DISEÑO INTERACTIVO DE INTERFACES GRÁFICAS DE USUARIO PARA PROGRESS

Ricardo Caballero Sánchez
Adrián Díaz Jiménez

GRADO EN INGENIERÍA INFORMÁTICA

GRADO EN INGENIERÍA DEL SOFTWARE

FACULTAD DE INFORMÁTICA

UNIVERSIDAD COMPLUTENSE DE MADRID

TRABAJO DE FIN DE GRADO

Director: Manuel Montenegro Montes

A nuestros familiares, amigos y compañeros

de universidad y trabajo por su interés y apoyo constante

Contenido

Tabla de ilustraciones

RESUMEN

Este proyecto se centra en la creación de una aplicación de escritorio que, mediante tecnologías web, nos permite crear, de manera visual, interfaces gráficas.

El proyecto nace de la necesidad de crear interfaces gráficas con un lenguaje que no está muy preparado para ello: Progress. Progress es un lenguaje de programación utilizado para el mantenimiento y uso de bases de datos. Es por ello que la creación de la capa visual con este lenguaje es una tarea tediosa y el resultado es difícil de mantener.

En este proyecto hemos desarrollado un programa que nos permite elegir una serie de características que tendrá la interfaz y las variables mostradas en ella con el objetivo de generar el código Progress correspondiente. Además, se crean los componentes visuales que pueden ser arrastrados y movidos a la zona de la pantalla donde finalmente se verán al ejecutar el código Progress. De esta forma, se simplifica mucho la elaboración y modificación de estas pantallas, viendo en todo momento el resultado visual final que tendrá este código al ser ejecutado. También nos permite ver en tiempo real el código Progress que se genera automáticamente con cada acción y guardar este código en un fichero. Los ficheros que se generan con la aplicación pueden ser cargados de nuevo para realizar modificaciones.

Enlace a proyecto GitHub.

PALABRAS CLAVE

Progress, frames, variable, temp-table, interfaz gráfica, Electron, Interact.JS, analizador sintáctico, PEG.JS.

ABSTRACT

This project is focused on the creation of a desktop application that, using web technologies, allows us to create graphical interfaces in a visual way. The project is born from the need to create graphical interfaces with a language that is not prepared for it, Progress. Progress is a programming language used for the maintenance and use of databases. That is why the creation of the visual layer with this language is a tedious task, and the resulting views are difficult to maintain.

In this project we have developed a program that allows us to choose a series of properties in the interface and a series of variables shown in it in order to generate the corresponding Progress code. Additionally, visual components created can be dragged and moved to the area of the screen where they will finally be displayed when the corresponding Progress code is run. In this way, this tool greatly simplifies the elaboration of these screens and allows us to modify them quickly, and to see at all times the final visual result this code will have when executed. It also allows us to see in real time the Progress code that is automatically generated with each action and save this code in a file. The files that have been generated with the application can be reloaded to make modifications.

Link to GitHub project.

KEYWORDS

Progress, frames, variable, temp-table, graphical user interface, Electron, Interact.JS, parser, PEG.JS

CAPITULO 1.- INTRODUCCIÓN

En este primer capítulo explicaremos qué es Progress, las razones para la realización de este proyecto y el uso que puede hacerse de nuestra aplicación en un entorno real de trabajo.

1.1.- ANTECEDENTES Y MOTIVACIÓN

Nuestro proyecto consiste en una aplicación que nos permite generar una interfaz gráfica en código Progress de una manera mucho más cómoda y rápida. Progress es un lenguaje de programación utilizado para el mantenimiento y uso de bases de datos.[9]

Este proyecto fue propuesto para su realización por uno de los participantes del trabajo, Ricardo. Este alumno comenzó a trabajar en una empresa que utiliza un ERP. Un ERP (*"Enterprise Resource Planning"*) es un software empresarial que administra aspectos de producción, distribución y otros en una compañía. Concretamente en la empresa de Ricardo utilizan Progress en el ERP. Las interfaces visuales son también creadas con Progress y al no estar este lenguaje orientado a la creación de una capa de presentación, sino a la gestión de las bases de datos, la programación de la interfaz visual es algo complicada. Teniendo este problema, nos surgió la idea de facilitar este proceso mediante la utilización de un software.

Al igual que existen diseñadores interactivos de interfaces de usuario para otros lenguajes de programación, nuestra aplicación permite crear los elementos visuales de una manera sencilla y situarlos en la pantalla con gran facilidad. De esta forma, se genera el código en el lenguaje deseado automáticamente, haciendo así la creación y la modificación de la capa de presentación mucho más rápida y sencilla.

Un ejemplo de software similar que facilita la creación de interfaces visuales es Netbeans. Netbeans es un entorno integrado de desarrollo que simplifica el desarrollo de aplicaciones para Java, e incluye un diseñador de interfaces de usuario en Swing.

1.2.- OBJETIVOS

Tal como hemos comentado, el objetivo principal de este proyecto es el desarrollo de una aplicación que nos permita crear interfaces visuales en código Progress. Nosotros desglosamos este objetivo principal en los siguientes:

- Construir una aplicación que nos permita crear frames de entrada y variables asignadas a estos. Un frame en Progress es una agrupación de variables que serán mostradas conjuntamente, junto con una serie de estilos y características aplicadas a éstas. Hablaremos detalladamente sobre estos conceptos en las secciones 4.1.1.- y 4.1.2.- Con este objetivo queremos tener la estructura

de datos necesaria para guardar la información que utilizaremos a la hora de generar el código Progress.

- Diseñar un panel que simula el frame y poder arrastrar las variables en la posición en la que queremos que se vean mostradas dentro de él. De esta forma tendremos la facilidad de crear la interfaz visual y de aquí tomaremos los parámetros necesarios para generar el código Progress.
- Permitir la carga de ficheros en lenguaje Progress con especificaciones de frames y variables ya creados anteriormente por la aplicación, para así poder modificarlos. Este objetivo implica el análisis sintáctico de código Progress para cargar la información ya existente y modificar a partir de la misma. Esto nos ayudará a realizar cambios siempre que queramos en códigos Progress generados anteriormente, sin tener que crear la estructura desde cero de nuevo.
- Validar los datos para que el código Progress generado no contenga errores. Con este propósito queremos asegurar que a la hora de generar el código no habrá ningún error que después nos impida compilar el código Progress.

1.3.- PLAN DE TRABAJO

En este capítulo explicaremos brevemente las fases de desarrollo en las que estructuramos la realización del proyecto:

1. **Búsqueda de e investigación de tecnologías.** El primer paso en todo desarrollo es la elección de las tecnologías a utilizar para crear la solución al problema al que nos enfrentamos. Finalmente elegimos Electron, porque nos permitía utilizar herramientas web para crear una aplicación de escritorio. Podemos encontrar información sobre esta parte del desarrollo en el CAPITULO 3.-
2. **Diseño de estructuras de datos.** En esta parte del desarrollo nos centramos en qué información debíamos guardar y cómo guardarla de la manera más eficiente para ser accedida tanto a la hora de crear la interfaz como a la hora de generar el código. Esta parte del desarrollo se comenta en los apartados 5.1.-Estructura de datos en JavaScript y 5.2.-.
3. **Investigación de librerías para la movilidad de componentes visuales.** Una vez tenemos elegidas las herramientas y los datos, nos queda resolver el problema de cómo hacer una interfaz dinámica que nos permita crear y colocar los componentes en la pantalla. Podemos ver cómo hemos desarrollado finalmente esta funcionalidad en el apartado 4.2.2.-
4. **Generación de código**. El siguiente paso a desarrollar era la generación del código con las características indicadas por el usuario y con los componentes visuales en la posición deseada. Esta parte del desarrollo la encontramos en el capítulo 5.2.1.-
5. **Generador de análisis sintáctico.** Tras desarrollar la aplicación en un sentido, es decir, de generar la pantalla visualmente a código Progress, nos propusimos hacer el sentido contrario,

obtener una representación visual a partir de código. En esta fase tuvimos que investigar sobre los analizadores sintácticos. También tuvimos que aprender las restricciones semánticas del lenguaje y crear nuestras propias reglas para analizar las partes de código Progress que nos convenían. En el apartado 5.3.-encontramos esta parte del desarrollo explicada en profundidad.

6. **Validación y consistencia de datos.** En esta fase queríamos revisar y asegurar que tanto los datos introducidos como los datos leídos de un archivo fueran consistentes y sin errores, con el fin de asegurar el buen funcionamiento tanto de nuestra aplicación como del código Progress generado. Detallamos las validaciones realizadas en el apartado 6.1.-

7. **Ampliación y mejoras**. Finalmente añadimos una serie de mejoras para agregar algunas funcionalidades que no se vieron contempladas como objetivos al iniciar el proyecto.

    a. Movilidad con el teclado. Añadimos esta característica para hacer más precisa la colocación de las variables en el frame. Más información en el apartado 4.2.1.-

    b. Creación de frame de salida. Ya que teníamos la generación de frames de entrada, agregamos una funcionalidad extra para generar también frames de salida utilizados para reportes de información. Esta funcionalidad está comentada en profundidad en el capítulo 4.3.-

    c. Creación de log de carga de datos. Como parte adicional a la carga de datos, creamos un archivo donde se registrase cómo se iba leyendo el archivo y así dar una lista detallada de los fallos que pudiera contener para ser arreglados y poder cargar el código en nuestra aplicación. Podemos ver la estructura y las validaciones de la carga de datos en el punto 6.1.2.-

CAPITULO 2.- INTRODUCTION

In the first chapter we explain what Progress is, the motivations that took us to implement this project and the future usage of the application in a real working environment.

2.1.- BACKGROUND AND MOTIVATION

This project is based on an application that allows users to create source code describing a Progress graphical user interface (GUI) in a simple and faster way. Progress is a programming language used to maintain and implementation of databases.

The project was proposed by one of the participants, Ricardo. He started to work in a company where they use an ERP ("Enterprise Resource Planning"). An ERP is a business software that manages production, distribution and other aspects in a company. In this company they use Progress in the ERP. The interfaces are also implemented with Progress and because this language is not specifically designed to create a presentation layer, but to the management of databases, the creation of a visual interface is not an easy task. With this problem we thought about the idea to make this process easier using a tailor-made software.

Similarly, to GUI interactive designers for other programming languages, our application allows one to create visual items in a simple way and to place them on the screen easily. Thus the code is automatically generated in the target language, making the creation and modification of the presentation layer a faster and easier task.

Netbeans is an example of similar software that makes this kind of task regarding the visual interfaces. Netbeans is an integrated development environment that simplifies the development of Java applications, including a GUI designer for Swing.

2.2.- GOALS

As we have talked about, the primary goal of the project is the development of an application that allows to us the creation of graphical user interfaces in Progress code. We divide the main goal as follows:

- Building an application that allows to us to create input frames and variables assigned to them. A frame in Progress is a group of variables that will be displayed jointly, with a series of styles and features applied to them. We will talk in more detail about these concepts in sections 4.1.1.- and 4.1.2.-. With this goal we will have the data structure necessary for saving the information that will be used to generate the Progress code.

- Designing a panel that simulates the frame and let us drag and drop the variables in the position that we want them to be shown on it. Therefore, we will be able to create the visual interface easily and to get the parameters required in order to generate the Progress code.

- Allowing the data loading in Progress language with frames and variables specifications previously created by the application, in order to modify them. This goal requires parsing Progress code to load and modify the current information. This will help us make changes whenever we want in previously generated Progress code without creating the whole structure from the beginning.
- Validating the data to guarantee that the Progress code generated does not have errors. With this goal we ensure that the generated Progress code will not have any errors resulting in compilation failures.

2.3.- WORKING PLAN

In this section we will explain shortly the development phases in which we divide the project:

1. **Related technology research.** The first step in every development is the decision about the technologies that we will use to create the solution to the problem exposed. Finally, we chose Electron because it allows us to use web technologies to develop a desktop application. We can find more information about this in Chapter 3.- .
2. **Design of data structures.** In this step of the development we focus on the information: which information will be saved and how we will save it in the most efficient way to be able to access it in the creation of the interface step and in the code generation step. We can find more information about this part of the development in sections 5.1.- and 5.2.-
3. **Research of libraries for the movement of visual components.** Once we have chosen the tools and the data we have to solve the problem about how to make a dynamic interface that allows us to create and place the components on the window. We have more information about the development of this functionality in section 4.2.2 –.
4. **Code generation.** The next step to be developed is the code generation with the features chosen by the user and the visual components in the desired position. We have more information about this development in section 5.2.1 .
5. **Parser.** At this time, we have the application developed in one way: the generation of Progress code from a visual window. After this we propose to develop the opposite way: generate the visual window from the code. In this step we had to research about parsers. We also had to learn the semantic constraints of the language and create our own rules to analyze the Progress code that we need. We can find more information about this in section 5.3. .
6. **Validation and data consistency.** In this step we wanted to ensure both data entered and data read from a file were consistent and without errors to be sure that both application and Progress code generated will work properly. We detail the validations that we develop in section

7. **Expansion and improvements**. At the end we incorporate a series of improvements to append functionalities that were not considered as goals at the beginning of the project.

    a. **Keyboard mobility.** We added this feature to improve the accuracy of the variables in the frame. More information in section 4.2.1. .

    b. **Output frame creation.** Since we generate input frames, we added an extra functionality to include the generation of output frames that are used to report information. This functionality is described more deeply in section 4.3.- .

    c. **Data loading log creation.** As an additional part in the data loading, we create a file where will be registered the way that the file is being read, with a detailed list of possible errors that could arise in order to be fixed and be able to load the code in our application. We can see more information about the structure and validations in the data loading phase in section 6.1.2.- .

CAPITULO 3.- TECNOLOGÍAS

Actualmente existen Frameworks para crear aplicaciones de escritorio, en las que se puede usar las tecnologías web conocidas, tales como HTML5, JavaScript, CSS, JSON entre otros.

Como ejemplo de aplicaciones de escritorio realizadas mediante tecnologías web, podemos mencionar Avast y Nod32.

Al usar este tipo de desarrollo se nos brinda la facilidad de crear interfaces gráficas con mucha más facilidad ya que podemos reutilizar nuestros conocimientos de HTML y CSS, cosa que con otros Frameworks de aplicaciones de escritorio podría resultar mucho más trabajoso y complicado.

Por esta razón, nuestro proyecto se ha desarrollado con muchas de estas herramientas, las cuales mencionaremos a continuación, todas ellas relacionadas con tecnologías web.

3.1.- ELECTRON

Para llevar a cabo nuestro proyecto, nos hemos decidido por el Framework Electron, ya que para nosotros nos ha parecido más adecuado dado nuestros conocimientos acerca de las tecnologías web que hay detrás.

Electron (anteriormente Atom Shell) es un Framework de código abierto. Permite el desarrollo de aplicaciones gráficas de escritorio usando un modelo cliente servidor similar al utilizado en aplicaciones web [13],[14].

Del lado del cliente nos encontraríamos con un motor Chromium como interfaz y de la parte del servidor el entorno de ejecución Node.js [10].

Electron es el Framework elegido para el desarrollo de algunas aplicaciones conocidas como, por ejemplo, Microsoft Visual Studio Code o Atom de GitHub.

3.1.1.- Node.js

Node.js es un entorno en tiempo de ejecución multiplataforma de código abierto. Está diseñado para la capa de servidor de las aplicaciones web, aunque no necesariamente se debe limitar a ello. Node.js, diseñado para crear aplicaciones web escalables, es un entorno de ejecución de JavaScript orientado a eventos asíncronos [16].

Al contrario que la mayoría de código JavaScript, el código escrito para Node.js no se ejecuta en la parte del navegador web, sino en el servidor.

Al estar basado en el lenguaje JavaScript, Node.js se puede combinar para un desarrollo homogéneo entre cliente y servidor, y permite la adaptación de patrones para desarrollo del lado del servidor como MVC, facilitando la reutilización de código entre el lado del cliente y del servidor.

3.1.2.- Chromium

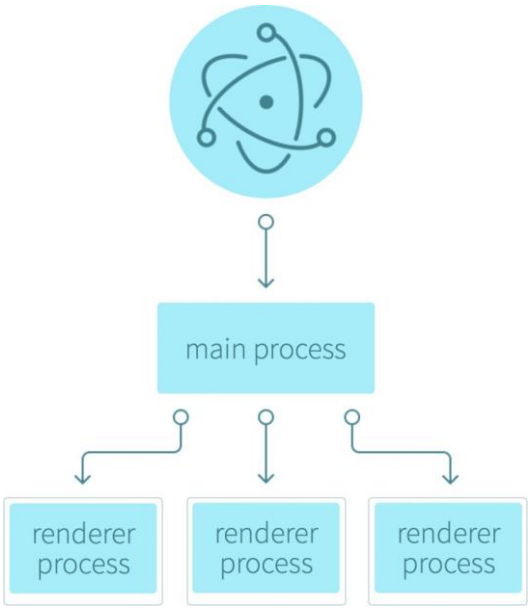
Es la versión de código abierto de Google Chrome. Este proyecto se creó para obtener la ayuda de la comunidad con el fin de mejorar entre todos el motor del navegador, o a su vez crear su propio navegador.

Chromium es usado por Electron como motor de la interfaz, es decir la visión de la aplicación en el lado del cliente está generada por Chromium [10].

3.1.3.- Estructura Electron.JS

El funcionamiento de Electron se divide en dos tipos de procesos, el proceso principal `main` y el proceso `renderer`. El proceso principal es el proceso de Node.js el cual se comunica con varias API de Electron.js para ayudarnos a comunicarnos con el Sistema Operativo, podemos decir que es nuestra aplicación.

El proceso `renderer` es un proceso de Chromium, pero este proceso tiene Node.js incorporado para poder acceder a los módulos y a los posibles módulos que instalemos con npm [10].



1.-Estructura Electron.js [11]

3.2.- PEG.JS

Es un simple generador de analizadores sintácticos para JavaScript. Genera rápidos analizadores con un excelente control de errores [15].

En nuestra aplicación usamos PEG.js para poder analizar fragmentos del lenguaje de Progress y poder crear nuestros objetos que serán tratados en nuestra aplicación.

3.3.- VISUAL STUDIO CODE

Es un editor de código fuente desarrollado por Microsoft, creado con Electron. Lo utilizamos para el desarrollo del proyecto.

3.4.- GITHUB

Es una plataforma web de gestión de proyectos y control de versiones de código.

Permite trabajar colaborativamente con cualquier persona del mundo, planificar proyectos y hacer un seguimiento de los mismos.

Está basado en el sistema de control de versiones Git, desarrollado por Linus Torvalds. Este sistema realiza el control de versiones mediante un registro de cambios realizados en el código.

Usamos esta tecnología ya que, como explicamos en el apartado de Electron, es la compañía desarrolladora de esta herramienta, además de por su facilidad de uso y su integración con Visual Studio Code.

3.5.- LENGUAJE DE PROGRAMACIÓN JAVASCRIPT

Como lenguaje de programación hemos elegido JavaScript. JavaScript es un lenguaje orientado a objetos, débilmente tipado y dinámico [2] ,[7].

Barajamos la posibilidad de usar TypeScript. TypeScript es un superconjunto de JavaScript que esencialmente añade tipos estáticos y objetos basados en clases. Es por eso que nos podía interesar para definir una estructura de datos más rígida. Finalmente decidimos no utilizar TypeScript y perder la capacidad de tener un fuerte tipado porque:

- TypeScript está más indicado para proyectos de mayor tamaño mientras que JavaScript es ideal para proyectos de menor envergadura.
- TypeScript necesita ser compilado.
- Tener que aprender un lenguaje nuevo.

Para definir nuestra estructura de datos, utilizamos clases de JavaScript, las cuáles nos ofrecen lo que necesitábamos para este proyecto. Comentaremos estas clases posteriormente, en el capítulo 5.1.- Estructura de datos en JavaScript.

3.6.- FRAMEWORK PARA LA CAPA DE PRESENTACIÓN JQUERY

Nos decidimos a usar jQuery para el manejo de los componentes visuales (HTML).

jQuery es una biblioteca de JavaScript que nos permite principalmente manipular el árbol DOM (Document Object Model) de una página web, simplificando la manera de interactuar con los documentos HTML y manejar eventos [4],[5].

3.7.- CAPA DE PRESENTACIÓN

3.7.1.- HTML5

HTML5 es la última versión de HTML ("*HyperText Markup Language*"). Podemos definir HTML como un estándar que sirve para definir la estructura y el contenido de una página web [8].

3.7.2.- CSS

CSS (*"Cascading Style Sheets"*), en español «Hojas de estilo en cascada», es un lenguaje de diseño gráfico que nos permite definir y crear la presentación de un documento HTML [2]

3.7.3.- Bootstrap

Bootstrap es un Framework CSS y JavaScript diseñado para la creación de interfaces más limpias de una manera más rápida y cómoda. Incluye plantillas de diseño basadas en HTML y CSS con la que es posible modificar tipografías, formularios, botones, tablas, navegaciones, menús desplegables, etc [1],[3].

Una de las mayores ventajas que aporta es el sistema de grid o cuadrículas de Bootstrap. Se trata de un sistema de cuadrículas que crea diseños de página mediante el uso de varias filas y columnas en las que se inserta el contenido. El tamaño de la cuadrícula se adapta a medida que se redimensiona al tamaño de la ventana. También nos ofrece la posibilidad de definir el tamaño de las columnas en función del tamaño de las ventanas. Esto nos permite crear un diseño adaptado a varios dispositivos.

3.7.4.- jQuery UI

jQuery UI es una biblioteca de componentes para jQuery.

Esta librería añade un conjunto de plug-ins, widgets y efectos visuales que nos permite tener una capa de presentación interactiva desarrollada de manera más rápida y sencilla. Por ejemplo, tenemos la función `sortable`, que utilizamos en el proyecto, la cual nos permite ordenar los elementos de un componente visual, como por ejemplo una lista. Uno de los elementos que también utilizamos en el proyecto es el `datepicker`, que nos hace aparecer un pequeño calendario en una variable y nos permite seleccionar una fecha en diferentes formatos. Aparte de estos ejemplos tiene otras muchas funcionalidades y efectos que no utilizamos en el proyecto como por ejemplo las funciones `draggable, droppable, resizable`, que nos permiten, arrastrar, soltar y redimensionar respectivamente [6].

3.7.5.- Interact.JS

Es una librería de JavaScript la cual está creada para facilitar la captura de eventos de arrastrar y soltar con el ratón, reescalado y gestos multitáctiles [12]. Su API es simple y flexible, está unificada para gestos de pantallas táctiles y eventos de ratón. Esta tecnología es la que usamos a la hora de mover los objetos de nuestra aplicación.

CAPITULO 4.- CREACIÓN DE INTERFACES DE USUARIO

En este punto hablaremos del desarrollo y uso de la interfaz de usuario, que debido a la naturaleza del proyecto es una parte muy importante de éste. Recordemos que nuestro proyecto consiste en crear una aplicación que nos permita desarrollar una interfaz gráfica de manera visual, en lenguaje Progress para el ERP de QAD.

4.1.- FORMULARIOS

Antes de hablar de los formularios es necesario ver las estructuras elegidas en Progress para la creación de interfaces visuales y sus características. Así podremos entender mejor los tipos de campos que hay en los formularios y sus casos de uso.

4.1.1.- Frames

Un frame en Progress se puede definir como un panel donde se engloban las variables a mostrar. Puede concebirse como un conjunto de variables que se muestran en el mismo momento, tanto para introducir información como para mostrar información ya guardada.

Podemos distinguir dos clases de frames: Un frame de entrada es una agregación de varias variables que son mostradas para añadir información. Un frame de salida sirve exclusivamente para mostrar la información.

Un frame puede ser usado tanto para entrada como para salida de datos. Sin embargo, en nuestro proyecto consideramos a los frames de entrada como paneles para la introducción de datos, es decir para pantallas de mantenimiento del ERP, y los frames de salida para pantallas de reporte de información.

4.1.2.- Variables

En Progress tenemos principalmente los siguientes tipos de variables:

- **Integer:** un entero que emplea 4 bytes de almacenamiento. El rango de valores que puede representar va de -2^{31} a $2^{31}-1$.
- **Character:** un carácter o cadena de caracteres.
- **Logical:** variable booleana con valores true o false.
- **Decimal:** dato numérico de hasta 50 dígitos de longitud, incluyendo 10 dígitos a la derecha del punto decimal.
- **Date:** utilizado para representar fechas en diferentes formatos.

En nuestra aplicación hemos creado un formulario para la obtención de datos del frame a crear (como, por ejemplo, su título o sus propiedades) y otro formulario para la información relativa a las variables del frame.

4.1.3.- Formulario de Frame

En un frame puede se pueden definir muchas características, como el color de fondo, elegir si tiene o no borde que englobe el frame, dónde colocar la etiqueta que define la variable, el tamaño del frame, etc. En nuestra aplicación, nosotros hemos elegido una serie de características fijas y otras modificables.

Las características fijas son las siguientes:

- **Tamaño:** Un frame puede ser definido por el número de columnas que puede haber en su interior. Cada columna es del tamaño de un carácter, por lo que se utiliza un formato monoespaciado de letra para mantener la proporción. La mayoría de los frames en QAD son de 80 columnas porque en sus inicios las pantallas que utilizaban los ordenadores no admitían un tamaño mayor. Hoy en día se permite tener más tamaño, pero el estándar sigue siendo 80 columnas de ancho por lo que dejamos esta característica fija a este valor.

- **Posición de las etiquetas:** Las etiquetas son un atributo de las variables que permiten una breve descripción de la variable a la que van asociadas. Es decir, si tenemos una variable llamada “a”, le podemos poner la etiqueta “Número de coches” y así permitimos al usuario que utilice la interfaz comprender qué dato está modificando, independientemente del nombre que utilice esa variable en el código.

  Para los frames de entrada hemos elegido que esta etiqueta se posicione a la izquierda del valor que va a mostrar ya que cumple con el estándar. Otras opciones son: mostrar por encima del valor, o que no exista.

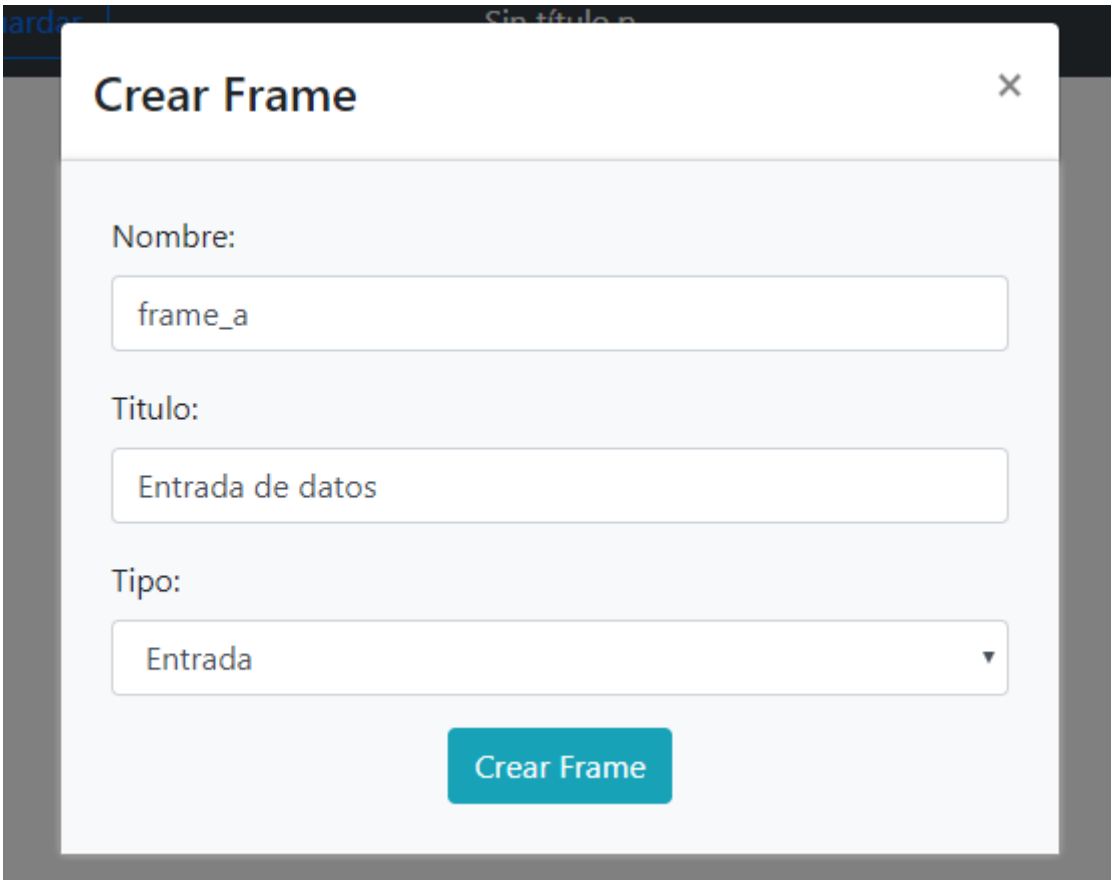
Los frames de salida están pensados como reportes que muestran mucha información en forma de lista, por lo que no hemos seleccionado la posibilidad de ajustar la posición de la etiqueta en estos casos. Cada atributo se representa como una columna, debajo de la cual aparece la información de todas las variables mostradas sin repetirse la etiqueta. Más adelante veremos un ejemplo para clarificar este punto.

Las características modificables de un frame son:

- **Nombre:** Nombre que vamos a dar al frame en el código.

- **Tipo:** Indica si el frame es de entrada o de salida.

- **Título:** Al frame se le puede agregar una descripción que indique para qué servirá el frame si así se desea. Dejamos esta característica a elección del usuario.



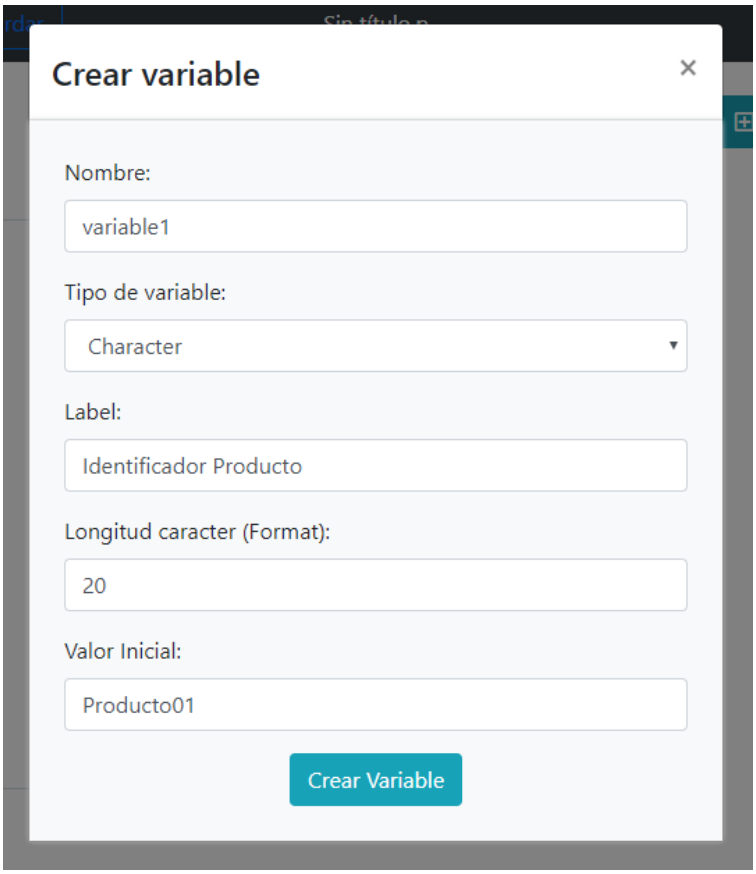
2.- Formulario de Frame

4.1.4.- Formulario de Variable

En una variable tenemos una serie de características que el usuario define al crearla:

- **Nombre:** nombre de la variable a utilizar en el código.
- **Tipo:** Progress ofrece muchos más tipos a elegir que los descritos en la sección 4.1.2.-, pero para simplificar la creación de las variables nos hemos quedado con los tipos básicos: integer, character, decimal, logical y date.
- **Label o etiqueta:** como comentábamos anteriormente, en Progress se permite tener asociada a una variable un nombre o descripción que se muestra en el frame como sustituto del nombre de la variable en el código. Si esta característica se deja vacía toma por defecto el nombre de la variable.
- **Valor Inicial:** Valor que queremos que tenga por defecto nuestra variable al ser creada.
- **Formato:** el formato es una característica que nos indica cómo será mostrada la variable en el frame. En función del tipo de variable tenemos diferentes formatos. Por ejemplo, si la variable es de tipo decimal podemos elegir cuantos decimales debe mostrar la variable; si es de tipo

character, podemos elegir cuántos caracteres se visualizan El ejemplo más claro sería el de las variables tipo date, en el que podemos mostrar las fechas en diversos formatos. Hablaremos más en detalle de los diferentes formatos en el CAPITULO 6.-.



3.- Formulario de Variable

4.2.- FRAME DE ENTRADA

Este apartado lo vamos a dedicar a describir el funcionamiento y la vista del frame de entrada.

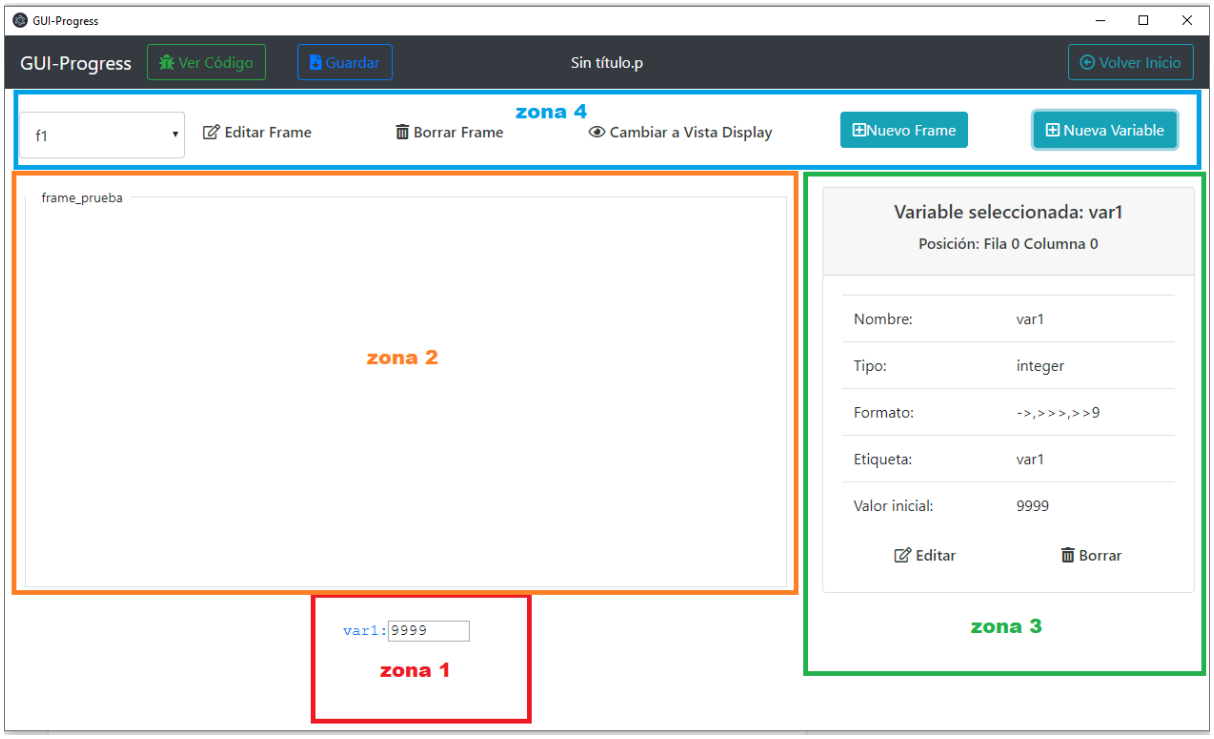
Como se ha comentado en el punto anterior, usaremos el frame de entrada para introducir los datos con los que queremos trabajar, y con ello crearemos una vista de mantenimiento del ERP.

4.2.1.- Zonas principales

Una vez creado el frame, debemos proceder a crear las variables necesarias tal y como se ha indicado en la sección 4.1.4.-. En este apartado podemos diferenciar a grandes rasgos cuatro zonas dentro de la vista del frame de entrada:

- **Zona 1**, en la cual se crean las variables. Si creamos varias variables se van posicionando una debajo de la otra.

- Zona 2 o frame, es la zona limitada en la cual se pueden situar las variables cuando las arrastramos.
- Zona 3, zona en la cual podemos ver un resumen de los datos de la variable actualmente seleccionada.
- Zona 4, controles de selección, edición y creación de frames.



4.-Vista del frame de entrada

4.2.2.- Posicionamiento de Variables

Uno de los puntos clave de este proyecto es la facilidad que damos al posicionamiento de las variables. Estas se pueden mover sobre el frame creado, bien sea arrastrando con el ratón o bien una vez dentro del frame, moviéndolas con las flechas del teclado. Al moverlas con el ratón restringimos su movilidad gracias a la funcionalidad de Interact.js:

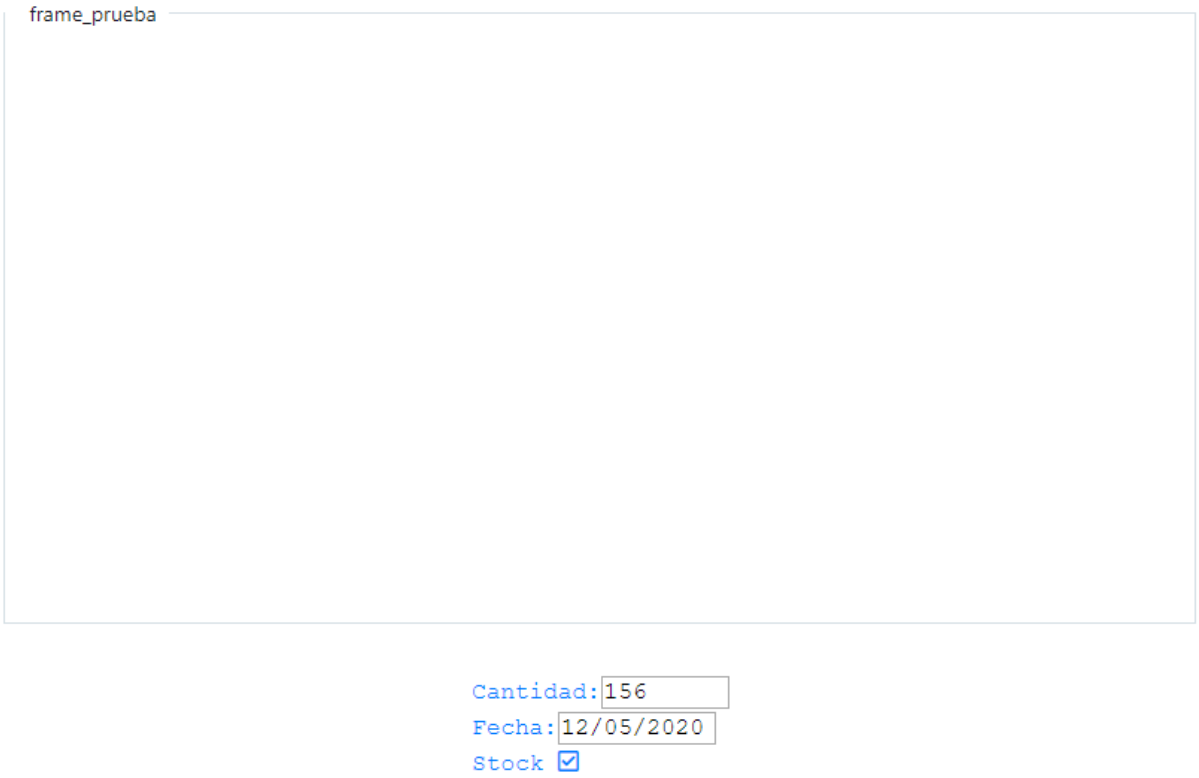
```
interact.modifiers.restrict({
    restriction: document.getElementById("inner-dropzone"),
    endOnly: false
}
),
```

Se aprecian dos restricciones:

- **restriction:** aquí indicamos sobre qué elemento vamos a restringir el movimiento, en este caso lo restringimos a la zona del frame ("`inner-dropzone`")
- **endOnly:** esta restricción se usa a nivel visual. Si se encuentra en "`false`" la variable no va a salir de la zona de restricción cuando la arrastramos. Si se encuentra a "`true`" la variable se puede salir de la zona de restricción cuando la arrastramos, pero en cuanto la soltemos se colocaría automáticamente dentro de la zona restringida.

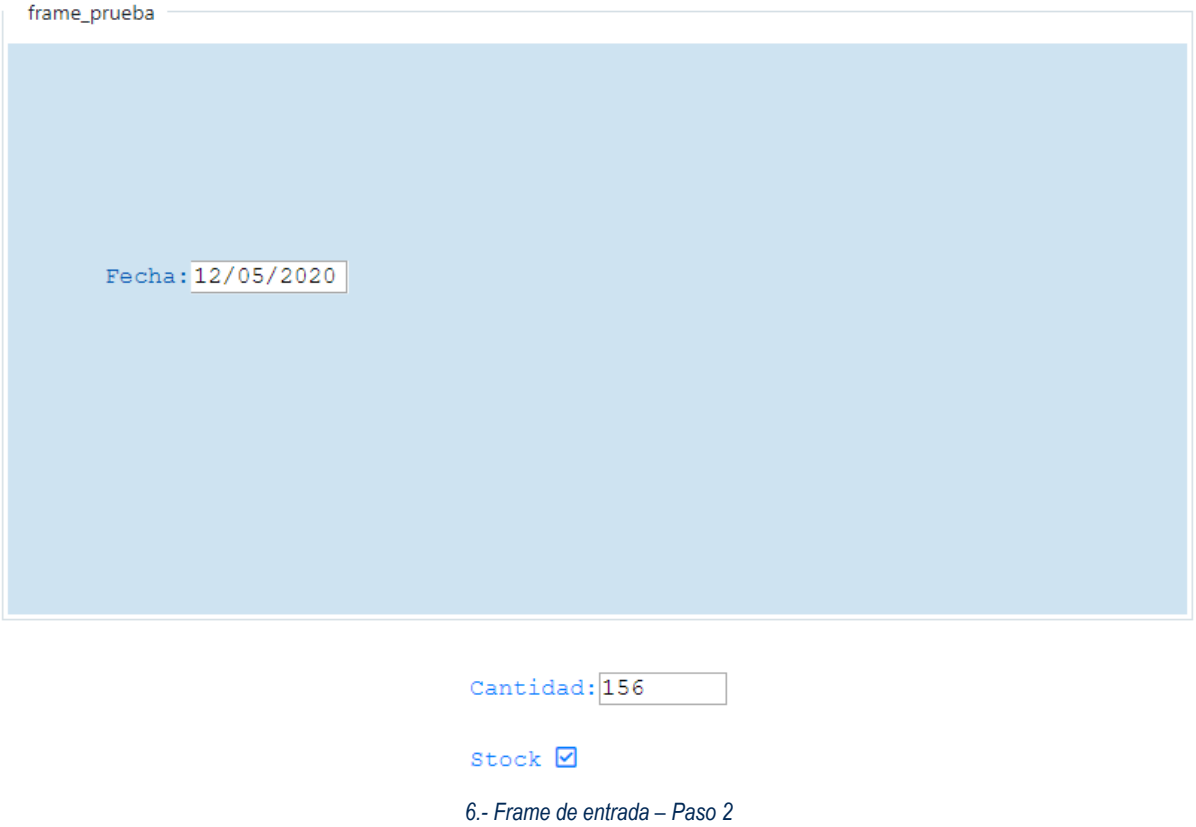
Procedemos a indicar el funcionamiento relativo al arrastre de variables:

- Pongamos el caso en el cual se han creado varias variables. Estas aparecerán en la Zona 1 anteriormente explicada, una debajo de la otra.



5.-Frame de entrada – Paso 1

En este punto sólo podemos mover las variables dentro del frame arrastrándolas con el ratón. Para saber si la variable se puede arrastrar, una vez que estamos arrastrando la variable en la zona del frame, este se colorea de azul.



6.- Frame de entrada – Paso 2

La movilidad de las variables se hace respetando los tamaños de fila y de columna, con lo que cada vez que se mueva la variable avanzará el tamaño que hemos definido para filas y columnas: filas `24px` y columnas `9px`. Esto lo conseguimos de manera muy fácil al usar Interact.js. Dentro de los eventos de la clase `drag-drop` encontramos este fragmento de código:

```
interact.modifiers.snap({
    targets: [
    interact.createSnapGrid({
        x: columnasdis, y: filasdis
    }
    )
    ],
```

Una vez arrastrada la variable, todas las que queden por debajo de ésta se colocarán automáticamente subiendo hacia arriba una posición. En la imagen siguiente podemos apreciar cómo se ha situado la variable “Stock” en la antigua posición de “Fecha”.

7.- Frame de entrada – Paso 3

A continuación, explicamos por encima cómo se consigue esta funcionalidad, ya que en realidad las variables, mientras se están trasladando, no se mueven en el documento HTML, sino que su representación visual se traslada la distancia que deseemos.

En nuestro código HTML debemos diferenciar tres zonas:

- Zona Final en la que se situará la variable desplazada. (`id="movend"`)
- Zona del frame, en la que se encontrará la representación visual de la variable desplazada. (`id="inner-dropzone"`)
- Zona de creación de variables, en la que vemos las variables una vez creadas. (`id="varsMov"`)

```
<div class="col-8 col-sm-8 placeholder">
    <div class="row">
       <div id="movend">        </div>
    </div>
    <div class="row">
       <fieldset>
          <legend class="tituloFrameActual">        </legend>
          <div id="inner-dropzone" class="dropzone">    </div>
       </fieldset>
    </div>
    <div class="row">
```

```
      <div id="varsMov">   </div>
    </div>
 </div>
```

Una vez se ha creado la variable, su `div` correspondiente se crea en `varsMov`. Cuando la variable se mueve a una zona válida del frame, dentro de `inner-dropzone`, creamos un clon del div de la variable. Este clon lo añadimos en `movend`, calculamos las distancias de traslación de la variable para la posición nueva del `div`, y a continuación eliminamos el div original que se encontraba en `varsMov`. Con esto conseguimos que las demás variables de la Zona 1 que aún no se han movido suban las posiciones correspondientes.

```
[…]
var objcln=objeto.cloneNode(true);
$('#movend').append(objcln);
posx=parseInt(objeto.getAttribute('data-x'));
posy=parseInt(objeto.getAttribute('data-y'));
// x=objeto.offsetLeft+posx
// -> dataX=posx+objeto.offsetLeft-clone.offsetLeft
// x=clone.offsetLeft+dataX
var x1=posx+objeto.offsetLeft-objcln.offsetLeft;
// y=objeto.offsetTop+posy
// -> datay=posy+objeto.offsetTop+clone.offsetTop
// y=datay-clone.offsetTop
var y1=posy+objeto.offsetTop+objcln.offsetTop;
// trasladar elemento
objcln.style.webkitTransform =
objcln.style.transform =
'translate(' + x1 + 'px, ' + y1 + 'px)'
// actualizar atributos de posicion
objcln.setAttribute('data-x', x1)
objcln.setAttribute('data-y', y1)
objeto.parentNode.removeChild(objeto);
[…]
```

Además, debemos comentar que una vez seleccionada la variable para arrastrar y esta haya sido situada dentro del frame, ya no podremos sacarla de este. En este punto podemos comentar dos opciones en el caso de intentar arrastrar la variable, con el ratón o con las flechas, fuera de la zona del frame:

- Si intentamos arrastrar la variable fuera de la zona con el ratón:
    - Si la posición anterior estaba dentro de la zona del frame, la variable volverá a situarse de nuevo en la última posición válida, es decir, en la posición en la que se encontraba antes de arrastrarla.

    - Si la variable la movemos por primera vez e intentamos arrastrarla fuera del frame, la variable volverá a su situación inicial, es decir a la posición en la cual se ha creado.

- Si intentamos mover la variable fuera de la zona con las flechas del teclado.
    - En este caso el movimiento se restringe y si llegamos a los límites de la zona la variable no se mueve.

Para esto hemos tenido que crear código específico para saber si la variable se ha arrastrado dentro de la zona válida del frame o no. A continuación detallamos el fragmento de código en el que controlamos si la variable se intenta arrastrar fuera de la zona válida, tanto con ratón como con las flechas del teclado:

- **Mover con el ratón:**

```
//Variable aún no se ha movido
if(variable['movido']==0){
    //Controlamos las X y las Y
    if(posright>(zone.offsetWidth+zone.offsetLeft) ||
    (objeto.offsetLeft+posx)<zone.offsetLeft ||
    (objeto.offsetTop+posy)<zone.offsetTop ||
    posbottom>(zone.offsetHeight+zone.offsetTop)){
        objeto.style.webkitTransform =
        objeto.style.transform =
        'translate(' + 0 + 'px, ' + 0 + 'px)'
        objeto.setAttribute('data-x', 0)
        objeto.setAttribute('data-y', 0)
    }
}
else {
    //Variable movida con anterioridad a zona válida
    if(posright > (zone.offsetWidth+zone.offsetLeft) ||
    (objeto.offsetLeft+posx)<zone.offsetLeft ||
    (objeto.offsetTop+posy)<zone.offsetTop ||
    posbottom>(zone.offsetHeight+zone.offsetTop)){
        //columnasdis y filasdis -> pixeles tamaño de fila y columna
        let datx=(variable["columna"]*columnasdis +
        document.getElementById('inner-dropzone').offsetLeft) -
        document.getElementById(variable["name"]).offsetLeft;
        let daty=(variable["fila"]*filasdis +
        document.getElementById('inner-dropzone').offsetTop) -
        document.getElementById(variable["name"]).offsetTop;
        objeto.setAttribute('data-x', datx);
        objeto.setAttribute('data-y', daty);
        // trasladar elemento
        objeto.style.webkitTransform =
        objeto.style.transform =
```

```
        'translate(' + datx + 'px, ' + daty + 'px)'
    }
}
```

- **Mover con las flechas:**

```
// La variable porcentaje indica el porcentaje mínimo de ocupación que aceptam
os para considerar que la variable se encuentra dentro de la zona para poder m
overla, en nuestro caso es el 95%
if (e.keyCode == '38') {
    // flecha arriba
    let daty = y - filasdis;
    posy = daty;
    if(daty>porcentaje*(document.getElementById('inner-
      dropzone').offsetTop)) {
        calcularPosicionesFlechas(objeto, obj,x,daty);
    }
}
else if (e.keyCode == '40') {
    // flecha abajo
    let daty = y + filasdis;
    posy=daty;
    if(daty+obj.offsetHeight<document.getElementById('inner-
      dropzone').offsetTop+document.getElementById('inner-
      dropzone').offsetHeight){
        calcularPosicionesFlechas(objeto, obj,x,daty);
    }
}
else if (e.keyCode == '37') {
    // flecha izquierda
    let datx =x - columnasdis;
    posx=datx;
    if(datx+obj.offsetLeft>porcentaje*(document.getElementById('inner-
      dropzone').offsetLeft)){
        calcularPosicionesFlechas(objeto, obj,datx,y);
    }
}
else if (e.keyCode == '39') {
    // flecha derecha
    let datx = x + columnasdis;
    posx=datx;
    if(datx+obj.offsetWidth+obj.offsetLeft<document.getElementById('inner-
      dropzone').offsetLeft+document.getElementById('inner-
      dropzone').offsetWidth){
        calcularPosicionesFlechas(objeto, obj,datx,y);
```

```
    }
}
```

4.2.3.- Modificación de variables

Una vez creadas las variables podemos editarlas en caso de necesidad. En el panel de la derecha (Zona 3) podemos ver los datos de la variable seleccionada, que será la última en crearse o la última en seleccionarse.

En la parte superior de este panel vemos el identificador de la variable seleccionada y la posición en la que se encontraría en el frame. Si la posición es columna 0 y fila 0, entendemos que esta variable únicamente ha sido creada y aún no se ha movido dentro del frame. Este dato se modifica una vez se ha situado la variable en una zona válida dentro del frame.

En la parte media de esta zona podemos ver los datos de la variable que hemos creado: etiqueta, tipo de variable, valor inicial y formato.

En la parte inferior del panel hay dos botones, uno para modificar la variable y otro para eliminarla. Si editamos la variable se nos abrirá una ventana idéntica a la de creación de variables, en la que podemos modificar los datos que deseemos. Una vez guardados estos datos la variable se modifica automáticamente.

8.- Panel visualización de variable

4.2.4.- Elementos del Frame.

En la Zona 4 explicada al principio de este capítulo nos encontramos con todas las opciones correspondientes al frame, tanto a nivel de configuración como a nivel visual. Vamos a explicar las diferentes opciones de izquierda a derecha.

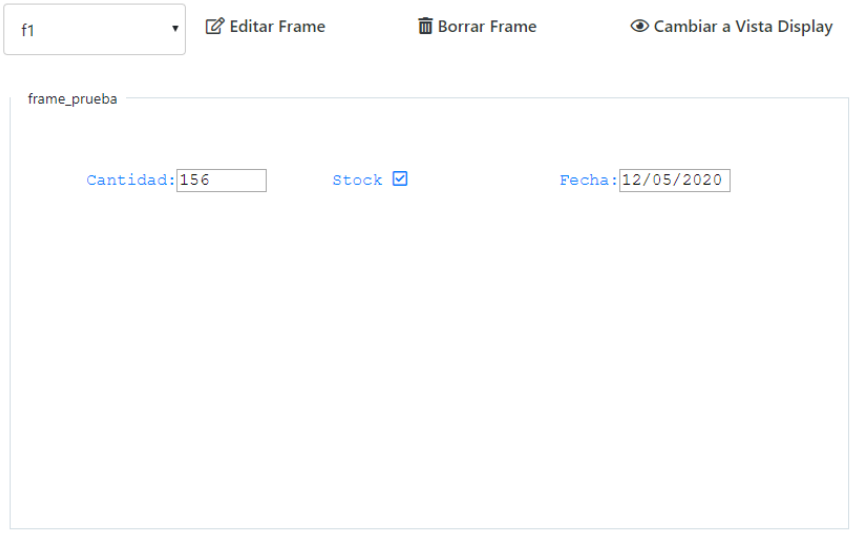
9.- Zona 4 (Elementos del Frame)

- Lista desplegable en la cual podemos seleccionar el frame en el que queremos trabajar. En esta lista se nos mostrarán todos los frames creados en el proyecto actual, tanto de entrada como de salida.
- Botón "Editar Frame". Si seleccionamos esta opción se nos mostrará una ventana con la configuración del frame en el que nos encontramos. Esta información se puede modificar y los cambios se aplicarán en cuanto los aceptemos. La ventana que se abre es muy parecida a la que se nos abre al crear Frame, explicado en puntos anteriores. La diferencia con este, es que ahora no podemos modificar si el frame es de entrada o salida:

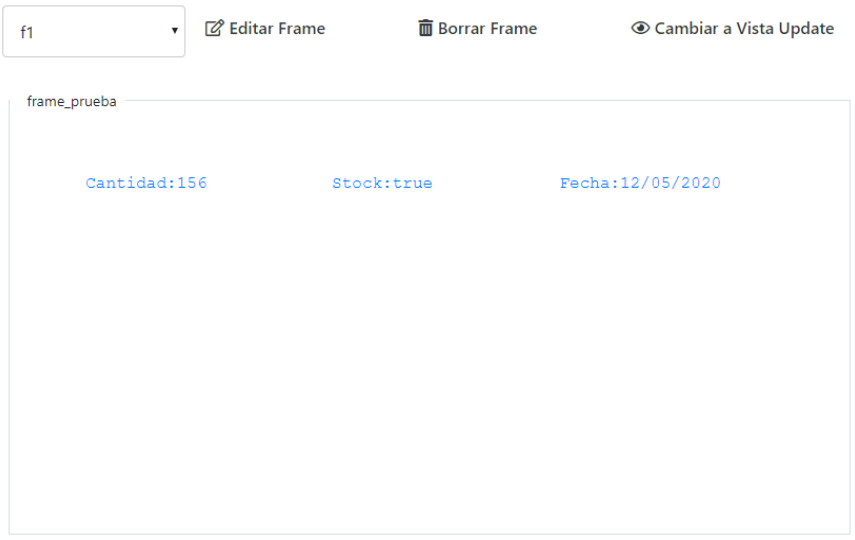
10.- Ventana Editar Frame

- Borrar Frame. Si seleccionamos esta opción el frame se elimina por completo, no pudiendo recuperar nada de lo que teníamos creado dentro de este frame.
- Cambiar a vista Display o cambiar a vista Update. Este botón tiene la funcionalidad de cambiar la vista del frame. Su texto cambia en función de la vista en la que nos encontremos, mostrando el cambio hacia la vista contraria. La vista que tenemos por defecto es la vista Update, en la que se nos muestran las variables con los campos editables. Si, por el contrario, deseamos que estos campos no se puedan editar, deberemos cambiar la vista a "display", en la que los campos dejan

de ser editables. A continuación, dejamos dos imágenes en las que se puede apreciar la diferencia entre ambas vistas:



11.- Vista Update



12.- Vista Display

4.2.5.- Código generado

A continuación, podemos ver un ejemplo del código creado en un frame de entrada. Según la figura 11.- Vista Update

```
/* BEGIN VARS Autogenerated */
define variable var1 as integer init 156 format "->,>>>,>>9" no-undo .
define variable var2 as date  init "12/05/2020" format "mm/dd/yy" no-undo .
define variable var3 as logical  init true no-undo .
/* END VARS Autogenerated */
```

```
/* BEGIN INPUT Frame Autogenerated */
define frame f1
   var1 label "Cantidad" at row 3 column 17
   var2 label "Fecha" at row 3 column 64
   var3 label "Stock" at row 3 column 40
with side-labels title "frame_prueba".
/* END INPUT Frame Autogenerated */
```

4.3.- FRAME DE SALIDA

Tal como hemos comentado, el frame de salida es un frame que será utilizado para hacer listas de reportes de información. En términos de diseño, son mucho más sencillos que los frames de entrada.

Para entender mejor esta interfaz es necesario introducir un nuevo concepto: las tablas temporales de Progress.

En Progress, existen unas estructuras, llamadas `temp-tables`, que son similares a un `struct` en C++. Es un tipo de dato compuesto que permite almacenar un conjunto de datos de diferente tipo. En lenguaje natural lo podríamos definir como una súper-variable que engloba diferentes campos, que son a su vez variables.

Cuando elegimos un frame de salida, utilizamos este tipo de estructura para agrupar las variables.

Ejemplo: si creamos un frame de salida para los productos que vendemos en un almacén, podríamos crear varias variables que corresponden a un mismo producto, pero que definen diferentes características, como, por ejemplo: identificador, cantidad, precio unitario y un valor booleano que nos diga si hay o no stock disponible en estos momentos.

En QAD nos mostraría el resultado de la siguiente forma:

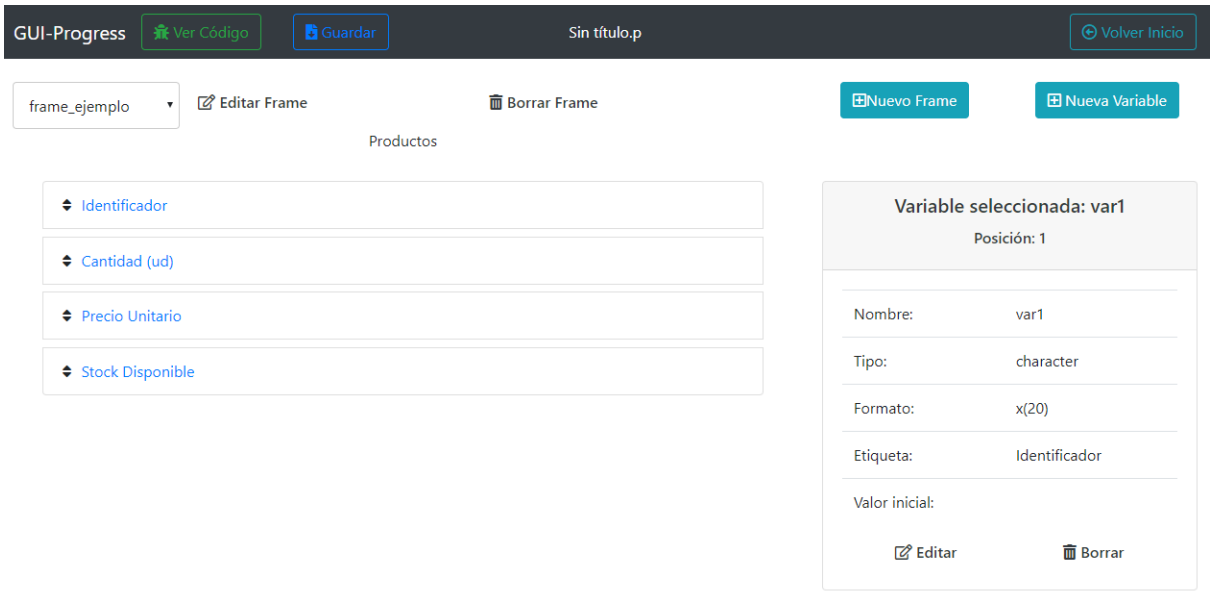
```
                          Productos
Identificador       Cantidad (ud) Precio Unitario Stock Disponible
------------------- ------------- --------------- ----------------
Producto1                     20           35.20 yes


                                                   End of Report
```

13.-QAD

Es por ello que, en esta interfaz (14.- Ejemplo Frame de Salida), las variables se crean como elementos de una lista. Y son intercambiables entre sí para elegir el orden en el que aparecen en el reporte.

Para permitir al usuario arrastrar y soltar los elementos de una lista con el fin de disponerlos en un determinado orden, hemos utilizado la función `sortable` de la librería jQuery UI. Esta librería permite al usuario ordenar de manera sencilla los elementos visuales de una lista de HTML y transformar esa información en un `array`. De esta forma, cuando el usuario intercambia visualmente las posiciones relativas de dos variables, este cambio se refleja en las posiciones de dichas variables dentro del `array`.

Siguiendo con el ejemplo anterior, la creación del frame en nuestra aplicación quedaría de la siguiente forma:



14.- Ejemplo Frame de Salida

El código generado para este frame es el siguiente:

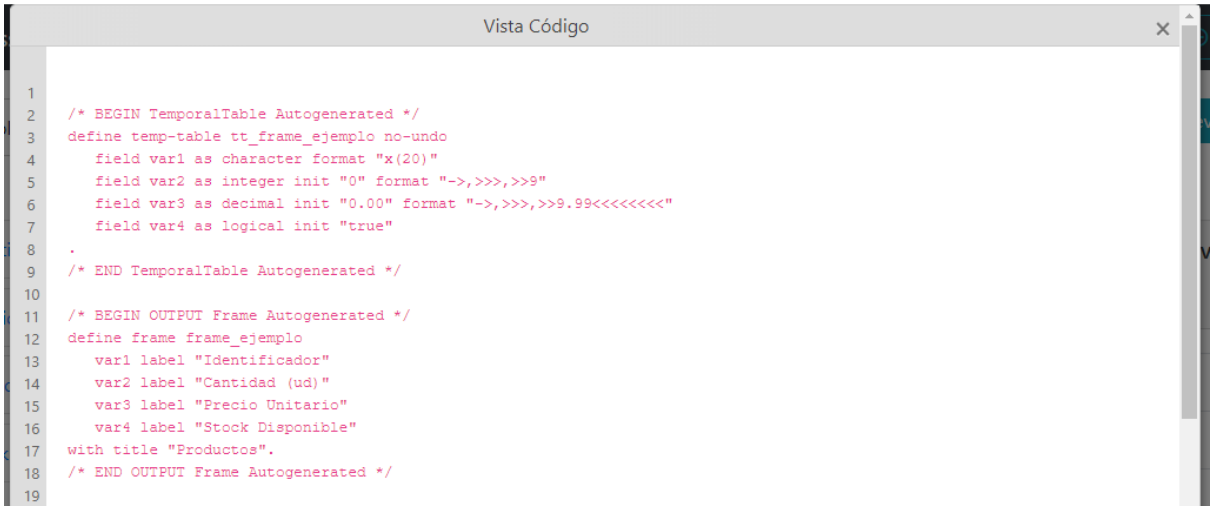
```
/* BEGIN TemporalTable Autogenerated */
 define temp-table tt_frame_ejemplo no-undo
    field var1 as character        format "x(20)"
    field var2 as integer  init 0        format "->,>>>,>>9"
    field var3 as decimal init 0.00 format "->,>>>,>>9.99<<<<<<<<"
    field var4 as logical   init true
 .
 /* END TemporalTable Autogenerated */

 /* BEGIN OUTPUT Frame Autogenerated */
 define frame frame_ejemplo
    var1 label "Identificador"
    var2 label "Cantidad (ud)"
    var3 label "Precio Unitario"
    var4 label "Stock Disponible"
 with title "Productos".
```

```
/* END OUTPUT Frame Autogenerated */
```

En esta tabla temporal se pueden crear diferentes registros, para luego ser mostrados con el frame `frame_ejemplo` del código anterior, tal como se muestra en la figura (13.-QAD)

4.4.- VISTA DE CÓDIGO

Como sabemos, nuestro objetivo es crear código Progress que nos permita hacer una interfaz visual para QAD, pero de una forma menos tediosa y mucho más rápida que si escribiéramos el código directamente. En parte, esto viene derivado de que Progress no estaba pensado en un principio para tener interfaces visuales amigables para el usuario, ya que es un lenguaje concebido para el manejo rápido y sencillo de grandes volúmenes de datos guardados en el sistema. Es por ello que hemos creado una vista de cómo quedaría el código Progress generado en el momento en el que lo estamos desarrollando. Así podemos ver rápidamente cómo quedará nuestro código antes de guardarlo realmente en un archivo. Esta utilidad ayuda incluso a entender qué consecuencias en código tiene la acción que el usuario está realizando visualmente.



15.- Vista de Código

CAPITULO 5.- GENERACIÓN Y ANÁLISIS SINTÁCTICO DE CÓDIGO PROGRESS

En este capítulo hablaremos de la conversión de código Progress a JavaScript y viceversa.

Como ya sabemos, nuestra aplicación nos permite crear interfaces visuales de una manera simple y obtener el código Progress correspondiente. Hablaremos primero de la estructura de datos, en JavaScript, que utilizamos para guardar esta información y más tarde de cómo se transforma esta estructura a código Progress.

5.1.- ESTRUCTURA DE DATOS EN JAVASCRIPT

Como elementos principales en el modelo de datos de la aplicación tenemos los frames y las variables. Cabe recordar que un frame en Progress nos permite dar un formato visual a las variables a la hora de presentarlas al usuario., es decir, nos dice cómo se verán las variables.

Nuestra estructura consta de tres entidades: los frames, las variables y una clase auxiliar que hemos denominado Progress.

5.1.1.- Clase Progress

Esta clase es la que creamos cuando el usuario elige crear una nueva plantilla o al cargar un archivo. Es la que guarda toda la información de variables y frames con los que estamos trabajando.

Esta clase tiene como atributos:

- **idFrames:** es un contador que utilizamos como identificador cuando creamos los diferentes frames.
- **frames:** es un Map de objetos Frame.
- **fileName:** atributo en el que guardamos el nombre del archivo en el que guardaremos el código Progress generado o en el que está guardado.

En cuanto a las funciones, podemos destacar las siguientes:

- **addFrame:** esta función recibe como parámetros las características del frame, crea un objeto de la clase Frame con el identificador correspondiente y lo guarda en el Map frames. Finalmente, actualiza el contador de frames.
- **addVartoFrame:** recibe como parámetro la información de una variable y el frame al que debe añadirse. Busca el frame correspondiente y llama a la función addVariable de éste.

- **addVartoOutputFrame**: es similar a la anterior, pero además de añadir la variable al frame de salida, añade la información de la variable en el array de posiciones. Esto se debe, a que como hemos hablado anteriormente, un frame de salida y uno de entrada son diferentes. El frame de salida guarda las posiciones en las que están las variables en un array, mientras que el frame de entrada no guarda las posiciones, sino que son las variables las que guardan dicha información.

Aquí podemos ver en detalle la clase Progress:

```
class Progress{
    constructor(){
        this.idFrames = 1;
        //inicializamos ids a 1
        this.frames = new Map();
        //inicializamos map de frames
        this.fileName = "Sin título.p" //inicializamos el nombre del archivo
    }
    //LLamamos al constructor de la clase frame para crear un nuevo frame, lo
      metemos en nuestro array y actualizamos idFrames
    addFrame(name, title, type){
        let newFrame=new classFrame.Frame(this.idFrames,name,title,type);
        this.frames.set(this.idFrames,newFrame);
        this.idFrames ++;
        return this.idFrames -1;
    }
    addVartoFrame(idFrame,varInfo){
        let frame = this.frames.get(parseInt(idFrame));
        return frame.addVariable(varInfo['name'],varInfo['type'],
        varInfo['format'],varInfo['label'],varInfo['initial'],
        varInfo['tam']);
    }
    addVartoFrameRead(idFrame,varInfo){
        let frame = this.frames.get(parseInt(idFrame));
        /* console.log("variable que me llega: ", varInfo); */
        return frame.addVariableRead(varInfo['name'], varInfo['type'],
        varInfo['format'],varInfo['label'],varInfo['initial'],
        varInfo['col'],varInfo['ro   w'],varInfo['movido']);
    }
    addVartoOutputFrame(idFrame,varInfo){
        let frame = this.frames.get(parseInt(idFrame));
        let id = frame.addVariable(varInfo['name'],varInfo['type'],
        varInfo['format'],varInfo['label'],varInfo['initial'],0);
        frame.addVariableOutput(id);
        return id;
```

```
    }
    getFrames(){
        return this.frames;
    }
    getFrame(idFrame){
        return this.frames.get(parseInt(idFrame));
    }
    getFileName(){
        return this.fileName;
    }
    setFileName(fileName){
        this.fileName = fileName;
    }
    deleteFrame(idFrame){
        this.frames.delete(parseInt(idFrame));
    }
    deleteVariable(idFrame, idVar){
        this.frames.get(parseInt(idFrame)).deleteVariable(idVar);
    }
    editFrame(idFrame, newData){
        let frame = this.frames.get(parseInt(idFrame));
        frame.editData(newData);
    }
    getVariableByKey(idFrame,keyVar){
        let frame=this.frames.get(parseInt(idFrame));
        return frame.getVariable(keyVar);
    }
}
```

5.1.2.- Clase Frame

Utilizamos esta clase para guardar la información de cada frame, con sus características y las variables que mostrará.

Sus atributos son los siguientes:

- **idVar:** contador que utilizaremos de identificador para las variables asignadas a este frame.
- **id:** el identificador con el que localizar a este frame en la clase Progress. Viene inicializado por esta última clase al crear un objeto de clase Frame.
- **name:** nombre del frame con el que será identificado en el código Progress.
- **title:** característica *title* que utilizaremos en el código Progress.
- **type:** atributo que nos indica si el frame es de entrada o de salida.

- **vars:** Map que contendrá las variables asignadas a este frame.
- **view:** a la hora de visualizar el frame en la aplicación, nos indica si se está visualizando en modo update o en modo display. Esto está explicado con más detalle en el punto 4.2.4.-. Utilizamos esta característica para ir cambiando de una vista a otra.
- **varsOutFrame:** este atributo es sólo utilizado por los frames de salida. Se trata de un array de identificadores de variables asignadas a este frame. Este array se ordena en función de la posición en la que queramos que se vean las variables en la salida. Para entender mejor esta característica revisar el punto 4.3.-.

El código de esta clase es el siguiente:

```
class Frame {
    /*Type  = 0 entrada
    = 1 salida
    */
    constructor(id,name,title,type){
        this.idVar=1;
        // inicializamos ids variables
        this.id=id;
        // mi id me viene dado de la clase Progress
        this.name=name;
        this.title=title;
        if (type == "0")
        this.type="input";
        if (type == "1")
        this.type="output";
        this.vars= new Map();
        //inicializamos map de variables
        this.view = "update";
        this.varsOutFrame = new Array(0);
        //inicializamos array para variables de salida
    }
    addVariable(name, type, format, label, initial, tam){
        let newVar = new classVar.Variable(this.idVar,name, type, format,
                                           label, initial, tam);
        this.vars.set(this.idVar,newVar);
        this.idVar ++;
        return this.idVar -1;
    }
    addVariableRead(name, type, format, label, initial,col,row,movido){
        let newVar = new classVar.Variable(this.idVar,name, type, format,
                                           label, initial);
        if(movido==1){
```

```
            newVar.setMovido();
        }
        if(col!=0)
        newVar.setCol(col);
        if(row!=0)
        newVar.setFila(row);
        this.vars.set(this.idVar,newVar);
        this.idVar ++;
        return this.idVar -1;
    }
    addVariableOutput(idVar){
        this.varsOutFrame.push(idVar.toString());
    }
    actualizarPosiciones(arrayActualizado){
        this.varsOutFrame = arrayActualizado.slice();
    }
    getPosicionOutput(idVar){
        return this.varsOutFrame.indexOf(idVar.toString()) + 1;
    }
    getVista(){
        return this.view;
    }
    setVista(vista){
        this.view = vista;
    }
    getFrame(){
        return this;
    }
    getVariables(){
        return this.vars;
    }
    getVariablesOutput(){
        return this.varsOutFrame;
    }
    getVariable(idVar){
        return this.vars.get(parseInt(idVar));
    }
    getNombre(){
        return this.name;
    }
    getTitulo(){
        return this.title;
    }
    getTipo(){
```

```
        return this.type;
    }
    editData(newData){
        this.name=newData.name;
        this.title=newData.titulo;
        this.type=newData.tipo;
    }
    editVar(newData){
        this.vars.get(parseInt(newData.idVar)).editData(newData);
    }
    deleteVariable(idVar){
        this.vars.delete(parseInt(idVar));
        if(this.type == "output"){
            this.varsOutFrame.splice(this.varsOutFrame.indexOf(idVar.toString(
            )),1);
        }
    }
}
```

5.1.3.- Clase Variable

Esta clase es utilizada para guardar la información de las variables.

Sus atributos son los siguientes:

- **id:** es el identificador usado para encontrarla en el Map de variables de la clase Frame a la que corresponde.
- **name:** en este campo guardamos el nombre con el que será identificada la variable al generar el código Progress.
- **type:** nos indica el tipo de variable. Los tipos son los mencionados en la sección 4.1.2.-.
- **format:** usamos esta característica para guardar el formato con el que el usuario de la aplicación ha indicado que deberá verse la variable en el frame. Veremos más en profundidad los diferentes formatos en la sección 6.1.1.-Validación de datos en la creación de código.
- **label:** en este atributo guardamos la etiqueta con la que aparecerá el valor de la variable en el frame.
- **initial:** nos indica el valor inicial de la variable.
- **posx:** guardamos la distancia respecto a la X que se ha movido la variable.
- **posy:** guardamos la distancia respecto a la Y que se ha movido la variable

- **movido:** indica si la variable ha sido arrastrada dentro del frame.
- **fila:** en los frames de entrada este atributo guarda la posición de la fila en la que se encuentra la variable. Recordamos que hay 16 filas.
- **columna:** en los frames de entrada este atributo guarda la posición de la columna en la que se encuentra la variable. Hay 80 columnas entre las que puede colocarse.

Para ver más detalles de la clase variable aquí tenemos su código:

```
class Variable{

    constructor(id,name, type, format, label, initial, tam){
        this.id=id;
        this.name=name;
        this.type=type;
        this.format=format;
        this.label=label;
        this.initial=initial;
        this.posx=0;
        this.posy=0;
        this.fila=0;
        this.columna=null;
        this.movido=0;
        this.tam = tam;
    }
    editData(newData){
        this.name=newData.name;
        this.type=newData.type;
        this.format=newData.format;
        this.label=newData.label;
        this.initial=newData.initial;
    }
    getVar(){
        return this;
    }
    getNombre(){
        return this.name;
    }
    getTipo(){
        return this.type;
    }
    getFormato(){
        return this.format;
    }
    getLabel(){
```

```
    return this.label;
}
getInitial(){
    return this.initial;
}
getPosition(){
    let position={
        'x':this.posx,
        'y':this.posy
    }
    return position;
}
setPosition(x,y){
    this.posx=x;
    this.posy=y;
}
setFilaCol(x,y){
    this.fila=x;
    this.columna=y;
}
getFilaCol(){
    let position={
        'x':this.fila,
        'y':this.columna
    }
    return position;
}
getFila(){
    return this.fila;
}
getColumna(){
    return this.columna;
}
setFila(i){
    this.fila=i;
}
setCol(i){
    this.columna=i;
}
getMovido(){
    return this.movido;
}
setMovido(){
    this.movido=1;
```

```
    }
    getTam(){
        return this.tam;
    }
}
```

5.2.- ESTRUCTURA DE DATOS EN PROGRESS

Una vez aclaradas las estructuras de datos utilizadas en la aplicación vamos a hablar de las tres estructuras de Progress que hemos escogido para la creación de las interfaces visuales.

Estas estructuras son: ***variables, frames y tablas temporales***.

Ya hemos comentado anteriormente el concepto que representan cada una de estas estructuras así que nos centraremos en la sintaxis utilizada para después entender mejor el parseador de código Progress.

5.2.1.- Generación de código Progress

Pondremos un ejemplo para entender mejor la sintaxis de Progress. Vamos a crear un frame de entrada, al que llamaremos artículo y que contendrá 2 variables: `variable var1` y `variable var2`.

Para definir las variables utilizamos las siguientes líneas de código:

```
define variable var1 as integer init 156 format "->,>>>,>>9" no-undo .
define variable var2 as logical init true no-undo .
```

Como podemos observar empezamos con la palabra reservada `define` seguida de lo que vamos a definir, en este caso una variable. Después se especifica el nombre de la variable y el tipo. Adicionalmente, podemos incluir el valor inicial con `init` seguido del valor y el formato con el que deseamos que se muestre. Finalmente se acaba la definición con la palabra reservada `no-undo`, que indica que el valor de esta variable no se perderá si hay algún error y se está utilizando la variable en una transacción. Rara vez es necesario que se pierda el valor y por defecto siempre se definen las variables con `no-undo.`

Para definir el frame de entrada en el que se sitúan las variables previamente definidas tenemos el siguiente código:

```
define frame articulo
   var1 label "Cantidad" at row 3 column 17
   var2 label "Stock"      at row 3 column 40

with side-labels title "Ejemplo".
```

Aquí podemos ver que de nuevo empezamos con la palabra `define` seguida de `frame` y el nombre que queremos dar a éste. A continuación, se colocan las variables que se quieren mostrar con este frame,

en este caso `var1` y `var2`. A las variables le añadimos el atributo `label` para indicar la etiqueta con la que queremos que se muestren en la pantalla y además utilizamos `at row [valor] column [valor]` para indicar la posición de la pantalla donde deben colocarse. Finalmente acabamos con las características de `side-label` y `title` que son las elegidas por defecto para estos frames, tal como comentamos en el apartado 4.1.3.-.

Para el frame de salida, la sintaxis es algo diferente como se menciona en el capítulo 4.3.-

Utilizaremos el siguiente ejemplo: creamos un frame de salida para los productos que vendemos en un almacén: crearemos varias variables que corresponden al producto pero que definen diferentes características, como, por ejemplo: identificador, cantidad, precio unitario y un booleano que nos diga si hay o no stock disponible en estos momentos.

El código es el siguiente:

```
define temp-table tt_frame_ejemplo no-undo
    field var1 as character        format "x(20)"
    field var2 as integer  init 0        format "->,>>>,>>9"
    field var3 as decimal init 0.00 format "->,>>>,>>9.99<<<<<<<<"
    field var4 as logical   init true
.
define frame frame_ejemplo
    var1 label "Identificador"
    var2 label "Cantidad (ud)"
    var3 label "Precio Unitario"
    var4 label "Stock Disponible"
with title "Productos".
```

Como vemos en el frame de salida utilizamos las tablas temporales para definir las variables, en lugar de definir las variables por separado. Pero la estructura es muy similar, a excepción de que primero definimos la tabla temporal con su nombre y dentro no ponemos `define` variable sino `field`.

En la definición del frame también encontramos diferencias con respecto a los frames de entrada. Por ejemplo, en las variables no ponemos la posición, sino que serán colocadas en el orden en el que aparecen ordenadas dentro del frame y la característica *side-labels* es eliminada.

Hay otras formas de definir los frames y se pueden utilizar diferentes estructuras para crear una interfaz visual en Progress. Además, hay muchos más atributos aplicables al frame, como por ejemplo el borde, color de fondo, transparencia, etc. Al igual que para las variables, como por ejemplo otros tipos y otras maneras de definir los tipos, pero hemos elegido estas características y sintaxis para realizar este proyecto.

Además, colocamos algunas marcas en el código para ser utilizadas a la hora de delimitar las regiones de código que serán analizadas sintácticamente. Las marcas generadas son las siguientes:

- `/* BEGIN VARS Autogenerated */` y `/* END VARS Autogenerated */` que engloban las definiciones de variables de los frames de entrada.

- `/* BEGIN TemporalTable Autogenerated */` y `/* END TemporalTable Autogenerated */` que son colocadas en cada definición de tabla temporal que es utilizada en un frame de salida.
- `/* BEGIN INPUT Frame Autogenerated */` y `/* END INPUT Frame Autogenerated */` nos marcan los frames de entrada.
- `/* BEGIN OUTPUT Frame Autogenerated */` y `/* END OUTPUT Frame Autogenerated */` nos señalan cada frame de salida.

Con estas marcas sabemos qué parte de código mandar a cada uno de los analizadores sintácticos que utilizamos para obtener los datos del código como se comenta en la próxima sección.

5.3.- ANALIZADOR SINTÁCTICO DE PROGRESS

Una de las funcionalidades de la aplicación es poder leer código Progress y cargar los frames y variables correspondientes en nuestra aplicación para ser modificados.

Una vez entendida la representación de las estructuras en los diferentes lenguajes podemos meternos en profundidad en el análisis de código Progress.

Como se menciona en el CAPITULO 3.- utilizamos el analizador sintáctico PEG.js.

5.3.1.- Gramática y Semántica de PEG.js

Una gramática en PEG.js consiste en una serie de reglas. Cada regla tiene un nombre que identifica esa regla y una expresión de análisis que define el patrón que debe cumplir la entrada.

También se puede introducir código en JavaScript para definir acciones cuando la entrada cumple el patrón, es decir, al pasar la regla satisfactoriamente.

Una regla puede contener también una etiqueta que puede ser usada para la gestión de errores. El análisis comienza en la primera regla, que también es llamada la regla de inicio. La regla de inicio puede ser precedida por un inicializador, esto no es más que código JavaScript entre llaves “{” “}”. Este código se ejecuta antes de que el generador de análisis comience. Todas las variables y funciones definidas en el inicializador son accesibles en las reglas.

Tipos de expresiones de análisis:

- **"literal" o 'literal'**: Comprueba que los siguientes caracteres de la entrada se ajustan exactamente al `string` indicado y lo devuelve. Añadiendo i a la derecha del literal hacemos el ajuste insensible a mayúsculas.
- **.** : Ajusta exactamente con un carácter y lo devuelve como string.

- **[characters]**: Ajusta la entrada con un carácter del conjunto dado y lo devuelve como `string`. La lista de caracteres puede contener también rangos (por ejemplo, [a-z] significa “todas las letras en minúscula”). Poner delante del carácter “^” niega la condición (ej: [^a-z] significa “todos los caracteres que no sean letras minúsculas”). Añadiendo i a la derecha lo hacemos insensible a mayúsculas.
- **regla:** Ajusta la entrada con el análisis de una expresión o una regla recursivamente y devuelve su resultado.
- **(expresión):** Ajusta con una subexpresión y devuelve el resultado.
- **expresión *:** Ajusta con cero o más repeticiones de la expresión y devuelve sus resultados en un array.
- **expresión +:** Ajusta con una o más repeticiones de la expresión y devuelve su resultado en un array.
- **expresión ?:** Intenta ajustar con la expresión dada. Si lo hace con éxito, devuelve el resultado, en otro caso, devuelve `null`.
- **& expresión :** Intenta ajustar la entrada con la expresión dada. Si lo hace con éxito, devuelve `undefined` y no consume ninguna entrada, en otro caso, considera la verificación fallida.
- **! expresión :** Intenta ajustar con la expresión dada. Si falla devuelve `undefined` y no consume ninguna entrada.
- **{ predicado}:** El predicado es una parte de código JavaScript que se ejecuta dentro de una función.
- **$ expresión:** Ajusta con la expresión dada y si el ajuste tiene éxito devuelve el texto parseado como resultado.
- **etiqueta: expresión:** Devuelve el resultado bajo la etiqueta especificada. La etiqueta debe ser un identificador de JavaScript.
- **expresión1 expresión2... expresiónN:** Ajusta con una secuencia de expresiones y devuelve el resultado en un `array`.
- **expresión{ acción}:** Ajusta la entrada con la expresión dada, si el ajuste tiene éxito, ejecuta la acción.
- **expresión1 / expresión2 / ... / expresiónN:** Intenta ajustar con la primera expresión. Si falla, sigue con la siguiente y así sucesivamente.

5.3.2.- Reglas utilizadas

A continuación, veremos un ejemplo de uno de los analizadores sintácticos que hemos utilizado en el proyecto. Hemos realizado tres analizadores diferentes. Uno para frames, otro para `temp-tables` y otro para las variables. En este ejemplo vemos como analizamos sintácticamente las variables:

```
variables = variable+

variable = _ defineVar __ line:line _ noUndo _ "."_{
      return {vars: line}
}

defineVar = "define variable"

noUndo  = "no-undo"

line = v:var _ opciones:Opcion+ _{
      return {name: v, opciones: opciones}
}

var = !(reservedWords __) ("_" / [a-zA-Z0-9] / "-")*  {
      return text()
}

reservedWords = "field" / defType / "temp-table"
               / noUndo / "as " / "init " / "format "

Opcion = _ opcion:(type/OpcionLabel/OpcionInit/OpcionFormat) {
      return opcion
}

type = "as" __ t:$defType {
      return { type:"type", value: t}
}

defType = "integer" / "character" / "date" / "logical" / "decimal"

OpcionLabel = "label" __ cadena:LiteralCadena {
      return { type: "label", label: cadena }
}

OpcionInit = "init" __ result:(LiteralCadena/Integer) {
      return { type: "init", value: result }
}
```

```
OpcionFormat = "format" __ cadena:LiteralCadena {
      return { type: "format", format: cadena }
}

LiteralCadena = "\"" texto:([^\"]*) "\"" {
      return texto.join("")
}
Integer "integer"
= _ [0-9]+ { return parseInt(text(), 10); }

_ "whitespace"
= [ \t\n\r]*

__ "whitespace_mandatory"
= [ \t\n\r]+
```

A continuación, vamos a analizar las reglas utilizadas. Para hacerlo más cómodo obviamos las reglas "_" y "__" que permiten los espacios y saltos de línea. Como primera regla tenemos "*variables*", que comprueba que puede haber una o más apariciones de la regla "*variable*". La regla "*variable*" es una sucesión de reglas, por lo que debe cumplir cada una de ellas. Dentro de ésta, tenemos la regla `defineVar`, la regla `noUndo` y la expresión ".", todas ellas comprueban un literal. Después nos encontramos con la regla `line`. Esta regla consta de la regla `var` que comprueba que el identificador de una variable sea cualquier carácter. Aquí tenemos que tener en cuenta la regla `reservedWords` que comprueba que el identificador no sea una palabra reservada. También tenemos la regla `Opciones`, en esta regla comprobamos si aparecen cada una de las características que esperamos en una variable y devuelve el resultado como texto con la característica correspondiente.

Como ejemplo de una de las características tendríamos label. Con la regla `OpcionLabel` comprobamos el literal `label` y el literal que lo acompañe entre comillas. De esta forma, devolvemos un objeto con dos campos: el campo `type` es igual `label` y el campo label es igual al texto entre comillas que lo acompañe.

De esta forma conseguimos tener una estructura de datos que nos devuelve la información en un objeto JavaScript que posteriormente utilizaremos a la hora de abrir el archivo.

CAPITULO 6.- OTRAS FUNCIONALIDADES

Para poder conseguir un correcto funcionamiento de toda la aplicación hemos tenido que ser muy rigurosos con ciertos parámetros, como, por ejemplo, la validación de datos.

Una gran debilidad de seguridad de una aplicación web es la falta de validación de datos asociada a la entrada de datos por parte del cliente. Nunca debemos confiar en los datos que introduce el cliente, sino que tenemos que garantizar que la aplicación sea robusta frente a la obtención de datos por parte del usuario.

La validación de datos es muy importante para que no haya problemas con la funcionalidad de nuestro proyecto. En los siguientes puntos vamos a comentar como se ha desarrollado el control de los datos de entrada.

6.1.- VALIDACIÓN DE DATOS

Este punto lo debemos tratar desde dos puntos de vista diferentes. En primer lugar, vamos a comentar la validación de datos desde la parte de creación de código por el usuario, y la segunda parte nos vamos a centrar en la validación de datos a la hora de cargar un archivo de Progress.

Toda validación de datos se ejecuta en la parte del servidor. Como comentamos antes, nuestra aplicación está creada con JavaScript, pero si validamos los datos en la parte cliente podríamos seguir estando expuestos a un ataque por parte del usuario, ya que se puede usar software para la desactivación de esta validación.

6.1.1.- Validación de datos en la creación de código.

Comenzamos comentando las validaciones de datos que realizamos a la hora de generar código con la aplicación.

- **Validación de tamaño:** Tenemos que controlar que el tamaño entre el label y los formatos introducidos no superen las 80 columnas del Frame
- **Validación de nombre de Frame**: No permitimos que se puedan crear dos frames con el mismo nombre, además de no poder crear un Frame con nombre vacío. Esto lo conseguimos recorriendo toda la clase del proyecto en el que estamos trabajando y revisamos que no exista una clase Frame que contenga el mismo nombre. A la hora de editar dicho Frame ejecutamos el código igual que el que se muestra a continuación, pero con la diferencia que comprobamos si el nombre ha cambiado:

```
exports.validateNewFrame = function validateNewFrame(frameInfo, callback){
```

```
    let encontrado = false;
    if (frameInfo["nombre"] == ""){
        callback("Nombre de frame vacío, debe rellenar este campo");
    }
    else{
        nuevaPlantilla.getFrames().forEach(function (elemF, indexF, array) {
            if (elemF.getNombre() == frameInfo["nombre"]){
                encontrado = true;
            }
        }
        );
        if (encontrado){
            callback("Nombre de frame ya existente, elija otro");
        }
        else
        callback("Ok");
    }
}
```

Diferencia a la hora de editar el Frame.

```
if(frameInfoNew["name"] !== frameInfoOld["name"]){
    //Si ha cambiado el nombre, compruebo que no sea uno existente
    nuevaPlantilla.getFrames().forEach(function (elemF, indexF, array){
        if (elemF.getNombre() == frameInfoNew["name"]){
            encontrado = true;
        }
    }
    );
}
if (encontrado){
    callback("Nombre de frame ya existente, elija otro");
}
else
callback("Ok");
```

- **Validación en la creación de variables:** cuando se crean variables no sólo debemos comprobar el nombre, sino que debemos comprobar que todos los formatos y campos introducidos coincidan entre sí con el formato definido. Por lo tanto, la validación de datos de variables podemos separarla en dos apartados diferentes:
    - **Validación del nombre:** Como en el paso anterior con el Frame, revisamos que la variable no se encuentre creada en todo el proyecto, es decir, no puede existir la variable en dos frames diferentes.

```
nuevaPlantilla.getFrame(idFrame).getVariables().forEach(function (elemF, index
F, array) {
    if (elemF.getNombre() == varInfo["name"]){
        validaciones["nombre"]="repetido"
    }
}
);
```

- **Validación y consolidación de formatos:** Con los formatos debemos ser más cautelosos para que realmente los datos que se introduzcan correspondan con los valores que se pueden poner.
  - **Enteros:** comprobamos que el valor introducido es un número. Si es correcto el tamaño del div de la variable corresponderá con el valor mayor entre 8 y la longitud del número introducido.

```
if(varInfo["type"]=="integer"){
    //integer
    validaciones["ini"]=!isNaN(varInfo["initial"]);
    if(validaciones["ini"]){
        varInfo["tam"]=Math.max(8,varInfo["initial"].length);
        varInfo["initial"]=parseInt(varInfo["initial"]).toFixed();
    }
}
```

  - **Fecha:** La fecha se puede introducir con varios formatos diferentes como:
    - dd/mm/yyyy - mm/dd/yyyy - yyyy/mm/dd

    Además, se puede introducir la fecha de forma manual, o seleccionando en un calendario. Si la fecha es manual podría haber problemas de consistencia de datos, como días que no existen en cierto mes, por ejemplo. Aun así, siempre comprobamos que la fecha sea correcta y válida, por lo que hemos creado una función para comprobar estos datos:

```
function isValidDate(dateString,format){
    var parts = dateString.split("/");
    if(format=="dd/mm/yyyy"){
        if(!/^\d{1,2}\/\d{1,2}\/\d{4}$/.test(dateString) )
            return false;
        var day = parseInt(parts[0], 10);
        var month = parseInt(parts[1], 10);
        var year = parseInt(parts[2], 10);
    }else if(format=="mm/dd/yyyy"){
        if(!/^\d{1,2}\/\d{1,2}\/\d{4}$/.test(dateString) )
            return false;
```

```
        var day = parseInt(parts[1], 10);
        var month = parseInt(parts[0], 10);
        var year = parseInt(parts[2], 10);
    }else if(format=="yyyy/mm/dd"){
        if(!/^\d{4}\/\d{1,2}\/\d{1,2}$/.test(dateString))
            return false;
        var day = parseInt(parts[2], 10);
        var month = parseInt(parts[1], 10);
        var year = parseInt(parts[0], 10);
    }else
        return false;
    // Creamos rangos para años y meses
    if(year < 1000 || year > 3000 || month == 0 || month > 12)
        return false;
    var monthLength = [ 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 ];
    // Ajustamos años bisisestos
    if(year % 400 == 0 || (year % 100 != 0 && year % 4 == 0))
        monthLength[1] = 29;
    // Comprobamos el día con el mes
    return day > 0 && day <= monthLength[month - 1];
};
```

- **Decimales:** Cuando creamos una variable de tipo decimal debemos comprobar que la parte decimal obligatoria sea un número entero mayor que 0 y el valor inicial sea un número entero o decimal. Si este número es decimal y se introducen más decimales de los que hemos indicado en la "parte decimal obligatoria" el número se redondea a los decimales que hemos indicado.

```
if(varInfo["type"]=="decimal"){
    //decimal
    let initial=!isNaN(varInfo["initial"]);
    let decimal=!isNaN(varInfo["decimal"]);
    if(initial && decimal){
        if(parseInt(decimal)<=0){
            varInfo["decimal"]=1;
        }
        else{
            varInfo["decimal"]=parseInt(varInfo["decimal"]);
        }
        varInfo["initial"]=parseFloat(varInfo["initial"]).toFixed(parseInt(
                                        varInfo["decimal"]));
        varInfo["format"]=getFormatDecimal(parseInt(varInfo["decimal"]));
        varInfo["tam"]=varInfo["initial"].toString().length;
    }
    else if(initial==false && decimal){
```

```
        validaciones["ini"]="decimalInitial";
    }
    else if(initial==false && decimal==false){
        validaciones["ini"]="ini&decimal";
    }
    else{
        validaciones["ini"]="decimal"
    }
}
```

- **Caracteres:** En este caso sólo verificamos que el formato, es decir la longitud, sea un formato válido, un número. Si está vacío o es 0 este formato se crea con valor X(8). Una vez que comprobamos que es un formato válido, creamos el formato para Progess de la forma X(nº caracteres).

```
if(varInfo["type"]=="character"){
    if(!isNaN(varInfo["format"])){
        if(parseInt(varInfo["format"])<=0){
            varInfo["format"]='x(8)';
        }
        else{
            varInfo["tam"]=parseInt(varInfo["format"]);
            varInfo["format"]='x('+parseInt(varInfo["format"])+')';
        }
    }
    else validaciones["ini"]="character";
}
```

6.1.2.- Validación de datos en la carga de archivo.

A la hora de cargar un archivo de Progress, lo analizamos a través del analizador sintáctico como explicamos en la sección 5.3.-Analizador sintáctico de Progress, con lo que obtenemos los datos estructurados en un formato JSON, en el que podemos diferenciar tres tipos de datos:

- **Datos de las *'temp-tables'*:** en estos datos recogemos las variables que se han inicializado para poder crear los frames de salida.
- **Datos de las variables:** almacenamos las definiciones de las variables de los frames de entrada.
- **Datos de los frames:** se recoge toda la información de los frames creados, además de sus variables correspondientes.

Una vez tenemos todos los datos estructurados es la hora de tratarlos para poder convertir ese código a las vistas de nuestro proyecto. Cabe destacar que la validación de datos es muy parecida que la comentada anteriormente incluso llegamos a utilizar las mismas funciones:

- Comenzamos recorriendo la estructura de datos de los frames. Por cada frame que nos encontramos comprobamos sus datos básicos: nombre, tipo y título.
- Continuamos con el tratamiento de las variables que contiene cada frame. Para esto es necesario saber el tipo de frame, ya que si es de entrada o de salida este tratamiento es diferente.
    - **Frame de Entrada:** como hemos comentado anteriormente, estos datos están divididos en la parte de definición de variables en la cual nos encontramos con el tipo, formato y valor inicial (estos datos se encuentran en la estructura de datos de variables), y por otra parte tenemos la label y la posición en el frame (estos datos se encuentran en los datos de los frames).

      Recorremos estas estructuras e inicializamos o corregimos ciertos valores que nos vamos encontrando. Por ejemplo, cuando el valor de la variable no coincide con el formato de la inicialización. Más adelante comentamos la lista de valores corregidos que son compartidos con los frames de salida.
    - **Frame de Salida:** estos datos se encuentran divididos en dos partes. Una de ellas es la definición de las variables en la estructura de las `temp-tables` y la otra se encuentra en la estructura de los frames.
- Lista de valores corregidos en los frames:
    - **Integer:** si el valor de la variable no es un número lo inicializamos a 0. Además, en cualquier caso, le aplicamos el formato indicado en el fichero Progress.
    - **Decimal:** si al leer el documento la variable tipo decimal no contiene formato rectificamos el valor de la parte obligatoria decimal a 1. De no ser vacío tratamos el formato para saber cuántos decimales son necesarios. Además, al igual que con los Integer, si el valor inicial no es un número lo inicializamos a 0.
    - **Character:** recogemos el valor del formato para saber el tamaño de la variable.
    - **Date:** comprobamos que la fecha correcta al igual que en la sección 6.1.1.-

A continuación, se puede observar el fragmento de código que realiza esta corrección de valores:

```
function preparaDatos(varInfo){
    if(varInfo["type"]=="integer"){
        if( isNaN(parseInt(varInfo["initial"])))
        varInfo["initial"]=0;
        else
```

```
        varInfo["initial"]=parseInt(varInfo["initial"]);
        //obligamos este formato
        varInfo["format"]="->,>>>,>>9"
    }
    if(varInfo["type"]=="decimal"){
        if(varInfo["format"]=="" )
        varInfo["decimal"]=1
        else varInfo["decimal"]=parseInt(getFormat(varInfo));
        if( isNaN(parseFloat(varInfo["initial"])))
        varInfo["initial"]=0;
        else
        varInfo["initial"]=parseFloat(varInfo["initial"]);
    }
    if(varInfo["type"]=="character"){
        varInfo["format"]=parseInt(getFormat(varInfo));
        varInfo["tam"]=varInfo["format"];
    }
    if(varInfo["type"]=="date"){
        varInfo["tam"]=10;
    }
    return varInfo;
}
```

- Además, debemos destacar que cada vez que se carga un archivo Progress se crea un fichero de texto con el `log` de los pasos que hemos ido siguiendo a lo largo de la importación. En este log podemos ver tanto los errores encontrados, como los procesos que se han ejecutado correctamente. A continuación, dejamos un ejemplo de la salida generada en este `log`.

Código Progress que se va a importar:

```
/* BEGIN VARS Autogenerated */
define variable v1 as integer init "2" no-undo.
/* END VARS Autogenerated */

/* BEGIN INPUT Frame Autogenerated */
define frame a
   v1 label "v1" at row 2 column 8
   sad label "sa" at row 4 column 17
with side-labels.
/* END INPUT Frame Autogenerated */

/* BEGIN OUTPUT Frame Autogenerated */
define frame b
   v2 label "v2"
   v3 label "v3"
```

```
   v4 label "v4"
with title "salida".
/* END OUTPUT Frame Autogenerated */
```

Contenido de `log` que se ha ido generando en la carga del fichero:

```
Comenzamos a leer el archivo
>OK:Se añade nuevo Frame de entrada con nombre: a con título:  con
id:1
>Empezamos a crear la variable de entrada: v1 con label: v1
>>OK:Se agrega correctamente la variable: v1 con id: 1 con label: v1
de tipo: integer con formato: ->,>>>,>>9 con valor inicial: 2 en
fila: 2 en columna: 8
>Empezamos a crear la variable de entrada: sad con label: sa
>>NOK: Error variable no encontrada en la tabla de variables.
------------------------------------------------------------
>OK:Se añade nuevo Frame de salida con nombre: b con título: salida
con id:2
>NOK: No se ha definido la temp-table.
------------------------------------------------------------
```

6.1.3.- Mensajes de Validación de Datos:

Como hemos comentado en el punto anterior, comprobamos los datos y a la vez vamos rellenando una variable de tipo diccionario en la cual se almacenan los datos y los posibles errores. Mostramos un mensaje u otro dependiendo del valor que obtengamos en el parámetro nombre para comprobar el nombre vacío o repetido y en el parámetro `ini` para controlar los formatos:

```
let validaciones={
    "nombre": null,
    "ini": null
};
if(validaciones["nombre"]=="vacio"){
    callback("Nombre de variable vacío, debe rellenar este campo");
}
else if(validaciones["nombre"]=="repetido"){
    callback("Nombre de variable ya existente, elija otro");
}
else if(validaciones["ini"]==false){
    callback("Valor inicial no corresponde con el tipo elegido");
}
else if(validaciones["ini"]=="decimalInitial"){
    callback("Valor inicial no corresponde con el tipo elegido");
}
else if(validaciones["ini"]=="ini&decimal"){
    callback("Valor inicial no corresponde con el tipo elegido y valor decimal
             debe ser un número");
```

```
}
else if(validaciones["ini"]=="decimal"){
    callback("Valor decimal debe ser un número");
}
else if(validaciones["ini"]=="character"){
    callback("El formato debe ser un número entero");
}
else{
    callback("Ok");
}
```

CAPITULO 7.- CONCLUSIONES Y TRABAJO FUTURO

A lo largo del desarrollo de nuestro proyecto hemos encontrado dificultades, las cuales hemos abordado todo lo bien que hemos podido, pero llegando finalmente al resultado esperado, además de hacer un desarrollo más amplio del que habíamos pensado en el planteamiento inicial del proyecto.

En este apartado vamos a centrarnos en las dificultades encontradas, en el cumplimiento o no de los objetivos que nos planteamos al inicio, y por qué no, en un apartado en el que hablaremos de las ampliaciones que se podrían hacer en el proyecto más adelante.

7.1.- CUMPLIMIENTO DE OBJETIVOS

Como hemos comentado en la sección 1.2.-, al comienzo de este proyecto planteamos unos objetivos asumibles para poder finalizarlo en el tiempo previsto. Llegado este punto resumiremos el objetivo principal y los subjetivos.

Construir una aplicación para crear frames de entrada para Progress. Este era el objetivo principal del proyecto: llegar a crear una aplicación la cual nos permitiera crear una interfaz gráfica de Progress simplemente arrastrando los objetos que deseamos sobre un frame. Este objetivo se ha cumplido, pues llegamos a desarrollar una aplicación con la que creamos una estructura de frames y variables.

Este es el punto clave del proyecto, ya que aquí podríamos resumir todos los demás objetivos, pero los dividiremos en otras tres partes.

- **Creación de interfaz gráfica:** Este podemos considerar que es el punto más especial del proyecto. Hemos conseguido lo que queríamos al principio de este. Ahora somos capaces de crear variables y poder interactuar con ellas con gestos y movimientos. Hemos sido capaces de crear una aplicación con la que el usuario final se puede sentir muy cómodo a la hora de desplazar y colocar las variables creadas dentro de un panel que previamente se ha generado en unos sencillos pasos.

  Además de que el usuario pueda interactuar de forma visual con las variables, el usuario puede ver en tiempo real el código Progress que se le está generando con cada movimiento o creación de frames o variables.

- **Importación de código:** queríamos conseguir que una vez creado el código con nuestra aplicación fuéramos capaces de volver a importarlo. Este punto se ha conseguido. Además, si el usuario siguiera creando código con las mismas pautas con las que se ha creado, no hay ningún problema a la hora de cargar dicho archivo.

  Por el momento no se puede cargar un archivo Progress con etiquetas diferentes a las que tratamos en el proyecto. El lenguaje Progress es demasiado extenso para abordarlo en una primera fase, por eso este punto lo comentamos más adelante en la sección 7.3.-Trabajo futuro.

- **Validación de datos:** Este es uno de los puntos más complejos de este proyecto. El tratamiento de datos como hemos comentado en la sección 6.1.-Validación de datos era bastante complejo, ya que debíamos tratar bastantes apartados. Somos capaces tanto de tratar datos que el usuario introduce a través de nuestros formularios, como de controlar los errores que encontramos a la hora de cargar un archivo.

7.2.- DIFICULTADES ENCONTRADAS

El proyecto se planteaba complicado para conseguir los objetivos debido al desconocimiento del lenguaje Progress por uno de los componentes del equipo y el desconocimiento de algunas de las herramientas utilizadas.

El primer problema al que nos encontramos fue ¿cómo desarrollar una aplicación de escritorio utilizando herramientas de desarrollo web? Este era uno de nuestros principales objetivos, el cual se resolvió bastante rápido en cuanto tuvimos una reunión con el director del proyecto, Manuel. Este nos planteó varias opciones para poder desarrollar nuestra aplicación como queríamos.

A lo largo que avanzamos en el aprendizaje, ya que, aunque supiéramos programación web, teníamos que aprender a realizar un proyecto con Electron.JS y comprender su funcionalidad, nos encontramos con varias dificultades a la hora de plantear el desarrollo principal, es decir, a planificar cómo íbamos a plasmar lo que teníamos en nuestra cabeza.

Una vez decididas las tecnologías y el planteamiento del problema, comenzamos con el desarrollo. A la hora de desarrollar nos hemos encontrado con los problemas que comentamos a continuación:

- **¿Cómo habilitar la funcionalidad de arrastrar y soltar (drag and drop)?** Este fue uno de los principales problemas que teníamos. Una vez que nos pusimos a investigar sobre este tema nos encontramos con una librería bastante fácil de utilizar, por lo que nos decidimos por usar Interact.JS y no código nativo.

- **Situación de las variables una vez movidas**. Cuando creábamos las variables y las arrastrábamos, en el panel inferior permanecía el hueco que ocupaba esta variable, ya que durante el arrastre realmente no se movía el objeto 'div' correspondiente. Encontramos una solución clonando este objeto y moviéndolo a un apartado superior. Además, así podíamos diferenciar las variables que habíamos movido de las que no. Este proceso se explica detalladamente en la sección 4.2.1.-.

- **Calcular distancias respecto al origen de variable.** En nuestras pruebas iniciales con Interact.JS no sabíamos cómo poder conseguir la distancia que se había movido la variable con respecto a su posición inicial. Leyendo la documentación nos encontramos con la solución a nuestros problemas. Una vez tuvimos estas distancias tuvimos que desarrollar en un papel cómo se desplazaba una variable y calcular sus nuevas coordenadas para poder resolver a qué fila y qué columna corresponde. Este punto lo hemos detallado en la sección 4.2.2.-.

- **Ajuste de letra para tamaño.** Teníamos un problema con las letras de las variables, ya que cada una ocupaba una cantidad de píxeles diferente. Necesitábamos un tipo de letra en la que todos sus caracteres ocuparan exactamente la misma cantidad de espacio horizontal. Nos centramos en el uso de letras monoespaciadas, por lo que nos decidimos en usar "Courier New Monospace".

- **Creación de ventanas complementarias**. Al comienzo del proyecto nos encontramos con el problema de pedir datos al usuario, por lo que nos decidimos, al ser una aplicación de escritorio, en crear ventanas complementarias para este propuesto. Nos encontramos con muchos problemas al estar trabajando con tecnología web ya que nos implicaba perder una gran parte del tiempo en crear servicios que pudieran comunicarse con el principal de Electron.JS, que era en el que se ejecutaba la aplicación. Por lo que al final nos decidimos por usar ventanas modales, que son cuadros de diálogo que aparecen sobre la página principal, bloqueando las funciones para centrar el foco en esta acción en particular.

- **Controlar tamaños de *'divs'* para que cumplan el formato**. Una vez que generamos los `divs` de las variables, estos ocupaban toda una línea, por lo que tuvimos que ajustarlos para que no hubiera problemas a la hora de arrastrar o mover las variables y estas se pudieran colocar en el frame correctamente.

- **Cómo representar los frames de salida para ser editables cómodamente**. Como hemos comentado los frames de salida son reportes de información en columnas. Para representarlos en nuestra aplicación finalmente elegimos una lista con las variables ordenadas en el orden en el que se mostrarían las columnas. De esta forma su edición y visualización era mucho más cómoda.

7.3.- TRABAJO FUTURO.

En este apartado comentaremos las posibles mejoras o extensiones que se podrían hacer en el proyecto presentado. Nos centraremos exclusivamente en las que nosotros creemos que se podrían hacer en un corto plazo.

7.3.1.- Ampliación de tipos.

Principalmente nos hemos centrado en una serie de tipos que son los más habituales, pero un punto que se podría extender es incorporar más tipos de datos para las variables creadas, como, por ejemplo: `datetime, recid, int64`, etc.

También se puede definir el tipo de una variable en Progress a través de un campo de una tabla de una base de datos, por lo que también sería una ampliación a tener en cuenta.

7.3.2.- Efectos visuales.

Se podrían generar diversos efectos visuales a la hora de arrastrar variables, como, por ejemplo, marcar las variables seleccionadas, o poder seleccionar varias variables y moverlas a la vez.

7.3.3.- Alineaciones por defecto.

Otra ampliación interesante sería agregar diferentes botones para alinear todas las variables de un mismo modo. Por ejemplo, que se coloquen todas a la izquierda, a la derecha o centradas.

7.3.4.- Añadir rejilla

Se podría añadir un efecto de rejillas para ver diferenciadas las filas y columnas en el frame, para facilitar su alineación.

7.3.5.- Ordenar visualización de frames.

Se puede agregar la funcionalidad de ordenar el orden en el que queremos que se vayan mostrando los frames en el ERP.

7.3.6.- Ampliar características del frame.

Como hemos comentado, hemos elegido una serie de características muy concretas con las que personalizar el frame, pero hay muchas más posibilidades, como permitir cambiar color de fondo, la transparencia del borde, la colocación de las etiquetas de cada variable, etc. Una ampliación interesante sería tener en cuenta algunas de estas características.

7.3.7.- Ampliación de lectura de documentos.

Nos hemos centrado en poder leer un documento Progress con las variables y formatos que hemos creado en el proyecto. Un punto muy fuerte como ampliación sería que fuéramos capaces de poder leer un código Progress bastante más completo, además de poder hacer una revisión por si hubiera problemas con esta nueva lectura.

7.3.8.- Vista simulada del frame de salida.

Como hemos comentado en el apartado 4.3.- los frames de salida se visualizan como columnas con información ya que están pensados para hacer reportes. Para simplificar la creación de estos frames en la aplicación, nosotros creamos una lista en la que se muestra el orden en el que aparecerán a la hora de ser mostradas. Podríamos añadir una funcionalidad para que el usuario pudiera visualizar cómo quedaría realmente la representación de este frame en el ERP.

CAPITULO 8.- CONCLUSIONS AND FUTURE WORK

Throughout the development of our project we have encountered difficulties, which we have addressed as well as we could, but finally reaching the expected result. We have also achieved a development broader than the one initially planned.

In this section we are going to focus on the problems encountered and explain to what extent the objectives of the project have been meet. in the last section we shall talk about the extensions that can be made in the future.

8.1.- ACHIEVEMENT OF GOALS

As we have commented in section 2.2.- at the beginning of this project, we set some acceptable objectives to be able to finish it in the scheduled time. At this point we will summarize the main objective and the secondary ones.

Build an application to create input frames for Progress. This was the main objective of the project: to create an application which would allow us to create a Progress graphical user interface simply by dragging the objects we want on a frame. This objective has been met, as we have developed an application with which we create a frame and variable structure.

This is the key point of the project, in which we could summarize all the other objectives, but we will divide them into three other parts.

- **Creation of graphical user interface:** we can consider this point the most important of the project. We are able to create variables and we are able to interact with them with gestures and movements. We have created an application with which the end user can feel very comfortable when moving and placing the created variables within a panel that has previously been generated in a few simple steps.

  The user can interact visually with the variables, the user can see in real time the Progress code that is being generated with each movement or creation of frames or variables.

- **Code import:** we wanted to ensure that once the code was created with our application we were able to import it again. This point has been achieved. In addition, if the user continues creating code with the same guidelines that we use in our application, the code could be loaded into the application successfully.

  At the moment it is not possible to load a Progress file with different labels than the ones we deal with in the project. The Progress language is too broad to deal with in a first phase, so we discuss this point later in section.

- **Data validation:** this is one of the most complex points of this project. The data processing as we discussed in section 6.1.-was quite complex. We can process data that user enters through our forms and control the errors that we find when code is imported from an input file.

8.2.- DIFFICULTIES ENCOUNTERED

The goals of the project were difficult to achieve due to the lack of knowledge of Progress language by one of the team members and the lack of knowledge of some of the tools used.

The first problem we encountered was how to develop a desktop application using web development tools. This was one of our main objectives, which was resolved fairly quickly as soon as we had a meeting with the project director, Manuel. He presented us several options to develop our application as we wanted.

Once the technologies and the problem approach had been decided, we started with the development itself. When developing we have encountered the problems that we discuss below:

- **How to enable drag and drop functionality?** This was one of the main problems we had. Once we started to investigate on this topic, we found a fairly easy to use library, so we decided to use Interact.JS and not native code.

- **Situation of the variables once moved.** When we created the variables and dragged them, the gap that this variable occupied still was shown in the bottommost pane because during the drag the corresponding 'div' object did not really move. We found a solution by cloning this object and moving it to a higher section. Also, this way we could differentiate the variables that we had moved from those that we had not. This process is explained in detail in section 4.2.1.-

- **Calculate distances from the origin of the variable.** In our initial tests with Interact.JS we did not know how to get the distance that the variable had been moved with respect to its initial position. Reading the documentation, we find the solution to our problems. Once we had these distances we had to develop on paper how a variable moved and calculate its new coordinates to be able to solve which row and which column it corresponds to. We have detailed this point in the section 4.2.2.-

- **Letter fit for size.** We had a problem with the letters of the variables, since each one occupied a different amount of pixels. We needed a typeface in which all of its characters took up exactly the same amount of horizontal space. We focus on the use of monospaced letters, so we decided to use "Courier New Monospace".

- **Creation of complementary windows.** At the beginning of the project we encountered the problem of requesting data from the user, so we decided to create complementary windows for this proposal. We ran into a lot of problems when working with web technologies, as it involved spending a great deal of time creating services that could communicate with the Electron.JS main process, where the application was running. So, we decided to use modal windows, which are

dialog boxes that appear on the main page, blocking the rest of the interface in order to focus on this particular action.

- **Control div sizes to fit the format.** Once we generated the `divs` of the variables, they occupied a whole line, so we had to adjust them to solve problems when dragging or moving the variables so that they could be placed in the frame correctly.
- **How to represent the output frames so they are easily editable.** As we have commented, the output frames are reports of information in columns. To represent them in our application we finally choose a list with the variables ordered in the order in which the columns would be displayed. In this way, the editing and viewing is much more comfortable.

8.3.- FUTURE WORK

In this section we will discuss the possible improvements or extensions that could be made. We will focus exclusively on what we believe could be done in the short term.

8.3.1.- Type extension

We have mainly focused on a series of types that are the most common, but one point that could be extended is to incorporate more data types for the created variables, such as: `datetime, recid, int64`, etc. You can also define the type of a variable in Progress through a field in a table in a database, so it would also be an extension to consider.

8.3.2.- Visual effects

Various visual effects could be generated when dragging variables, such as marking selected variables, or being able to select several variables and move them at the same time.

8.3.3.- Default alignments

Another interesting extension would be to add different buttons to align all the variables in the same way. For example, they can be aligned to the left, right, or centered.

8.3.4.- Add grid

A grid effect could be added to see the rows and columns in the frame differentiated, to facilitate their alignment.

8.3.5.- Order display of frames

We can add the functionality of specifying the order in which we want the frames to be displayed in the ERP.

8.3.6.- Expand characteristics of the frame

As we have commented, we have chosen some specific properties to customize the frame, but there are many others, such as the background color, the transparency of the border or the placement of the labels of each variable. An interesting extension would be to take into account some of these properties.

8.3.7.- Extension of document reading

We have focused on being able to read a Progress document with the variables and formats that we have created in the project. An extension could be to be able to read a much more complete Progress code, in addition to being able to do a review in case there were issues with this new reading process.

8.3.8.- Simulated view of the output frame

As we have commented in section 4.3.- the output frames are visualized as columns with information, since they are intended to make reports. To simplify the creation of these frames in the application, we created a list showing the order in which they will appear when they are displayed. We could add a functionality to visualize the representation of how this frame would actually look in the ERP.

CAPITULO 9.- CONTRIBUCIONES DE CADA PARTICIPANTE

9.1.- CONTRIBUCIÓN RICARDO CABALLERO SÁNCHEZ

- **Propuesta de trabajo.**

  Como hemos comentado en la introducción, la propuesta de este proyecto nace al tener un problema con el lenguaje nuevo que estaba utilizando en el trabajo, Progress.

  En un principio tomé de referencia Netbeans, programa que había utilizado en cursos anteriores en la carrera, como en la asignatura Modelado de Software en la que implementamos aplicaciones de escritorio con Java. Partiendo de esta base, comencé a investigar un poco más sobre las interfaces en Progress, qué se podía hacer y qué no, ya que las interfaces que solíamos crear en el trabajo eran bastante simples, pero quería saber hasta qué punto se podía llegar. Después hablé con mi jefe, ya que él tiene mucha más experiencia con el lenguaje y sabe de las ventajas y las carencias de éste. Hicimos unos bocetos de cómo podría ser la interfaz para que fuese cómoda de utilizar, a la vez que simple y también de las funcionalidades que estábamos buscando. La mayor de estas funcionalidades era utilizar la aplicación para crear pantallas de mantenimiento, lo que llamamos en el trabajo frames de entrada. Estas pantallas son bastante utilizadas en un ERP y nos permiten acceder a ciertos datos e ir modificándolos. Por ejemplo: una pantalla de mantenimiento de órdenes de ventas está compuesta de muchos campos que se van rellenando para hacer la venta. Tenemos datos en la cabecera, como fecha efectiva, cliente al que va dirigida, datos sobre impuestos, etc. Después en cada línea tenemos el producto, la cantidad, el precio, descuentos y otras características. Como vemos, son muchos datos y se van presentando en distintos frames para que el usuario vaya rellenando los datos de forma cómoda o que se rellenen por defecto en base a otros datos introducidos anteriormente.

  Programar estas pantallas en Progress es muy tedioso. Al ver este problema pensé que podríamos aportar una solución a partir de un Trabajo de Fin de Grado y así hacer un proyecto que sea útil y que no sea simplemente un proyecto con el que cumplir con los créditos, sin motivación y sin un propósito real. Por esta razón, opino que nuestro proyecto tiene un gran valor al estar basado en un problema real y al resolverlo de una manera eficiente.

- **Estudio de herramientas y tecnologías.**

  Al tener una idea de la interfaz que queríamos construir nos pusimos a investigar sobre librerías de JavaScript que nos permitieran arrastrar y soltar elementos en la pantalla. Encontré una librería, jQuery UI, que finalmente descartamos para este propósito pero que utilizamos para mostrar un calendario a la hora de elegir el valor inicial en variables de tipo fecha. De esta misma librería también utilizamos la función `sortable` en los frames de salida.

También estuve investigando el uso de otros lenguajes en lugar de JavaScript, como TypeScript.

La parte que más tiempo de estudio me llevó fue el uso de PEG.JS como analizador sintáctico. En la carrera no había visto nunca cómo se definían unas reglas para analizar un lenguaje y construir un objeto con los datos que nos interesan, por lo que al principio me costó algo más entender el funcionamiento, pero utilizando el parseador online con algunos ejemplos que nos ofrecían y la ayuda de Manuel, ha sido más llevadero.

- **Desarrollo de interfaces visuales.**

  He desarrollado en este proyecto las interfaces de los formularios de creación y edición de variables y frames, el menú principal, la cabecera, la vista de código y los frames de salida.

  La parte visual que no he desarrollado es de la que se ha encargado mi compañero, en los frames de entrada, la visualización de las variables y el movimiento de éstas.

  El cambio entre pantallas y la ocultación de ciertas partes de las vistas en función de la pantalla en la que nos encontremos también han sido parte de mi trabajo en este apartado.

  Para la parte visual como hemos comentado he utilizado librerías con las que ya había trabajado anteriormente, como Bootstrap y otras nuevas, como jQuery UI. El uso de estas librerías me ha permitido crear las interfaces de una manera bastante sencilla y cómoda de ampliar. En la pantalla de frames de salida he utilizado la función `sortable` de jQueryUI que tiene una función para transformar el elemento ordenado en un array, lo cual ha hecho bastante cómodo el trabajo con la interfaz y la capa de datos.

- **Desarrollo de backend.**

  En este punto podemos destacar el desarrollo de la creación de las variables y frames a partir de los datos obtenidos en los formularios y la edición de los datos ya introducidos. De la validación de estos datos se ha encargado mi compañero.

- **Analizador sintáctico de código Progress.**

  En este apartado he desarrollado las reglas utilizadas para obtener las variables y los frames de salida. Los analizadores sintácticos se generan automáticamente en base a estas reglas y utilizarlos es muy sencillo. Simplemente se llama a su método `parse` y te devuelve un objeto con la información obtenida.

- **Lectura del archivo.**

  En relación a este apartado he desarrollado el flujo completo: la lectura del archivo en sí, el uso de los parseadores para obtener los datos y finalmente la transformación del objeto devuelto por los parseadores a mis objetos de las clases Frame y Variable.

  De nuevo, la validación de éstos datos ha sido desarrollada por Adrián.

- **Definición de estructura de datos.**

  Como hemos comentado en la memoria, utilizamos clases de JavaScript para definir nuestra estructura de datos, compuesta de una clase Progress, que contendrá un `Map` de clases Frame y que cada uno de éstos tiene otro `Map` de clases Variable. La definición de estas clases ha sido llevada en gran medida por mí, exceptuando algunos atributos y funciones que se utilizan para la visualización de las variables en los frames de entrada.

- **Generación de Código Progress.**

  La generación de código Progress la he desarrollado yo completamente, ya que Adrián no conocía el lenguaje previamente.

  A la hora de generar este código hay ciertas características que no son elegidas por el usuario. Estas características fijas se pueden encontrar en la sección 4.1.1.-Frames y en la sección 4.1.2.-Variables y son usadas para simplificar la creación de código y ajustarnos a los estándares que define el ERP para sus pantallas.

- **Pruebas de código Progress.**

  En este punto cabe mencionar que la empresa en la que trabajo, Devel C11, nos cedió un entorno de desarrollo del ERP, en el que poder hacer pruebas con el código Progress que íbamos generando y ver si compilaba y el resultado de la pantalla en el ERP era el esperado. El desarrollo de estas pruebas ha sido parte de mi trabajo en el proyecto.

9.2.- CONTRIBUCIÓN ADRIÁN DÍAZ JIMÉNEZ

- **Estudio de tecnologías.**

  En cuanto se planteó el proyecto al director y fue aprobado, la primera tarea que debíamos hacer era el estudio de tecnologías para llevarlo a cabo.

  Esta fue mi primera contribución: pensar cuales iban a ser las tecnologías más importantes que podríamos usar para este proyecto, por lo que puedo enumerar las principales:

  - **Electron.JS**: estudio e investigación sobre cómo desarrollar una aplicación con este Framework, además estaría pensado para usarlo con JavaScript, aunque mi compañero estudió la posibilidad de utilizar TypeScript.
  - **Elección del analizador sintáctico:** revisar varios analizadores sintácticos, ANTLR y PEG.js, al final opté por elegir PEG.js. Por otro lado, este paso se decidió definitivamente en una parte más adelantada del proyecto, ya que mi compañero Ricardo fue el que finalmente estudió este analizador.
  - **Elección de librerías para poder arrastrar y soltar:** tenía dudas sobre la conveniencia de utilizar JavaScript nativo o, en su lugar, facilitar el trabajo usando una librería. Al final me decanté por la elección de Interact.JS ya que una vez consultada la documentación y su utilización, me resultó bastante más sencilla que utilizar directamente el código nativo en JavaScript.

- **Planteamiento del desarrollo del proyecto**

  En este apartado cabe destacar la planificación y planteamiento del desarrollo del proyecto. Este punto es consecuencia del punto anterior.

  Una vez tenía elegidas las tecnologías que se utilizarían en este proyecto, tuve que plantear cómo se iba a desarrollar, por ejemplo, organización mediante clases, elegir cómo se iban a mostrar los formularios al usuario (ventana nueva o modal), etc.

  Comencé a generar la aplicación, un proyecto desde cero, y creé las ventanas básicas para poder ver el funcionamiento de Electron.JS, la importación de Bootstrap, y experimentar con jQuery, mediante pruebas muy sencillas.

- **Desarrollo de la vista de Frames arrastrar-soltar con el ratón / mover con el teclado.**

  Este es el apartado en el cual invertí mayor parte del tiempo, tanto en el estudio de la librería Interact.JS como en la correcta implantación y funcionamiento de la misma. Lo puedo dividir en varias fases ya que como he comentado ha sido la parte más amplia:

  - **Comienzo de movilidad de variables:** Estructurar el código HTML, además de comprender las funciones para arrastrar las variables. Desarrollo de las funciones de

cálculo de traslación de las variables. En este punto las variables se podían mover a lo largo de toda la ventana.

- **Movimiento y reposicionamiento de varias variables:** volver a estructurar el código HTML y volver a calcular las distancias de traslación respecto al origen, ya que ahora teníamos la fila y la columna en la que se encontraba, pero al clonar los `divs` y reubicarlos cambiaban las distancias de traslación. Esto se explica en la sección 4.2.1.- Zonas principales.

- **Restringir el movimiento de las variables:** en este apartado puedo comentar que quise restringir el movimiento a ciertas coordenadas, pero no era capaz, por lo que después de muchos intentos opté por restringir el movimiento al 'div' del frame:

  ```
  'restriction: document.getElementById("inner-dropzone")'
  ```

  Una vez tenía una pequeña restricción tuve que crear ciertas reglas para que aun así la variable no se saliera de su zona.

- **Movimiento con teclado:** cuando terminé con el movimiento con el ratón, comencé a crear el código para que podamos mover las variables con el teclado. Después de intentar centrar el foco en la variable sin éxito conseguí crear el código centrándome en los datos de la variable que tenemos seleccionada en la vista del Frame.

  Una vez conseguimos mover las variables con el teclado tuve que crear unas nuevas restricciones para estos movimientos.

- **Creación de analizadores sintácticos de temp-tables y frames de entrada:** he desarrollado el código para poder analizar sintácticamente el código Progress tanto de las 'temp-tables' como de los frames de entrada. A raíz de la explicación de Ricardo sobre el funcionamiento de PEG.JS me resultó más sencillo este desarrollo, generando unas reglas muy sencillas siguiendo una serie de pasos.

- **Validación de datos de entrada:** una vez creada toda la funcionalidad del proyecto, era hora de conseguir adaptarnos a las reglas necesarias para que no hubiera problemas a la hora de crear código Progress, por lo que tuve que crear una validación de los datos de entrada del usuario, de modo que se cumplieran ciertas restricciones, por ejemplo:

  - Comprobar que hay consistencia entre los datos introducidos y el tipo seleccionado para cada variable.

  - Avisar al usuario de problemas con los datos introducidos.

  - No permitir ciertos valores vacíos como el nombre de la variable o frame, además de intentar corregir automáticamente los valores omitidos por el usuario como, por ejemplo, los valores iniciales.

- **Validación de datos lectura de código:** al igual que el punto anterior, tuve que adaptar las funciones de validación de datos para que ahora funcionaran también con la lectura de un archivo. Además, tuve que desarrollar unos ciertos pasos anteriores a la validación para que esta no fallara. Por ejemplo, al igual que he comentado en el paso anterior, tuve que tratar los datos y adaptarlos para que estos no fallaran en caso de errores, en algunas ocasiones corrigiendo automáticamente el código.

- **Creación de log de carga:** este ha sido el último paso que he desarrollado. Al trabajar en una empresa en la cual tenemos que implantar muchos sistemas, siempre que he tenido un problema me ha sido muy útil recurrir a los ficheros de `logs` que se generan, ya sea a la hora de la instalación como a la hora de funcionamiento. Por esta razón decidimos ampliar la carga de un fichero de código Progress con este proceso.

  A la hora de cargar un fichero de Progress se siguen ciertos pasos para poder validar estos datos. He añadido una serie de información según la lectura de este fichero sea correcta o no. Una vez creado un `String` con la información que hemos obtenido, se crea un fichero en el que se incluyen estos datos. En este último fichero podemos encontrar líneas que comienzan con 'OK' para un proceso cuya ejecución ha tenido éxito, y líneas que comienzan con 'NOK' para un proceso que ha fallado.

BIBLIOGRAFÍA

- [1] Bhaumik, S. **Bootstrap essentials: use the powerful features of bootstrap to create responsive and appealing web pages.** Birmingham, England: Packt Publishing. ISBN: 9781784395179 - 9781784396336 (e-book), 2015.
- [2] Cameron, D. **HTML5, JavaScript, and jQuery 24-hour trainer.** Indianapolis, Indiana: Wrox. ISBN: 9781119001164 - 9781119001188 (e-book), 2015.
- [3] Cochran, D, **Twitter Bootstrap Web Development How-To.** England: Packt Publishing, Limited, 2012. ISBN: 9781849518826. 2012.
- [4] Bibeault, B., Katz, Y., De Rosa, A., **jQuery in Action**, 3rd Edition,Manning Publications, ISBN 9781617292071, 2015.
- [5] jQuery, http://api.jquery.com/
- [6] jQueryUI, https://jqueryui.com/
- [7] Goodman, D. **Javascript Bible**, 7th ed.; Wiley: Hoboken, N.J., 2010.
- [8] Shah, N.; Balda, G. **Html5 Enterprise Application Development**; Packt Pub: Birmingham, 2013.
- [9] Progress, https://knowledgebase.progress.com/
- [10] Electron, https://www.electronjs.org/docs
- [11] Imagen de Electron 3.1.3,https://platzi.com/blog/aplicaciones-escritorio-electron-js/
- [12] Interact JS, https://interactjs.io/docs/
- [13] Kinney, Steven, **Electron in Action**, Manning Publications Co.20 Baldwin Road PO Box 761 Shelter Island, NY 11964 ISBN: 9781617294143, 2019.
- [14] Electron wikipedia, https://es.wikipedia.org/wiki/Electron_(software)
- [15] PEGJS Documentation, https://pegjs.org/documentation
- [16] Power, S. **Learning Node, 2nd edition** O`Reilly,ISBN: 9781491943120, 2016.