UNIVERSIDAD DE CONCEPCIÓN
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE INGENIERÍA INFORMÁTICA Y CIENCIAS DE LA COMPUTACIÓN

Generación automática de visualizaciones de datos: Integración con fuentes de datos

POR

JORGE ANTONIO JARA INOSTROZA

Memoria de Título presentada a la Facultad de Ingeniería de la Universidad de Concepción para optar al título profesional de Ingeniero Civil Informático

Profesor Guía
Gonzalo Rojas Durán

Abril de 2024
Concepción, Chile

Resumen

Para poder analizar correctamente la información obtenida de alguna fuente de datos, sea cual sea su tamaño o complejidad, es necesario definir una visualización adecuada a las características de dicha información, para que las personas interesadas puedan analizarla y extraer conclusiones significativas, las cuales les permitan tomar decisiones informadas respecto a los procesos de los que emanan estos datos. Con este objetivo en mente, en el marco de este proyecto se desarrolló una herramienta gráfica para generación de visualizaciones de datos, denominada `dashboardgen`, la cual permite generar *dashboards* por medio de un lenguaje específico de dominio gráfico, a partir de los cuales se pueden generar aplicaciones ejecutables que tomen los datos de las fuentes indicadas y puedan generar visualizaciones, como gráficos y tarjetas. Los resultados obtenidos durante la evaluación de la herramienta fueron generalmente positivos, donde los usuarios destacaron la facilidad al realizar los ejercicios propuestas, indicando, sin embargo, algunas dificultades en aspectos más complejos o en la presentación de errores, entre otros. El proyecto realizado tiene la posibilidad de ser mejorado en próximas versiones, incorporando nuevas características y mejorando la experiencia de usuario, utilizando la información provista por los usuarios que la probaron y considerando limitaciones que fueron encontradas durante el proceso de desarrollo. La herramienta puede ser considerada un punto de partida para la implementación de una solución de *software* que permita generar visualizaciones de datos interactivas entendiendo el dominio de los datos a utilizar y sin necesitar mayores conocimientos de herramientas de programación.

Índice general

Índice general **3**

Índice de figuras **5**

Índice de cuadros **6**

1 Introducción **7**
1.1. Solución propuesta y alcances 8
1.2. Estructura del informe 9

2 Marco teórico **10**
2.1. Lenguajes específicos de dominio 10
2.2. Visualización de datos 11
2.3. Trabajo previo 15

3 Arquitectura de la propuesta **17**

4 Definición del lenguaje específico de dominio **21**
4.1. Análisis del dominio 21
4.2. Definición del metamodelo 22
4.3. Representación gráfica del lenguaje específico de dominio 32
4.4. Tarjetas 33
4.5. Vistas 34
4.6. Fuentes de datos 34
4.7. Relaciones entre elementos 35

5 Sistema de generación automática de código **36**
5.1. Esquema de transformación de modelo a texto 36
5.2. Aplicación generada 40

6 Evaluación de la propuesta **46**
6.1. Cobertura de las visualizaciones 46
6.2. Evaluación de usabilidad 48

7 Conclusiones **56**

Bibliografía **58**

A Resumen de los elementos del DSL **60**

B Diagrama del metamodelo **65**

C Encuesta de satisfacción **67**

Índice de figuras

2.1. Elementos de un lenguaje específico de dominio. 10
2.2. Ejemplo de gráfico de líneas, generado con la biblioteca Chart.js. 12
2.3. Ejemplo de gráfico de área, generado con la biblioteca Chart.js. 12
2.4. Ejemplo de gráfico de barras, generado con la biblioteca Chart.js. 13
2.5. Ejemplo de gráfico de dispersión, generado con la biblioteca Chart.js. 13
2.6. Ejemplo de histograma, generado con la biblioteca Plotly. Ejemplo extraido de [8]. 14
2.7. Conjunto de tarjetas, obtenido desde una instancia de la aplicación Pi-hole. . . . 15

3.1. Diagrama de contexto de la arquitectura C4 propuesta. 18
3.2. Diagrama de contenedores de la arquitectura C4 propuesta. 19
3.3. Diagrama de componentes de la arquitectura C4 propuesta. 20

4.1. Diagrama del lenguaje específico de dominio a implementar. 21
4.2. Metaclase principal del *dashboard*. 22
4.3. Diagrama de las fuentes de datos. 23
4.4. Diagrama de las columnas virtuales. 24
4.5. Diagrama del *dashboard*. 25
4.6. Diagrama de los gráficos. 25
4.7. Diagrama de las tarjetas. 27
4.8. Diagrama de las transformaciones. 28
4.9. En el sentido de las agujas del reloj, empezando desde la esquina superior izquierda: Íconos utilizados para los gráficos de dispersión, gráficos de área, gráficos de barras, histogramas y gráficos de línea. 32
4.10. En el sentido de las agujas del reloj, desde la esquina superior izquierda: Gráfico de líneas, gráfico de área, histograma, gráfico de dispersión, gráfico de barras. . . 33
4.11. Elemento de tarjeta en la representación visual. 34
4.12. Elemento de vista en la representación visual. 34
4.13. Elemento de fuente de datos en la representación visual. 35
4.14. Vista conectada a una fuente de datos. 35

5.1. Estructura de los archivos del generador de modelo a texto. 37
5.2. Proceso de generación de código a partir de un modelo de *dashboard* 41
5.3. Vista del *dashboard* generado. 42
5.4. Gráfico de dispersión generado. 43
5.5. Acercamiento al selector de series de un gráfico. 43
5.6. Gráfico de líneas generado utilizando valores de tiempo para el eje X. 43
5.7. Histograma generado. 44

5.8. Flujo de ejecución del *dashboard* generado. 44

6.1. *Dashboard* esperado después de realizar el Ejercicio 1. 49
6.2. *Dashboard* esperado después de realizar el Ejercicio 2. 50
6.3. *Dashboard* esperado después de realizar el Ejercicio 3. 52

B.1. Diagrama del metamodelo completo. 66

Índice de cuadros

2.1. Ventajas y desventajas entre los tipos de gráficos mencionados. [9] [10] 15

4.1. *Dataset* de ejemplo. 30
4.2. *Dataset* de ejemplo después de agrupar por `nombre`, dejando la `nota` mínima. . . 30
4.3. *Dataset* de ejemplo después de filtrar las filas con `nota` mayor a 2. 31
4.4. *Dataset* de ejemplo después filtrar las `notas` mayores a 2. 31
4.5. *Dataset* de ejemplo después de agrupar las filas por `nombre`, dejando la `nota` mínima. 31

A.1. Resumen de los elementos base del DSL. 61
A.2. Resumen de los elementos del DSL relacionados a gráficos. 62
A.3. Resumen de los elementos del DSL relacionados a gráficos. (cont.) 63
A.4. Resumen de los elementos del DSL relacionados a transformaciones. 64

Capítulo 1

Introducción

En una era de crecimiento exponencial de datos impulsado por el auge de la internet, la visualización de datos ha adquirido una importancia crítica. Cada día se generan enormes cantidades de datos procedentes de diversas fuentes, como plataformas de redes sociales, transacciones digitales y dispositivos IoT[1]. Dentro de esta avalancha de datos se pueden encontrar patrones, tendencias y perspectivas que pueden impulsar estrategias, mejorar la toma de decisiones y dar a las empresas una ventaja competitiva. Sin embargo, extraer información significativa de este océano de datos no es una tarea sencilla. Es aquí donde entra en juego la visualización de datos, la que permite generar representaciones simplificadas pero más comprensibles a partir de grandes volúmenes de datos.

Gracias al enorme incremento en capacidad de cómputo que han adquirido los computadores en las últimas décadas, se ha vuelto posible que hasta un equipo de uso doméstico tenga la capacidad de digerir grandes volúmenes de datos y obtener de estos indicadores y análisis relevantes. Sin embargo, es necesario que estos resultados sean presentados al usuario de una forma legible y comprensible. Es aquí donde la visualización de datos resulta indispensable, ya que un buen *dashboard* le mostrará al usuario todos los indicadores, gráficos y tablas que requiera sin abrumarlo innecesariamente, permitiéndole tomar decisiones adecuadas a partir de estos. Al convertir conjuntos de datos grandes y complejos en patrones y tendencias visuales y accesibles, la visualización de datos permite a asesores, supervisores, gerentes y cualquier otro responsable de la toma de decisiones poder percibir y comprender rápidamente las correlaciones, los valores atípicos y patrones complejos que pudieran contener los datos a analizar.

Sin embargo, la generación de herramientas de visualización de datos suele ser una actividad relegada exclusivamente a programadores e ingenieros de *software*, quienes, por medio de su conocimiento en programación de computadores, pueden crear dichas herramientas y publicarlas a posibles usuarios. Desarrollar una herramienta *software* de visualización de datos requiere que el programador tenga suficientes conocimientos en uno o varios lenguajes de programación, conozca de herramientas para obtener los datos a analizar, realizar análisis sobre ellos y generar gráficos e indicadores útiles para su público objetivo, algo que no cualquier interesado en el análisis de datos sabría hacer. Por lo mismo, se vuelve importante darle la posibilidad al público en general de poder desarrollar programas a la medida sin necesitar de profundos conocimientos en desarrollo de *software*.

[1] Abrevación para *Internet of Things*, "Internet de las cosas" en inglés.

1.1. Solución propuesta y alcances

El proyecto, una vez finalizado, comprenderá los siguientes componentes:

- Un **metamodelo** capaz de modelar los requerimientos de cualquier *dataset*, así como los *dashboards* que desee generar el usuario, junto con los gráficos, vistas y demás elementos que incluyan.
- Una **herramienta gráfica de generación de modelos**, que permita la creación de modelos de *dashboard* y definición de *datasets*, los cuales describirán la aplicación final.
- Un **transformador de modelo a texto** (M2T, *model to text* en inglés), que convertirá el modelo creado por el usuario en archivos de código fuente, los cuales, al ejecutarse, permitirán al usuario y público objetivo acceder al *dashboard* diseñado, el cual utilizará los datos del o los *datasets* definidos para alimentar sus vistas y componentes.
- Estos elementos serán combinados en un único entregable, una **extensión para Eclipse EMF**, la cual permitirá la generación de *dashboards* sin necesidad de tener conocimientos de programación.

Objetivo general

El objetivo general de este proyecto es integrar fuentes de datos a entornos de generación automática de código para la visualización de datos a partir de estas.

Objetivos específicos

Los objetivos específicos son los siguientes:

1. Especificar una arquitectura de sistema de visualización de datos, basada en el paradigma de desarrollo dirigido por modelos, que considere bibliotecas de visualización y fuentes de datos (*datasets*).
2. Especificar requerimientos de integración de fuentes de datos con descripciones abstractas de visualizaciones de datos.
3. Integrar conocimientos de visualización de datos e ingeniería de *software* en un proceso de generación automática de código.
4. Evaluar la integración y usabilidad de la herramienta de modelado y generación automática, con desarrolladores y usuarios de visualizaciones.

Metodología

Se adoptará un método de desarrollo iterativo e incremental, con adaptaciones de propuestas ágiles a un entorno de desarrollo unipersonal. El desarrollo se basa en un análisis de tecnologías existentes y referencias bibliográficas en el área de desarrollo dirigido por modelos y visualización de datos, que sean pertinentes a los objetivos de la memoria.

1.2. Estructura del informe

Después de este capítulo de introducción, se presenta el marco teórico, que define los conceptos fundamentales que utilizará este informe, además de su conexión e importancia para el trabajo realizado.

En los capítulos posteriores, se define el contenido de la propuesta, que comprende una explicación de la arquitectura de *software* utilizada, una descripción a detalle de sus componentes y su funcionamiento, además de presentar los artefactos generados por la aplicación y su uso.

Luego, en el capítulo de evaluación, se explica la metodología utilizada para evaluar la experiencia de uso del proyecto con usuarios de prueba, además de entregar los resultados de dicha evaluación y las conclusiones que pueden extraerse de esta.

Finalmente, se presenta la conclusión de este informe, donde se entregan las reflexiones en torno al trabajo realizado, las posibles mejores a este y las experiencias aprendidas a lo largo de su elaboración.

Capítulo 2

Marco teórico

2.1. Lenguajes específicos de dominio

Un lenguaje específico de dominio, también denominado DSL (abreviatura de *Domain Specific Language* en inglés), se define como un lenguaje de programación o especificación ejecutable que ofrece, a través de las notaciones y abstracciones apropiadas, una capacidad expresiva enfocada y, usualmente restringida, a un dominio de problemas en particular [1].

En el contexto del desarrollo de *software* dirigido por modelos, un paradigma de programación que se centra en la utilización de modelos para representar soluciones de *software* en lugar de código, los DSL cumplen el rol de lenguajes de modelado, con los cuales se puede diseñar un lenguaje a la medida para detallar los elementos de un sistema y sus relaciones, los cuales pueden representar las abstracciones de un dominio de problemas de una forma familiar para expertos en el tema [2]. Así, todo DSL se compone de tres elementos: una sintaxis abstracta, representada por un metamodelo, que define las características a abstraer del dominio y sus relaciones, una sintaxis concreta (notación), que le indica al usuario la forma en que utilizará el lenguaje, y una semántica, que conecta la sintaxis abstracta y concreta, dotándolas de significado [3]. La Figura 2.1 presenta un diagrama mostrando la estructura de un DSL.

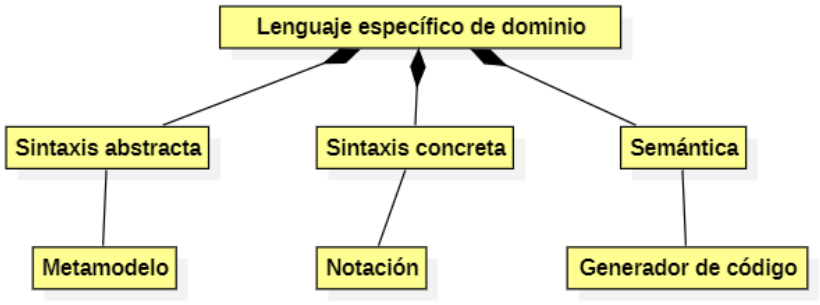

Figura 2.1: Elementos de un lenguaje específico de dominio.

Sintaxis abstracta

La sintaxis abstracta se puede definir como un modelo de lenguaje que define los elementos que van a formar parte de los modelos que van a poder definirse con el DSL y sus relaciones [2]. Haciendo una analogía, la sintaxis abstracta correspondería a todas las palabras que existen en un idioma y las reglas gramaticales que lo rigen.

Una sintaxis abstracta puede definirse por medio de un metamodelo, que puede definir los componentes del dominio por medio de diagramas, como puede ser un diagrama UML, que es la técnica recomendada según [2].

Sintaxis concreta

La sintaxis concreta es la forma en la que los usuarios van a interactuar con sus modelos para construirlos y visualizarlos [2]. Siguiendo con la analogía anterior, la sintaxis concreta equivale al sistema de escritura de un idioma humano, donde las palabras pueden representarse mediante distintos tipos de símbolos, como alfabetos, ideogramas o incluso mediante representaciones numéricas, por medio de sistemas de codificación de caracteres.

Una sintaxis concreta, en este contexto, puede estar definida mediante un sistema gráfico, donde los elementos de la sintaxis abstracta se representan como elementos gráficos, como figuras geométricas, imágenes, texto, etc. También es posible utilizar una sintaxis basada en texto, como es el caso de los lenguajes de programación convencionales, en el que los elementos de la sintaxis abstracta se pueden representar mediante palabras o símbolos.

Semántica

La semántica se puede definir como el pegamento entre la sintaxis abstracta y la sintaxis concreta. Es la que revela el significado de las expresiones sintácticamente válidas definidas por el usuario en un determinado lenguaje [3]. En este contexto, el rol de la semántica lo cumple una herramienta de generación de código, que toma los elementos de la sintaxis concreta y los relaciona con la sintaxis abstracta [2], para así generar artefactos ejecutables que representen el problema definido por el desarrollador.

2.2. Visualización de datos

La visualización de datos se define la representación de información en un formato sistemático, incluyendo atributos y variables para la unidad de información a representar [4]. Para lograr esto, es posible utilizar distintos tipos de técnicas (visualizaciones), como pueden ser gráficos, tablas, diagramas o tarjetas. La visualización de un conjunto de datos debiera realizarse de forma clara y adecuada al contexto de los datos, para así facilitarle a los *stakeholders* el análisis de la información representada y, así, la toma de decisiones a partir de esta.

Tipos de gráficos

Existen múltiples tipos de gráficos, donde cada uno tiene sus propias particularidades y casos de eso. Algunos ejemplos de gráficos son los siguientes:

Gráficos de líneas y de área

El gráfico de líneas (también llamado gráfico de curvas) es uno de los tipos de gráficos más utilizados para visualizar datos [5]. Se compone de un plano cartesiano, donde cada eje representa una componente distinta de los datos, en el que se grafican los puntos de datos conectados mediante líneas, para dar la sensación de continuidad. Estas líneas pueden componerse de múltiples segmentos rectos o ser una única curva suavizada.

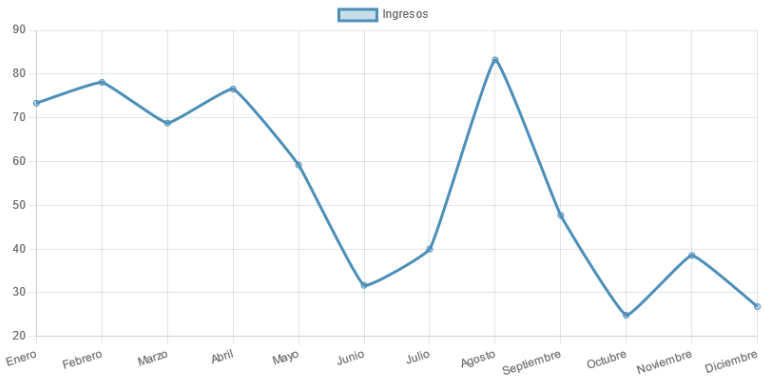

Figura 2.2: Ejemplo de gráfico de líneas, generado con la biblioteca Chart.js.

Por otro lado, también existen los gráficos de área (también llamados gráficos de superficie), que son similares a un gráfico de líneas, pero que sombrean la porción del plano por debajo de la curva, lo que permite dar énfasis a la magnitud de una tendencia o mostrar la importancia relativa de múltiples variables que componen un valor más grande (p. ej. cuánto aporta cada país en el cálculo del producto interno bruto mundial) [5].

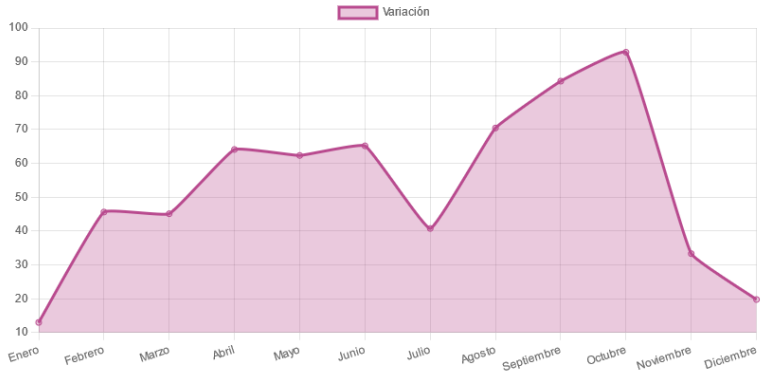

Figura 2.3: Ejemplo de gráfico de área, generado con la biblioteca Chart.js.

Gráfico de barras

Los gráficos de barras, de forma análoga a los gráficos de líneas y de área, utilizan un plano cartesiano como base para presentar su información, con la diferencia de que esta se presenta mediante el uso de barras verticales u horizontales, apoyadas contra algunos de los

dos ejes del plano. En el caso de que las barras estén en vertical, el gráfico toma también el nombre de gráfico de columnas.

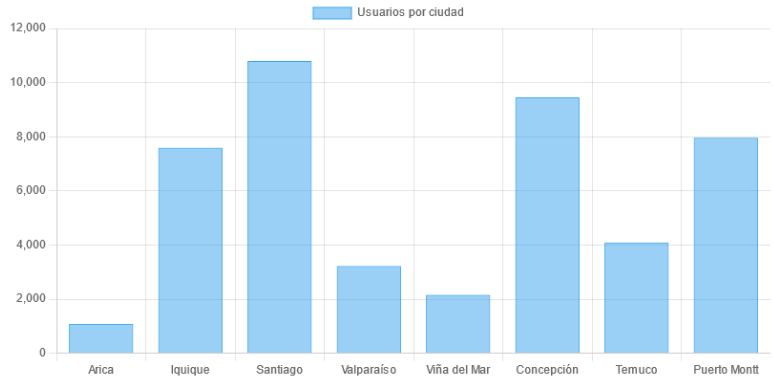

Figura 2.4: Ejemplo de gráfico de barras, generado con la biblioteca Chart.js.

Los gráficos de barras son bastante versátiles como método de visualización, pues pueden utilizarse, por ejemplo, para graficar la evolución de una variable a lo largo del tiempo (como podría hacerse con un gráfico de líneas o de área) o comparar el valor de una variable en un instante dado entre distintas categorías (p. ej. el precio promedio de las casas vendidas el año pasado en distintas ciudades) [5].

Gráfico de dispersión

Los gráficos de dispersión, al igual que los ejemplos anteriores, toman como base un plano cartesiano, con la diferencia que los datos se representan en este por medio de puntos, donde cada uno de los ejes representa una variable distinta.

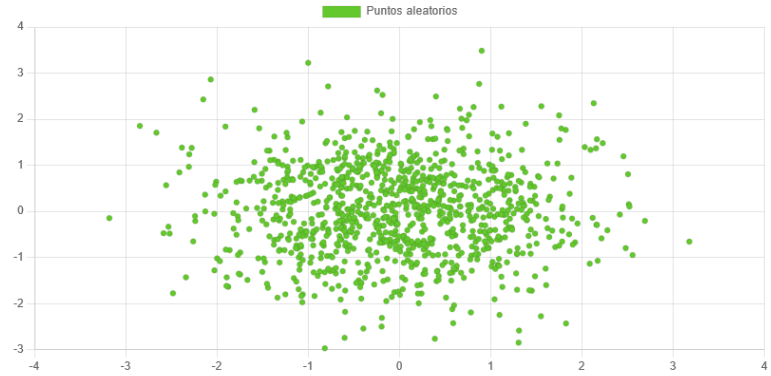

Figura 2.5: Ejemplo de gráfico de dispersión, generado con la biblioteca Chart.js.

A pesar de presentar la información de una manera más densa que otros tipos de gráficos, los gráficos de dispersión permiten visual de forma más clara la presencia de *outliers*, así como el nivel de variabilidad que existe para una de las variables con respecto a la otra [6]. También es posible visualizar de manera intuitiva tendencias sobre los datos, lo que puede ser posteriormente respaldado por medio de cálculos sobre el conjunto de datos graficado.

Histograma

Un histograma es un tipo especial de gráfico de barras, que permite visualizar la forma de una distribución de datos [7]. A diferencia de un gráfico de barras, el histograma utiliza una única variable, a partir de la cual se genera la distribución, la que se representa usualmente con barras verticales contiguas. Cada barra representa un intervalo (*bin*) de la distribución, cuya altura indica la frecuencia del intervalo dentro del conjunto de datos. El número de intervalos que puede contener el histograma queda a elección de quien lo diseña, aunque es usual realizar algún tipo de análisis sobre los datos para calcular la cantidad óptica de intervalos.

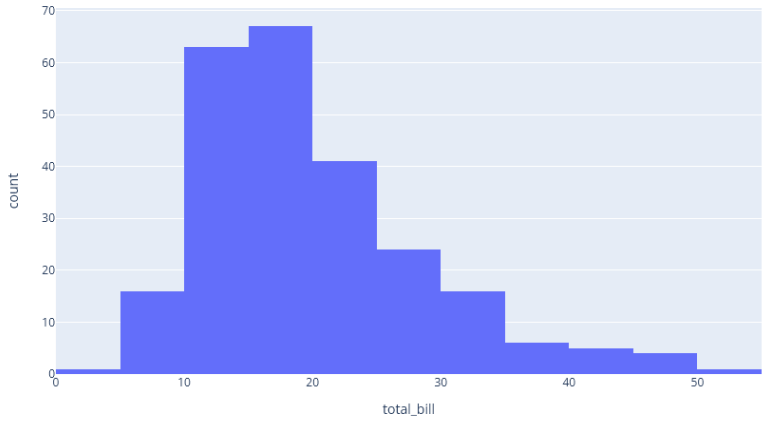

Figura 2.6: Ejemplo de histograma, generado con la biblioteca Plotly. Ejemplo extraido de [8].

Resumen

Considerando las características mencionadas de cada uno de los tipos de gráfico mencionados, además de otras que van en función de los tipos de datos a graficar, es posible generar la siguiente tabla de ventajas y desventajas para cada uno de ellos:

	Ventajas	Desventajas
Líneas/área	• Funciona bien presentando tendencias en orden cronológico • Presenta de forma clara las relaciones de datos continuos y periódicos	• No es recomendable para datos que no sean periódicos • Utilizar muchas categorías dispares puede complicar la lectura del gráfico
Barras/columnas	• Útil para presentar información categórica • Permite comparar valores entre distintas categorías o grupos	• Uso limitado para datos continuos • Limitado a una única variable • No es tan efectivo para presentar datos de series de tiempo
Dispersión	• Permite analizar la distribución de una variable	• Dependiendo de los datos, el gráfico puede quedar muy abarrotado
Histograma		• Un número de intervalos mal definido puede ocultar patrones en los datos

Cuadro 2.1: Ventajas y desventajas entre los tipos de gráficos mencionados. [9] [10]

Tarjetas

Una tarjeta puede definirse como una unidad de información autocontenida que puede mostrar uno o más gráficos, tablas, mapas u otros elementos visuales [11]. Para el contexto de este trabajo, el concepto de tarjeta se referirá específicamente a un elemento rectangular que va a indicar el valor numérico de una variable, como puede ser el valor actual de una divisa en términos de otra, el número de nuevos clientes de una empresa en el trimestre actual o el monto total de la deuda de un crédito. Este valor estará acompañado de una descripción corta que dotará de significado a dicho número.

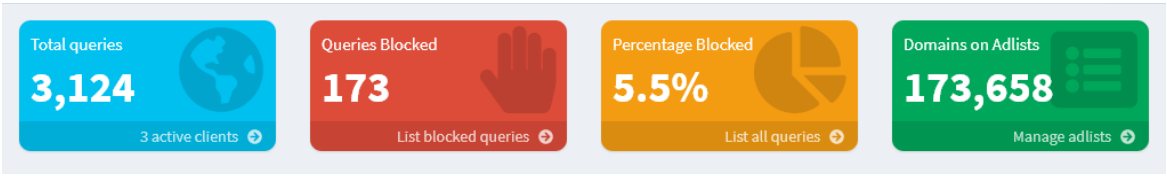

Figura 2.7: Conjunto de tarjetas, obtenido desde una instancia de la aplicación Pi-hole.

2.3. Trabajo previo

Para la realización de este proyecto de memoria de título, fue fundamental el trabajo desarrollado por Braulio Quiero, magíster en Ciencias de la Computación de la Universidad de Concepción, quien desarrolló una herramienta gráfica para la generación de *dashboards* orientados al monitoreo de salud estructural (SHM, del inglés *Structural Health Monitoring*) en construcciones, orientándose específicamente a puentes [12]. Su trabajo consistió en desarrollar un DSL orientado a definir los tipos de sensores utilizados en el monitoreo de puentes,

los cuales funcionan como fuentes de datos para la generación de *dashboards* que muestren la información obtenida por dichos sensores por medio de elementos como tarjetas, gráficos y tablas.

De manera adicional al DSL, también fue desarrollada una herramienta de modelado y generación de código, denominada Vis4bridge, que permite a usuarios sin mayores conocimientos de programación desarrollar visualizaciones de datos mediante un lenguaje de modelado gráfico que, además, permite la generación automática de código, pudiendo así desarrollar *dashboards* sin la necesidad de estar familiarizado con lenguajes de programación o bibliotecas de visualización de datos.

Este proyecto sirvió de base para la realización de este proyecto, cuyo propósito fue desarrollar una herramienta que permita la generación de visualizaciones de datos para cualquier contexto en particular, por medio de un DSL más generalizado y una refactorización de la estructura para representar los elementos que contendrán los *dashboards* a generar.

Capítulo 3

Arquitectura de la propuesta

No ha sido posible dentro de la industria establecer una definición exacta para el concepto de arquitectura de *software*. Sin embargo, en el contexto de este informe, se considerará como la estructura o estructuras de un sistema, integradas por componentes de *software*, sus características visibles desde el exterior y las interconexiones entre dichos componentes. Las çaracterísticas visibles desde el exterior"hacen referencia a las expectativas que otros componentes pueden tener de uno en particular, incluyendo los servicios que ofrece, aspectos de rendimiento, gestión de errores, utilización de recursos comunes, entre otros aspectos.

La arquitectura que utilizará la propuesta fue definida por medio de un diagrama de modelo C4, un tipo de notación utilizada para definir la arquitectura de *software* de un sistema, el cual la presenta en múltiples niveles, desde el más amplio (diagrama de contexto de sistema, que muestra los distintos sistemas de la plataforma) al más específico (diagrama de código, que puede consistir de un diagrama de clases UML). [13]

Para el diagrama C4 creado para este proyecto, fueron utilizados sólo los primeros tres niveles (contexto de sistema, contenedores y componentes). El cuarto nivel (código) fue omitido, pues será definido en los capítulos 4 y 5 por medio del metamodelo utilizado y la definición del sistema de generación automática de código, respectivamente.

Para el primer nivel (de contexto), el diagrama es el siguiente:

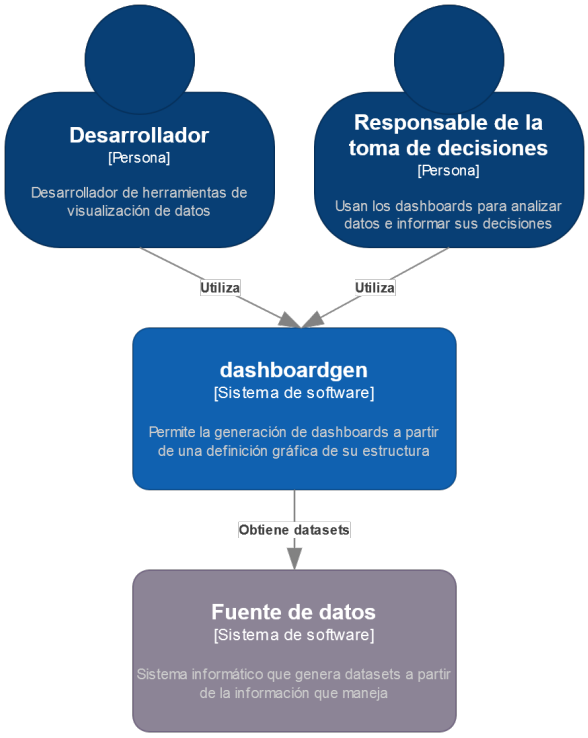

Figura 3.1: Diagrama de contexto de la arquitectura C4 propuesta.

En el diagrama se pueden apreciar dos tipos de *stakeholders*: los desarrolladores, quienes estarán a cargo de generar herramientas de visualización de datos y los responsables de la toma de decisiones, quienes utilizarán las herramientas generadas por los desarrolladores para informarse sobre los sistemas que administran y, así, tomar decisiones informadas.

Con respecto a los sistemas de *software*, están la aplicación como tal (dashboardgen), la cual va a permitir la generación de herramientas de visualización de datos por medio de *dashboards* y las fuentes de datos, las cuales generan *datasets* con la información que reciban, la cual será consumida por el sistema de dashboardgen.

Las fuentes de datos corresponden, para la implementación actual de la herramienta, a archivos de texto en formato CSV, los cuales pueden ser generados por el sistema de *software* del cual se desea analizar sus datos. Estos archivos deben incluir una fila de cabecera, la cual nombre a cada una de las columnas definidas. Además, los tipos de datos soportados para las columnas son, a saber: valores numéricos, cadenas de caracteres y *timestamps* (formato UNIX e ISO 8601).

Con respecto al segundo nivel (de contenedores), la propuesta contempla lo siguiente:

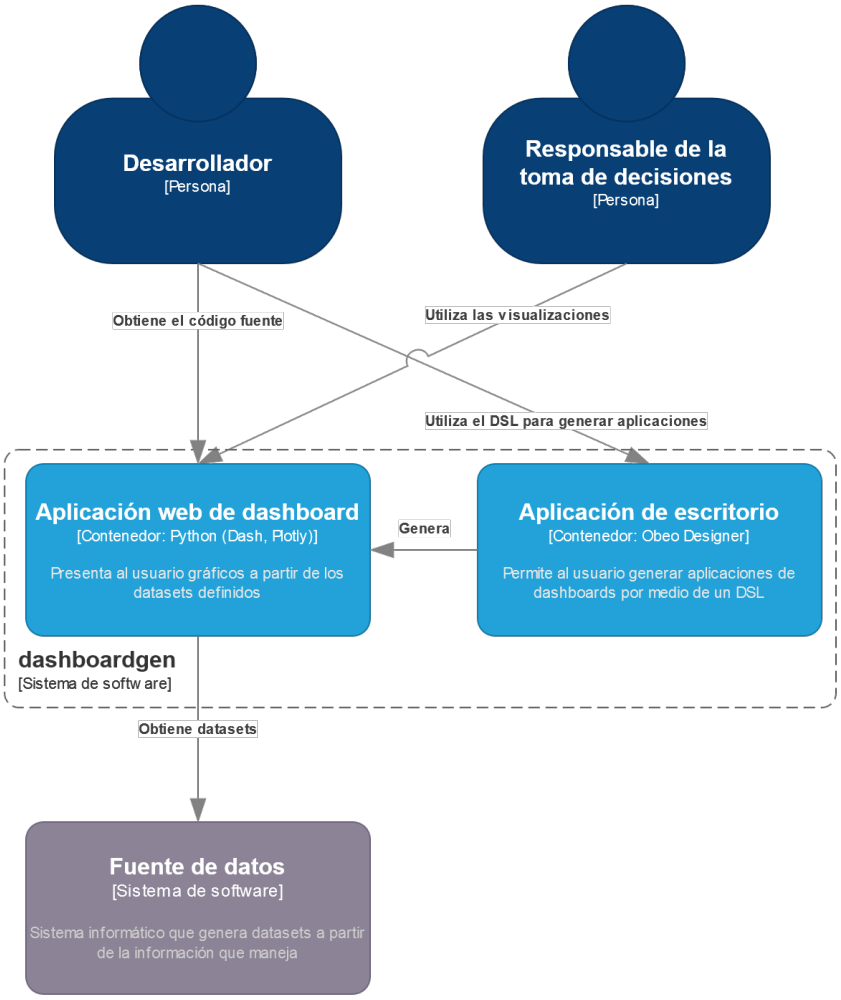

Figura 3.2: Diagrama de contenedores de la arquitectura C4 propuesta.

El sistema de dashboardgen se descompone en dos contenedores: una aplicación de escritorio, la cual es utilizada por los desarrolladores para generar aplicaciones de *dashboard* por medio de un lenguaje específico de dominio (DSL) gráfico, utilizando el entorno de desarrollo integrado (IDE) Obeo Designer como base, y una aplicación web de *dashboard* en Python, generada a partir del DSL descrito en la aplicación de escritorio. El código generado para la aplicación web será obtenido por el desarrollador, el cual podrá usarse para publicar la aplicación y que, así, las visualizaciones que genere puedan ser utilizadas por el responsable de la toma de decisiones. Para lograr esto, la aplicación va a obtener los *datasets* necesarios de la fuente de datos y convertirlos en elementos gráficos.

Finalmente, para el tercer nivel (de componentes), su estructura es la siguiente:

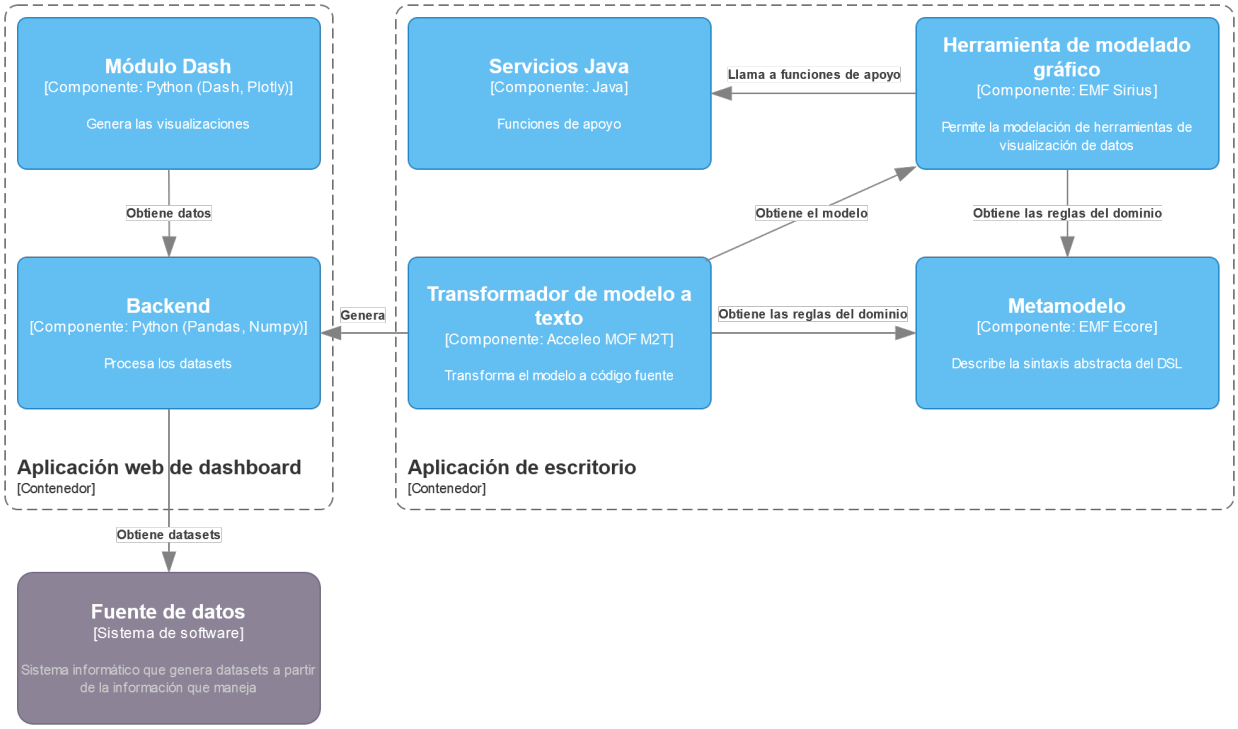

Figura 3.3: Diagrama de componentes de la arquitectura C4 propuesta.

La aplicación web hace uso de dos componentes: un *backend* a cargo del procesamiento de los *datasets*, utilizando las bibliotecas de Python Pandas y NumPy, y el módulo Dash, el cual utiliza los valores procesados por el *backend* para generar las visualizaciones y presentárselas al usuario por medio de una página web que utiliza los componentes gráficos provistos por la biblioteca del mismo nombre.

Capítulo 4

Definición del lenguaje específico de dominio

Para este proyecto, fue necesario desarrollar un lenguaje específico de dominio (DSL) que pudiera representar de manera suficientemente descriptiva el dominio del problema a resolver. Así, la sintaxis abstracta quedó definida por medio de un metamodelo, que contiene los distintos elementos del dominio y sus relaciones. Por otro lado, la sintaxis concreta fue definida como un lenguaje gráfico que representara de manera visual los elementos del dominio. Y, finalmente, la semántica se definió como un conjunto de reglas de transformación de modelo a texto (M2T, *Model to Text* en inglés), las cuales permiten generar una aplicación ejecutable que contenga las visualizaciones definidas por el usuario con la sintaxis concreta.

Siguiendo la misma estructura que la Figura 2.1, la Figura 4.1 muestra los elementos utilizados para definir el DSL definido para este proyecto.

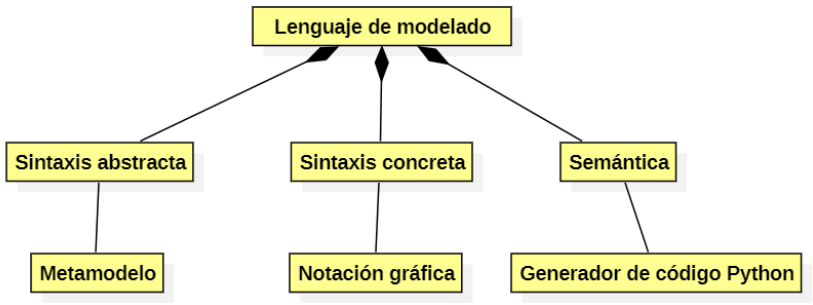

Figura 4.1: Diagrama del lenguaje específico de dominio a implementar.

4.1. Análisis del dominio

Para determinar las abstracciones necesarias que definirán la sintaxis abstracta del DSL, es necesario realizar un análisis del dominio en cuestión. Para esto, se utilizó como base para este proyecto el metamodelo definido en [14], al tener este ya determinada una estructura para la generación de las visualizaciones, sólo teniendo que realizar adaptaciones al metamodelo

para generalizar su funcionamiento a cualquier fuente de datos tabular y agregarle otras funcionalidades.

Con respecto a la generalización del metamodelo para trabajar con fuentes de datos tabulares, se realizó una búsqueda bibliográfica con el objetivo de encontrar proyectos que hayan implicado definir fuentes de datos tabulares de forma flexible en un metamodelo. Sin embargo, no fue posible encontrar mayor información al respecto, siendo lo más cercano a algo útil el trabajo realizado en [15], donde se define un metamodelo de *datasets* utilizado para transformaciones de modelo a modelo. Utilizando ese metamodelo como referencia, se definió una especificación que permita definir la estructura de un *dataset* para la generación de visualizaciones de datos.

4.2. Definición del metamodelo

En esta sección se va a presentar la estructura del metamodelo, definido a partir del modelo de características descrito en la sección anterior. Para esto, el metamodelo fue construido por medio de un diagrama de clases UML, utilizando Eclipse Ecore. El diagrama completo del metamodelo está disponible en el Apéndice B.

La estructura del metamodelo se puede dividir en dos partes: una que permite definir los *datasets* a utilizar en el modelo, junto con su estructura, y otra para las visualizaciones del *dashboard* generado. Ambas partes juntas permiten generar una aplicación completa, capaz de procesar uno o múltiples *datasets* y generar gráficos y tarjetas con los datos que contienen. Para esto, el diagrama incorpora las metaclases `View` y `DataSource`, las cuales emanan de la metaclase principal `Dashboard`, la cual representa el *dashboard* a generar.

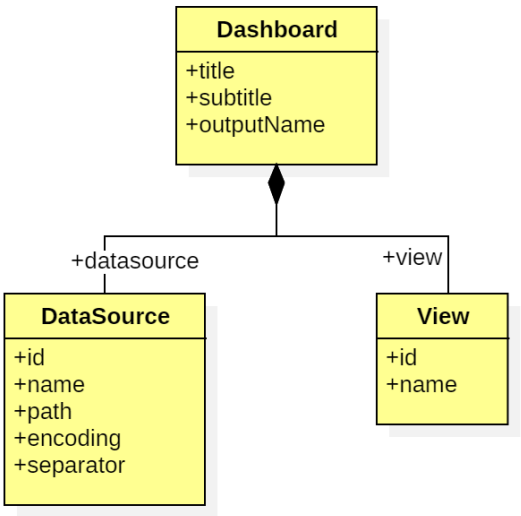

Figura 4.2: Metaclase principal del *dashboard*.

La metaclase `Dashboard` permite definir un título y un subtítulo para este, además de definir un nombre personalizado para el ejecutable Python a generar, siendo este por defecto `app.py`.

Estructura de los *datasets*

Para la definición de los *datasets* en el metamodelo, se hace uso de la metaclase `DataSource`, la cual corresponde a una fuente de datos cualquiera. Una instancia de `DataSource` apuntará a un *dataset* en específico, debiendo ser actualmente un archivo CSV almacenado en disco, el cual puede ser accedido mediante un atributo `path`, que apunta a un determinado fichero en el sistema de archivos. Adicionalmente, existen los atributos `encoding` y `separator`, los cuales permiten definir la codificación de caracteres utilizada en el archivo (UTF-8 por defecto) y el separador entre columnas (el carácter `,` por defecto). Es necesario que el archivo CSV incluya una fila de cabecera, la cual indique el nombre de todas sus columnas.

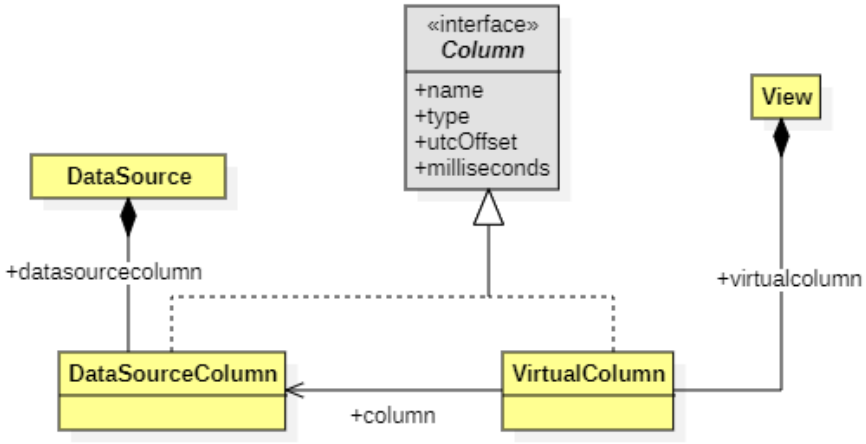

Figura 4.3: Diagrama de las fuentes de datos.

Cada `DataSource` se compone de una o múltiples instancias de `DataSourceColumn`, metaclase que implementa la metainterfaz `Column` (cuyo funcionamiento se explica más adelante), que corresponden a las columnas a utilizar del *dataset*. No es necesario que todas las columnas del archivo tengan su correspondiente instancia de `DataSourceColumn`; el usuario puede especificar sólo las que necesite usar. Es posible indicar para cada columna el tipo de dato que contiene, además de definirle un nombre, el cual debe corresponder al utilizado en el archivo CSV para identificar la columna.

El tipo de una columna se representa mediante la enumeración `DataSourceColumnType`, que contiene los siguientes literales:

- **`STRING`**: Una cadena de caracteres.
- **`NUMBER`**: Un número, ya sea entero o de coma flotante.
- **`TIMESTAMP_ISO8601`**: Un valor de tiempo, utilizando la representación indicada por la norma ISO 8601.
- **`TIMESTAMP_UNIX`**: Un valor de tiempo, representado como un número, cuyo valor corresponde a la diferencia de tiempo entre el 1 de enero de 1970 a la medianoche (UTC) y el momento en cuestión.

Para las columnas de tipo `TIMESTAMP_UNIX`, la metaclase también contiene el atributo `utcOffset`, que permite indicar la zona horaria utilizada para representar los valores de tiempo de cada columna. El desfase con respecto a UTC se representa como un flotante que indica la diferencia en horas. De esta forma, una columna que contiene valores en UTC debe usar el valor `0.0` en `utcOffset`. En cambio, para valores en UTC-4, `utcOffset` debe valer `-4.0`. Para zonas horarias no entreras, como la hora estándar de la India (UTC+5:30), el valor sería `5.5`, representando los minutos como fracciones de hora. Adicionalmente, también existe el *flag* booleano `milliseconds`, que indica si los valores de la columna están representados en milisegundos en lugar de segundos.

Columnas virtuales

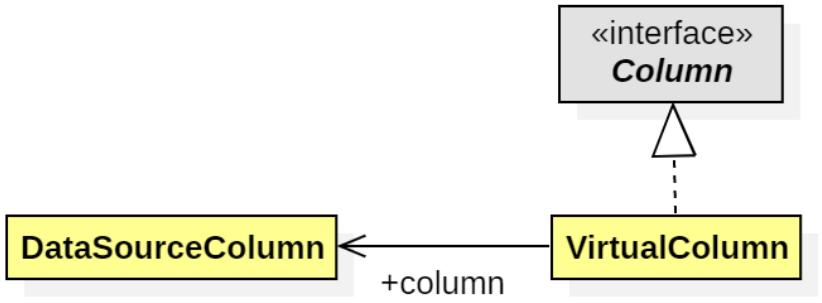

Figura 4.4: Diagrama de las columnas virtuales.

Como se mencionó anteriormente, la metaclase `DataSourceColumn` implementa la metainterfaz `Column`. La idea de utilizar una interfaz es permitir la creación de columnas virtuales, las cuales van a rescatar información de una columna real (una instancia de `DataSourceColumn`) y pueden ser ocupadas como si fueran directamente instancias de `DataSourceColumn`. Para lograr esto, se creó la metaclase `VirtualColumn`, que también implementa a `Column`, pero a la vez referenciando a una `DataSourceColumn`, comportándose de forma idéntica a una al momento de generar el código del *dashboard*.

La diferencia entre `VirtualColumn` y `DataSourceColumn` es que la primera permite definir transformaciones, por medio de una composición de una o más instancias de `Transformation`, metaclase que será explicada en la siguiente subsección. El propósito de esto es permitir el procesamiento previo de los datos del *dataset* antes de graficarlos, pudiendo reutilizarse estos datos procesados en múltiples gráficos o series.

Al momento de realizar la generación automática de código, como se explica en el siguiente capítulo, en lugar de utilizar los atributos que `VirtualColumn` implementa de `Column`, se utilizan los correspondientes a la columna referenciada por `column`, haciendo así que, en la práctica, la columna virtual actúe como si fuera una columna real del *dataset*.

Estructura del *dashboard*

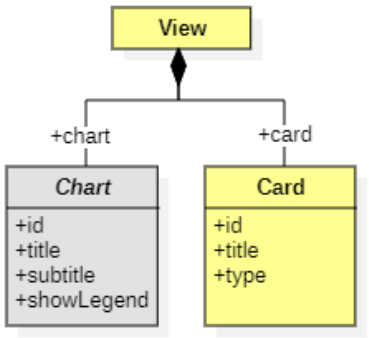

Figura 4.5: Diagrama del *dashboard*.

La metaclase `Dashboard`, elemento raíz de este diagrama, representa un *dashboard* a generar con esta herramienta. Contiene los atributos `title` y `subtitle`, cadenas de caracteres que indican el título y subtítulo que tendrá el *dashboard* generado. Además, por medio de otro parámetro (`outputName`, también una cadena de caracteres), se puede asignar el nombre de archivo del *script* que contendrá el *dashboard*, teniendo por defecto el valor `app.py`.

Una instancia de `Dashboard` se compone de múltiples vistas (`View`), que corresponden a pantallas que puede visualizar el usuario. Una vista puede contener múltiples gráficos (`Chart`) y tarjetas (`Card`), así como requiere estar asociada a un *dataset* en particular, el cual será utilizado por los elementos que contenga. Además, contienen un atributo identificador (`id`) y un nombre (`name`), el cual se utilizará como título para distinguir las vistas entre sí.

Gráficos

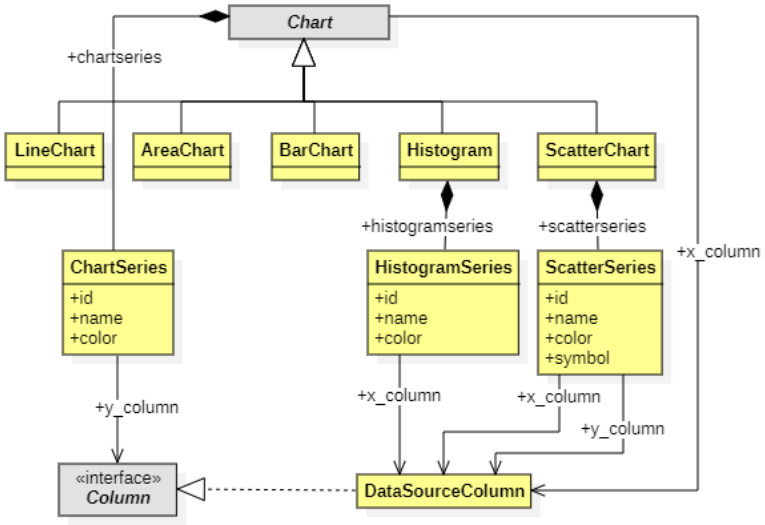

Figura 4.6: Diagrama de los gráficos.

Los gráficos que puede contener un modelo son clases que heredan de `Chart`, donde cada una representa un tipo de gráfico distinto. Actualmente, el metamodelo define cinco tipos de gráfico, a saber:

- **Gráfico de líneas (`LineChart`)**: Contiene una o más líneas de datos, representadas mediante pares de valores (x, y).
- **Gráfico de área (`AreaChart`)**: Similar a un gráfico de líneas, pero el área debajo de cada línea también es coloreado.
- **Histograma (`Histogram`)**: Permite visualizar distribuciones de valores como barras verticales.
- **Gráfico de dispersión (`ScatterChart`)**: Representa pares de valores (x, y) como puntos en el plano cartesiano.
- **Gráfico de barras (`BarChart`)**: Utiliza barras (rectangulares o verticales) para graficar las magnitudes que se le indican. Puede ser utilizado con pares de valores (x, y) (como si fuera un gráfico de líneas o área) o para asociar valores numéricos a valores categóricos (p. ej. el producto interno bruto en dólares de un país).

Además, todos los tipos de gráfico, a excepción de los gráficos de dispersión e histogramas, referencian a una columna del *dataset* asociado (`x_column`), para utilizar sus valores para la componente de las abscisas. Todo gráfico deberá contener una o más series, las cuales proveerán los valores a representar en pantalla.

Para los gráficos de líneas, área y barras, se utiliza la metaclase `ChartSeries`, la cual va a representar a una única línea, área o conjunto de barras, respectivamente. Cada serie referencia a una columna del `dataset` asociado a la vista, de la cual extraerá sus valores, utilizándolos como valores para las ordenadas, combinándolos con la columna de las abscisas indicada en el gráfico para generarlo. En el caso de los gráficos de barras, sus series también pueden utilizar columnas virtuales para obtener sus valores.

Para el caso de los gráficos de dispersión, a pesar de referenciar también a una columna para los valores de X, esta es ignorada. Los puntos que contendrá el gráfico se definen únicamente a través de una o más series de dispersión (`ScatterSeries`) que contendrá cada instancia de `ScatterChart`. Estas series contendrán las referencias a columnas para definir sus valores de X e Y (`x_column` e `y_column`, respectivamente).

Por otro lado, los histogramas pueden contener una o más series de histograma (`HistogramSeries`), donde cada una apunta a una columna del *dataset* (`x_column`) para analizar su distribución. Al igual que con los gráficos de dispersión, la referencia a la columna del eje X en el gráfico es ignorada.

Los tres tipos de series contienen atributos identificadores (`id`) y un nombre para mostrar en pantalla (`name`), ambos cadenas de caracteres. En el caso de las series de dispersión, se incluye también un atributo `symbol`, que indica el tipo de símbolo a utilizar para los puntos en el gráfico. Los valores posibles para este están contenidos en la enumeración `ScatterSymbol`, que contiene los siguientes valores:

- **Círculos** (`CIRCLE`)

- **Cuadrados** (`SQUARE`)
- **Diamantes** (`DIAMOND`)
- **Estrellas** (`STAR`)
- **Equis** (`X`)
- **Asteriscos** (`ASTERISK`)

Adicionalmente, todos los tipos de serie también pueden especificar el color que van a utilizar en el gráfico. Los colores disponibles para tales efectos están contenidos en la enumeración `SeriesColor`, que contiene los siguientes valores:

- Automático[1] (`AUTO`)
- Rojo (`RED`)
- Azul (`AZUL`)
- Verde (`GREEN`)
- Morado (`PURPLE`)
- Naranja (`ORANGE`)
- Amarillo (`YELLOW`)
- Café (`BROWN`)
- Rosa (`PINK`)
- Gris (`GRAY`)

Por defecto, el color utilizado por todas las series es `AUTO`.

Tarjetas

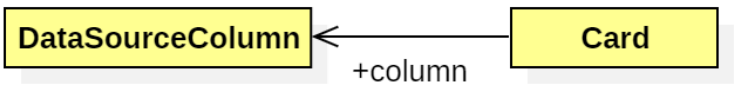

Figura 4.7: Diagrama de las tarjetas.

Las tarjetas corresponden a indicadores numéricos simples, cuyo valor se obtiene a partir de una operación determinada sobre los elementos de una columna. Toda tarjeta contiene un identificador (`id`), un título (`title`) y un tipo (`type`). Los tipos válidos para una tarjeta están contenidos en la enumeración `CardType`, a saber:

[1] De utilizar este valor, el color de la serie quedará determinado por Plotly al momento de mostrar el gráfico. De haber múltiples series usando el color `AUTO`, Plotly les asignará colores diferentes para poder distinguirlos.

- **Promedio** (`AVERAGE`): Calcula el valor promedio de la columna.
- **Mínimo** (`MIN`): Calcula el valor mínimo de la columna.
- **Máximo** (`MAX`): Calcula el valor máximo de la columna.

Toda tarjeta requiere estar conectada a una columna del *dataset*, por medio de su atributo (`column`).

Transformación de datos

Para el caso de los gráficos de barra, puede existir la necesidad de procesar los datos antes de ser presentados, considerando, por ejemplo, que los *datasets* pueden consistir de datos desagregados, desordenados o derechamente en bruto. Sabiendo lo versátiles que pueden ser los gráficos de barra para presentar información, se consideró necesario definir procesos para transformar los valores asociados a cada serie antes de ser mostrados al usuario. De esta forma, el usuario tendrá la posibilidad de definir gráficos más complejos e interesantes, que no solo tomen información del *dataset* y la grafique directamente, sino que puedan rescatar más valor de los datos originales.

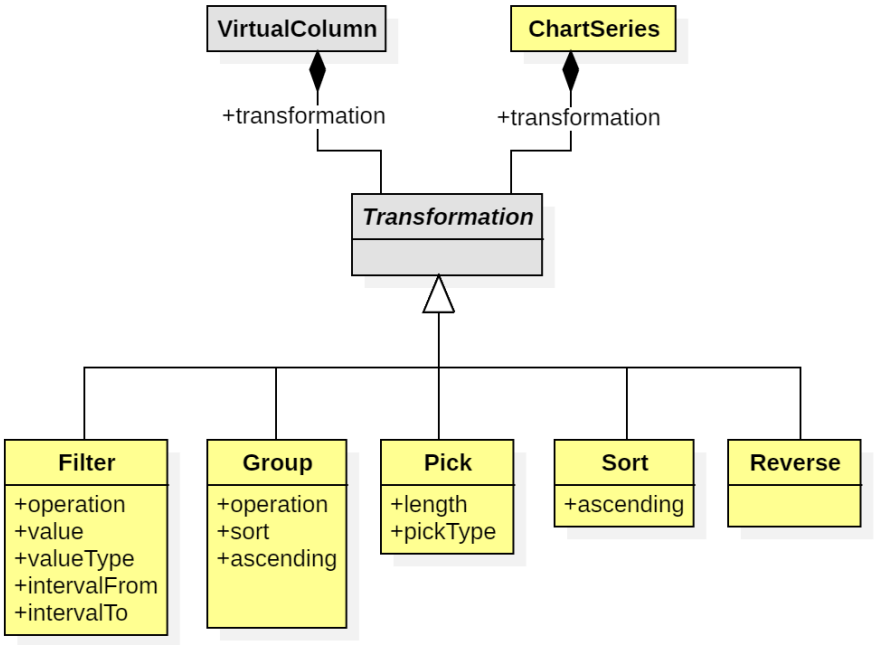

Figura 4.8: Diagrama de las transformaciones.

Para lograr esto, el metamodelo incorpora a `Transformation`, una metaclase abstracta que representa todos los tipos de transformación disponibles. Los tipos de transformación, que pueden entenderse como operaciones sobre los datos de cada serie, además de los atributos que utilizan, son los siguientes:

- **Filtrar** (`Filter`): Filtra filas según el valor de una determinada columna y una operación asociada.

- **operation**: Indica el tipo de operación para el filtro. Las operaciones soportadas están definidas en la enumeración `FilterType`, con los siguientes valores:
  - **Igual** (`EQUAL`)
  - **Mayor que** (`GREATER`)
  - **Menor que** (`LESS`)
  - **Mayor o igual que** (`GREATER_EQUAL`)
  - **Menor o igual que** (`LESS_EQUAL`)
  - **No igual** (`NOT_EQUAL`)
  - **Intervalo** (`INTERVAL`)
- **value**: Contiene el valor a comparar. Es utilizado en todas las operaciones excepto `INTERVAL`.
- **valueType**: Indica el tipo de valor de `value`. Soporta los mismos valores que `DataSourceColumnType`.
- **intervalFrom**: Si la operación es `INTERVAL`, el valor de este atributo es utilizado para definir el inicio del intervalo (cerrado).
- **intervalTo**: Al igual que `intervalFrom`, este valor se utiliza para definir el fin del intervalo (cerrado).
- **column**: Apunta a la columna cuyos valores van a ser comparados por el filtro.

- **Agrupar** (`Group`): Agrupa filas según el valor de una columna.
  - **operation**: Indica el tipo de agrupamiento a realizar. Los tipos de agrupamiento soportados son los siguientes, indicados en la enumeración `GroupType`:
    - **Promedio** (`AVERAGE`): Reemplaza múltiples filas por una que contenga el promedio de una columna de comparación.
    - **Mínimo** (`MIN`): Reemplaza múltiples filas por una que contenga el mínimo de una columna de comparación.
    - **Máximo** (`MAX`): Reemplaza múltiples filas por una que contenga el máximo de una columna de comparación.
    - **Cantidad** (`COUNT`): Reemplaza múltiples filas por una que contenga la cantidad de veces que se repite el valor de una columna. Para este de tipo de operación, se le agrega una nueva columna `value` al *dataset*, la cual va a contener los valores de cantidad. El archivo CSV no será modificado en este caso; la columna nueva sólo es almacenada en memoria.
  - **sort**: *Flag* booleano que permite ordenar los valores una vez agrupados.
  - **ascending**: Si el ordenamiento está activado, este *flag* indica si el ordenamiento es ascendente o descendente.
  - **grouping_column**: Este atributo apunta a la columna que será utilizada para generar las agrupaciones (múltiples filas con el mismo valor en esta columna serán agrupadas).
  - **comparing_column**: Si la operación es `AVERAGE`, `MAX` o `MIN`, los valores de esta columna serán utilizados para computar el promedio, máximo o mínimo, respectivamente. No es utilizado si la operación es `COUNT`.

- **Escoger** (`Pick`): Permite escoger un número determinado de valores del inicio o final de una serie.
  - **`length`**: Indica la cantidad de elementos a escoger.
  - **`pickType`**: Indica el extremo desde el que se van a extraer los valores. Utiliza la enumeración `PickType`, con los valores `HEAD` (extraer desde el inicio) y `TAIL` (extraer desde el final).
- **Ordenar** (`Sort`): Ordena las filas de la serie utilizando el valor de alguna de las columnas como punto de referencia.
  - **`ascending`**: Indica si el ordenamiento es ascendente o descendente.
  - **`column`**: Apunta a la columna que usará para el ordenamiento. Puede tomar un valor nulo, si el usuario desea ordenar con respecto a un agrupamiento por cantidad previo en la secuencia de transformaciones.
- **Invertir** (`Reverse`): Invierte el orden en el que se presentan los valores.

Debido a la naturaleza de estas transformaciones, el orden en el que se representan es importante, ya que no todas estas operaciones son conmutativas. Para ilustrar este ejemplo, se observa a continuación un *dataset* de ejemplo:

nombre	nota
Martín Pérez	3,4
Martín Pérez	3,4
Claudia Navarro	1,2
Claudia Navarro	5,3
Claudia Navarro	3,1
Antonia Gutiérrez	4,7

Cuadro 4.1: *Dataset* de ejemplo.

Si se agrupan las filas del *dataset* respecto a su `nombre`, dejando solamente la `nota` más baja, se obtiene lo siguiente:

nombre	nota
Martín Pérez	3,4
Claudia Navarro	1,2
Antonia Gutiérrez	4,7

Cuadro 4.2: *Dataset* de ejemplo después de agrupar por `nombre`, dejando la `nota` mínima.

Luego, si se eliminan las filas con `nota` menor a 2, utilizando un filtro, el *dataset* queda así:

nombre	nota
Martín Pérez	3,4
Antonia Gutiérrez	4,7

Cuadro 4.3: *Dataset* de ejemplo después de filtrar las filas con `nota` mayor a 2.

Por otro lado, si al *dataset* del Cuadro 4.1 se le aplica primero el filtro, se obtiene este resultado:

nombre	nota
Martín Pérez	3,4
Martín Pérez	3,4
Claudia Navarro	5,3
Claudia Navarro	3,1
Antonia Gutiérrez	4,7

Cuadro 4.4: *Dataset* de ejemplo después filtrar las `nota`s mayores a 2.

Luego, si se agrupan las filas de la misma forma que en el Cuadro 4.2, el *dataset* queda así:

nombre	nota
Martín Pérez	3,4
Claudia Navarro	3,1
Antonia Gutiérrez	4,7

Cuadro 4.5: *Dataset* de ejemplo después de agrupar las filas por `nombre`, dejando la `nota` mínima.

Como se puede apreciar, los *datasets* resultantes de los Cuadros 4.3 y 4.5 no son idénticos, pues aunque se hayan aplicado las mismas transformaciones, estas no se efectuaron en el mismo orden.

Visto esto, en el metamodelo, la metaclase `Transformation` es utilizada por `ChartSeries` y `VirtualColumn`, en ambos casos por medio de una composición de uno o más elementos. De esta forma, se almacena una secuencia de transformaciones que pueden aplicarse paso a paso. Como se mencionó anteriormente, la idea de utilizar columnas virtuales y que estas puedan contener transformaciones por sí mismas es para permitir separar el *pipeline* de transformaciones que desee implementar el usuario en dos etapas: una de preparación y una de presentación. Así, se puede generar una secuencia inicial de transformación de los datos que puede ser reutilizada en múltiples partes de la misma vista, ya sea en distintas series del mismo gráfico o en gráficos separados. Posteriormente, cada serie puede contener sus propias transformaciones, las cuales van a modificar los datos según los requerimientos de cada una, usando como base las transformaciones indicadas en la columna virtual misma.

La ventaja de utilizar columnas virtuales y no simplemente agregar las transformaciones a las columnas directamente es permitir acceder a los tanto a los valores de una columna como a los valores ya procesados, en caso de que sea necesario graficar ambos.

Resumen

Un resumen sobre los elementos utilizados para definir el DSL descrito para este proyecto se puede encontrar en el Apéndice A.

4.3. Representación gráfica del lenguaje específico de dominio

Una vez definidas las reglas que darán forma al DSL implementado, es posible definir una representación gráfica para este, la cual permitirá a los desarrolladores determinar la estructura de sus *dashboards* en un lenguaje visual y más intuitivo, aprovechándose de la abstracción con el que fue construido el DSL con el objetivo de no tener que enfocarse en los aspectos de código de la aplicación a generar, ya que será el sistema de generación automática de código quien se encargará de esos detalles.

Dentro de la aplicación, los elementos a representar gráficamente son principalmente los gráficos, tarjetas, fuentes de datos, además de sus relaciones y elementos contenedores.

Gráficos

Los gráficos se representan como contenedores, elementos en los que pueden agregarse otros elementos. Estos contenedores, al igual que la mayor parte de los componentes descritos en esta sección, hacen uso de íconos para mostrarle al usuario qué tipo de gráfico representan, a la vez que igual indican en una etiqueta el nombre del gráfico. Los íconos corresponden a representaciones simplificadas de los tipos de gráficos soportados, como se puede ver en los siguientes ejemplos:

Figura 4.9: En el sentido de las agujas del reloj, empezando desde la esquina superior izquierda: Íconos utilizados para los gráficos de dispersión, gráficos de área, gráficos de barras, histogramas y gráficos de línea.

Casi todos los íconos utilizados fueron diseñados originalmente para la *suite* de ofimática de código abierto LibreOffice, siendo algunos adaptados para reflejar mejor las necesidades del proyecto.

Dentro de cada gráfico pueden incluirse una o más series, las cuales son representadas mediante rectángulos u otros contenedores. En el caso de los gráficos de área, líneas, barras e histogramas, sus series son representadas con rectángulos, los que incluyen la etiqueta de la serie más un ícono. Adicionalmente, dicho rectángulo tendrá adosado a un costado un rectángulo para mostrar la columna (virtual o real) que utiliza para obtener sus valores. Para el caso de los gráficos de barras, también podrá incluirse un círculo rojo en una esquina, el cual indica que la serie hace uso de transformaciones para procesar sus datos. Por otro lado, los gráficos de dispersión hacen uso de un subcontenedor, en el cual se incluyen dos rectángulos, utilizados para indicar las columnas a utilizar para los valores del eje X e Y.

Adicionalmente, todos los gráficos, a excepción de los de dispersión e histogramas, deben incluir un rectángulo de color rojo, el cual indica la columna ocupada para los valores del eje X en los gráficos. Análogamente, las series de dichos gráficos utilizan rectángulos de color verde para indicar que sus columnas representan los valores del eje Y. Los histogramas funcionan de forma similar, pero sólo utilizando rectángulos de color rojo en sus series ya que, al ser gráficos basados en frecuencia, no necesitan de otra dimensión para representar sus valores.

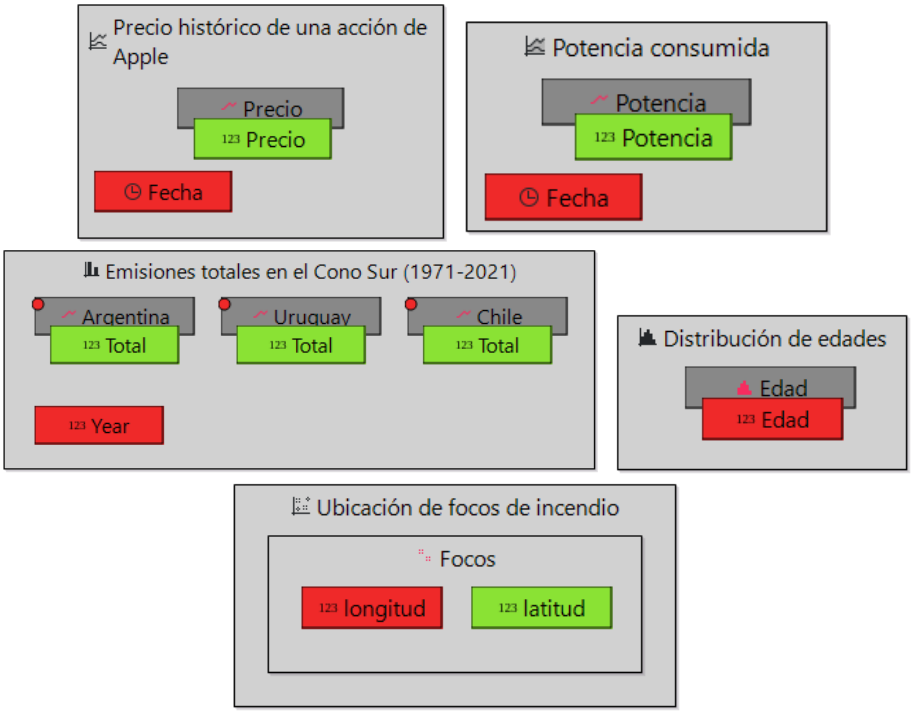

Figura 4.10: En el sentido de las agujas del reloj, desde la esquina superior izquierda: Gráfico de líneas, gráfico de área, histograma, gráfico de dispersión, gráfico de barras.

4.4. Tarjetas

Las tarjetas se también se representan como contenedores, los cuales indican en su etiqueta de título tanto el título de la tarjeta como el tipo de operación que utilizan. Dentro del contenedor se incluye una única columna, representada por un rectángulo morado, la cual indica la columna utilizada para calcular los valores de la tarjeta.

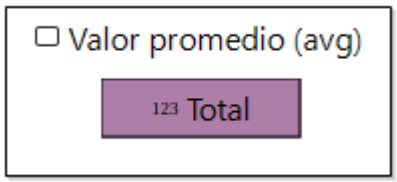

Figura 4.11: Elemento de tarjeta en la representación visual.

4.5. Vistas

Las vistas corresponden a contenedores de mayor nivel, los cuales pueden contener múltiples gráficos y tarjetas. Cada vista puede entenderse como una pantalla separada, la cual dota de un contexto común a los elementos que contiene.

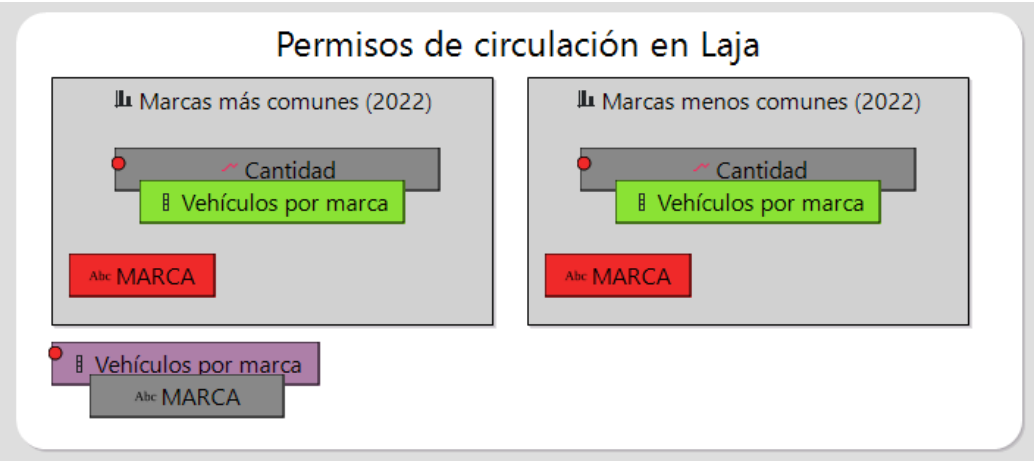

Figura 4.12: Elemento de vista en la representación visual.

Dependiendo de las necesidades del desarrollador, una vista, además de poder incluir gráficos y tarjetas, puede incluir columnas virtuales, las cuales pueden ser reutilizadas por múltiples gráficos de barras, como una forma de preprocesar sus valores antes de ser presentados. Cada columna virtual se representa como un rectángulo morado con una columna adosada a un costado por medio de otro rectángulo de color gris. Al igual que las series de gráfico, una columna virtual va a incluir un círculo rojo en una esquina si esta contiene transformaciones asociadas.

4.6. Fuentes de datos

Las fuentes de datos, al igual que las vistas, son contenedores, esta vez de color gris claro, que representan los *datasets* utilizados. Cada uno equivale a un único *dataset*, el cual va a mostrar un nombre descriptivo más un ícono y una lista de columnas, donde cada columna utiliza un ícono diferente para representar el tipo de valor que representa.

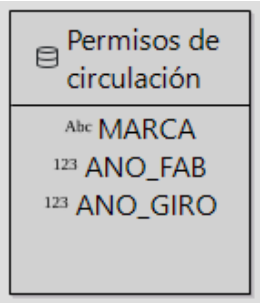

Figura 4.13: Elemento de fuente de datos en la representación visual.

4.7. Relaciones entre elementos

La mayor parte de las relaciones entre los elementos mencionados se representan directamente como objetos anidados. Así, una serie va a estar inequívocamente asociada a un gráfico ya que va a estar contenido dentro de este, al igual que un gráfico va a estar contenido dentro de una vista. Sin embargo, la relación entre fuentes de datos y las vistas no es representada de esta forma, sino a través de flechas, las cuales indican que una vista determinada utiliza los datos de una cierta fuente de datos. Una fuente de datos sólo puede ser utilizada por una única vista y viceversa.

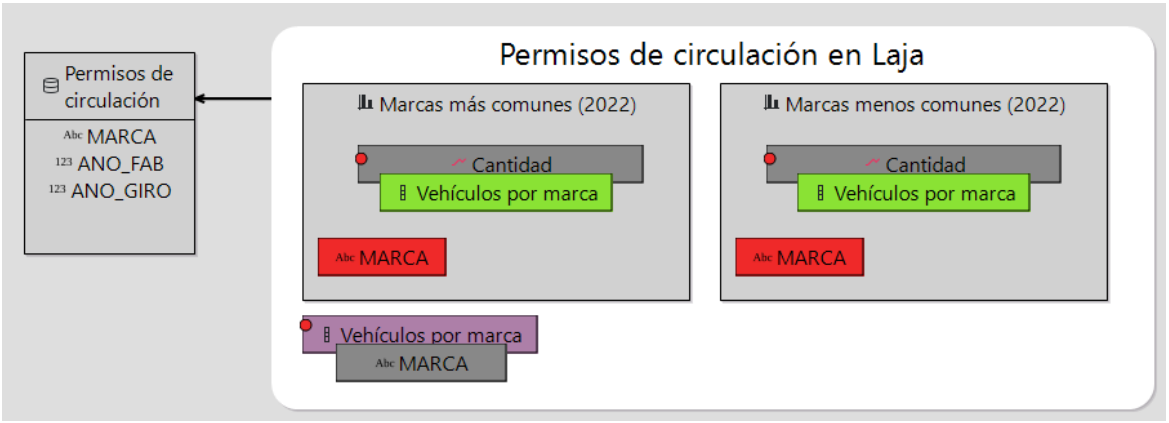

Figura 4.14: Vista conectada a una fuente de datos.

Capítulo 5

Sistema de generación automática de código

5.1. Esquema de transformación de modelo a texto

Para la generación de código a partir de un modelo cualquiera, se hace uso de un proyecto Acceleo, que define las transformaciones necesarias para generar código a partir de un modelo.

Acceleo [16] es una implementación de la norma MTL (*Model to Text Language*, Lenguaje de Modelo a Texto en inglés), definido por el Object Management Group (OMG) [17], un consorcio internacional dedicado a la publicación de normas y especificaciones relacionadas con distintas ramas del desarrollo de software en diversas industrias.

La implementación realizada permite la generación de una aplicación Python, utilizando las bibliotecas Dash y Plotly para la creación de los elementos gráfico del *dashboard* final. De todas formas, considerando la separación que existe entre el DSL y la transformación de modelo a código, sería posible crear otros proyectos Acceleo que pudieran generar *dashboards* utilizando distintos lenguajes o bibliotecas.

A continuación, se explicará el funcionamiento de la transformación implementada, cuyos archivos de transformación utilizados comprenden la siguiente estructura, como se puede ver en la Figura 5.1:

```
xyz.jorgejarai.dashboardgen.acceleo/
├── files/
│   ├── dash/
│   │   ├── callbacks/
│   │   │   ├── generateCallbackAreaChart.mtl
│   │   │   ├── generateCallbackBarChart.mtl
│   │   │   ├── generateCallbackFunction.mtl
│   │   │   ├── generateCallbackHistogram.mtl
│   │   │   ├── generateCallbackLineChart.mtl
│   │   │   └── generateCallbackScatterChart.mtl
│   │   ├── cards/
│   │   │   ├── generateCard.mtl
│   │   │   └── generateCardUtilFunctions.mtl
│   │   ├── charts/
│   │   │   ├── generateAreaChart.mtl
│   │   │   ├── generateBarChart.mtl
│   │   │   ├── generateChart.mtl
│   │   │   ├── generateHistogram.mtl
│   │   │   ├── generateLineChart.mtl
│   │   │   └── generateScatterChart.mtl
│   │   └── generateView.mtl
│   ├── transformations/
│   │   ├── generateFilter.mtl
│   │   ├── generateGroup.mtl
│   │   ├── generateLiteral.mtl
│   │   ├── generatePick.mtl
│   │   ├── generateReverse.mtl
│   │   ├── generateSort.mtl
│   │   └── generateTransformations.mtl
│   ├── generateCss.mtl
│   ├── generateDashCode.mtl
│   ├── generateDataSource.mtl
│   ├── generateRequirements.mtl
│   └── generateUtilFunctions.mtl
└── main/
    ├── Services.java
    ├── generate.mtl
    └── services.mtl
```

Figura 5.1: Estructura de los archivos del generador de modelo a texto.

El proyecto Acceleo se compone de múltiples paquetes, cada uno conteniendo uno o más archivos MTL, más un par de archivos Java de apoyo. Cada archivo MTL contiene un módulo de Acceleo del mismo nombre que el archivo.

Paquete `xyz.jorgejarai.dashboardgen.acceleo.files`

Contiene los archivos base para la generación del código, los que incluyen el esqueleto del archivo Python a generar, además de otros archivos anexos a este.

- `generateCss.mtl`: Genera un archivo CSS para la personalizar los estilos de la página web que sirve la aplicación Dash. El generador crea un archivo `assets/custom.css` en el directorio raíz indicado al momento de ejecutar el proyecto sobre un modelo.
- `generateDashCode.mtl`: Genera la base del archivo Python final, incluyendo las dependencias a importar para el programa, la inicialización de Dash, las funciones de apoyo, el análisis de los *datasets* que utilizará el *dashboard*, la generación del árbol de componentes para el *dashboard* y las funciones de *callback* para la actualización de los gráficos. Para esto, llama a otros módulos, contenidos en el paquete `xyz.jorgejarai.dashboardgen.acceleo.files.dash`.
- `generateDataSource.mtl`: Genera el código para el análisis de cada uno de los *datasets* utilizados por el *dashboard*, lo que incluye la lectura del archivo CSV asociado, la especificación de las columnas a utilizar de este y la interpretación de los valores, de corresponder a marcas de tiempo.
- `generateRequirements.mtl`: Genera un archivo de requerimientos (`requirements.txt`), el cual contiene las bibliotecas de las que depende la aplicación generada. Este archivo puede ser utilizado por gestores de paquetes como `pip` para instalar las dependencias del proyecto.
- `generateUtilFunctions.mtl`: Genera funciones de apoyo, utilizadas en tiempo de ejecución para la generación de los gráficos.

Paquete `xyz.jorgejarai.dashboardgen.acceleo.files.dash`

Contiene los módulos necesarios para generar los componentes del *dashboard*, ya sean parte de la interfaz gráfica a generar o código interno para el procesamiento de los datos.

- `generateView.mtl`: Genera el código HTML para insertar una vista en el *dashboard*, incluyendo todos los gráficos y tarjetas que contenga esta.

Paquete `xyz.jorgejarai.dashboardgen.acceleo.files.dash.callbacks`

Contiene los *callbacks* utilizados para actualizar los gráficos del *dashboard*, donde cada tipo de gráfico cuenta con su propio módulo.

- `generateAreaChart.mtl`: Genera el *callback* para un gráfico de área.
- `generateBarChart.mtl`: Genera el *callback* para un gráfico de barras. Si alguna de las series contiene transformaciones, estas son utilizadas para procesar los datos asociados.
- `generateHistogram.mtl`: Genera el *callback* para un histograma.
- `generateLineChart.mtl`: Genera el *callback* para un gráfico de líneas.

- `generateScatterChart.mtl`: Genera el *callback* para un gráfico de dispersión.

Paquete `xyz.jorgejarai.dashboardgen.acceleo.files.dash.cards`

Contiene el código utilizado para generar una tarjeta. A diferencia de los gráficos, los valores de las tarjetas son generados una única vez al arrancar la aplicación, por lo que no necesitan de *callbacks*.

- `generateCard.mtl`: Inserta en el árbol de componentes el código HTML para generar una tarjeta.
- `generateCardUtilFunctions.mtl`: Genera funciones de apoyo para el cálculo de los valores de una tarjeta. Se utilizan en tiempo de ejecución para computar el mínimo, máximo o promedio de la columna asociada, una vez los *datasets* ya fueron interpretados.

Paquete `xyz.jorgejarai.dashboardgen.acceleo.files.dash.charts`

Contiene el código utilizado para generar los componentes HTML que compondrán los gráficos del *dashboard*.

- `generateAreaChart.mtl`: Genera el *callback* para un gráfico de área. Incluye, además del gráfico como tal, un selector para escoger qué series de las definidas el usuario desea mostrar. Si el gráfico utiliza un valor de tiempo para el eje X, incluirá también un selector de rango deslizante para definir el rango de tiempo a mostrar.
- `generateBarChart.mtl`: Genera el *callback* para un gráfico de barras. Incluye un selector para las series a mostrar en el gráfico/
- `generateChart.mtl`: Genera el código HTML para insertar un gráfico en el árbol de componentes de la página. Dependiendo del tipo de gráfico, llama a uno de los módulos definidos en este paquete para generarlo.
- `generateHistogram.mtl`: Genera el *callback* para un histograma. Incluye un selector deslizante para indicar el número de intervalos del histograma, además de un *checkbox* para utilizar el valor por defecto, que se calcula en función del intervalo total de los valores del histograma.
- `generateLineChart.mtl`: Genera el *callback* para un gráfico de líneas. Incluye los mismos componentes extra que un gráfico de área.
- `generateScatterChart.mtl`: Genera el *callback* para un gráfico de dispersión. Incluye un selector para escoger qué series debe mostrar el gráfico.

Paquete `xyz.jorgejarai.dashboardgen.acceleo.files.transformations`

Contiene el código utilizado para crear un *pipeline* de transformaciones, tanto para las columnas virtuales como para las series de gráficos. Cada transformación soportada cuenta con su propio módulo.

- `generateFilter.mtl`: Genera una transformación de filtro, eliminando las filas de una serie o columna virtual que no cumplan con una determinada condición.
- `generateGroup.mtl`: Genera una transformación de agrupamiento, agrupando las filas de una serie o columna.
- `generateLiteral.mtl`: Genera el literal Python correspondiente a un valor determinado, dependiendo de su tipo. Por ejemplo, un *string* se imprime en el código final envuelto entre comillas, mientras que un número puede ser insertado directamente.
- `generatePick.mtl`: Genera una transformación de elección, escogiendo los primeros o últimos n elementos de una serie o columna virtual.
- `generateReverse.mtl`: Genera una transformación de reversa, inviertiendo el órden de los elementos de una serie o columna virtual.
- `generateSort.mtl`: Genera una transformación de ordenamiento, ordenando de forma ascendente o descendente los elementos de una serie o columna virtual.
- `generateTransformations.mtl`: Función base para la generación de transformaciones dada una serie de gráfico o columna virtual. Llama a los demás modulos de este paquete dependiendo del tipo de transformación. Si la serie o columna virtual tiene más de una transformación, estas se ejecutan en órden secuencial.

Paquete `xyz.jorgejarai.dashboardgen.acceleo.main`

Contiene el archivo Acceleo principal, el cual es llamado para la generación de un *dashboard*, además de una clase Java de servicios.

- `Services.java`: Contiene un método para la generación del nombre del archivo Python con el código del *dashboard*.
- `generate.mtl`: Corresponde al punto de llamada inicial para la generación de un *dashboard*. Llama a los módulos `generateRequirements`, `generateCss` y `generateDashCode` del paquete `xyz.jorgejarai.dashboardgen.acceleo.dash`, que generan todos los archivos necesarios para el *dashboard* final.
- `services.mtl`: Genera las *queries* de Acceleo para acceder al servicio definido en `Services.java`, para así usar el método que contiene dentro de los módulos.

5.2. Aplicación generada

Generación de los artefactos

El proceso de transformación de modelo a texto sobre un modelo de *dashboard* se puede dividir en las siguientes etapas, como muestra la Figura 5.2:

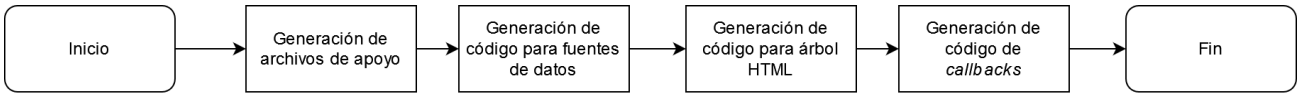

Figura 5.2: Proceso de generación de código a partir de un modelo de *dashboard*

1. **Generación de archivos de apoyo**: Incluye la generación de archivos de contenido estático (no varían en función del modelo) que utiliza la aplicación generada para su ejecución.

2. **Generación de código de para fuentes de datos**: Esta y las siguientes etapas comprenden la generación del *script* final que contendrá el *dashboard* final. En esta etapa se lee el modelo para obtener información sobre las fuentes de datos y generar el código que leerá los archivos indicados y cargará sus contenidos en memoria.

3. **Generación de código para árbol HTML**: En esta etapa, se lee el modelo para obtener información sobre las vistas, gráficos y tarjetas que compondrán el *dashboard* con el propósito de generar el código HTML necesario para desplegar los gráficos y la interfaz de usuario. A diferencia de los gráficos, como se explica a continuación, los valores de las tarjetas se generan de forma estática en esta etapa.

4. **Generación de código de *callbacks***: En esta última etapa, utilizando la información del modelo sobre los gráficos requeridos, se generan los métodos de *callback*, los cuales hacen uso de las fuentes de datos y selectores en la aplicación web, realizan transformaciones de ser necesario y retornan la información requerida en un formato entendible por la biblioteca de gráficos, los cuales serán mostrados al usuario.

Artefactos generados

Una vez es ejecutado el sistema de generación automático de código sobre un archivo de modelo, se generan múltiples archivos, a saber:

- `app.py`: El *script* de Python con el código de la aplicación. Puede tomar otro nombre (siempre incluyendo la extensión `.py`) si el usuario define un valor personalizado en la propiedad `outputName` del `Dashboard` dentro del modelo. El código contiene todas las llamadas necesarias para leer el o los archivos de los *datasets* definidos, generar las visualizaciones y procesar los datos que estas van a consumir.

- `requirements.txt`: Un archivo *lock* utilizado por el administrador de paquetes `pip` para definir las versiones de las dependencias que requiere el ejecutable.

- `assets/custom.css`: Un archivo CSS con estilos personalizados de acuerdo a las necesidades del proyecto. El nombre y ruta del archivo están definidos por la biblioteca Dash, la cual requiere que cualquier CSS personalizado deba tener ese nombre y ubicación con respecto al ejecutable, aunque el nombre del directorio puede ser personalizado por el desarrollador.

Estos tres archivos, más los *datasets* en formato CSV, son requeridos para poder ejecutar la aplicación de manera consistente entre sistemas ya que, con el archivo `requirements.txt`, es posible definir un entorno virtual de Python (`venv`), lo que permite generar una copia del intérprete de Python y su administrador de paquetes, la cual pueda tener instaladas las dependencias del proyecto sin causar conflictos con las versión que el usuario pueda tener instalados a nivel global en su sistema.

Interfaz de la aplicación

Al iniciar la aplicación, ejecutando el *script* de Python generado, esta levantará un servidor HTTP, usando por defecto el puerto TCP 8050. Al acceder a este servidor con un navegador web, el usuario podrá visualizar el *dashboard* que generó, el que incluirá todas las vistas, gráficos y tarjetas que definió.

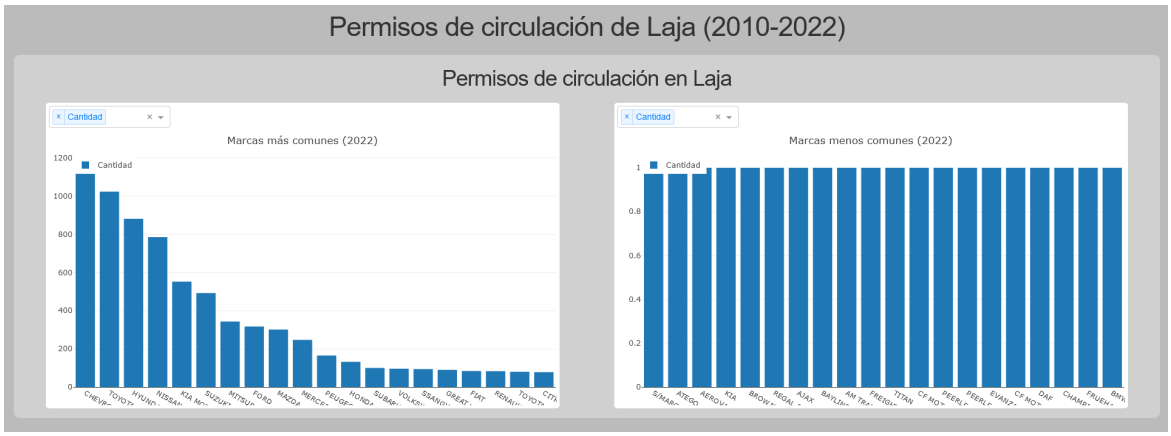

Figura 5.3: Vista del *dashboard* generado.

El título y subtítulo del *dashboard* se muestra en la parte superior de la pantalla, por encima de las vistas. Las vistas se presentan como rectángulos de color gris claro, las cuales contienen las tarjetas y gráficos. Tanto los gráficos como las vistas son presentadas como rectángulos de color blanco.

Como se mencionó en la subsección 4.3, cada gráfico, dependiendo de su tipo, mostrará distintos controles para interactuar con él. Todos los tipos de gráfico muestran en la parte superior un selector para filtrar las series definidas en el modelo, en caso de que el usuario desee ver sólo algunas en particular. Por defecto, todas las series son presentadas.

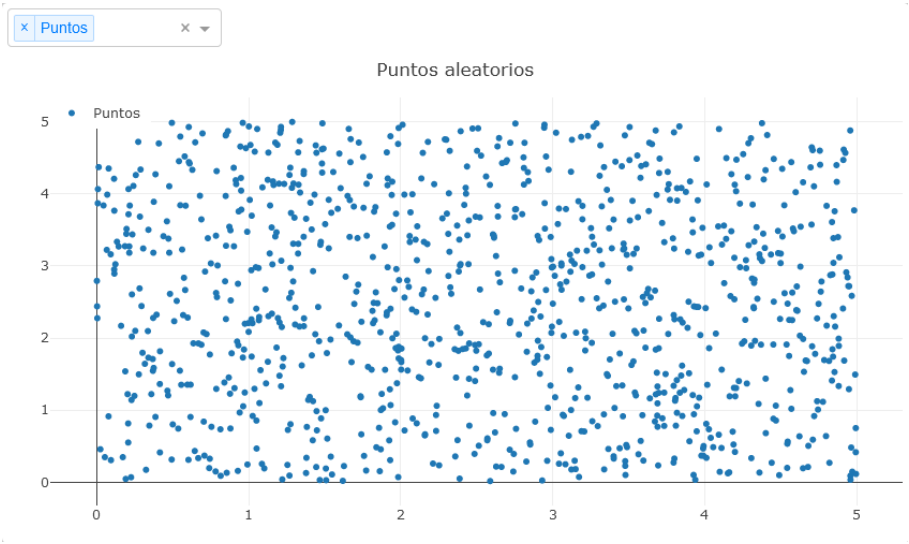

Figura 5.4: Gráfico de dispersión generado.

Figura 5.5: Acercamiento al selector de series de un gráfico.

En el caso de los gráficos de líneas y área, si su eje X fue definido con una columna de valores de tiempo, se incluirá en la parte inferior un selector de rango, el cual permite escoger el intervalo de tiempo a mostrar, presentando el rango completo de tiempo por defecto.

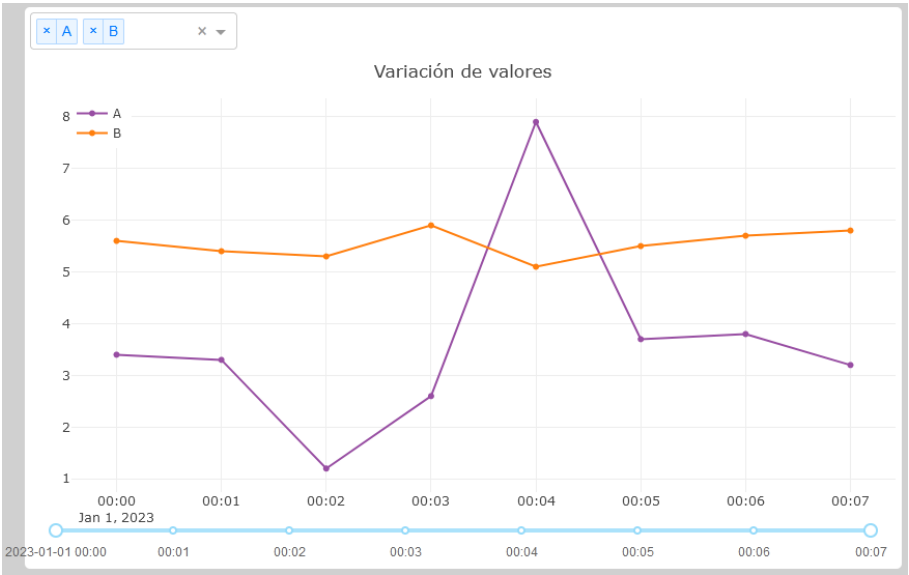

Figura 5.6: Gráfico de líneas generado utilizando valores de tiempo para el eje X.

Por otro lado, los histogramas mostrarán en su parte inferior un selector deslizante para definir el número de intervalos a utilizar y un *checkbox* para determinar si el gráfico va a utilizar un valor precalculado para el número de intervalos o el valor definido por el selector, utilizando por defecto el primero.

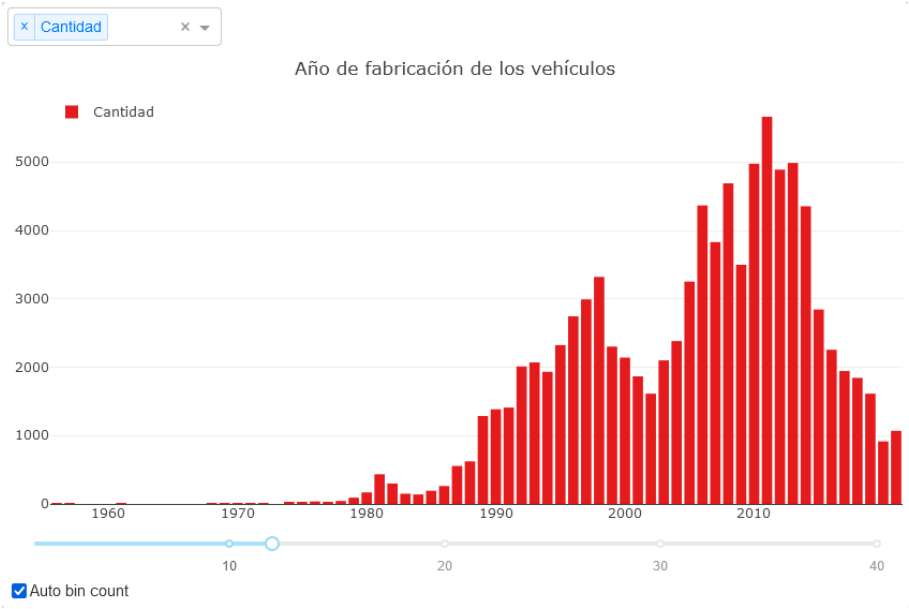

Figura 5.7: Histograma generado.

Funcionamiento de la aplicación

El siguiente diagrama presenta el funcionamiento de la aplicación web generada desde su generación hasta el despliegue de las visualizaciones especificadas.

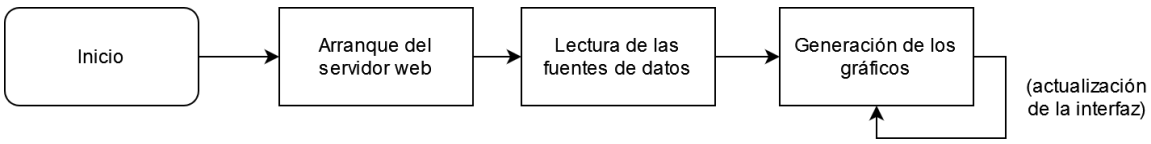

Figura 5.8: Flujo de ejecución del *dashboard* generado.

Las etapas que componen este flujo se explican a continuación:

1. **Arranque del servidor web**: La aplicación Python levanta un servidor web por medio de la biblioteca Dash, lo que permite que los usuarios puedan acceder a la aplicación desde sus navegadores.

2. **Lectura de las fuentes de datos**: Los archivos CSV que contienen la información de las fuentes de datos son leídos por la aplicación, los cuales son cargados en memoria.

3. **Generación de los gráficos**: Una vez cargados los datos, los métodos de *callback* mencionados anteriormente son llamados para generar el contenido de los gráficos. El proceso

de generación vuelve a ser llamado siempre que el usuario recargue la página o indique por medio de la interfaz web que desea mostrar u ocultar las series de los gráficos o modificar el número de intervalos en los histogramas.

Capítulo 6

Evaluación de la propuesta

6.1. Cobertura de las visualizaciones

En esta sección, se hará un análisis de los tipos de visualización implementados y su nivel de usabilidad al compararlos con implementaciones que utilizan directamente las bibliotecas utilizadas en este proyecto.

Tipos de visualización implementados

Los tipos de visualización actualmente soportados por la herramienta corresponden a:

- Gráficos de línea
- Gráficos de área
- Gráficos de dispersión
- Histogramas
- Gráficos de barras
- Tarjetas, que pueden presentar el valor máximo, mínimo o promedio de una serie de datos

Para todos los gráficos soportados, existe un cierto nivel de personalización al gusto del usuario, pudiendo definir colores para las series (a partir de una paleta predeterminada), incluir u omitir un cuadro de leyenda, seleccionar las series a visualizar en tiempo de ejecución y seleccionar un rango de tiempo para los gráficos de línea/área que utilicen series de tiempo para el eje X.

Tipos de visualización disponibles en las bibliotecas utilizadas

Como se ha mencionado anteriormente, este proyecto hace uso de la biblioteca de Python Plotly para la generación de los gráficos. Para la versión utilizada en este proyecto (la versión 5.18.0), están disponibles, entre otros, los siguientes tipos de gráfico:

- Gráficos de dispersión
- Gráficos de línea/área
- Gráficos de barras
- Gráficos de torta
- Gráficos de burbuja
- Diagramas de caja (*box plot*)
- Barras de error
- Histogramas
- Gráficos de contorno
- Mapas de calor (*heat map*)
- Diagramas ternarios (gráficos triangulares)
- Gráficos de vela (*candlestick*)
- Gráficos OHLC (*open-high-low-close*)
- Mapas

Soporte para tipos de visualización no implementados

Es posible modificar la herramienta para incorporar soporte a nuevos tipos de visualización. Para esto, es necesario realizar los siguientes pasos:

1. **Agregar visualización al metamodelo**: Dependiendo de su complejidad, incorporar una visualización nueva al metamodelo puede implicar simplemente agregar una nueva subclase a `Graph` y referenciarla a una o más instancias de `ChartSeries`, o definir una nueva estructura en el metamodelo, si el tipo de gráfico no encaja bien dentro de la arquitectura definida.

2. **Agregar componente al editor gráfico**: Dentro del proyecto Sirius, es necesario agregar un nuevo nodo al modelo, agregando también los elementos gráficos que se estimen necesarios (subnodos, colores personalizados, etc.). Para que la nueva visualización esté disponible para los usuarios, se debe agregar también un elemento de herramienta que permita su creación, así como su sección de propiedades para permitir modificar valores como su título, color o columnas del *dataset* a referenciar.

3. **Agregar transformación al proyecto Acceleo**: Definido todo eso, se debe modificar la transformación de modelo a texto para incorporar la nueva visualización, lo que implica definir una estructura HTML para este en el *dashboard* y el código que utilizará su *callback*, para entregar la información del *dataset* en el formato apropiado a Plotly u otra herramienta de visualización.

6.2. Evaluación de usabilidad

Para la validar la usabilidad de la herramienta propuesta, se diseñó un conjunto de ejercicios de prueba, a través de los cuales los potenciales usuarios podrán hacer uso de ella para generar *dashboards*, conocer su funcionamiento y así evaluar su experiencia. La evaluación se realizó por medio de un cuestionario de preguntas, basado en los cuestionarios definidos en [18], para evaluar la opinión de los usuarios sobre la experiencia utilizando la herramienta y su nivel de satisfacción con los resultados obtenidos. Las preguntas del cuestionario están disponibles en el Apéndice C.

Participantes de la evaluación

Para esta evaluación, participó un grupo de personas con conocimiento de herramientas de desarrollo de software dirigido por modelo, entre los cuales participaron alumnos de Ingeniería Civil Informática que cursaron el ramo de Desarrollo de Software Dirigido por Modelos. Los participantes de este proceso aceptaron realizar una prueba de la herramienta desarrollada siguiendo una guía predefinida de ejercicios, para después completar una encuesta de satisfacción expresando sus opiniones sobre los ejercicios realizados.

Estructura de las pruebas

Las pruebas realizaron consistieron en reuniones telemáticas con los participantes, en las cuales se les explicó el objetivo de este proyecto y el funcionamiento de la herramienta, así como la instalación de las dependencias utilizadas para su ejecución. Luego, se procedió a realizar los ejercicios definidos, los cuales implicaban completar los pasos indicados y mostrar los *dashboards* generados en cada uno. Posteriormente, una vez finalizados los ejercicios, los participantes rellenaron sus encuestas, por medio de la plataforma Google Forms, indicando su opinión sobre la experiencia y entregando comentarios al respecto.

Escenarios de evaluación

Para este ejercicio, se diseñaron tres casos de ejemplo que los usuarios debieron replicar. Cada ejemplo, que prueba distintas funcionalidades de la herramienta, incluye una lista de instrucciones junto con una imagen del resultado esperado.

Ejercicio 1: *Dashboard* con un gráfico de barras y una tarjeta

Para este ejemplo, se utilizó un *dataset* basado en [19], que contiene información sobre los permisos de circulación emitidos por la Municipalidad de Laja entre los años 2010 y 2022.

El objetivo de este ejercicio es crear un *dashboard* que contenga un gráfico de barras, que debe mostrar el *top* 10 de marcas más registradas el año 2020, presentado como un conjunto de barras horizontales, ordenadas en orden descendente de arriba a abajo, y una tarjeta que indique el valor promedio de la cuota del permiso pagado a lo largo de todo el período registrado.

El resultado esperado es el siguiente:

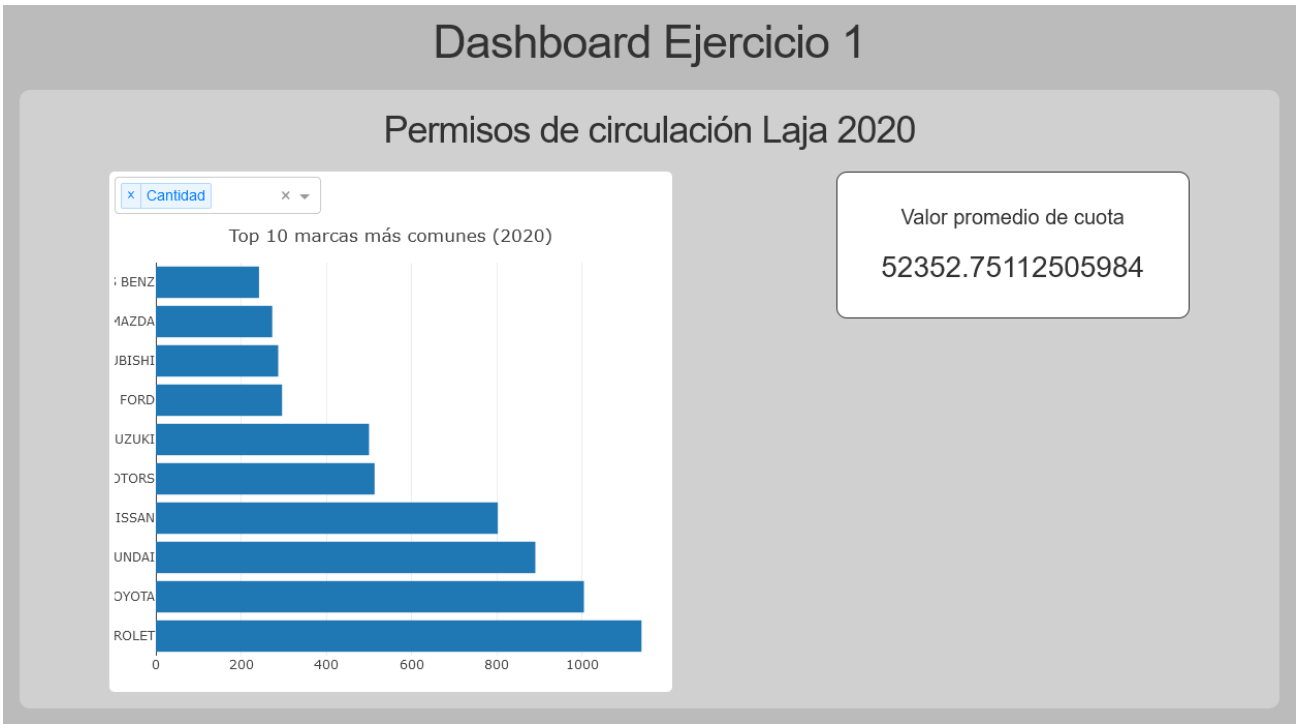

Figura 6.1: *Dashboard* esperado después de realizar el Ejercicio 1.

Para este ejercicio, las instrucciones fueron las siguientes:

1. Cree un modelo de *dashboard* nuevo usando el proyecto base incluido, con el título "Dashboard Ejercicio 1".
2. Agregue una fuente de datos llamada "Permisos de circulación Laja 2020", la cual debe apuntar al archivo `./permisos.csv` (incluyendo el `./`), utilizando el separador `;` y la codificación ISO 8859-1 (escrita como `iso8859-1`).
3. A la fuente de datos recién creada, agregue las siguientes columnas:
   - `MARCA`, de tipo `STRING`
   - `ANO_GIRO`, de tipo `NUMBER`
   - `MONTO_PAGO`, de tipo `NUMBER`
4. Cree una vista nueva, llamada "Permisos de circulación Laja 2020" y conéctela con la fuente de datos.
5. En la vista, agregue un gráfico de barras, que deberá llamarse "Top 10 marcas más comunes (2020)", el cual debe usar la columna `MARCA` para el eje X, no debe mostrar una leyenda, pero sí utilizar barras horizontales.
6. Dentro del gráfico de barras, agregue una serie de gráfico nueva, llamada "Cantidad", la cual debe ser de color automático y que debe utilizar la columna `MARCA`. A esta serie se le deben agregar las siguientes transformaciones en el mismo orden que se presentan:

a) Filtrar (*Filter*) valores iguales a 2020 (`NUMBER`) en la columna `ANO_GIRO`.

b) Agrupar (*Group*) las filas por `MARCA`, utilizando la operación `COUNT` (contar).

c) Ordenar (*Sort*) las filas en orden descendente, dejando la columna a ordenar en blanco, de forma que utilice el resultado del agrupamiento anterior.

d) Escoger (*Pick*) las primeras 10 filas.

7. Agregue dentro de la vista una tarjeta, la cual debe titularse "Valor promedio de cuota", que debe calcular el valor promedio (`AVERAGE`) y utilizar la columna `MONTO_PAGO`.

8. Genere el código del *dashboard*.

Ejercicio 2: *Dashboard* con un histograma y un gráfico de líneas con series de tiempo

Para este ejemplo, se utilizó un *dataset* con información de variación anual de la inflación en múltiples países del mundo desde 1980 a la fecha [20], generado a partir de información recopilada por el Banco Mundial. El *dataset* fue modificado para incluir sólo la información relativa a Chile en un formato más acorde a los requerimientos del programa.

El objetivo de este ejercicio es generar un *dashboard* que contenga un histograma, el cual presente la distribución de la variación anual de la inflación y un gráfico de líneas que muestre la variación anual en orden cronológico. El resultado esperado es el siguiente:

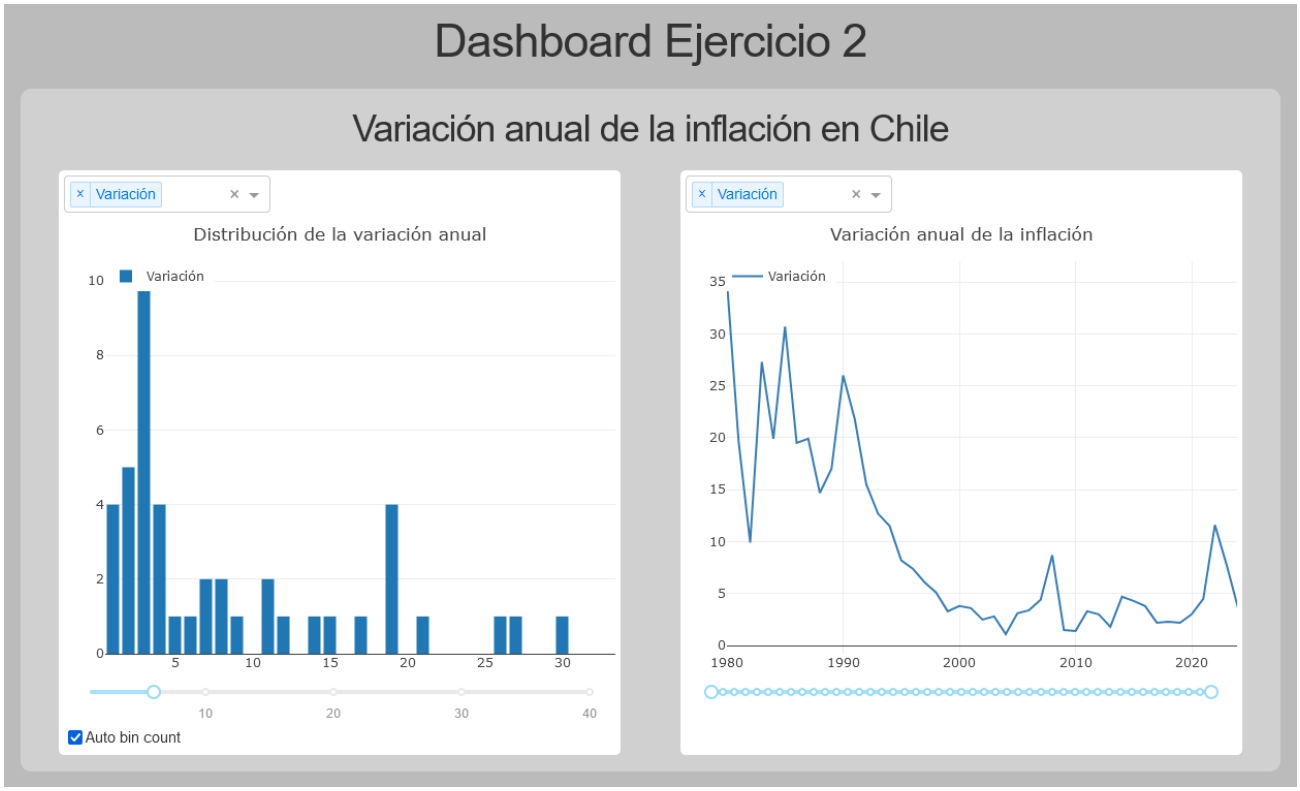

Figura 6.2: *Dashboard* esperado después de realizar el Ejercicio 2.

Para este ejercicio, las instrucciones fueron las siguientes:

1. Cree un modelo de *dashboard* nuevo usando el proyecto base incluido, con el título "Dashboard Ejercicio 2".
2. Agregue una fuente de datos llamada "Variación anual de la inflación Chile", la cual debe apuntar al archivo `./inflacion.csv` (incluyendo el `./`), utilizando los valores de separador y codificación por defecto.
3. A la fuente de datos recién creada, agregue las siguientes columnas:
   - `fecha`, de tipo `TIMESTAMP_ISO8601`
   - `variacion`, de tipo `NUMBER`
4. Cree una vista nueva, llamada "Variación anual de la inflación en Chile" y conéctela con la fuente de datos.
5. En la vista, agregue un histograma, que deberá llamarse "Distribución de la variación anual", el cual debe usar mostrar una leyenda.
6. Dentro del histograma, agregue una serie de histograma de nombre "Variación", que debe usar la columna `variacion` y debe usar un color automático.
7. Agregue en la misma vista un gráfico de líneas, llamado "Variación anual de la inflación", que debe usar la columna `fecha` para el eje X, además de mostrar una leyenda.
8. Dentro del gráfico de líneas, agregue una serie de gráfico nueva, llamada "Variación", la cual debe ser de color azul y debe utilizar la columna `variacion`.
9. Genere el código del *dashboard*.

Ejercicio 3: *Dashboard* que utilice dos fuentes de datos y columnas virtuales

Para este ejemplo, se utilizaron dos *datasets* no relacionados: uno con información sobre las emisiones de dióxido de carbono por países desde 1750 hasta 2022 [21] y otro con información sobre las canciones más reproducidas en la plataforma de *streaming* Spotify durante el año 2023 [22].

El objetivo de este ejercicio es generar un *dashboard* con más de una vista, donde cada una utilizará una fuente de datos diferente, además de hacer uso de columnas virtuales para preprocesar la información antes de ser presentada a un gráfico. El resultado esperado es el siguiente:

Figura 6.3: *Dashboard* esperado después de realizar el Ejercicio 3.

Para este ejercicio, las instrucciones fueron las siguientes:

1. Cree un modelo de *dashboard* usando el proyecto base incluido, con el título "Dashboard Ejercicio 3".

2. Agregue una fuente de datos, llamada "Emisiones de CO2 por país", la cual debe apuntar al archivo `./emisionesco2.csv` (incluyendo el `./`), utilizando los valores de separador y codificación por defecto.

3. Para esta fuente de datos, agregue las siguientes columnas:
   - `Country`, de tipo `STRING`
   - `Year`, de tipo `NUMBER`
   - `Total`, de tipo `NUMBER`
4. Agregue otra fuente de datos, que deberá llamarse "Canciones más reproducidas 2023", la cual debe apuntar al archivo `spotify.csv` (incluyendo el `./`), utilizando los valores de separador y codificación por defecto.
5. Para esta fuente de datos, agregue las siguientes columnas:
   - `track_name`, de tipo `STRING`
   - `streams`, de tipo `NUMBER`
6. Cree una vista nueva, llamada "Emisiones de CO2 en el Cono Sur" y conéctela con la fuente de datos "Emisiones de CO2 por país".
7. Dentro de esta vista, cree una columna virtual, que deberá llamarse "Emisiones", debe referenciar a la columna `Total` y debe incluir una transformación de tipo Ordenar (*Sort*), en orden ascendente, usando los valores de la columna `Year`.
8. Agregue en la vista un gráfico de barras, con el título "Emisiones en el Cono Sur (1973-2022)", que use la columna `Year` para el eje X, sin leyenda, con barras verticales y apiladas, que debe incluir las siguientes tres series de gráficos:
   - Una serie llamada "Chile", de color rojo, que utilice la columna virtual "Emisiones en 2022", más las siguientes transformaciones, en el mismo orden en que se presentan:
     - Filtrar (*Filter*) valores iguales a "Chile" (`STRING`) en la columna `Country`.
     - Escoger (*Pick*) las últimas 50 filas.
   - Una serie llamada "Argentina", de color azul, que utilice la columna virtual "Emisiones en 2022", más las siguientes transformaciones, en el mismo orden en que se presentan:
     - Filtrar (*Filter*) valores iguales a "Argentina" (`STRING`) en la columna `Country`.
     - Escoger (*Pick*) las últimas 50 filas.
   - Una serie llamada "Uruguay", que utilice la columna virtual "Emisiones en 2022", más las siguientes transformaciones, en el mismo orden en que se presentan:
     - Filtrar (*Filter*) valores iguales a "Uruguay" (`STRING`) en la columna `Country`.
     - Escoger (*Pick*) las últimas 50 filas.
9. Cree otra vista, llamada "Canciones más reproducidas" y conéctela con la fuente de datos "Canciones más reproducidas 2023".
10. Dentro de esta vista, agregue un gráfico de barras llamado "Top 10 canciones más reproducidas", que use la columna `track_name` para el eje X, sin leyenda, con barras verticales y sin apilar.

11. Agregue una serie de gráficos a este último gráfico, que debe llamarse "Reproducciones", de color automático, que utilice la columna `streams` e incluya las siguientes transformaciones, en el mismo orden en que se presentan:
    - Ordenar (*Sort*) las filas en orden descendente, usando los valores de la columna `streams`.
    - Escoger (*Pick*) las primeras 10 filas.
12. Genere el código del *dashboard*.

Resultados de la encuesta

Para la realización de encuesta participaron cuatro personas, las cuales tenían conocimiento de las herramientas utilizadas y el funcionamiento de los componentes que utilizó este proyecto para su desarrollo.

Con respecto a las primeras tres preguntas, que evalúan la satisfacción al completar las tareas de los ejercicios, se destaca la evaluación de las tareas "Ubicar elementos en el lienzo" y "Conectar objetos", en las cuales todos los participantes respondieron "Totalmente de acuerdo" en las preguntas 1 y 2 y tres respondieron de la misma forma en la pregunta 3, habiendo sólo uno que respondió "De acuerdo". Es posible concluir que los usuarios no tienen mayor dificultad para ensamblar los elementos de sus *dashboards* (p. ej. agregar vistas, incluir gráficos en estas, etc.), así como al conectar las vistas con las fuentes de datos.

Por otro lado, las tareas "Editar propiedades de los objetos" y "Generar el código del *dashboard*" obtuvieron resultados teniendo más a la neutralidad. Se puede concluir de esto que tanto la edición de las propiedades de los elementos en el lienzo (nombre, columnas a referenciar, lista de transformaciones, etc.) como la generación del código ejecutable a partir de los modelos podrían servirse de mejoras. Sin embargo, dadas las limitaciones de Sirius al momento de definir las secciones de propiedades de sus elementos, podría no ser posible solucionar todas las dificultades que los usuarios podrían haber encontrado. Con respecto a la generación del código del *dashboard*, el procedimiento propuesto para gatillar la ejecución de Acceleo y llevar a cabo la transformación de modelo a texto puede no haber sido muy sencilla para los usuarios, probablemente debido a que esta debió hacerse de manera manual durante la evaluación. Esto podría resolverse ofreciendo a los usuarios un procedimiento más automático para la generación automática de código, lo que podría lograrse integrando mejor este proceso a la interfaz del IDE, de forma que pudiera hacerse con menos pasos y sin tener que configurar manualmente los parámetros necesarios para esto.

Satisfacción con respecto al sistema

Las siguientes preguntas (de la 3 a la 21) califican la satisfacción del usuario con aspectos generales del sistema a probar. La mayor parte de los usuarios respondió con puntajes neutrales o positivos. Solamente uno de los usuarios respondió con puntajes negativos ("En desacuerdo" y "Totalmente en desacuerdo"), debido a que no pudo completar a cabalidad los ejercicios en su sistema, probablemente debido a problemas de portabilidad del IDE Eclipse con el código (la herramienta fue desarrollada en Windows para procesadores x86 y el usuario utilizó macOS para procesadores ARM). Sin embargo, este respondió positivamente con

respecto a los aspectos que sí pudo probar durante sus pruebas, al igual que el resto de los usuarios.

Por otro lado, con respecto al manejo de errores al realizar los ejercicios, los usuarios tendieron a responder de forma más negativa. La herramienta incorpora validaciones en determinadas propiedades de los elementos para así evitar, por ejemplo, evitar dejar un nombre vacío en un gráfico o utilizar un tipo de columna inválida en una transformación (como filtrar todas las columnas mayores a un *string*, lo cual no es posible). Sin embargo, esto parece no ser suficiente para los usuarios. Es probable que la forma en que se presentan los errores no les parezca muy intuitiva y no sepan cómo resolverlos. Esto podría solucionarse utilizando otras técnicas para presentar los mensajes de error, como cuadros de diálogo o mensajes en un color llamativo debajo de los controles que presenten errores, así como proponer soluciones al usuario, las que puedan seleccionarse y que resuelvan automáticamente los errores que pudiese haber.

También se pueden destacar de los comentarios adicionales en la pregunta 22 algunas falencias encontradas en los entregables, como conflictos en las versiones de Java especificadas en el código y las versiones utilizadas por los usuarios, dependencias no incluidas en el archivo de requerimientos de Python generado y poca flexibilidad en las rutas de archivos. Algunos de estos errores (como el problema con las versiones de Java y los requerimientos no definidos) pueden ser resueltos sin mayor dificultad. Sin embargo, aunque es factible la generación de los entregables del *dashboard* en rutas determinadas por el usuario (el código ya es capaz de ello), podrían presentarse problemas al momento de utilizar este código en sistemas distintos al utilizado para generarlo, ya que las rutas definidas para las fuentes de datos pueden no ser las mismas ni seguir el mismo formato (una ruta en un sistema Windows no sigue las mismas especificaciones que una ruta en un sistema tipo Unix, como GNU/Linux o macOS). Es necesario considerar estos detalles al momento de habilitar de forma más explicita esta funcionalidad, para evitar problemas de portabilidad entre plataformas.

Adicionalmente, uno de los comentarios menciona posibles mejoras en la configuración de las transformaciones de parte del usuario. Esto podría lograrse explicando de mejor forma los distintos atributos de cada transformación y qué realiza cada uno, además de mejorar la interfaz con la que estos se configuran.

Otro aspecto que podría mejorarse según lo indicado en las respuestas es la documentación de la herramienta. Para propósitos de la evaluación, a los usuarios se les realizó una inducción verbal de las características de la herramienta y los procedimientos para construir los *dashboards* y generarlos. Para mejorar esta parte, se podría agregar más documentación a la herramienta, ya sea documentación en línea, manuales de instrucciones o tutoriales en video, que puedan explicar su funcionamiento y cómo realizar los procedimientos más habituales.

A pesar de todo esto, se puede concluir que la experiencia de los usuarios fue en buena parte satisfactoria, pudiendo completar los ejercicios propuestos a pesar de las dificultades o limitaciones que pudieron haber encontrado durante la evaluación. Fue posible extraer bastante información a partir de los puntajes y comentarios de los usuarios, los cuales pueden servir para determinar los aspectos a mejorar del proyecto.

Capítulo 7

Conclusiones

Durante la realización de este trabajo, fue posible llevar a cabo en gran medida los objetivos iniciales propuestos, pudiendo desarrollar una herramienta que permita la generación de visualizaciones de datos por medio de un lenguaje específico de dominio. Dicha herramienta es capaz de presentarle al usuario una interfaz que permite describir de forma gráfica sus requerimientos (tanto de visualización como de los datos que utilizarán estas), los que después pueden ser convertidos a código ejecutable, el cual va a generar las visualizaciones que pidió inicialmente.

Sin embargo, a pesar de haber podido cumplir con varios de los objetivos, se encontraron algunas dificultades en el proceso que implicaron acotar el alcance de los objetivos o, derechamente, omitir su implementación. Por ejemplo, debido a la escasa documentación que acompañaba a las herramientas utilizadas, como Acceleo y Sirius, no fue factible empaquetar el código realizado en una extensión autocontenida que incluyera todas las características implementadas. También, se encontraron algunas limitaciones con el funcionamiento de la herramienta Sirius, que dificultaban la realización de algunas características, debiendo redefinirse o darse por no factibles. De todas formas, aun considerando estas problemáticas, fue posible desarrollar la herramienta como tal, pudiendo ser utilizada y siendo posible generar código a partir de las representaciones del usuario.

Aun considerando las limitaciones encontradas durante el desarrollo de este proyecto, es posible seguir implementando nuevas funcionalidades, como puede ser la generación de tablas, más tipos de gráficos, un mejor diseño para los elementos de los diagramas, la obtención de los *datasets* por medio de fuentes de datos en red en lugar de archivos en disco, un sistema de transformaciones más flexible, mejorar la generación de errores para reflejar de manera más clara las restricciones definidas en el metamodelo, así como mejoras en la experiencia de usuario al interactuar con cuadros de diálogo, propiedades y la ejecución de la transformación de modelo a texto, entre otras. Dependiendo de cómo estas se implementen, podría ser posible continuar este trabajo utilizando las mismas herramientas, aunque no se debería descartar buscar otras plataformas de *software* mejor mantenidas que pudieran utilizarse como base para una nueva versión.

La evaluación de la herramienta por parte de usuarios reales permitió confirmar los aspectos positivos de la herramienta, así como mostrar funcionalidades en las que se podría mejorar su funcionamiento, los cuales, al resolverse, podrán ofrecer una experiencia de usuario más simple, así como también resolver algunos fallos que fueron ignorados durante el proceso de desarrollo. Se puede destacar la opinión generalmente positiva de los participantes de la

evaluación, la cual demuestra el potencial que tiene este proyecto a futuro, el cual podría ser aún más útil y funcional en futuras versiones.

En conclusión, este proyecto puede servir como punto de partida para una aplicación mucho más completa, la cual permita generar *dashboards* sin necesitar un mayor dominio de lenguajes de programación o bibliotecas de visualización de datos, por medio de un lenguaje gráfico lo suficientemente simple pero a la vez extensible para cumplir las necesidades de la mayor cantidad de usuarios posible.

Bibliografía

[1] A. van Deursen, P. Klint y J. Visser, «Domain-specific languages: an annotated bibliography,» *ACM SIGPLAN Notices*, vol. 35, n.º 6, págs. 26-36, jun. de 2000, ISSN: 1558-1160. DOI: 10.1145/352029.352035. dirección: http://dx.doi.org/10.1145/352029.352035.

[2] U. Zdun, «Concepts for Model-Driven Design and Evolution of Domain-Specific Languages,» en *Proceedings of the International Workshop on Software Factories at OOPSLA 2005*, San Diego, CA, USA, oct. de 2005, págs. 1-6. dirección: http://eprints.cs.univie.ac.at/2365/.

[3] A. Rodrigues da Silva, «Model-driven engineering: A survey supported by the unified conceptual model,» *Computer Languages, Systems & Structures*, vol. 43, págs. 139-155, oct. de 2015, ISSN: 1477-8424. DOI: 10.1016/j.cl.2015.06.001. dirección: http://dx.doi.org/10.1016/j.cl.2015.06.001.

[4] L. Wang, G. Wang y C. A. Alexander, «Big Data and Visualization: Methods, Challenges and Technology Progress,» *Digital Technologies*, vol. 1, n.º 1, págs. 33-38, 2015. DOI: 10.12691/dt-1-1-7. dirección: http://pubs.sciepub.com/dt/1/1/7.

[5] M. E. Spear, *Charting Statistics*. McGraw-Hill, 1952.

[6] J. Utts, *Seeing Through Statistics*. Thomson Brooks/Cole, 2004, ISBN: 0534394027.

[7] D. Howitt y D. Cramer, *Introduction to statistics in psychology*, en. Philadelphia, PA: Pearson Education, 2008.

[8] *Histograms — plotly.com*, https://plotly.com/python/histograms/, 2022. (visitado 08-03-2023).

[9] J. Freeman, *Line Graph - When to Use It? - Edraw — edrawsoft.com*, https://www.edrawsoft.com/chart/when-to-use-line-graph.html. (visitado 08-04-2024).

[10] T. BiGNet, *Advantages and Disadvantages of Bar Diagram — bignet.in*, https://bignet.in/blog/429/advantages-and-disadvantages-of-bar-diagram-in-english. (visitado 08-04-2024).

[11] A. Gupta, *An ultimate guide for designing responsive data visualization cards — linkedin.com*, https://www.linkedin.com/pulse/ultimate-guide-designing-responsive-data-cards-arpit-gupta-f5saf/, 2023. (visitado 08-03-2024).

[12] B. Quiero, «Visualización de datos para monitoreo estructural de puentes mediante desarrollo de software dirigido por modelos,» Tesis de magíster, Universidad de Concepción, Concepción, Chile, 2021.

[13] S. Brown, *The C4 model for visualising software architecture — c4model.com*, https://c4model.com/. (visitado 21-02-2024).

[14] B. Quiero y G. Rojas, «Automatic Code Generation of Data Visualization for Structural Health Monitoring,» *IEEE Latin America Transactions*, vol. 20, n.º 7, págs. 1041-1050, 2022. DOI: 10.1109/TLA.2021.9827466.

[15] A. de la Vega, P. Sánchez y D. Kolovos, «Pinset: A DSL for Extracting Datasets from Models for Data Mining-Based Quality Analysis,» en *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*, 2018, págs. 83-91. DOI: 10.1109/QUATIC.2018.00021.

[16] Eclipse Foundation, *Acceleo | Overview — eclipse.dev*, https://eclipse.dev/acceleo/overview.html. (visitado 06-02-2024).

[17] Object Management Group, *OMG | Object Management Group — omg.org*, https://www.omg.org/. (visitado 06-02-2024).

[18] J. R. Lewis, «IBM computer usability satisfaction questionnaires: Psychometric evaluation and instructions for use,» *International Journal of Human–Computer Interaction*, vol. 7, n.º 1, págs. 57-78, 1995. DOI: 10.1080/10447319509526110.

[19] Ilustre Municipalidad de Laja, *Base Permisos de Circulación Laja - Portal de Datos Abiertos — datos.gob.cl*, https://datos.gob.cl/dataset/base-permisos-de-circulacion-laja, 2023. (visitado 18-02-2024).

[20] S. Islam, *Global Inflation Dataset — kaggle.com*, https://www.kaggle.com/datasets/sazidthe1/global-inflation-data, 2024. (visitado 18-02-2024).

[21] R. M. Andrew y G. P. Peters, *The Global Carbon Project's fossil CO2 emissions dataset*, ver. 2023v43, ene. de 2024. DOI: 10.5281/zenodo.10562476. dirección: https://doi.org/10.5281/zenodo.10562476 (visitado 18-02-2024).

[22] N. Elgiriyewithana, *Most Streamed Spotify Songs 2023 — kaggle.com*, https://www.kaggle.com/datasets/nelgiriyewithana/top-spotify-songs-2023/, 2023. (visitado 18-02-2024).

Apéndice A

Resumen de los elementos del DSL

En la siguiente página se presenta un resumen de los elementos que conforman el DSL descrito, indicando una descripción breve sobre su rol y los atributos que estos utilizan.

Nombre	Descripción
`Dashboard`	Elemento raíz del metamodelo. Contiene los elementos base de un *dashboard* (fuentes de datos y vistas). ▪ `title`: Título del *dashboard* ▪ `subtitle`: Subtítulo del *dashboard* ▪ `outputName`: Nombre del *script* a generar con el código del *dashboard* ▪ `datasource`: Fuente(s) de datos utilizadas en el *dashboard* ▪ `view`: Vista(s) que va a presentar el *dashboard*
`DataSource`	Fuente de datos a consumir para generar los *dashboards*. ▪ `id`: Identificador único de la fuente de datos ▪ `name`: Nombre descriptivo de la fuente de datos ▪ `path`: Ubicación del archivo CSV (relativo al *script* generado) ▪ `encoding`: Codificación de caracteres del archivo CSV (UTF-8 por defecto) ▪ `separator`: Carácter separador de caracteres del archivo CSV (`,` por defecto) ▪ `datasourcecolumn`: Columnas utilizadas en la fuente de datos
`View`	Vista del *dashboard*. Puede contener múltiples gráficos y tarjetas. ▪ `id`: Identificador único de la vista ▪ `name`: Nombre de la vista ▪ `chart`: Gráficos contenidos en la vista ▪ `card`: Tarjetas contenidas en la vista ▪ `virtualcolumn`: Columnas virtuales utilizadas en la vista
`DataSourceColumn`	Columna de la fuente de datos. Corresponde a una columna del archivo CSV a utilizar. ▪ `name`: Nombre de la columna en el archivo CSV ▪ `type`: Tipo de datos almacenado en la columna ▪ `utcOffset`: Zona horaria del *timestamp*, representado como número flotante (sólo si `type` es `TIMESTAMP_UNIX`) ▪ `milliseconds`: Indica si la resolución del *timestamp* es en milisegundos en lugar de segundos enteros (sólo si `type` es `TIMESTAMP_UNIX`)
`VirtualColumn`	Columna virtual, la cual apunta a un `DataSourceColumn` y puede contener una o más transformaciones. ▪ `column`: Columna base ▪ `transformation`: Transformaciones utilizadas en la columna

Cuadro A.1: Resumen de los elementos base del DSL.

Nombre	Descripción
`LineChart`, `AreaChart`, `BarChart`	Representa un gráfico de líneas, de área o de barras, respectivamente. ▪ `id`: Identificador único del gráfico ▪ `title`: Título del gráfico ▪ `subtitle`: Subtítulo del gráfico ▪ `showLegend`: Indica si se muestra o no la leyenda del gráfico ▪ `x_column`: Columna utilizada para los valores del eje X ▪ `chartseries`: Series utilizadas por el gráfico
`ChartSeries`	Representa una serie de un gráfico de líneas, área o barras. Toma los valores de una columna de la fuente de datos para los valores del eje Y. ▪ `id`: Identificador único de la serie ▪ `name`: Nombre de la serie ▪ `color`: Color de la serie ▪ `y_column`: Columna utilizada para los valores del eje Y ▪ `transformation`: Transformaciones utilizadas en la serie (sólo para `BarChart`)
`Histogram`	Representa un histograma. ▪ `id`: Identificador único del histograma ▪ `title`: Título del histograma ▪ `subtitle`: Subtítulo del histograma ▪ `showLegend`: Indica si se muestra o no la leyenda del histograma ▪ `histogramseries`: Series utilizadas por el histograma
`HistogramSeries`	Representa una serie de un histograma. Toma los valores de una columna de la fuente de datos para los valores del eje Y. ▪ `id`: Identificador único de la serie ▪ `name`: Nombre de la serie ▪ `color`: Color de la serie ▪ `x_column`: Columna utilizada para los valores del eje X

Cuadro A.2: Resumen de los elementos del DSL relacionados a gráficos.

Nombre	Descripción
`ScatterChart`	Representa un gráfico de dispersión. ▪ `id`: Identificador único del gráfico ▪ `title`: Título del gráfico ▪ `subtitle`: Subtítulo del gráfico ▪ `showLegend`: Indica si se muestra o no la leyenda del gráfico ▪ `histogramseries`: Series utilizadas por el gráfico
`ScatterSeries`	Representa una serie de un gráfico de dispersión. Toma los valores de dos columnas, una para los valores del eje X y otra para los del eje Y. ▪ `id`: Identificador único de la serie ▪ `name`: Nombre de la serie ▪ `color`: Color de la serie ▪ `symbol`: Símbolo a utilizar para los puntos de la serie ▪ `x_column`: Columna utilizada para los valores del eje X ▪ `y_column`: Columna utilizada para los valores del eje Y
`Card`	Representa una tarjeta, la cual puede mostrar un valor numérico. ▪ `id`: Identificador único de la tarjeta ▪ `title`: Título de la tarjeta ▪ `type`: Tipo de la tarjeta (máximo, mínimo o promedio) ▪ `column`: Columna utilizada para calcular el valor de la tarjeta

Cuadro A.3: Resumen de los elementos del DSL relacionados a gráficos. (cont.)

Nombre	Descripción
`Filter`	Filtra filas según el valor de una determinada columna y una operación asociada. ▪ `operation`: Tipo de operación para el filtro ▪ `value`: Valor de comparación para el filtro ▪ `valueType`: Tipo de valor para el filtro ▪ `intervalFrom`: Si la comparación es de tipo `INTERVAL`, define el inicio del intervalo a filtrar ▪ `intervalTo`: Si la comparación es de tipo `INTERVAL`, define el fin del intervalo a filtrar ▪ `column`: Columna cuyos valores serán utilizados para el filtrado
`Group`	Agrupa filas según el valor de una columna. ▪ `operation`: Tipo de agrupación a realizar ▪ `sort`: Indica si los valores agrupados deben ordenarse ▪ `ascending`: Indica si el ordenamiento debe ser ascendente ▪ `grouping_column`: Columna utilizada para generar las agrupaciones ▪ `comparing_column`: Si la operación no es de tipo `COUNT`, indica la columna utilizada para calcular el máximo, promedio o mínimo, respectivamente
`Pick`	Permite escoger un número determinado de valores del inicio o final de una serie. ▪ `length`: Cantidad de elementos a escoger ▪ `pickType`: Indica si los elementos se escogen del inicio o final de la columna
`Sort`	Ordena las filas de la serie utilizando el valor de alguna de las columnas como punto de referencia. ▪ `ascending`: Indica si el ordenamiento debe ser ascendente ▪ `pickType`: Columna utilizada para realizar el ordenamiento
`Reverse`	Invierte el orden en el que se presentan los valores.

Cuadro A.4: Resumen de los elementos del DSL relacionados a transformaciones.

Apéndice B

Diagrama del metamodelo

En la siguiente página se presenta el diagrama del metamodelo utilizado para este proyecto.

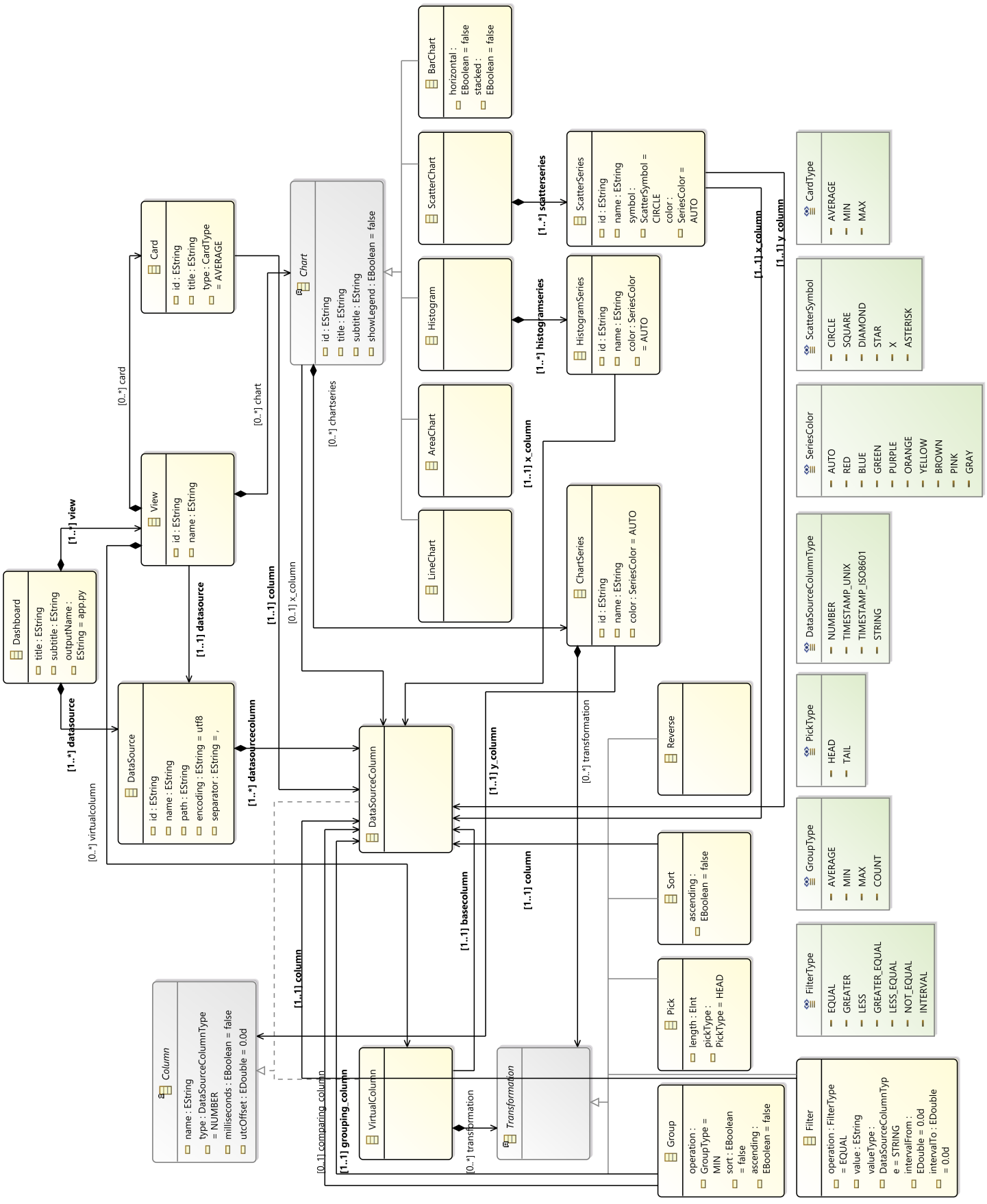

Figura B.1: Diagrama del metamodelo completo.

Apéndice C

Encuesta de satisfacción

La encuesta de satisfacción se compone de 22 preguntas, donde las primeras 21 corresponden a preguntas de selección múltiple. Cada una contiene una afirmación sobre la experiencia de uso del sistema, la cual puede ser calificada en una escala de cinco puntos, que van desde “totalmente de acuerdo” hasta “totalmente en desacuerdo”. La pregunta 23 corresponde a un cuadro de texto para comentarios adicionales.

Las primeras tres preguntas se descomponen en cuatro subpreguntas, las cuales corresponden a tareas en específico que el usuario debió realizar al completar los escenarios. En estos casos, cada tarea debe ser calificada individualmente.

La encuesta fue acompañada por el siguiente mensaje de bienvenida:

> ¡Hola! Muchas gracias por haber participado de las pruebas para mi proyecto de memoria de título. Una vez realizadas las pruebas, necesito que llene este formulario para conocer su opinión sobre la experiencia, para así entender bien las fortalezas y debilidades de mi herramienta.
>
> Esta encuesta se compone de 23 preguntas, donde en las primeras 22 se deben puntuar afirmaciones sobre la herramienta y la experiencia que tuvo al utilizarla en una escala que va desde "Totalmente de acuerdo"hasta "Totalmente en desacuerdo". En la última pregunta (opcional) puede dejar comentarios adicionales, los que me van a servir para complementar las preguntas anteriores.
>
> ¡Muchas gracias por participar!
>
> - Jorge Jara

En tanto, las preguntas de la encuesta fueron las siguientes:

1. En general, estoy satisfecho(a) con la facilidad al completar las tareas.
    - Ubicar elementos en el lienzo
    - Conectar objetos
    - Editar propiedades de los objetos
    - Generar el código del *dashboard*
2. En general, estoy satisfecho(a) con la cantidad de tiempo que tomó completar cada tarea.

   - Ubicar elementos en el lienzo
   - Conectar objetos
   - Editar propiedades de los objetos
   - Generar el código del *dashboard*

3. En general, estoy satisfecho(a) con la información de soporte (ayuda en línea, mensajes, documentación) al completar cada tarea.
   - Ubicar elementos en el lienzo
   - Conectar objetos
   - Editar propiedades de los objetos
   - Generar el código del *dashboard*
4. En general, estoy satisfecho(a) con la facilidad de uso del sistema.
5. Fue sencillo utilizar este sistema.
6. Pude completar efectivamente las tareas y escenarios usando este sistema.
7. Fui capaz de completar las tareas y escenarios eficientemente utilizando este sistema.
8. Me sentí cómodo(a) utilizando este sistema.
9. Fue fácil aprender a utilizar este sistema.
10. Creo que podría volverme productivo(a) rápidamente utilizando este sistema.
11. El sistema me entregó mensajes de error que me decían claramente cómo resolver problemas.
12. Cuando cometía un error al utilizar el sistema, podía recuperarme fácil y rápidamente.
13. La información (como ayuda en línea, mensajes en pantalla y otra documentación) provista con este sistema fue clara.
14. Fue sencillo encontrar la información que necesitaba.
15. La información provista por el sistema fue fácil de entender.
16. La información fue efectiva en ayudarme a completar las tareas y escenarios.
17. La organización de la información en las pantallas del sistema era clara.
18. La interfaz del sistema era agradable.
19. Me gustó usar la interfaz de este sistema.
20. El sistema tiene todas las funciones y capacidad que esperaba que tuviera.
21. En general, estoy satisfecho(a) con el sistema.
22. ¿Tiene algún comentario adicional que quiera agregar?