util.List;

public class LineUtils {
    public static List<Punto> getPuntosManhattan(List<Punto>
    ↪ puntos) {
        List<Punto> puntosManhattan = new ArrayList<>();
        if (puntos.size() < 2) {
            puntosManhattan = puntos;
        } else {
            puntosManhattan.add(puntos.get(0));
            for (Punto[] pareja : getParejas(puntos)) {
                Punto intermedio = manhattan(pareja[0],
                ↪ pareja[1]);
                if (intermedio != null) {
                    puntosManhattan.add(intermedio);
                }
                puntosManhattan.add(pareja[1]);
            }
```

```
        }

        return puntosManhattan;
    }

    /**
     * Calcula el punto intermedio para conectar p1 y p2
        ↪ rectangularmente
     *
     * @param p1 primer punto a conectar
     * @param p2 segundo punto a conectar
     * @return {@code null} si los puntos ya se puden
        ↪ conectar rectangularmente. Si no, el punto
     * intermedio para conectarlos.
     */
    private static Punto manhattan(Punto p1, Punto p2) {
        Punto intermedio = null;
        if (p1.getX() != p2.getX() && p1.getY() != p2.getY()
            ↪ ) {
            //primero el eje m s largo
            if (Math.abs(p1.getX() - p2.getX()) >= Math.abs(
                ↪ p1.getY() - p2.getY())) {
                intermedio = new Punto(p2.getX(), p1.getY())
                    ↪ ;
            } else {
                intermedio = new Punto(p1.getX(), p2.getY())
                    ↪ ;
            }
        }

        return intermedio;
```

```
    }

    public static List<Punto[]> getParejas(List<Punto>
        ↪ puntos) {
        List<Punto[]> parejas = new ArrayList<>();
        Punto p1 = puntos.get(0);
        for (int i = 1; i < puntos.size(); i++) {
            Punto p2 = puntos.get(i);
            parejas.add(new Punto[]{p1, p2});
            p1 = p2;
        }

        return parejas;
    }
}
```

## B.33. Fichero `utils/Punto.java`

```
package caponera.uned.tfm.lizardclips.utils;

import jakarta.persistence.Embeddable;
import jakarta.persistence.Transient;
import lombok.Getter;
import lombok.NoArgsConstructor;

import java.awt.Point;
import java.util.Objects;

@Embeddable
@NoArgsConstructor
public class Punto {
```

```
//Coordenadas de referencia en las que se sit a el
    ↪ panel
@Transient
private static int referenciaX = 0, referenciaY = 0;
@Transient
@Getter
private static double escala = 1f;
@Transient
private static final float ESCALA_MIN = 0.1f, ESCALA_MAX
    ↪  = 2f;

//Coordenadas virtuales en un hipot tico plano infinito
private double x, y;

public Punto(Point p) {
    this(p.x, p.y);
}

public Punto(Punto otro) {
    this.x = otro.x;
    this.y = otro.y;
}

public Punto(int x, int y) {
    setXPanel(x);
    setYPanel(y);
}

public static Punto puntoCoordenadasVirtuales(double x,
    ↪ double y) {
    Punto p = new Punto(0, 0);
```

```
        p.setXVirtual(x);
        p.setYVirtual(y);
        return p;
    }

    public static void reescalar(float dZ) {
        escala = Math.min(ESCALA_MAX,
                Math.max(ESCALA_MIN, escala + dZ)); //
                    ↪ Asegurar que escala est  entre min y
                    ↪ max
        System.out.println("Nueva escala: " + escala);
    }

    public static void desplazarReferencia(int dX, int dY) {
        referenciaX += dX;
        referenciaY += dY;
    }

    public static void resetReferencia() {
        referenciaX = 0;
        referenciaY = 0;
    }

    public int getX() {
        return (int) (x * escala + referenciaX);
    }

    private void setXPanel(int xPanel) {
        x = (xPanel - referenciaX) / escala;
    }
```

```
public int getY() {
    return (int) ((y * escala + referenciaY));
}

private void setYPanel(int yPanel) {
    y = (yPanel - referenciaY) / escala;
}

public int getXVirtual() {
    return (int) x;
}

public int getYVirtual() {
    return (int) y;
}

public void setXVirtual(double x) {
    this.x = x;
}

public void setYVirtual(double y) {
    this.y = y;
}

public Point getPoint() {
    return new Point(getX(), getY());
}

public void translate(int dx, int dy) {
    x += dx;
    y += dy;
```

```
    }

    @Override public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return
            ↪ false;
        Punto punto = (Punto) o;
        return Double.compare(punto.getX(), getX()) == 0 &&
            ↪ Double.compare(punto.getY(), getY()) == 0;
    }

    @Override public int hashCode() {
        return Objects.hash(getX(), getY());
    }
}
```