


# Trabajo de fin de título en Ingeniería Informática

## Generador Sintético de datos para una aplicación de Fintech: Finbook

Juan Alberto Ureña Rodríguez

Tutor: José Juan Hernández Cabrera

Fecha: 1 de Julio de 2020

# Resumen


## Español

Este proyecto es un generador sintético de datos para una aplicación Fintech, que simula un entorno empresarial en el que se produce compras y

ventas de productos desde un punto de vista realista y lógico. En este caso, debido a la extensa cantidad de sectores de empresas que hay actualmente,

me he querido centrar en sector de la restauración.

Dentro de este sector, he realizado un entorno centralizado en los restaurantes con los distintos individuos y empresas conocidos en el

sector, como son los comensales, los trabajadores del restaurante, los proveedores de alimentos y los proveedores de servicios. Estos

interactúan entre sí, generando comercio, consumo, y como consecuencia, facturas entre estos, el principal objetivo de esta simulación.

## English

This project is a synthetic data generator for a Fintech application, that simulate a business environment in which occurs purchases and sales of

products from a realistic and logic point of view. In this case, cause of the extensive amount of company sectors that exist nowadays, I have decided to focus in the restoration sector.

Inside this sector, I have performed an environment centralized in the restaurants with the different know individuals and companies in the

sector, like the commensals, the workers of the restaurant, the providers of aliments and the providers of services. These interact between each

other, generating commerce, consumption and due to this, bills between them, the main objective in this simulation.

# Contenido

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Escuela de Ingeniería Informática

# 1. Explicación del ámbito del proyecto

## 1.1 Introducción

La internacionalización de los mercados y el comercio electrónico demandan, cada vez más, el uso generalizado de la factura electrónica y el abandono del formato en papel. En este apartado se estudia la factura, la normativa que regula su documentación, su uso esencial en el ámbito tributario, las ventajas que ofrece la factura electrónica, su consideración en el ámbito español y especialmente en el de México, por estar implantada de forma obligatoria para todos los contribuyentes, así como la presentación de la tecnología aplicada a las finanzas para la automatización de los procesos de facturación.

### 1.1.1 Hacia la digitalización

Tradicionalmente, las facturas se han venido generando en papel y también ha sido el método más usado durante las últimas décadas. En el siglo XX y sobre todo antes de la creación del ordenador y las redes de internet, el papel era la mejor opción para documentar las transacciones comerciales. Sin embargo, actualmente es necesario plantearse si este sustento formal sigue siendo la manera óptima teniendo en cuenta el apoyo de las nuevas tecnologías, ya que la factura electrónica es un instrumento que permite reducir costes, reduce el impacto medioambiental y permite, al igual que la generada en papel, garantizar la realidad de las operaciones que documentan.

La factura electrónica es aquella que se ha expedido y recibido en formato electrónico. En el marco legal español, el artículo 1 de la Ley 56/2007, de 28 de diciembre, de Medidas de Impulso de la Sociedad de la Información se define la factura electrónica, indicando que es un documento electrónico que cumple con los requisitos legal y reglamentariamente exigibles a las facturas y que, además, garantiza la autenticidad de su origen y la integridad de su contenido. Al contrario de la factura en papel, la factura electrónica se crea, se guarda y se distribuye virtualmente, lo que permite una mayor facilidad y seguridad en mundo socioeconómico actual, como veremos a continuación. La factura electrónica funciona como cualquier fichero o archivo de datos que guarda información, pero que en este caso almacena datos de una operación comercial.

La factura electrónica tiene la ventaja que controla el flujo de facturas generados virtualmente y permite a la Administración obtener información en línea en el momento de realizada la operación y cruzar información entre Administraciones tributarias de distintos países (Intercambio de información), lo que mejora el control de las evasiones fiscales frente a las facturas en papel que no son declaradas.

El formato en que se genera la factura electrónica más extendido es XML (Extensible Markup Language) debido a su simplicidad y la claridad a la hora de leer una factura por

una persona que conozca un poco el formato o para un programa que la procese. Este uso de formato permite generalizar la factura, es decir, facilita o, más bien, obliga a incorporar un conjunto de reglas formales que seguir para su creación formando una estructura común y constante entre todas ellas.

Las ventajas que ofrece este formato de factura electrónica motivan el que sea ampliamente utilizado en el ámbito profesional o empresarial, así como el ámbito tributario, como es el caso de la Administración tributaria española.

Por último, tenemos la ventaja que se va a aprovechar y demostrar en este proyecto, la capacidad de procesar y analizar estas facturas. Esto se debe como ya se ha mencionado anteriormente, al formato que utiliza, que favorece el procesamiento y el análisis, ya que su estructura es única, por lo que todas las facturas se pueden leer y obtener datos de la misma forma, de forma iterativa y constante y, finalmente, generar un reporte de global mucho más rápido, que el que pudiera ofrecer un profesional experto.

La técnica que automatiza el análisis de datos se denomina Big Data, que como su nombre indica, permite analizar grandes volúmenes de datos de forma rápida y eficaz en todo tipo de sectores, uno de ellos la facturación. Esta es una de las grandes ventajas de la factura electrónica con respecto a la de papel, y es el Big Data el futuro en muchos sectores que trabajen con grandes volúmenes de datos.

La factura electrónica es utilizada de forma obligatoria en España desde 2015 a partir de la publicación de la Ley 25/2013, de 27 de diciembre, de impulso de la factura electrónica y creación del registro contable de facturas en el Sector Público. Las empresas proveedoras de productos o servicios a las Administraciones Públicas en España están obligadas a presentar las facturas de importe igual o superior a 5.000 euros. También se está empezando a extender en países de América Latina. Sin embargo, en España sólo es obligatoria en el ámbito del sector público. Es cuestión de tiempo que el uso de la factura electrónica sea extendido en el ámbito privado, propiciado por los avances de la ciencia y la tecnología. En cambio, en México desde 2014 oficialmente la factura electrónica es obligatoria para todos los contribuyentes, con el objetivo de evitar fraudes fiscales.

### 1.1.2 Liquidación automatizada del IVA

Según la Ley 58/2003, de 17 de diciembre, General Tributaria, en su artículo 106.4, los gastos deducibles y las deducciones que se practiquen, cuando estén originados por operaciones realizadas por empresarios o profesionales, deberán justificarse, de forma prioritaria, mediante la factura. Asimismo, la importancia de la factura como medio de prueba en el ámbito tributario se explica en la exposición de motivos de la Real Decreto 1496/2003, de 28 de noviembre, por el que se aprueba el Reglamento por el que se regulan las obligaciones de facturación, y se modifica el Reglamento del Impuesto sobre el Valor Añadido. A este respecto se señala lo siguiente:

> *"En lo concerniente al Impuesto sobre el Valor Añadido, al Impuesto General Indirecto Canario y al Impuesto sobre la*

*Producción, los Servicios y la Importación, la expedición de la factura tiene un significado especialmente trascendente, ya que en estos tributos la factura va a permitir el correcto funcionamiento de su técnica impositiva, pues a través de ella va a efectuarse su repercusión, a la vez que la posesión de una factura que cumpla los requisitos que se establecen en el Reglamento por el que se regulan las obligaciones de facturación va a permitir, en su caso, que el destinatario de la operación practique la deducción de las cuotas soportadas"*.

Según el trabajo realizado por [1], la factura constituye un documento fundamental en el Impuesto sobre Valor Añadido (IVA) en orden a la deducción del IVA soportado pues "sin factura no hay IVA deducible que valga, por mucho que la operación se haya realizado, figure contabilizada, existan otros medios probatorios de la realidad de la transacción".

La normativa sobre facturación en el Derecho tributación español tiene sus orígenes en la legislación del IVA. Así todo cambio en la VI Directivas del IVA ha comportado modificación de nuestra legislación del impuesto en España y las sucesivas introducciones y modificaciones de los reglamentos de facturación, estando actualmente regulado en la Real Decreto 1619/2012, de 30 de noviembre, por el que se aprueba el Reglamento por el que se regulan las obligaciones de facturación.

El Impuesto de Valor Añadido (IVA) es un tributo que grava la entrega de bienes y las prestaciones de servicios y su hecho imponible se encuentra regulado en el artículo 13 de la Ley 37/1992, de 28 de diciembre, del Impuesto sobre el Valor Añadido. En Canarias la imposición indirecta se contempla en Ley 20/1991, de 7 de junio, de modificación de los aspectos fiscales del Régimen Económico Fiscal de Canarias.

Los sujetos pasivos del impuesto deben determinar la deuda tributaria e ingresarla en el lugar, forma, plazos e impresos establecidos legalmente. Este procedimiento se denomina liquidación del IVA, el cual se ha de realizar trimestralmente, si bien las grandes empresas lo realizarán mensualmente. El modelo fiscal en que se formaliza es el modelo 303 y se presenta telemáticamente a la Agencia Tributaria. Teniendo en cuenta [2], cada entrega de bienes y servicios estará gravada sobre el IVA, de tal manera que la empresa cuando venda sus mercancías o preste sus servicios deberá consignar en la factura el importe del IVA repercutido. De otro lado, cuando la empresa compre bienes y servicios, el proveedor le entregará una factura por el importe gravado por el IVA, denominado IVA soportado. La base imponible del IVA se determina por el precio facturado menos todos los descuentos. A dicho importe se le aplicará un tipo de gravamen el cual dependerá del tipo de operación.

Tal como se ha indicado anteriormente, la liquidación del IVA consiste en determinar la deuda tributaria, que consiste en la diferencia entre el IVA repercutido y el IVA soportado. Tal diferencia, si es positiva, deberá ingresarse en la Agencia Tributaria a

través del referido modelo 303. Si la diferencia es negativa, se generará un derecho de cobro que se compensará en la siguiente liquidación, a menos que nos encontremos al cierre del ejercicio económico.

La información fiscal que se desprende de las operaciones económicas que vienen gravadas por el IVA también se representan en la contabilidad, el IVA soportado que es deducible del IVA repercutido, es decir, que minora la base imponible del impuesto vendrá representado en una cuenta denominada Hacienda Pública, IVA soportado del grupo 4 del cuadro de cuentas del Real Decreto 1514/2007, de 16 de noviembre, por el que se aprueba el Plan General de Contabilidad (Plan General de Contabilidad). Según la normativa contable, cuando el IVA soportado no puede deducirse del IVA repercutido, formará parte del precio de adquisición y, por tanto, de la cuenta de gasto del elemento o de activo. Ello significa que la empresa al no poder reducir el IVA soportado no tendrá ninguna cuenta específica en los estados financieros.

Tras una breve introducción de la liquidación, llega unos de los enfoques del Big Data en el sector fiscal, la liquidación automática del IVA, que consiste en el análisis y cálculo de IVA de repercutido y el IVA soportado de forma automática, rápida y eficaz del precio facturado de una empresa o de cualquier entidad económica que ofrezca bienes o servicios. Además de suponer una enorme facilidad para la Agencias tributarias a la hora comprobar esos datos, supone un ahorro de tiempo a la hora de presentar el modelo fiscal 303 ante la agencia tributaria telemáticamente.

Este avance permite un ahorro económico importante en las empresas y una facilidad para estas a la hora de hacer las desgravaciones ya que nos ahorramos “la mano de obra” que se encarga de realizar tal análisis.

## 1.2 Factura

Tal como se ha comentado con anterioridad la factura es un documento que permite justificar la operación de entrega de un bien o servicio y esencial para la deducibilidad fiscal. La deducibilidad del gasto a través de su conveniente justificación formal a través de la factura no sólo está contemplada en la imposición indirecta, sino también en la imposición directa, impuesto sobre sociedades, a efectos de desgravar los gastos incurridos por las sociedades de los ingresos obtenidos. A continuación, se estudia los tipos de facturas, así como el caso especial de la factura en México.

### 1.2.1 Factura genérica

El sistema de facturación está basado en dos tipos de facturas: [3]

- Factura completa u ordinaria.
- Factura simplificada: Las facturas simplificadas, que sustituyen a los tiques, tienen un contenido más reducido que las facturas completas y por lo general, están permitidas solo para facturas no superiores a unos 400 euros.

La factura completa tiene los siguientes campos:

- Número y, en su caso, serie. La numeración ha de ser correlativa dentro de cada serie.
- Fecha de su expedición.
- Fecha de realización de la operación documentada en la factura (si no coincide con la fecha de expedición).
- Número de identificación fiscal (NIF), nombre y apellidos o denominación social tanto del obligado a expedir la factura como del destinatario de las operaciones.
- Domicilio del expedidor y del destinatario.
- Descripción de las operaciones y todos los datos necesarios para la determinación de la base imponible y su importe y precio unitario sin impuesto, así como cualquier descuento o rebaja que no esté incluido en el precio unitario.
- Tipos impositivos aplicados a las operaciones.
- Cuota tributaria, que deberá consignarse por separado.
- Sede de actividad o establecimiento si alguna de las partes o ambas disponen de varias sedes fijas del negocio.
- Moneda
- Lengua, Las facturas pueden expedirse en cualquier lengua. Existe la posibilidad de exigir traducción al castellano u otra lengua oficial.
- Firma y localidad (opcional).
- Medios de expedición, papel o formato electrónico que permita garantizar la autenticidad de su origen, la integridad de su contenido y su legibilidad.

También existe la denominada factura proforma que es simplemente un anticipo de la posterior factura real o la factura recapitulativa que se genera por periodos mensuales, es decir, pueden incluirse en una sola factura las operaciones realizadas en distintas fechas para un mismo destinatario. Deben ser expedidas, como máximo, el último día del mes natural en el que se realizaron las operaciones documentadas en la factura.

La factura electrónica es el documento electrónico equivalente a la factura en papel. Consiste en una transmisión telemática entre un emisor y un receptor por medio de dispositivos electrónicos capaces de generar las facturas adecuadas, que será la que estudiaremos más a fondo en el caso de México.

Las entidades o empresas deberán conservar los siguientes documentos:

• Las facturas recibidas.

• Las copias de las facturas expedidas.

La obligación de conservación puede cumplirse mediante la utilización de medios electrónicos. Las obligaciones se pueden cumplir materialmente por un tercero que actúe en nombre de la empresa. El Código de Comercio obliga a los empresarios a conservar los libros contables, la correspondencia, la documentación y los justificantes de sus operaciones durante seis años, a partir del último asiento realizado en los libros.

## 1.2.2 Factura electrónica en México

La factura electrónica en México es ya una realidad en este país, que debido a varias causas de índole fiscal y económica ha traído consigo indirectamente un gran cambio en la manera de crear y administrar las facturas.[4]

La facturación electrónica en México se inició en el 2004, cuando el Servicio de Administración Tributaria (SAT) creó el marco legal que definió la implantación del Comprobante Fiscal Digital a través de Internet (CDFI). Con el tiempo se ha convertido en uso obligatorio para la contabilidad electrónica de México y actualmente es utilizado en el 100% de las transacciones económicas. El éxito de este sistema implementado en México ha hecho que otros gobiernos de Latinoamérica estén siguiendo sus pasos y trabajan en la implantación de una facturación electrónica propia. El sistema de facturación electrónica de México ha destacado gracias a su rápida difusión, que le ha permitido extenderse en pocos años por toda la sociedad mexicana.

El CFDI es un documento XML, el cual está diseñado para cumplir con la especificación y exigencias del SAT y son las siglas que indican 4 características importantes de esta factura:

- Comprobante: justifica ante la administración fiscal que realmente se efectuó un proceso de compra venta y que se pagaron los impuestos designados.
- Fiscal: garantiza que el comprobante se encuentra en el marco legal, que tiene que ajustarse a la legislación aplicable y enviarse en plazo al SAT. Este punto es muy importante ya que fue una las causas más importantes que motivaron este cambio en la facturación debido a los fraudes fiscales.
- Digital: las facturas están creadas en un sistema binario y son almacenadas en un disco duro donde se puede hacer una búsqueda de una factura, de un archivo y encontrarla en segundos tan solo indicando su nombre, fecha o alguna otra característica de esta. Además, se puede mantener archivada fácilmente sin llegar a perderla en ningún momento.
- Internet: Las empresas que emiten CFDI, las emiten con su propio sistema de facturación, descentraliza la facturación y, por lo tanto, se ahorra un gran capital en términos de seguridad informática. Por lo tanto, este sistema permite emitir CFDI desde todo tipo de dispositivos electrónicos.

La Plataforma Interna del SAT es la encargada de coordinar, administrar y sincronizar a todos los proveedores que están autorizados, también se encarga de administrar la información concerniente a cada uno de los contribuyentes y sus respectivas facturas electrónicas asociadas. Poseen un buen sistema de seguridad basado en el sistema de criptografía de clave pública, por lo que le otorga a cada contribuyente algunas claves, una pública y otra privada.

Como conclusión de este sistema, podemos ver que tenemos un sistema bien desarrollado y con vistas de futuro que con el tiempo el resto de los países seguirá y posiblemente se convierta en la base de la facturación en el mundo.

## 1.3 Explicación de la aplicación “Fintech”

La Tecnología Financiera o Fintech es una industria que aplica las nuevas tecnologías de la información, concretamente el creciente sector del Big Data, al sector financiero [5]. Esta industria este dividida en cuatro segmentos:

- Herramientas de operación y medios de pago
- Conocimiento del cliente y Big Data
- Seguridad e identificación de personas
- Dinero electrónico

Las aplicaciones de Tecnología financiera (o Aplicación Fintech) es un tipo de aplicación financiera que proporciona herramientas tecnológicas a este sector para poder facilitar la labor de los usuarios de la aplicación a la hora de tratar con grandes volúmenes de datos.

Estas aplicaciones tienen como objetivo automatizar procesos que hoy en día se producen manualmente como la facturación, mediante la digitalización de los datos y el procesamiento automático de éstos por parte de procesadores y células inteligentes. Para ello, esta entidad inteligente tiene la facultad de procesar lenguaje natural y derivados, o leer archivos usados para el almacenamiento de datos, como XML o JSON, para poder realizar adecuadamente su labor; esto suele llevar detrás el desarrollo de una inteligencia que puede entender tales lenguajes. Esta tecnología tiene diversos usos como:

- Pagos y remesas.
- Préstamos.
- Gestión de finanzas empresariales.
- Gestión de finanzas personales.
- Crowdfunding (financiamiento de proyectos).
- Gestión de inversiones.
- Seguros.
- Educación financiera y ahorro.
- Soluciones de scoring, identidad y fraude.
- Trading y mercados.

# 2. Motivación y objetivos iniciales

En este apartado se presentan los motivos que me han llevado a elegir este proyecto para mi trabajo final de grado y la satisfacción que me ha producido llevarlo a cabo, así como los objetivos que se pretenden alcanzar con el mismo

## 2.1 Motivación

Este proyecto ha supuesto un enorme interés desde el inicio de su desarrollo debido a varios factores. El primero de ellos es enfrentarme al reto de crear por primera vez una aplicación Fintech, un campo completamente novedoso para mí, siguiendo una arquitectura de Big Data, otro sector completamente nuevo que, aplicado al sector financiero, supuso estudiar un nuevo enfoque de la programación que conllevaría aprender mucha información para el desarrollo del proyecto.

Otro aspecto para destacar es el tamaño del proyecto completo, que también ha sido un nuevo reto para mí, porque el haber desarrollado un módulo de este proyecto ha llevado mucho tiempo, además de la posterior integración con el resto de mis compañeros. Es sin duda el proyecto más ambicioso que he realizado hasta el momento y al que le he dedicado mucho esfuerzo y dedicación con entusiasmo y emoción.

Centrándonos en mi proyecto personal, el hecho de realizar una simulación en la que los elementos de este interactúen entre sí, realizando unas acciones realistas y aceptables dentro del marco de la normalidad, era de antemano, un desafío para mí. Llevar a cabo un proyecto tan avanzado y sólo con la ayuda de mi tutor era una idea que creía imposible al inicio del trabajo, debido a mi nivel de formación de partida, inseguridad que fue disipándose a medida que profundizaba más en el proyecto.

Este proyecto parte desde cero, sin ningún tipo de trabajo de referencia, teniendo yo la responsabilidad de la arquitectura y las dependencias a utilizar, de realizar todas las funcionalidades necesarias de la simulación. Siguiendo con las ideas del tutor, se ha considerado oportuno que el proyecto tuviera el mínimo de dependencias externas, ya que son factores que luego pueden afectar directamente al mismo. Por ello, todos los distintos módulos y funcionalidades que necesitaré para desarrollar el proyecto en el futuro serán implementadas, en la medida de lo posible, por mí. Sin embargo, tengo claro que quiero buscar un equilibrio en el que también se puedan utilizar ciertas librerías ya existentes que no supongan grandes dependencias en mi proyecto final.

## 2.2 Objetivos iniciales

Este proyecto tiene como propósito generar un Publisher (feeder) de la Plataforma de Datos, es decir, un generador de datos financieros realistas con el fin de crear facturas de compra de bienes y servicios a proveedores, de venta a clientes y de nóminas de personal, todos ellos involucrados en el desarrollo económico de las empresas.

Estas empresas o agentes pondrán a disposición un gran volumen datos financieros que serán utilizados por otros proyectos posteriormente, para su estudio y análisis. Este objetivo está enfocado al ámbito de la restauración, y es posible a través de un ciclo iterativo basado en tres fases, Modelación, Simulación y Validación, que se definen a continuación:

Modelación: Consiste en estructurar y abstraer los datos, elementos y eventos de la realidad para poder simularlos posteriormente, ya sea el consumo de restaurantes por parte de clientes y las facturas generadas, además de las nóminas y la compra de materias primas.

Simulación: Trata de añadir estos elementos dentro del entorno y que interactúen entre sí mediante compra/venta de materias primas, contratación/despido de personal y consumo de los clientes.

Validación: Consiste en el análisis de los datos recibidos de la simulación y en comprobar que lo datos recibidos son acordes y se reflejan en la realidad.

A partir de lo anterior, han surgido nuevos requisitos que se han tenido que cumplir, tales como la generación de facturas de servicios, de devolución de productos, además de tener que añadir suficientes elementos en la simulación: contratos de trabajadores, nacimiento y muerte de individuos y empresas que están relacionadas con el negocio, jubilación de los trabajadores, pago de impuestos por parte de las empresas, pago de hipoteca. Todo ello teniendo en cuenta un aspecto muy importante en las simulaciones, que según la situación de los individuos y empresas de la simulación, se toman una serie de decisiones adecuadas para mejorar el realismo de la simulación.

# 3. Desarrollo

## 3.1 Metodología aplicada

El lenguaje de programación que utilizaremos principalmente para elaborar la aplicación "Fintech" será Java, utilizando el IDE "Intellij". Se hará uso de la herramienta "Intino", que permite la creación de la plataforma de datos. Se usará una arquitectura de programación funcional, aplicando un desarrollo guiado por pruebas (TDD – Test-Driven Development) e iterativo, en el que se establecerán una serie de objetivos/hitos a cumplir en cada iteración. Cada iteración tendrá una duración aproximada de 2 semanas, en la que se le dedicará un total de 50 horas en cada una.

Se hará uso de un "bus" de mensajería proporcionado por "Intino" para comunicar las distintas partes funcionales del proyecto realizadas por los distintos participantes. Cabe destacar que "Intino" usa la tecnología JMS (Java Messaging Service) que consiste en una solución para el uso de colas de mensajes. Es un estándar de mensajería que permite a los componentes de aplicaciones crear, enviar, recibir, y leer mensajes, y que permite una comunicación confiable de manera asíncrona.

Para el control de versiones del proyecto se utilizará la tecnología "Git", y dada la naturaleza grupal del proyecto, se aprovechará el modelo de ramificación "GitFlow" para facilitar el desarrollo concurrente y organizado por parte de todos los integrantes del proyecto.

## 3.2 Planificación

He dividido mi proyecto en 3 fases, estudio del ámbito y las tecnologías que se usaran, desarrollo e implementación del proyecto, y validación y prueba del proyecto en el entorno de usuario. Por último, se encuentra la documentación final.

| ***Fases*** | ***Duración Estimada (horas)*** | ***Tareas (nombre y descripción, obligatorio al menos una por fase)*** |
|---|---|---|
| Primera Iteración: Estudio previo y Análisis de las tecnologías y la metodología que se usaran. | 50 | Iteración 1.1: Estudio y búsqueda de un generador de personas aleatorias usadas como clientes de la simulación |
| | | Iteración 1.2: Estudio de la tecnología JSOUP para la obtención de datos de restaurantes la página web TripAdvisor, usados como restaurantes de la simulación. |
| | | Iteración 1.3: Análisis de datos relacionados con la parametrización de la simulación. Por ejemplo, distribuciones de sueldos, del tamaño del restaurante, del grado de consumo de restaurantes por parte de la población y el número de personas que son invitadas en función de sus relaciones sociales. |

| | | |
|---|---|---|
| | | Iteración 1.4: Estudio de las facturas electrónicas generadas en México usando CFDI, que es un documento XML que cumple con las especificaciones proporcionadas por la SAT (Servicio de Administración Tributaria). |
| Segunda Iteración: Diseño, Desarrollo e Implementación de la Simulación | 250 | Iteración 2.1: Desarrollo de un módulo que extraiga de internet, una lista de restaurantes con sus datos para ser usados posteriormente en la simulación. |
| | | Iteración 2.2: Desarrollo de un módulo que extraiga de un archivo CSV, una lista de clientes y empresas secundarias con sus datos para ser usados posteriormente en la simulación. |
| | | Iteración 2.3: Guardar los datos de clientes, empresas secundarias y restaurantes en una base de datos fácilmente accesible para su rápida lectura durante la simulación. |
| | | Iteración 2.4: Diseño y desarrollo de un sistema de tiempo que permita controlar un calendario artificial con días meses y años, en el que sucederán todos los actos dentro de la simulación. |
| | | Iteración 2.5: Diseño y desarrollo de un sistema de simulables que interactuarán independientemente cada día de la línea de tiempo, todos los elementos que formen parte de la simulación serán simulables. |
| | | Iteración 2.6: Diseño e implementación de un sistema de consumo de restaurantes por parte de los clientes con cierta temporalidad parametrizada, que genere facturas de comidas y que sean incluidas en sus datos financieros para su posterior análisis, esto generará facturas de compras. |
| | | Iteración 2.7: Adición de empresas relacionadas con el sector de la Restauración como proveedores y empresas de servicios que provean de materias primas y servicios a los restaurantes, esto generará facturas de compras de productos, servicios y devoluciones. |
| | | Iteración 2.8: Desarrollo de un sistema de contrataciones de los trabajadores que trabajaran en los restaurantes a cambio de un salario que influya también en los datos financieros de la empresa, esto generará facturas de nóminas. |
| | | Iteración 2.9: Diseño e implementación de un sistema de estrategias a la hora de tomar decisiones en las distintas situaciones que se presentan. Por ejemplo, a la hora de contratar los trabajadores, los restaurantes elegirán quien contratar según la situación. |
| | | Iteración 2.10: Desarrollo de un sistema de contrataciones de los trabajadores que trabajaran en los restaurantes a cambio de un salario que influya también en los datos financieros de la empresa, esto generará facturas de nóminas. |

| | | |
|---|---|---|
| | | Iteración 2.11: Implementación de un sistema de impuestos y pagos de hipoteca del local por parte de las empresas. |
| | | Iteración 2.12: Creación de una vista en web para mostrar la simulación en la que el usuario puede ver los eventos que se producen, e interactuar y cambiar los ajustes de este. |
| Tercera Iteración: Evaluación, Validación y Prueba de la Simulación en el Entorno de Finbook | 60 | Iteración 3.1: Testeo y búsqueda de errores. |
| | | Iteración 3.2: Integración del Publisher con la Plataforma de datos |
| Cuarta Iteración Documentación y Presentación del Proyecto realizado | 40 | Iteración 4.1: Documentar todo el proyecto realizado. |
| | | Iteración 4.2: Preparación de la presentación del trabajo. |

# 4. Herramientas usadas

## 4.1 Herramientas de hardware

Las herramientas de hardware utilizadas son las siguientes:

- Ordenador de Sobremesa: herramienta fundamental y principal para el desarrollo de este proyecto, ya que ha sido el lugar de creación, desarrollo y almacenamiento de este, además del almacenamiento y uso de todos los programas necesarios para el desarrollo de la simulación.
- Ordenador Portátil: herramienta adicional para continuar con el desarrollo de este proyecto cuando no me encuentro cerca del lugar de trabajo principal.
- Conexión a internet: herramienta esencial en todo tipo de proyectos hoy en día, para la documentación y la descarga de aplicaciones necesarias.

## 4.2 Herramientas de software

Las herramientas de software utilizadas son las siguientes:

- J**ava:** lenguaje de programación principal usado durante el desarrollo y la implementación de la simulación.
- **Javascript + JSP + CSS:** lenguajes y herramientas principales en el entorno gráfico y visual del usuario, ya que he usado el entorno web para mostrar la simulación.
- **Intellij IDEA:** programa principal de desarrollo del proyecto tanto en la parte de servidor con Java como la parte del cliente en web.
- **Tomcat Server:** herramienta de Tomcat que he usado como servidor del proyecto.
- **Javax Web-socket:** herramienta de Javax para el desarrollo de los sockets entre el servidor y el cliente con el pasar información directa entre archivos de Javascript y archivos Java.
- **Jsoup:** librería para “Web Scrapping”, es decir, leer datos de los archivos html de las páginas web. [6]
- **Generador de datos aleatorios:** he usado un generador de datos para crear los perfiles de las distintas empresas y personas de la simulación, exceptuando los restaurantes. [7]
- **TripAdvisor:** página web usada para leer los datos de todos los restaurantes activos en Gran canaria, a través de la librería Jsoup. [8]
- **Jquery:** librería para complementar en Javascript en la administración de la web. [9]
- **DB Browser for SQLite:** programa para administrar leer, y guardar los datos en bases de datos SQL que se manejaran durante la simulación, esto incluyen datos de clientes, empresas secundarias, facturas y restaurantes. [10]

# 5. Arquitectura

## 5.1 Arquitectura Publisher/Subscriber

Esta arquitectura es muy usada para el desarrollo de proyectos Big Data y es la arquitectura elegida para este proyecto de Finbook [11]. La arquitectura está dividida en tres módulos: los Publisher (Publicadores), los Subscriber (Suscriptores) y el Data Hub (Centro de datos). A continuación, se explicarán las funciones de cada módulo:

- Centro de datos: como su nombre indica, es el centro donde se guardan los datos. Se encarga de recibir los datos de los publicadores y proveer de esos datos a los suscriptores. Cuando se quiere publicar un dato, genera un evento indicando los datos del envío y el dato a guardar. Luego manda un evento a los suscriptores de que se ha añadido un nuevo dato. Por último, el suscriptor obtiene el dato de ese evento para su posterior uso.
- Publicador: es un módulo que genera datos para que otros módulos puedan usarlos, puede haber varios publicadores conectados al centro de datos. A través de un módulo de conexión perteneciente al centro de datos, se conecta al servidor donde se encuentra desplegado el centro de datos, el módulo genera el dato, y el módulo de publicador genera el evento en el centro de datos.
- Suscriptor: este módulo es el encargado de recibir esos datos para su posterior uso, puede haber varios suscriptores suscritos al centro de datos. A través del módulo desarrollado como suscriptor del centro de datos, se conecta al servidor al igual que el publicador, pero para suscribirse al centro de datos. Así, cuando se genere algún evento, el centro de datos publicará el evento a todos los suscriptores, usando el módulo de conexión, y de este obteniendo el dato en cuestión.

A continuación, se muestra visualmente como es una arquitectura Publisher/Subscriber genérica.

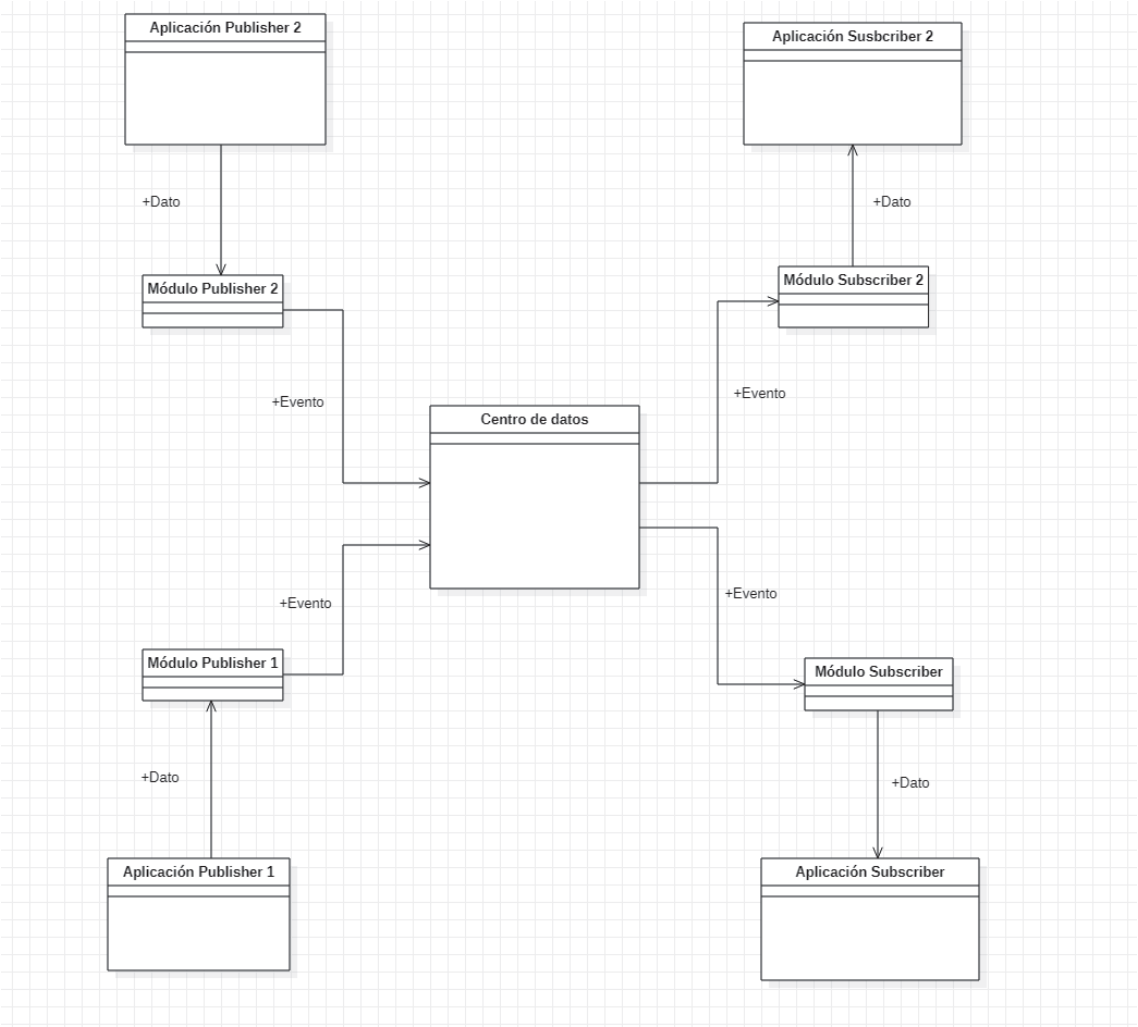



*Ilustración 1: Diagrama arquitectura Publisher/Subscriber*

A partir de esta arquitectura, se ha desarrollado Finbook, que está divida en 4 módulos: un centro de datos, un Publisher, un Subscriber y uno mixto (Publisher y Subscriber). A continuación, se explican los distintos módulos:

- El primero es un centro de datos encargado de guardar facturas. Realizado por Raúl Lozano Ponce.
- El segundo módulo se encarga de generar una gran densidad de facturas realistas a través de un entorno empresarial ficticio, generar los archivos de las facturas y publicarlas. Realizado por mí, Juan Alberto Ureña Rodríguez.
- El tercero es encargado de procesar las facturas para calcular las desgravaciones del IVA. Realizado por Juan Kevin Trujillo Rodríguez.
- El último es principalmente un Subscriber que se encarga de calcular el estado financiero, aunque también como Publisher permite subir de facturas ya creadas al centro de datos a través de un “Drag and Drop”. Realizado por Gerardo Santana Franco.

A continuación, se muestra un diagrama que muestra con más detalle la estructura de nuestro proyecto.

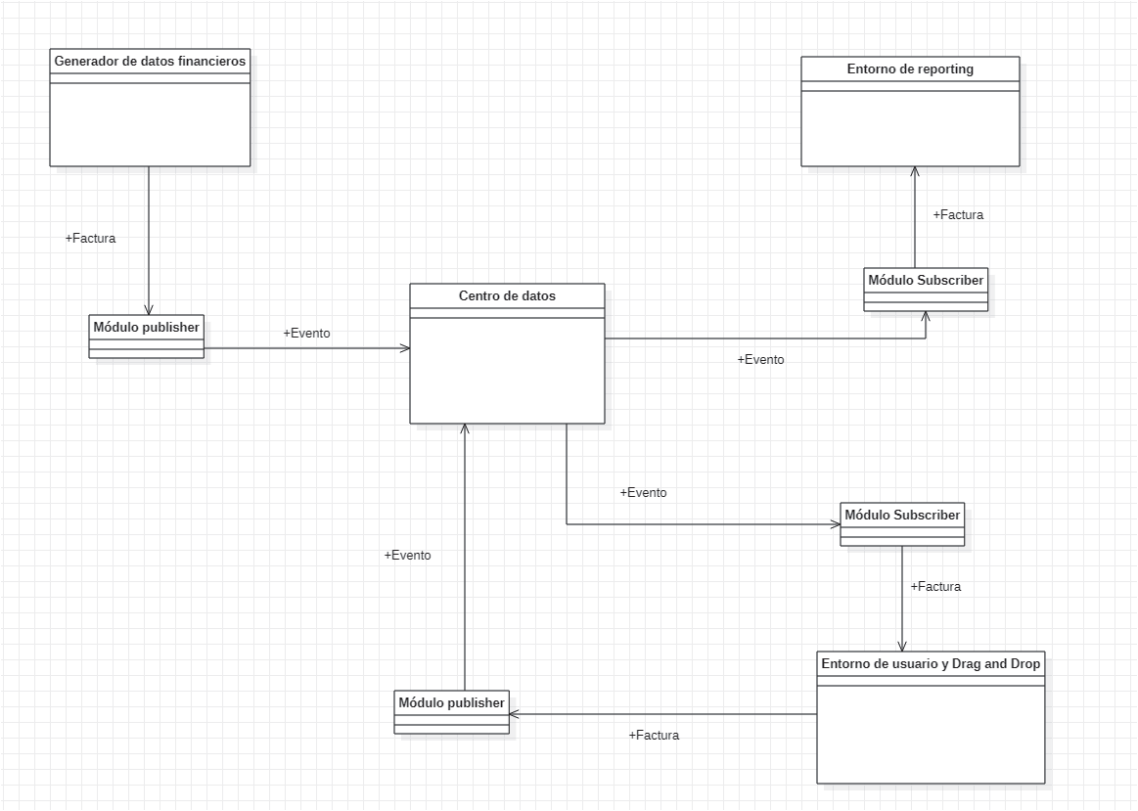



*Ilustración 2: Diagrama de Finbook*

# 5.2 Arquitectura del módulo desarrollado

## 5.2.1 Arquitectura Multi-Agente

Mi módulo desarrollado ha seguido una arquitectura Multi-Agente, muy común en el desarrollo de simuladores como es este caso [12]. La arquitectura consiste en la participación de múltiples agentes inteligentes de forma simultánea que realizan unas acciones en función del rol asociado. En cada ciclo de la simulación, todos los agentes realizan de forma independiente y asíncrona, las acciones que crean oportunas dentro del entorno de la simulación donde se sitúan. Estos actores de la simulación tienen cierta inteligencia ya que realizan acciones acordes a cómo los agentes actúan en la realidad.

En mi proyecto, el ciclo de la simulación lo dicta la **Línea de Tiempo**, en el que cada día del ciclo de la simulación cada uno de los agentes participará en el entorno. Esto se explicará durante el apartado del ciclo de vida de la Simulación (7.2.1). Los agentes y sus acciones serán también explicados más adelante, concretamente en la sección de Agentes activos o Simulables (7.2.2).

## 5.2.2 Diseño modular

En esta sección se ha desarrollado dos diagramas por cada módulo que reflejan correctamente los elementos estructurales más importantes del proyecto.

### 5.2.2.1 Módulo de la simulación

En cuanto al módulo de la simulación, se ha desarrollado un diagrama de clases sobre los elementos principales del mismo y sus dependencias entre estos. Debido al enorme tamaño de este módulo (más de 250 clases), en este diagrama se han omitido muchos detalles que serán mejor explicados durante la explicación de este módulo (apartado 7.2). Como detalle a comentar, en el segundo diagrama, en la parte superior izquierda se encuentra el paquete de la capa de datos se omite su arquitectura ya que se explica detalladamente en su sección y no requiere de diagramas de la base de datos ya que sus tablas y su organización son sencillas.

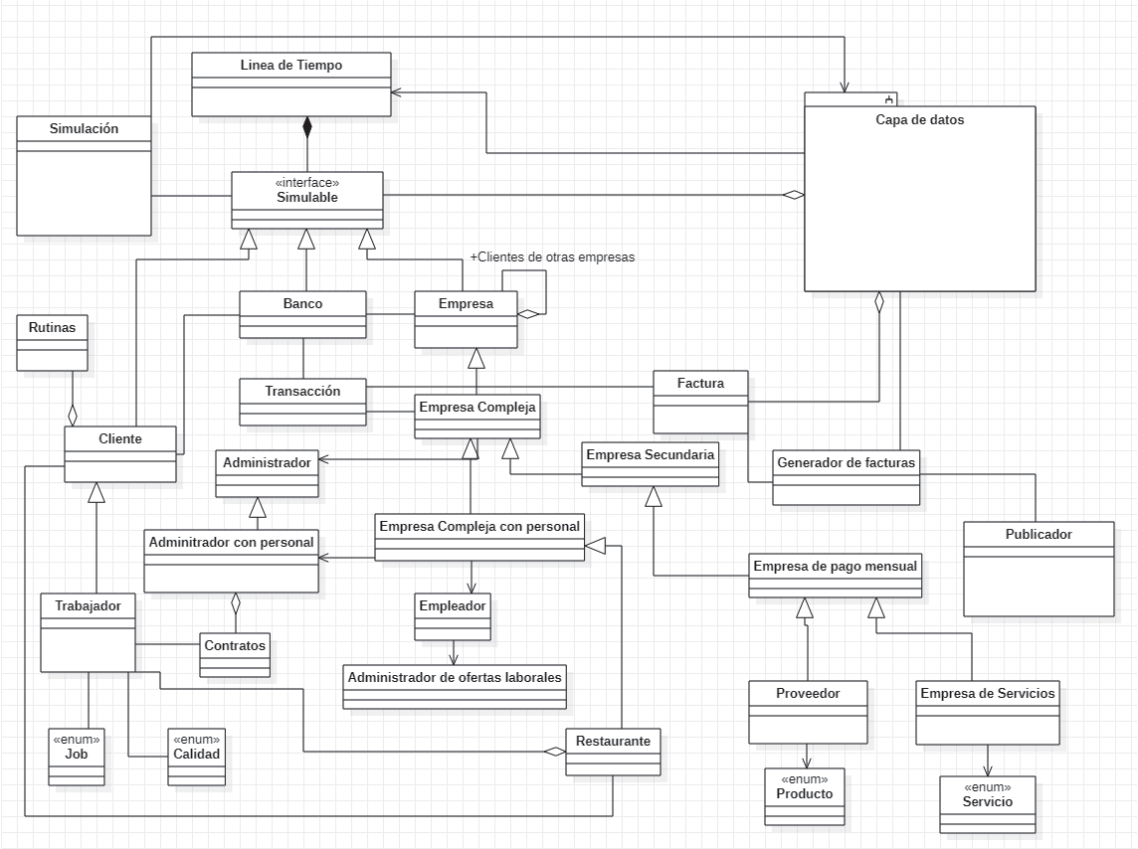



*Ilustración 3: Diagrama de clases de la simulación, capa de negocio*

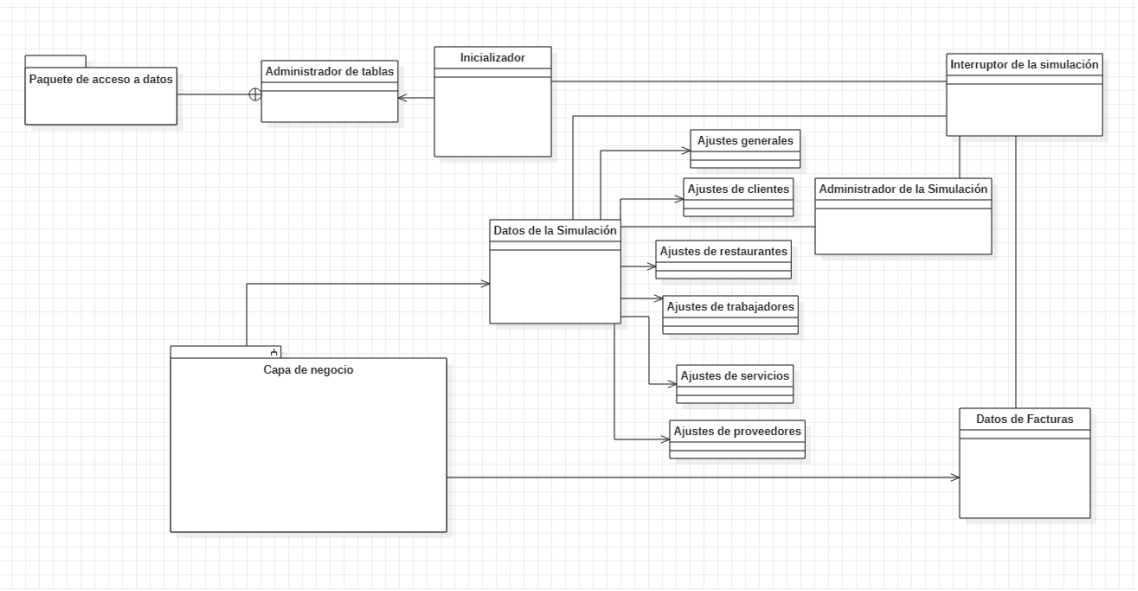



*Ilustración 4: Diagrama de clases de la Simulación, capa de datos*

Además, he realizado un diagrama de estados de la simulación para mostrar los estados principales en los que puede estar la simulación. Dentro del diagrama se puede observar el ciclo principal de la simulación que son, la fase de acción de los simulables, el nuevo día y la fase administración.

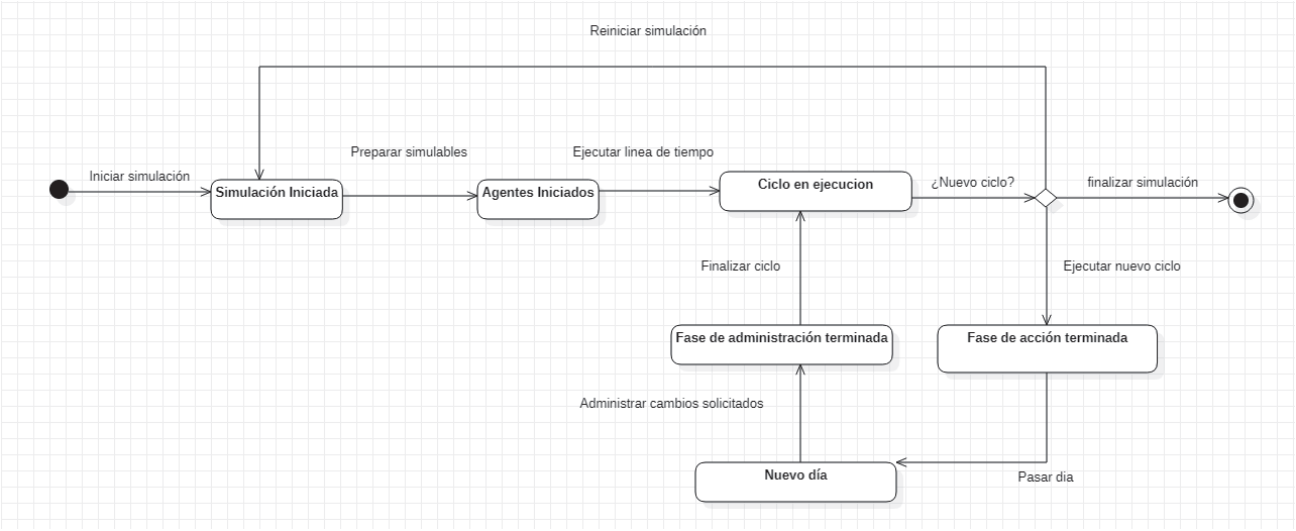



*Ilustración 5: Diagrama de estados de la simulación*

## 5.2.2.1 Módulo del entorno de usuario

En cuanto al segundo módulo, he hecho dos diagramas de clases, uno del servidor y otro del cliente. En el diagrama del servidor nos encontramos el paquete del servidor de tomcat con el contenedor de servlet, los servlets desarrollados en este proyecto y los comandos que pertenecen al Frontcontroller Servlet, para más detalles de esto último se explican en la sección de explicación del Servidor (7.3.2).

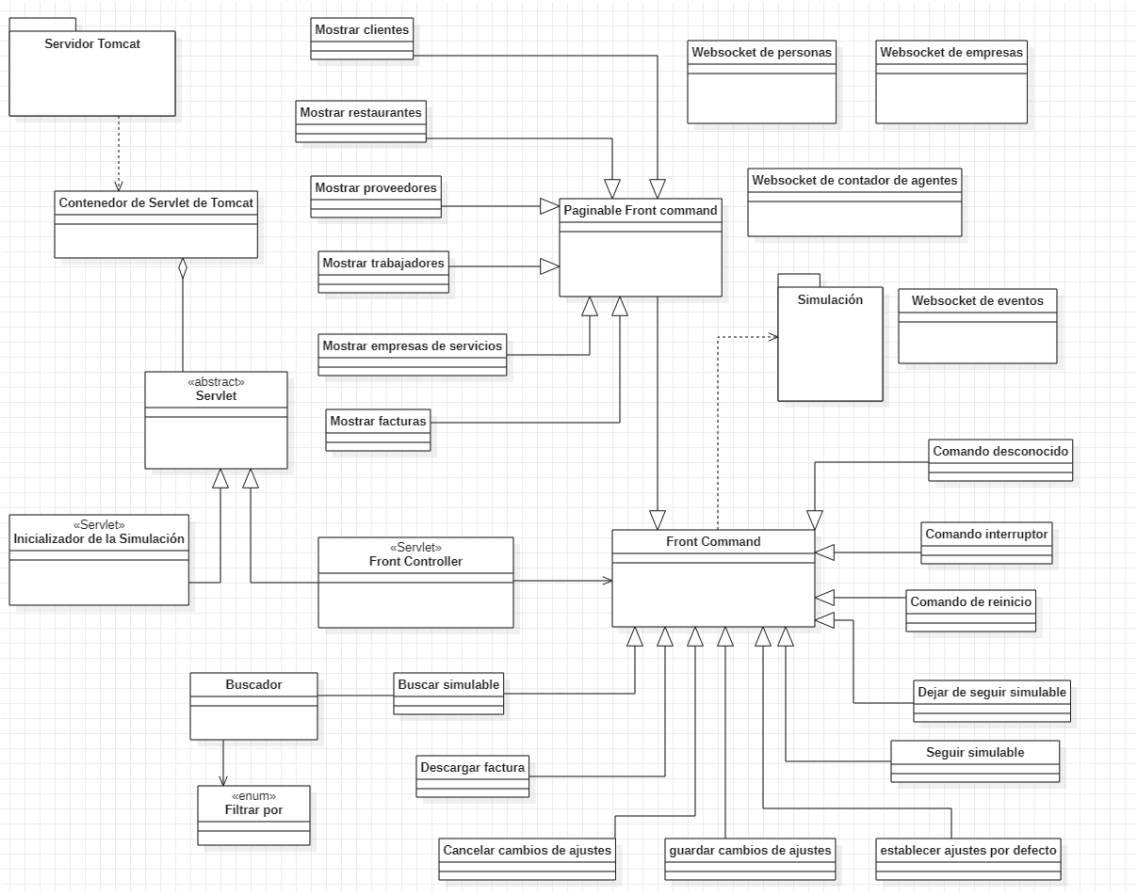



*Ilustración 6: Diagrama de clases del servidor*

En cuanto al diagrama del cliente, se muestran todas las vistas de la aplicación (archivos JSP) y todos los principales administradores y procesadores de datos (archivos JS o Javascript). Todas las vistas se conectan al servidor mediante peticiones HTTP al servlet del Frontcontroller con el comando solicitado (apartado 7.3.2).

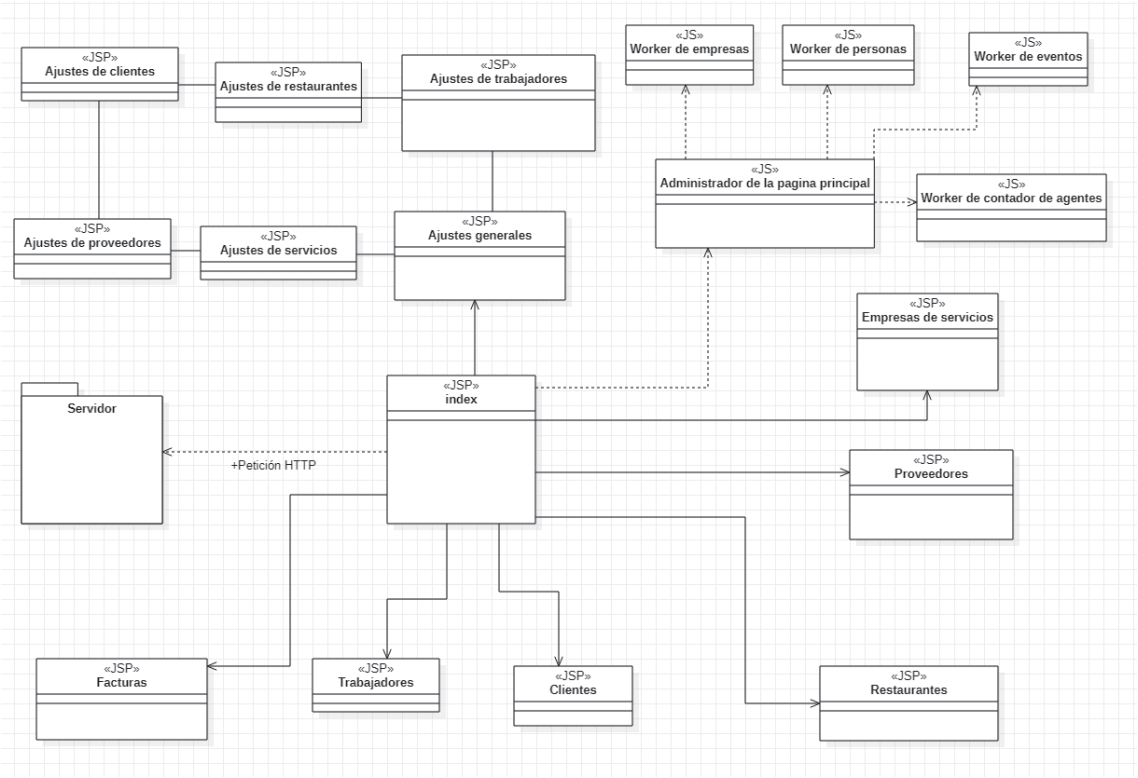



*Ilustración 7: Diagrama de clases del cliente*

## 5.2.2 Pila tecnológica

La pila tecnológica son las dependencias externas de tecnologías que se requieren para la ejecución del proyecto en las que, si faltara alguna de ellas, no se podría ejecutar éste. Las dependencias son las siguientes:

- **Tomcat Server**: es el servidor necesario para el despliegue del entorno de usuario que ejecutará la simulación. Para más detalles, en herramientas de Software (4.2).
- **DB Browser for SQLite:** es la aplicación de gestión de base datos necesario para guardar y acceder a los datos de los agentes de la simulación. Para más detalles, en herramientas de Software (4.2).
- **JVM:** Java Virtual Machine es la máquina virtual que permite interpretar y ejecutar el código binario de java (bytecode Java). [13]

# 6. Competencias especificas cubiertas

- **TFG01**: Ejercicio original a realizar individualmente y presentar y defender ante un tribunal universitario, consistente en un proyecto en el ámbito de las tecnologías específicas de la Ingeniería en Informática de naturaleza profesional en el que se sinteticen e integren las competencias adquiridas en las enseñanzas.

Justificación: este proyecto considero que es bastante original ya que el sector del Big data es un sector bastante nuevo y que no hemos aprendido durante la carrera.

- **CII01**: Capacidad para diseñar, desarrollar, seleccionar y evaluar aplicaciones y sistemas informáticos, asegurando su fiabilidad, seguridad y calidad, conforme a principios éticos y a la legislación y normativa vigente.

Justificación: se ha podido desarrollar un sistema informático de gran calidad que genere facturas realistas y fiables.

- **CII06**: Conocimiento y aplicación de los procedimientos algorítmicos básicos de las tecnologías informáticas para diseñar soluciones a problemas, analizando la idoneidad y complejidad de los algoritmos propuestos.

Justificación: se han usado y desarrollado una gran variedad de algoritmos avanzados que puedan calcular las acciones y los factores que afecten a la simulación.

- **CII07**: Conocimiento, diseño y utilización de forma eficiente los tipos y estructuras de datos más adecuados a la resolución de un problema.

Justificación: en este proyecto se han manejado una gran variedad de datos que se guardan tanto ficheros de texto como en bases de datos SQL.

- **CII08**: Capacidad para analizar, diseñar, construir y mantener aplicaciones de forma robusta, segura y eficiente, eligiendo el paradigma y los lenguajes de programación más adecuados.

Justificación: en este proyecto, como he explicado en el apartado de Arquitectura (5.), me he centrado en desarrollar una arquitectura completamente escalable, eficiente y robusta, para posteriormente, añadirle extensiones que aumenten el tamaño de esta manteniendo la misma estructura. Sobre todo, es muy importante priorizar este aspecto ya que este proyecto es infinitamente escalable debido a la gran extensión y complejidad de la economía, donde centrándose en un sector muy concreto, como es la restauración, ha llevado mucho tiempo desarrollarla de forma adecuada.

- **CII013**: Conocimiento y aplicación de las herramientas necesarias para el almacenamiento, procesamiento y acceso a los Sistemas de información, incluidos los basados en web.

Justificación: en este proyecto he usado sistemas para desarrollo web como Tomcat Server para el desarrollo del entorno de usuario, además de varias herramientas pertenecientes a éste y otras tecnologías.

- **CII014**: Conocimiento y aplicación de los principios fundamentales y técnicas básicas de la programación paralela, concurrente, distribuida y de tiempo real.

Justificación: tanto para el apartado de preparación de simulables (7.2.8), como la ejecución de procesos principales como la simulación o la preparación de eventos en el Threadpool (apartado 7.2.4.1 en el Interruptor de la Simulación), o como las acciones de cada uno de los agentes de la simulación, actúan simultáneamente en su propio hilo. Todo ello usando Threadpool y parallelStream de la programación de funcional.

- **CII015**: Conocimiento y aplicación de los principios fundamentales y técnicas básicas de los sistemas inteligentes y su aplicación práctica.

Justificación: este proyecto ha generado una simulación en el que los actores que se encuentran en su interior actúan conforme a unos patrones y factores que se tendrán en cuenta, convirtiendo cada uno de estos actores en agentes inteligentes.

- **CII016**: Conocimiento y aplicación de los principios, metodologías y ciclos de vida de la ingeniería de software.

Justificación: el desarrollo de este proyecto ha estado guiado por un modelo iterativo en el que cada iteración se cumplen unos objetivos marcados. Además, se han usado varios patrones y principios de diseño, como por ejemplo el patrón Strategy (Estrategía) o el principio de sustitución de Liskov.

- **IS03**: Capacidad de dar solución a problemas de integración en función de las estrategias, estándares y tecnologías disponibles.

Justificación: durante el desarrollo de la simulación, he tenido varios problemas para integrar ciertos elementos a la simulación, ya que no se contemplaban en este entorno, por lo que tuve que cambiar ciertos aspectos de la estructura y añadir otros para que la integración de estas dos partes fuera satisfactoria. Además, se ha integrado correctamente los distintos módulos de los distintos estudiantes con escasos problemas ya que hemos mantenido el mismo tipo de factura entre todos los proyectos, que es el elemento principal de información entre los módulos.

- **IS04**: Capacidad de identificar y analizar problemas y diseñar, desarrollar, implementar, verificar y documentar soluciones software sobre la base de un conocimiento adecuado de las teorías, modelos y técnicas actuales.

Justificación: en la resolución de los problemas de integración de submódulos y elementos de la simulación, dedique mucho tiempo a analizarlos correctamente y buscar las posibles soluciones hasta encontrar la más fiable, genérica y abierta a nuevas funcionalidades, para así desarrollar la arquitectura más robusta.

# 7. Generador sintético de datos

## 7.1 Introducción

La explicación de este proyecto realizado junto a sus funcionalidades se dividirá en 2 módulos: la simulación y el entorno de usuario que interactuará con la simulación. Cabe destacar que la simulación y el modelo de éste se ha desarrollado de tal manera que sea completamente independiente del exterior, es decir, que no tiene ningún tipo de dependencia con el servidor usado o el cliente que se ha desarrollado, las acciones del usuario o las librerías usadas. Por tanto, si en el futuro algunas de las librerías o herramientas usadas se quedaran obsoletas, no afectará en ningún modo al núcleo del proyecto, a la simulación.

En la primera parte, hablaremos del núcleo de la simulación, el modelo del proyecto donde se ejecuta internamente todos los procesos de la simulación y en el que cada uno de los agentes de la simulación realiza sus acciones pertinentes.

En la segunda parte, mostraré la interfaz gráfica del usuario, las funcionalidades que tiene y las opciones en las que el usuario puede influir en la simulación. Por lo que podemos ver, que el primer módulo tiene el Modelo del proyecto y el segundo la Vista y el Control.

Antes de empezar es importante aclarar 2 puntos importantes. Por un lado, tenemos el uso del concepto de simulables, que se usará a menudo durante esta explicación. Un simulable es un agente de la simulación, pero usamos esta palabra ya que refleja de forma más precisa el concepto, ya que éstos simulan elementos como por ejemplo un cliente, o un restaurante. Estos simulables tomarán decisiones y realizarán acciones acordes al elemento que son. Por ejemplo, un cliente consumirá los servicios de comida de un restaurante en función de su poder adquisitivo.

Por otro lado, durante la explicación, se mostrarán ilustraciones de secciones del código que ayudarán a entender lo explicado en ese momento. Estas ilustran no permiten visualizar la clase al completo, sino una sección del mismo como muestra para poder entender mejor la clase.

## 7.2 Simulación

### 7.2.1 Ciclo de vida de la simulación

Es indispensable explicar el proceso principal de la simulación, el ciclo de vida. Este proceso lo lleva a cabo los administradores de la simulación, un conjunto de clases que se explicarán en el punto 7.2.4.

El ciclo es esencialmente un bucle constante que mantiene una fecha que se incrementa un día en cada iteración. Esto tiene como objetivo crear una línea de tiempo en el cual

el ciclo o bucle es el “corazón que bombea” a modo de días que transcurren de forma sistemática en cada iteración de éste.

Dentro de cada simulación se produce cuatro procesos principales, en el siguiente orden:

- Se llama a cada uno de los agentes para que realicen la acciones que consideren oportunas (esto se explicará más adelante con detenimiento).
- Se adelanta la fecha en un día en la simulación.
- Los administradores de la simulación realizan ciertas tareas de mantenimiento y control (7.2.4)
- Se producirá una espera momentánea para que los días tengan una cierta duración de no más de 5 segundos, adaptable por el usuario en la interfaz.

En el primero, la línea de tiempo llama a cada uno de los agentes activos de la simulación. Estos agentes o simulables se encuentran guardados en una lista dentro de la línea de tiempo, que será siempre actualizada inmediatamente cada vez que se produzca el alta o la baja de algún simulable. A continuación, se puede ver la clase de la Línea de Tiempo, en la se puede visualizar la lista de los simulables, el método play() que

llama a los simulables para que realicen la acción que crean oportuna, y el método passDay() que hace transcurrir el día.

```
public class TimeLine extends EventGenerator{

    private static Speed speed = new Speed();
    private static SimulationDate date = new SimulationDate();
    private List<Simulable> simulableList;

    public static Simulable actualSimulable;

    public TimeLine(List<Simulable> simulableList) {
        this.simulableList = simulableList;
        date = new SimulationDate();
    }

    public List<Simulable> getSimulableList() {
        return simulableList;
    }


    public void play(){
        simulableList.forEach(Simulable::simulate);
        SimulableTester.changeSimulable(null);
        passDay();
    }


    private void passDay() {
        date.setDate(date.getDate()+1);
        addEvent((SimulationDate)date.clone());
    }
```

*Ilustración 8: Clase TimeLine (Línea de Tiempo)*

La línea de tiempo no sabe que simulable es realmente, ya que mediante el principio de sustitución de Liskov, la línea de tiempo no conoce las implementaciones que se encuentran detrás y para ésta son simples “Simulables”, es decir una interfaz con el método simulate(), como podemos ver en la foto que se encuentra a continuación. El resto de los métodos son de identificación del simulable que tendrán siempre todos los ellos.

```
1   package backend.model.simulables;
2
3   import backend.model.simulables.bank.EconomicAgent;
4
5   public interface Simulable extends EconomicAgent {
6       void simulate();
7       String[] getSimulable();
8       int getNIF();
9       String getName();
10
11  }
```

*Ilustración 9: Interfaz Simulable*

Los simulables serán llamados uno a uno por la línea de tiempo y realizarán la acción oportuna durante ese día. Estos simulables serán llamados por el método simulate() y en función de las circunstancias realizaran unas acciones u otras. Todos los tipos de simulables que existen y sus acciones serán explicadas durante el apartado 7.2.2.

Cuando terminen todos los elementos se terminará las acciones durante este día y se dará por finalizado este proceso.

Tras la fase de acción de los simulables, se llama al método passDay() que nombramos anteriormente, esta se encarga de actualizar la fecha que tiene la línea de tiempo, además de generar el evento de que ha pasado un día. Este tema de los eventos se explicará posteriormente en el apartado 7.2.6.

A continuación, nos encontramos con la fase de administración. En esta parte, se encarga de administrar la propia simulación y los simulables activos de esta. Debajo se encuentra el método principal de la fase de administración

```
public static void manageSimulation() {
    makeChanges();
    getSimulableController().manageSimulation();
}
```

*Ilustración 10: Método Principal de administración*

Este método como se puede observar se divide en dos partes. La primera se encarga de confirmar los cambios solicitados por los simulables y las bajas de simulables, el segundo se encarga de las posibles altas en la lista de simulables.

La necesidad de esta fase radica en el problema que trae consigo ciertos cambios durante la fase de acción de los simulable. Por ejemplo, las bajas de los propios simulables mientras la misma lista está siendo iterada por la línea de tiempo, traerá problemas de concurrencia en la ejecución de la simulación. Estos cambios de bajas de simulables se explicarán durante el apartado 7.2.3 y el resto de los cambios se explicarán durante la explicación de los simulables y los cambios que son solicitados por los mismos en el apartado 7.2.2.

En la segunda parte de la fase de administración, nos encontramos la sub-fase que controla las altas de simulables. Se encarga de comprobar si se quiere generar alguno de los simulables y si es afirmativo, se llamará al módulo concreto de creación e inicialización de simulables y actualiza la lista de simulables de la línea de tiempo. Estas altas de simulables se explicarán con más detenimiento durante el apartado 7.2.3 y la inicialización de estos en el apartado 7.2.8.

## 7.2.2 Agentes activos o simulables

En este apartado veremos todos los tipos de agentes o roles que hay dentro de la simulación, así como sus características sus acciones y su toma de decisiones. Algunos de estos agentes son eminentemente activos (realizan acciones de forma autónoma sin que nadie se lo haya solicitado), y otros son principalmente pasivos (están a la espera

de que algún otro agente solicite que realice una acción de forma directa o indirecta. Estos agentes se pueden dividir en 2 tipos principalmente, particulares o personas, y empresas. Además, se encuentra una entidad única, el Banco, que se explicará más adelante.

Empezaremos por las empresas, desde las menos importantes o con menos acciones con otros agentes, hasta los más importantes o con más acciones.

#### 7.2.2.1 Empresas

En esta sección se expone todas las empresas existentes en la simulación, los productos o servicios que ofrece y sus acciones. Estas empresas tienen algunos elementos en común:

- Datos de la empresa: Todos tienen unos datos comunes, como el nombre de la empresa, el NIF, la calle, el número de teléfono. Estos datos se generaron a través de generador de datos aleatorios, como expliqué anteriormente, que genera muchas filas de datos de empresas ficticias. Como excepción, están los restaurantes, que se obtuvieron directamente de Tripadvisor (para más detalles sobre los datos de los restaurantes, apartado 7.2.7.2).
- Administración de finanzas: todas las empresas controlan un patrimonio, unos gastos e ingresos, ya que se generan compras y ventas de forma periódica. Por lo tanto, todas las empresas tendrán un control de esto, que es sencillamente una clase que guarda todos esos datos y realizas toda administración financiera cada vez que se produzca alguna compra o venta. Todas las empresas tienen una instancia propia con datos del capital social inicial, el patrimonio neto, la tesorería, los gastos, las ventas, los beneficios, los datos del mes anterior… etc.

```java
public class FinancialData implements Cloneable {
    private double totalActive;   // total del activo
    private double totalPassive;  // total del pasivo

    private double netWorth;      // patrimonio neto
    private double treasury;      // tesoreria
    private final double socialCapital; // capital social

    private double purchases;     // compras
    private double sales;         // ventas
    private FinancialData lastMonthData;
    private Map<ComplexCompany,Double> debtsTable = new LinkedHashMap<>();
    private Map<Worker,Double> payrolls = new LinkedHashMap<>();


    public FinancialData(double socialCapital) {
        this.socialCapital = socialCapital;
        this.treasury = socialCapital;
        this.netWorth = socialCapital;
        this.totalActive = 0;
        this.totalPassive = 0;
        this.lastMonthData = null;
        reset();
    }

    public Map<ComplexCompany, Double> getDebtsTable() {
        return debtsTable;
    }

    public Map<Worker, Double> getPayrolls() {
        return payrolls;
    }

    public void addSale(double amount){
        sales += amount;
    }

    public void addPurchase(double amount){
        purchases += amount;
    }

    public void addDebt(double amount) {
        totalPassive+=amount;
    }

    public void removeDebt(double amount) {
        totalPassive-=amount;
    }
```

*Ilustración 11: Clase FinancialData (Datos financieros)*

- Clientes de otras empresas: todas las empresas son también clientes de otras que compran productos y/o servicios. Estas empresas serán pagadas mensualmente por los productos o servicios suministrados. Además, las empresas buscaran los servicios y productos que necesiten para su empresa seleccionando el mejor precio del mercado, cambiando de proveedor si es necesario.
- Pago de impuestos: para simplificar se ha considerado que cada empresa paga unos impuestos mensuales en función de sus beneficios, es decir, un porcentaje de estos que podrá ser cambiado por el usuario en todo momento.
- Pago de hipoteca: cada empresa pagará también mensualmente por el local que usa para su negocio. Este pago dependerá de factores concretos de la empresa.
- Creación y quiebra: las empresas pueden llegar a formarse de forma espontánea si las condiciones son idóneas para que la idea de negocio sea rentable. Además, estas

empresas pueden quebrar si incurren en grandes pérdidas. Esta parte se explicará con más detalle en el apartado de bajas y altas (7.2.3).

- Precio bajo demanda: las empresas suben y bajan los precios de sus productos o servicios en función de la demanda actual. Cuando sus productos o servicios están siendo muy solicitados subirán el precio, en caso contrario, lo bajarán para atraer a más clientes.
- Administrador: todas las empresas tendrán un administrador que les controlará todos los procesos de gestión internos respecto a las finanzas y a los suministradores de bienes y servicios.

```
public class Administrator {

    protected FinancialData financialData;
    protected ComplexCompany company;

    public Administrator(FinancialData financialData, ComplexCompany company) {
        this.company = company;
        this.financialData = financialData;
    }

    public void addProvider(Provider provider){
        SimulationAdministrator.addSimulableForCompany(company,provider);
    }

    public void removeProvider(Provider provider){
        SimulationAdministrator.removeSimulableForCompany(company,provider);
    }

    public void addService(ServiceCompany serviceCompany){
        SimulationAdministrator.addSimulableForCompany(company,serviceCompany);
    }

    public void removeService(ServiceCompany serviceCompany){
        SimulationAdministrator.removeSimulableForCompany(company,serviceCompany);
    }

    public List<Provider> getProvidersList() {
        return company.getProviders();
    }

    public void checkProducts() {
        if(getProvidersList().size()==0) return;
        if(ProviderSettings.isBadProduct()) refundProduct();
    }
```

*Ilustración 12: Clase Administrador*

## Empresas de servicios

Estas empresas ofrecen algún tipo de servicio a otras empresas, en concreto hay dos tipos de servicio en esta simulación, de transporte y de limpieza. La arquitectura desarrollada permite añadir más tipos de servicios incluso a particulares, pero en esta simulación centrada en la restauración, son los 2 tipos de servicios principales, complementarios a este sector. Ofrecen un servicio a cambio de un pago mensual.

## Proveedores de productos

Proveen de productos o materias primas a las empresas que las usan para crear otros productos o servicios. En este caso, al centrarse en el sector de la restauración, los productos son alimentos principales para los restaurantes como vegetales, carne, pescado, frutas y otros. Cada uno de ellos proveerá de un producto concreto.

La arquitectura permite en todo momento añadir nuevos productos que sean usados para otros motivos y empresas ya que los proveedores son genéricos y cada empresa elige cuales quiere comprar. Al igual que los servicios, estos consisten en un pago mensual a cambio del aprovisionamiento de productos. Además, para el transporte de estos productos, harán servicio de empresas de transporte, explicadas anteriormente.

## Restaurantes

Es la empresa principal de este sector y, por lo tanto, la que más interacciones tiene dentro de la simulación. Esta empresa compra todos los alimentos necesarios, por lo que recurrirá a diferentes proveedores. Además, también necesita servicios de limpieza.

Al ser un restaurante, también necesita de trabajadores que realicen distintas labores, tales como camareros, cocineros, ayudante de cocina, maître o chef. Los trabajadores no trabajan indefinidamente, ya que funcionan con contratos a término como en la realidad. Esta labor la controlan dos nuevos encargados en la administración: el empleador y el mánager de ofertas.

Al acabarse un contrato, el empleador con cierto tiempo de antelación ordena al mánager de ofertas a buscar posibles mejores opciones que el trabajador que finaliza su contrato. El día que acaba el contrato, el empleador toma una decisión, quedarse con el trabajador extendiéndole el contrato con una subida de sueldo, o contratar a otro. Esta decisión depende de ciertos factores que el empleador, al igual que la búsqueda por parte del mánager de ofertas tendrán:

- Grandes pérdidas: si el restaurante esta con muchas pérdidas económicas, el mánager buscará los trabajadores que acepten el menor sueldo. Este aspecto se explicará más detalladamente en la sección del trabajador. El empleador se quedará con el trabajador (entre las opciones del mercado) con menor sueldo de los que encontró el mánager y ordena a este último a elegir el trabajador final; si tienen el mismo sueldo, buscará el trabajador con mayor calidad (esta parte de calidad también se explicará mejor en la sección de los trabajadores). Por último, el mánager comprueba las características de los dos finalistas. Si el trabajador actual cobra menos o igual, se quedará con el actual y renovará el trabajador con un aumento de sueldo. En caso contrario, se despedirá al actual y se contratará al nuevo.
- Estado mediocre: si el estado del restaurante está dentro de un intervalo aceptable, el mánager buscará el trabajador con mejor proporción calidad/sueldo. El proceso es el mismo, pero con esta estrategia.

- Grandes beneficios: si el restaurante está con grandes beneficios, el mánager buscará el trabajador con mejor calidad. El proceso también es el mismo.

Esta toma de decisiones se lleva a cabo con el patrón "Strategy", https://refactoring.guru/design-patterns/strategy.

Los restaurantes mensualmente generan una nómina a todos sus trabajadores con el salario a retribuir, llamando a la clase banco para generar la transacción del salario reduciendo su patrimonio; esta parte se explicará con detenimiento en la sección de Otros (7.2.2.2) en la explicación del banco como agente de la simulación.

```
public abstract class Employer {

    protected AdministratorWithStaff administratorWithStaff;
    protected OfferManager manager;

    public Employer(AdministratorWithStaff administratorWithStaff) {
        this.administratorWithStaff = administratorWithStaff;
        this.manager = new OfferManager(administratorWithStaff);
    }


    public void checkExpiredSoonContracts() {
        administratorWithStaff.getWorkersWithExpiredSoonContracts().parallelStream()
                .filter(worker -> !WorkerSettings.isInRetireAge(worker))
                .forEach(worker -> manager.makeOffers(worker));

    }

    public void checkExpiredContracts() {
        administratorWithStaff.getWorkersWithExpiredContracts().forEach(this::checkExpiredContract);
    }

    protected void checkExpiredContract(Worker worker) {
        if(WorkerSettings.isInRetireAge(worker)) changeRetiredWorker(worker);
        else decideContract(worker);
        administratorWithStaff.removeWorker(worker);
        manager.removeOffers(worker);
    }

    protected void changeRetiredWorker(Worker worker) {
        administratorWithStaff.removeWorker(worker);
        checkStaff();
    }
```

*Ilustración 13: Clase Empleador*

```java
public class OfferManager {

    private Map<Worker,List<JobOffer>> workerOffers;
    private AdministratorWithStaff administratorWithStaff;

    public OfferManager(AdministratorWithStaff administratorWithStaff) {
        this.administratorWithStaff = administratorWithStaff;
        this.workerOffers = new LinkedHashMap<>();

    }

    void makeOffers(Worker current) {
        searchBetterWorkers(current).forEach(worker -> makeOffer(current,worker));
    }

    private void makeOffer(Worker current, Worker option) {
        JobOffer jobOffer = new JobOffer(administratorWithStaff.getCompany(), option, option.getSalaryDesired());
        if(!workerOffers.containsKey(current)) workerOffers.put(current, new LinkedList<>());
        workerOffers.get(current).add(jobOffer);
        jobOffer.acceptOfferRestaurant();
        option.addOffer(jobOffer);
    }

    Worker getBestOption(Worker worker){
        if (thereIsNoOffers(worker)) return worker;
        List<JobOffer> jobOffers = workerOffers.get(worker);
        JobOffer option = jobOffers.parallelStream()
                .filter(JobOffer::isAccepted)
                .filter(offer -> SimulationAdministrator.isNotAlreadyHired(offer.getWorker()))
                .filter(offer -> SimulationAdministrator.isNotAlreadyRetired(offer.getWorker()))
                .reduce(jobOffers.get(0),this::getBetterOffer);
        if (option == null) return worker;
        else return option.getWorker();
    }
```

*Ilustración 14: Clase Manager de Ofertas*

Los restaurantes aumentarán o disminuirán el precio mínimo y máximo del plato en función de los ingresos, todo ello administrado directamente por el administrador explicado anteriormente, con una extensión, ya que ahora también administran los trabajadores y sus contratos.

```java
public class AdministratorWithStaff extends Administrator {

    private List<Contract> contractList;
    private WorkerStrategy currentStrategy;

    public AdministratorWithStaff(FinancialData financialData, ComplexCompany company) {
        super(financialData,company);
        this.contractList = new CopyOnWriteArrayList<>();
        this.currentStrategy = new BestProportionScoreSalaryStrategy();
    }

    public void addWorker(Worker worker){
        addContract(worker);
        SimulationAdministrator.addSimulableForCompany(company,worker);
    }

    public void removeWorker(Worker worker){
        removeContract(worker);
        SimulationAdministrator.removeSimulableForCompany(company,worker);
    }

    private void addContract(Worker worker) {
        contractList.add(createContract(worker));
    }

    private void removeContract(Worker worker) {
        Contract contract = getContract(worker);
        if(contract!=null)contractList.remove(contract);
    }

    private Contract getContract(Worker worker) {
        return contractList.stream().filter(contract -> contract.getWorker().equals(worker)).findFirst().orElse(null);
    }
```

*Ilustración 15: Clase Administrador con trabajadores*

Los clientes de los restaurantes consumen sus servicios con una frecuencia que depende de su poder adquisitivo. La calidad de los restaurantes se mide a través de la nota media de los trabajadores. Los clientes elegirán entre los mejores restaurantes que están entre un intervalo de precios adecuado a su nivel económico. El cliente al elegir restaurante, este solicitará su reserva y dependiendo del espacio disponible se formalizará o no.

Los contratos tienen una duración variable que podrá especificar el usuario. Los restaurantes necesitan un número concreto de trabajadores para cada trabajo que depende del tamaño del restaurante, y éste se mide por el número de mesas. Cuantas más mesas se ofrece una mayor capacidad, y podrá obtenerse mayores ingresos, pero aumentará el número de nóminas de trabajadores y el pago de hipoteca. Además, el usuario puede controlar el porcentaje de ocupación del restaurante para adaptar la simulación a posibles estudios futuros que quieran tener en cuenta este aspecto, como en el caso de una pandemia.

Por último, cabe destacar que los restaurantes son restaurantes reales, ya que todos sus datos son obtenidos directamente de la página web de Tripadvisor a través de su web o “Web Scrapping”, como expliqué anteriormente.

```java
public class Restaurant extends ComplexWorkerWithStaff {
    private PriceRange priceRange;
    private int tables;
    private AtomicInteger tablesAvailable;

    public Restaurant(String companyName, String telephoneNumber, String street, PriceRange priceRange, int tables) {
        this(new RestaurantNIFCreator().create(), companyName, telephoneNumber, street, priceRange, tables);
    }

    public Restaurant(int NIF, String companyName, String telephoneNumber, String street, PriceRange priceRange, int tables) {
        super(NIF,companyName,street,telephoneNumber);
        this.priceRange = priceRange;
        this.tables = tables;
        this.tablesAvailable = new AtomicInteger((int)(tables* RestaurantSettings.EATINGS_PER_TABLE*RestaurantSettings.getCapacity()));
    }

    public double getScore(){
        return administratorWithStaff.getWorkerList().stream()
                .map(worker -> (double)worker.getQuality().getScore())
                .reduce(0.0,Double::sum)/ administratorWithStaff.getWorkerList().size();
    }

    public void collectEating(double amount){
        financialData.addSale(amount);
    }

    public double getPricePlateMean(){
        return MathUtils.twoNumberMean(this.getMinPricePlate(),this.getMaxPricePlate());
    }

    public double getMinPricePlate() {
        return priceRange.getMinPrice();
    }
```

*Ilustración 16: Clase Restaurante*

## 7.2.2.2 Personas

### Clientes

Los clientes se excluyen de la simulación en su vida diaria, y sólo se simula sus interacciones con el restaurante, como comensales. Cada uno tiene sus datos personales, como el NIF, nombre, apellido, fecha de nacimiento, trabajo, salario… etc. Este último, el salario, se genera en la propia simulación cuando se crea el cliente. El salario se genera a través de una distribución normal que se adapta a los datos actuales de salario en España.

Los clientes tienen una lista de rutinas, es decir, una lista de restaurantes que ir. El tamaño de la lista y el precio de los restaurantes dependerá del poder adquisitivo del cliente. Esto se lleva a cabo a través de una separación en grupos que, en función del salario del cliente, estará en un grupo u otro. Los clientes pueden aparecer espontáneamente durante la simulación, y también pueden fallecer cuando envejecen.

Los clientes dirigen un porcentaje de su sueldo al consumo de restaurantes, este porcentaje fue sacado de estudios al respecto para aumentar el realismo de la simulación. Este porcentaje es el presupuesto que tiene el cliente para el gasto de restauración de forma mensual.

Cada vez que llegue el día de consumo de algún restaurante de la rutina, el cliente primero comprueba si tiene presupuesto como para permitírselo, si no lo tiene, cancela la rutina.

Por último, el cliente puede traer invitados consigo y paga por todos. Este número de invitados varia aleatoriamente. El cliente y sus invitados piden una serie de platos con precio por plato variable en función del precio del restaurante, todo usando distribuciones normales que generan datos muy realistas tras muchas pruebas. Como consecuencia se generan facturas de grupos de clientes que van a comer a un restaurante. La parte de facturas se explicará más adelante en la sección de facturas (7.2.5).

```
public class Client implements Simulable, Collector{
    protected PersonalData personalData;
    protected RoutineList routineList;
    protected int peopleInvited;

    public Client(PersonalData personalData) {
        this.personalData = personalData;
    }

    public PersonalData getPersonalData() {
        return personalData;
    }

    @Override
    public int getNIF() {
        return personalData.getNIF();
    }

    public String getFirstName() {
        return personalData.getFirstName();
    }

    public String getLastName() {
        return personalData.getLastName();
    }

    @Override
    public String getName(){
        return personalData.getFirstName() + " " + personalData.getLastName();
    }

    public String getJob() {
        return personalData.getJob();
    }
```

*Ilustración 17: Clase Cliente*

## Trabajadores

Los trabajadores son extensiones de la clase Cliente, que además de ser clientes son trabajadores de restaurantes. Hacen las mismas acciones que el cliente salvo que, en este caso, irán a comer a restaurantes si tienen trabajo. Estos trabajadores también se pueden retirar cuando cumplen una edad concreta (normalmente 65 años, aunque puede ser cambiado por el usuario). Los trabajadores cuando se jubilan hay 2 opciones:

- Si están en paro en el momento de jubilación, obtendrán la pensión mínima.
- Si están trabajando, cuando se termine el contrato, el restaurante contratará directamente al mejor trabajador en el mercado según la estrategia actual, y jubilará al trabajador actual cobrando un porcentaje del salario actual como pensión, que podrá ser editado por el usuario.

Los trabajadores tienen una calidad que puede ser muy baja, baja, media, alta y muy alta. Esto afecta directamente a dos factores, a la hora de contratarlos, como hablamos anteriormente, y a la calidad del restaurante, ya que se hace una media de la calidad de todos los trabajadores.

Los trabajadores cuando están en paro tienen un salario deseado a la hora buscar trabajo. Este salario deseado baja conforme pasan los meses sin encontrar trabajo. Esta acción afecta directamente para ser contratado, porque como expliqué anteriormente, los restaurantes si hay problemas financieros, tendrán en cuenta este salario deseado a la hora de contratar personal. Dependiendo de la situación laboral:

- Mientras tiene trabajo, realizará el trabajo de forma adecuada y contribuirá al consumo de restaurantes como cualquier otro cliente.
- Mientras no tenga trabajo, esperará ofertas de los restaurantes y cuando encuentra alguna que le satisfaga en cuanto al salario deseado, aceptará. Si tiene varias, elegirá la mejor opción. Además, al no tener trabajo no consumirá ningún restaurante.

Estas ofertas funcionan de una manera muy simple. Los mánager de ofertas mandan una batería de ofertas a muchos trabajadores con el criterio de la estrategia actual. Esta oferta tiene 2 firmas, la del restaurante y la del trabajador. Cuando el trabajador acepta esta oferta, llama al método para aceptar la oferta. El mánager cuando es la hora de analizar la lista de candidatos obtiene de esa lista de ofertas sólo las que estén aceptadas por los trabajadores.

```java
public class JobOffer {
    private ComplexCompany company;
    private Worker worker;
    private double salary;
    private boolean canceled;
    private boolean acceptedByWorker;
    private boolean acceptedByRestaurant;

    public JobOffer(ComplexCompany company, Worker worker, double salary) {
        this.company = company;
        this.worker = worker;
        this.salary = salary;
        canceled = false;
        acceptedByWorker = false;
        acceptedByRestaurant = false;
    }

    public ComplexCompany getCompany() {
        return company;
    }

    public Worker getWorker() {
        return worker;
    }

    public double getSalary() {
        return salary;
    }

    public boolean isAccepted(){
        return acceptedByWorker && acceptedByRestaurant;
    }

    public void acceptOfferWorker() {
```

*Ilustración 18: Clase Oferta de Trabajo*

```java
public class Worker extends Client{
    private double salaryDesired;
    private Quality quality;
    private AtomicBoolean isWorking = new AtomicBoolean(false);
    private ComplexWorkerWithStaff company = null;
    private List<JobOffer> jobOfferList;


    public Worker(PersonalData personalData) {
        super(personalData);
        super.setJob("");
        salaryDesired = 0;
        setSalary(0);
        jobOfferList = new CopyOnWriteArrayList<>();
    }


    public ComplexCompany getCompany() {
        return company;
    }

    public Date getContractExpireDate(){
        if(company == null) return null;
        return company.getContractExpireDate(this);
    }


    public double getSalaryDesired() {
        return salaryDesired;
    }

    public void setSalaryDesired(double salaryDesired) {
        this.salaryDesired = salaryDesired;
    }
```

*Ilustración 19: Clase Trabajador*

### 7.2.2.3 Banco

Es una entidad central que controla el flujo de capital entre las distintas entidades. Cada vez que se produce algún tipo de compra o venta, esta entidad única en la simulación es la llamada para realizar la transacción, como elemento central. Todos los simulables conocen al banco y cada vez que haga falta alguna transacción en alguna compra o venta, llamarán al banco para que la realice.

```java
public class Bank extends EventGenerator implements Simulable, Event {

    public static void makeTransaction(Transaction transaction){
        transaction.makeTransaction();
    }

    @Override
    public void simulate() {
        SimulableTester.changeSimulable(this);
        if(TimeLine.isLastDay()){
            Simulation.getClientListCopy().forEach(Collector::collectSalary);
            Simulation.getCompanyListCopy().forEach(this::payMortgage);
            Simulation.getCompanyListCopy().forEach(this::payTaxes);
            addEvent(this);
        }
    }

    private void payTaxes(Company company) {
        company.pay(Company.getTaxes()*company.getFinancialData().getBenefits());
        addEvent(new TaxesPayedCompanyEvent(company));
    }


    @Override
    public int getNIF() {
        return RestaurantNIFCreator.getInitialValue()-1;
    }

    @Override
    public String getName() {
        return "Bank";
    }

    public void payMortgage(Payer payer) {
```

*Ilustración 20: Clase Banco*

Al contrario, el banco no sabe quiénes son los que solicitan la transacción. Los solicitantes están envueltos en una interfaz genérica que controla los cambios que se producen en la transacción en cada parte. Esta interfaz se llama Agente Económico (EconomicAgent) y tienes dos métodos a implementar por cada simulable, pay() y collect(). La propia interfaz simulable que vimos anteriormente, la cual todos los agentes tienen que implementar, también extiende de la interfaz EconomicAgent, por lo que todos los agentes de la simulación siempre serán agentes económicos que implementen un método de pago y de cobro.

```java
package backend.model.simulables.bank;

public interface EconomicAgent {
    void pay(double amount);
    void collect(double amount);
}
```

*Ilustración 21: Interfaz Agente Económico*

Las empresas implementan esta interfaz, incrementando o disminuyendo el patrimonio neto. Los clientes la implementan simplemente reduciendo o aumentado el presupuesto para mes actual.

En cuanto a las transacciones, son simplemente instancias que genera el simulable para emitir la factura. El simulable manda la transacción al banco para que la ejecute y así, realizar los pagos correspondientes. Estas transacciones son las encargadas de generar

la factura internamente llamando al generador de facturas que, como es habitual, se explicará en su sección correspondiente (7.2.5). Por cada tipo de factura, hay un tipo de transacción. Cuando se produce la transacción, se añade el porcentaje en impuestos en el desembolso al pagador.

```
public abstract class Transaction {
    protected EconomicAgent issuer;
    protected EconomicAgent receiver;
    protected double amount;
    protected double taxRate;

    public Transaction(EconomicAgent issuer, EconomicAgent receiver, double amount) {
        this.issuer = issuer;
        this.receiver = receiver;
        this.amount = amount;
    }


    public EconomicAgent getIssuer() {
        return issuer;
    }

    public EconomicAgent getReceiver() {
        return receiver;
    }

    public double getAmount() {
        return amount;
    }

    public void makeTransaction(){
        taxRate = generateBill();
        if(taxRate != 0)pay();
    }

    protected abstract boolean checkBill();

    protected abstract void pay();

    protected abstract double generateBill();
```

*Ilustración 22: Clase Abstracta Transacción*

El banco es también un agente activo ya que cuando es el último día del mes se encarga de hacer pagar la hipoteca del local y el pago de impuestos de todas las empresas activas actualmente en la simulación.

Por último, nos encontramos con dos interfaces que algunos de los agentes implementan la primera, y otros la segunda. Estas son las interfaces de pagadores (payers) y cobradores (collectors). Los payers son siempre empresas, que tienen que pagar mensualmente unos impuestos al Estado. Este impuesto, como explicamos anteriormente, es un porcentaje del beneficio de la empresa que puede ser cambiado por el usuario. Los collectors, por su parte, son clientes y trabajadores que mensualmente cobran el sueldo del banco, que realmente reinician el presupuesto de gasto para restaurantes. Estas juntos a las hipotecas son las tres acciones activas del banco.

```java
package backend.model.simulables.bank;

public interface Payer {
    void payMortgage();
}
```

*Ilustración 23: Clase Pagador*

```java
package backend.model.simulables.bank;

public interface Collector {
    void collectSalary();
}
```

*Ilustración 24: Clase Cobrador*

## 7.2.3 Bajas y altas de simulables

Los simulables, como explicamos anteriormente, pueden entrar y salir de la simulación. Estas entradas y salidas dependen de muchos factores. Los encargados de controlar estas entradas y salidas son los administradores de la simulación que, en cada iteración del ciclo de vida durante la fase de administración, se controla si alguien quiere salir, o si se cumplen los factores para entrar de algún simulable y posteriormente, se inicializa el simulable.

En cuanto a las salidas, las personas a partir de cierta edad (75 años), cada día de la simulación, hay una ínfima probabilidad de morir. Esta probabilidad está lo suficientemente ajustada para que las personas tengan una esperanza de vida que cumpla con el estándar de este país (84 años de media).

Las empresas, por otro lado, quebrarán si las finanzas no son favorables, es decir, si los beneficios son negativos e inferiores a una cantidad, que puede ser cambiados por el usuario.

En cuanto a las entradas, estos simulables se generan si la situación es propicia y rentable. Si se cumple los factores para alguno de los simulables, se inicializará un simulable de ese tipo y se añadirá a la simulación. Estas entradas dependen de una formula, que es una probabilidad que cuanto mayor sea, más probabilidades que se genere tal simulable. Esas probabilidades son todas proporciones que reflejen realista y lógicamente la realidad. Estas dependen de cada simulable:

- Cliente: Cuanta menor proporción de clientes por cada restaurante haya, más posibilidad de que se generen y viceversa.
- Restaurante: Es la proporción contraria al cliente, cuantos más clientes haya por cada restaurante, más probable de que se generen.
- Trabajador: cuanto menor sea el porcentaje de paro, mayor la probabilidad de que se genere un trabajador.
- Proveedor de productos: cuanto menor proporción haya de proveedores por cada restaurante, mayor la probabilidad de que se genere un proveedor.

- Empresa de servicios: cuanto menor sea la proporción de las empresas de servicios en el número de empresas, mayor la probabilidad de que se genere una empresa de servicios.

La manera en la que se generan estos simulables se explicará durante el apartado de Preparación de Simulables (7.2.8). Este mecanismo de generación de simulables es bastante realista y controlado para que el sistema sea estable y cumpla con lo visto en la realidad.

## 7.2.4 Administración

### 7.2.4.1 Control central

#### Interruptor del simulador

Es el encargado de controlar la ejecución de la simulación, la clase que actúa como interruptor de la simulación, que se llama para parar y encender la simulación. También, permite reiniciar la simulación.

```
public class SimulatorSwitcher {

    private static String uriProvider = "";
    private static String uriClient = "";


    public static void setUriProvider(String uriProvider) {
        SimulatorSwitcher.uriProvider = uriProvider;
    }


    public static void setUriClient(String uriClient) {
        SimulatorSwitcher.uriClient = uriClient;
    }

    public static void restart(){
        SimulationDataController.getRestart().set(true);
    }

    public static boolean isRunning() {
        return SimulationDataController.getExecuting().get();
    }


    public static void changeExecuting() {
        if(isRunning()) stopSimulation();
        else startSimulation();
    }

    public static void stopSimulation() {
        SimulationDataController.getExecuting().set(false);
    }

    public static void startSimulation(){
        SimulationDataController.getExecuting().set(true);
```

*Ilustración 25: Clase SimulatorSwitcher (Interruptor del Simulador)*

Cuando se llama a ejecutar por primera vez, se envía un boolean (true o false) que indica si se ejecuta en un hilo a parte o no. La ejecución se puede realizar en local usando simplemente la consola para mostrar los eventos (apartado 7.2.6), o en la web con un servidor de Tomcat. La primera opción, al ser en local y sin ningún tipo de interacción, se ejecuta sin hilo. La segunda al tener incorporado una interfaz de interacción con el usuario, se ejecuta en un hilo aparte.

```
private static void initExecution(boolean thread) {
    SimulationDataController.initSessionData();
    initSimulatorElements();
    //SimulatorTester.test();
    if(thread) executeWithThread();
    else executeLocal();
}


private static void initSimulatorElements() {
    List<Simulable> simulables = initSimulables();
    SimulationDataController.getSimulationData().initTimeLine(simulables);
}

private static List<Simulable> initSimulables() {
    return SimulationInitializerController.init();
}

private static void executeWithThread() {
    SimulatorThreadPool.executeTask(SimulatorSwitcher::executeLocal);
}


private static void executeLocal() {
    try {
        while (!SimulationDataController.getRestart().get()) {
            if (isRunning()) {
                SimulationDataController.getTimeLine().play();
                SimulationAdministrator.manageSimulation();
                delay();
            }
        }
        SimulationInitializerController.reset();
    } catch (Exception e){
```

*Ilustración 26: Inicialización de la Simulación*

La administración del hilo se lleva acabo usando un ThreadPoolExecutor de la librería Concurrent [14]. Esta te permite crear un “pool” de hilos que pueden ser usado dentro del proyecto. Este objeto permite mandarle tareas que puedan ser ejecutadas y éste las realizará en uno de sus hilos mientras haya espacio. Dependiendo de la situación, puedes ponerle un tamaño máximo de hilos o un tamaño variable.

En mi proyecto cree una clase que envolvía ese objeto y se usa como clase de acceso al ThreadPoolExecutor. El interruptor del simulador llama a esta clase mandando una tarea (Runnable Task) a la clase. También se usa para la generación de eventos (apartado 7.2.6)

```
private static void executeWithThread() {
    SimulatorThreadPool.executeTask(SimulatorSwitcher::executeLocal);
}
```

*Ilustración 27: Llamada de la clase mandando la tarea*

.

```
package backend.model.simulation.administration;


import java.util.concurrent.Executors;
import java.util.concurrent.ThreadPoolExecutor;

public class SimulatorThreadPool {

    private static ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newCachedThreadPool();


    public static void executeTask(Runnable task){
        executor.submit(task);
    }
}
```

*Ilustración 28: Clase SimulatorThreadPool*

Esto permite una arquitectura fácilmente adaptable a la ejecución de varias simulaciones simultáneas sin alterar el proyecto.

## Simulación

Esta clase es la clase principal de servicio a los simulables. Provee de todo tipo de métodos para obtener todo tipo de datos variados según las necesidades de los simulables. Por ejemplo, permite obtener la lista de los trabajadores sin empleo o con empleo, la lista de proveedores de un producto, el número de simulables de cada tipo, etc.

Estos datos tendrán acceso todos los simulables como los restaurantes a la hora de buscar trabajadores para su negocio, o empresas de servicios para la limpieza, o los clientes para buscar restaurantes al que ir a comer.

En definitiva, provee de una serie de métodos “Getters”, que los simulables podrán consumir en todo momento. Además, tendrán acceso a las estrategias genéricas para la selección de ofertas por parte de los trabajadores o la búsqueda de rutinas, por parte de los clientes.

```java
public class Simulation {

    public static final SelectOfferStrategy SEARCHER_STRATEGY = new AlwaysAcceptStrategy();
    public static final RoutineStrategy ROUTINE_STRATEGY = new BestRoutineStrategy();

    public static int getCompanySize() {
        return getCompanyList().size();
    }

    public static int getProviderSize() {
        return getProviderList().size();
    }

    public static int getServiceCompanySize() {
        return getServiceCompanyList().size();
    }

    public static int getSecondaryCompanySize(){
        return (int)getCompanyList().stream()
                .filter(company -> company instanceof SecondaryCompany)
                .count();
    }

    public static int getRestaurantSize() {
        return getRestaurantList().size();
    }

    public static int getClientSize() {
        return getClientList().size();
    }

    public static int getWorkerSize() {
        return getWorkerList().size();
    }

    public static List<Restaurant> getRestaurantList(int page) {
        int from = DatabaseManager.getFrom(page);
        int to = DatabaseManager.getTo(from,getRestaurantSize());
        return getRestaurantList().subList(from, to);
    }
```

*Ilustración 29: Clase Simulación*

## Administrador principal de la simulación

Es la interfaz principal de comunicación entre administración y los simulables en cuestión de solicitar cambios por parte de estos últimos y el encargado de llamar a ejecutar la fase de administración en el ciclo de vida, el que dirige todos los pasos de esta fase. Los simulables tienen una gran variedad de acciones posibles, pero muchos de estas acciones pueden causar problemas de concurrencia como por ejemplo la salida de estos en medio de la iteración de la lista de la fase de acción de los simulables o como cambios en el personal o de empresas clientes en una empresa, como ya se ha comentado anteriormente.

Por lo tanto, todos estos cambios de personal, de relaciones de empresas o de salida y entrada de simulables, se solicitan durante la fase de acción de los simulables y se confirman los cambios (“commit changes”) durante la fase de administración del ciclo de vida. Cuando un agente quiere realizar un cambio que afecte a la concurrencia, lo solicita directamente al administrador principal de la simulación. Esta manda al

**Controlador de Simulables** para que añade esta solicitud y que durante la fase de administración este controlador realice este cambio. Esta parte se explicará durante la administración de simulables (apartado 7.2.4.4) donde se explicará las funciones del Controlador de Simulables y otros controladores.

```
public class SimulationAdministrator {

    public static void manageSimulation() {
        getSimulableController().manageSimulation();
    }

    private static SimulableController getSimulableController(){
        return SimulationDataController.getSimulationData().getSimulableController();
    }

    public static boolean isNotAlreadyHired(Worker worker) {
        return getSimulableController().isNotAlreadyHired(worker);
    }

    public static boolean isNotAlreadyRetired(Worker worker) {
        return getSimulableController().isNotAlreadyRetired(worker);
    }

    public static void retire(Worker worker) {
        getSimulableController().retire(worker);
    }

    public static void addSimulableForCompany(ComplexCompany company, Simulable simulable) {
        getSimulableController().addSimulableForCompany(company,simulable);
    }

    public static void removeSimulableForCompany(ComplexCompany company, Simulable simulable) {
        getSimulableController().removeSimulableForCompany(company,simulable);
    }

    public static void makeChanges() {
        getSimulableController().makeChanges();
    }

    public static void isGoingToDie(Client client){
        getSimulableController().isGoingToDie(client);
    }

    public static void isGoingToClose(Company company){
        getSimulableController().isGoingToClose(company);
    }
}
```

*Ilustración 30: Clase SimulationAdministrator (Administrador de la Simulación)*

## 7.2.4.2 Control de datos

### Datos de simulación

Es la clase encargada de guardar todos los datos de simulación en ejecución: los simulables activos, la línea de tiempo, los datos de ejecución del interruptor, los datos de ajustes (apartado 7.2.4.5), el controlador de simulables (apartado 7.2.4.4) y la lista con los simulables que se están siguiendo. Esta última se explicará durante el entorno de usuario ya que es una funcionalidad del usuario.

El resto de las clases de administración como la clase Simulation o la clase SimulationAdministrator usan directamente estos datos a través del intermediario SimulationDataController (Controlador de datos de la simulación), que se explicará en el siguiente subapartado.

## Controlador de datos de la simulación

Esta clase es la clase de acceso directo a los datos de Simulación, esta clase tiene como objetivo dividir bien la estructura para poder cambiar fácilmente la simulación con el propósito de ejecutar múltiples simulaciones simultaneas, en el caso de necesitarlo en un futuro, y esta clase sea la encargada de coordinar a cuál de los datos de las simulaciones acceder en todo momento. A pesar de ello, esta clase sigue siendo muy útil para tratar el SimulationData como una instancia de los datos de la simulación y el SimulationDataController sea la clase estática de fácil acceso a los datos sin tener que instanciar ninguna clase.

```
public class SimulationDataController {

    private static SimulationData simulationData;

    public static void initSessionData() {
        simulationData = new SimulationData();
    }

    public static SimulationData getSimulationData(){
        return simulationData;
    }

    public static TimeLine getTimeLine() {
        return getSimulationData().getTimeLine();
    }

    public static GeneralData getGeneralData() {
        return getGeneralSessionData().getGeneralData();
    }

    public static ClientData getClientData() {
        return getClientSessionData().getClientData();
    }

    public static RestaurantData getRestaurantData() {
        return getRestaurantSessionData().getRestaurantData();
    }

    public static ProviderData getProviderData() {
        return getProviderSessionData().getProviderData();
    }

    public static ServiceData getServiceData() {
        return getServiceSessionData().getServiceData();
    }

    public static WorkerData getWorkerData(){
        return getWorkerSessionData().getWorkerData();
    }
```

*Ilustración 31: Clase SimulationDataController*

## Datos de facturas de la simulación

Contiene la lista de todas las facturas generadas durante la simulación actual. Permite añadir facturas que se hayan generado durante la simulación y reiniciar la lista al reinicio de la simulación.

```java
public class SimulationBillData {
    private List<XMLBill> billList;

    public SimulationBillData() {
        reset();
    }


    public void addBill(XMLBill bill){
        billList.add(bill);
    }

    public List<XMLBill> getBillList(int from, int to) {
        return billList.subList(from,to);
    }

    public void reset(){
        billList = new LinkedList<>();
    }

    public int getSize(){
        return billList.size();
    }
}
```

*Ilustración 32: Clase SimulationBillData (Datos de Facturas de la Simulación)*

## Controlador de facturas de la simulación

Al igual que el controlador de datos de simulación, gestiona el acceso a los datos, pero en este caso a los datos de las facturas generadas. Tiene el mismo objetivo que en el caso del controlador de datos, usar la clase que guarda los datos como una instancia que se genera en cada simulación y facilita el acceso a estos. Además, permite como en el caso anterior generar una arquitectura con simulaciones simultáneas.

```java
public class SimulationBillAdministrator {

    private static SimulationBillData simulationBillData = new SimulationBillData();
    private static final TableAdministrator administrator = new SQLiteTableAdministrator();

    public static void addBill(XMLBill bill){
        simulationBillData.addBill(bill);
    }

    public static void resetBills(){
        simulationBillData.reset();
        try {
            if(getBillCount() !=0)
                administrator.deleteAll(XMLBill.class);
        } catch (SQLException | ClassNotFoundException e) {
            System.out.println("Database is locked, could not delete Bills of last Simulation");
        }
        BillNIFCreator.initInitialValue();
        DatabaseManager.setBillInitialPrimaryKeyValue();
    }

    public static List<XMLBill> getBillPage(int page) {
        int from = DatabaseManager.getFrom(page);
        int to = DatabaseManager.getTo(from, simulationBillData.getSize());
        return simulationBillData.getBillList(from, to);
    }
```

*Ilustración 33: Clase SimulationBillAdministrator (Controlador de Facturas de la Simulación)*

### Controlador de seguidos de la simulación

Esta clase se encarga de controlar los simulables que sigue el usuario. Este concepto de seguir simulables se explicará durante la explicación del entorno de usuario. Además, se encarga de añadir unos simulables aleatorios al inicio de la simulación.

```
public class SimulationFollowAdministrator {

    public static void followSimulable(Simulable simulable){
        if(!SimulationDataController.getSimulationData().getFollowedSimulables().contains(simulable))
            SimulationDataController.getSimulationData().getFollowedSimulables().add(simulable);
    }

    public static void unfollowSimulable(Simulable simulable){
        SimulationDataController.getSimulationData().getFollowedSimulables().remove(simulable);
    }

    public static List<Simulable> getFollowedSimulables(){
        return SimulationDataController.getSimulationData().getFollowedSimulables();
    }

    public static void followRandomOptions() {
        followSimulable(Simulation.getCompanyListCopy().get(MathUtils.random(0,Simulation.getCompanySize())));
        followSimulable(Simulation.getClientListCopy().get(MathUtils.random(0,Simulation.getClientSize())));
    }
}
```

*Ilustración 34: Clase SimulationFollowAdministrator (Controlador de Seguidos de la Simulación)*

## 7.2.4.3 Inicialización de simulables

### Inicializador de simulables

Es el encargado de inicializar los simulables de la aplicación. Este solo controla la inicialización, pasándole el número de simulables de un tipo que inicializar, para luego realizar las funciones necesarias para devolver los simulables requeridos. Permite tanto crear una lista como uno sólo.

Para crear los simulables, primero tiene que acceder a estos que se encuentran guardados en la base de datos, para acceder a ellos usará el paquete desarrollado personalmente para el acceso a la base datos. Para ello usa la clase TableAdministrator que envuelve perfectamente toda la implementación de lectura de datos de las tablas que se encuentran en la base de datos. Esta sección se explicará con más detenimiento durante el apartado 7.2.7 (Capa de Datos).

Además de crearlos, es el encargado de organizar la preparación de los simulables para su entrada a la simulación. Dependiendo del simulable a preparar llamará a unos preparadores u otros que se encargan de darle los elementos necesarios al simulable

para que puede empezar a participar en la simulación. Esta parte de la preparación se explicará con más detalle durante el apartado de Preparación de Simulables (7.2.8).

Durante la simulación, se irá avanzando en la lista de los simulables de la base datos, que se van leyendo a la vez que van desapareciendo otros cuando se producen bajas. Cuando se llega al límite de datos de la base de datos, vuelve automáticamente a empezar desde el principio de la tabla de la base de datos.

```
public class SimulationInitializer {

    private static final TableAdministrator administrator = new SQLiteTableAdministrator();

    public static List<Provider> getProviders(int providerCount) throws SQLException, ClassNotFoundException {
        List<Provider> providerList = administrator.read(Simulation.getSecondaryCompanySize(),providerCount,Provider.class);
        ProductThread.initProducts(providerList);
        return providerList;
    }

    public static List<ServiceCompany> getServiceCompanies(int serviceCompanyCount) throws SQLException, ClassNotFoundException {
        List<ServiceCompany> companies = administrator.read(Simulation.getSecondaryCompanySize(),serviceCompanyCount,ServiceCompany.class);
        ServiceThread.initProducts(companies);
        return companies;
    }

    public static List<Restaurant> getRestaurants(int restaurantCount) throws SQLException, ClassNotFoundException {
        //return RestaurantThread.loadRestaurantsPage(restaurantCount/30);
        return administrator.read(Simulation.getRestaurantSize(),restaurantCount,Restaurant.class);
    }

    public static List<Client> getClients(int clientCount) throws SQLException, ClassNotFoundException {
        List<Client> clientList = administrator.read(Simulation.getClientSize(),clientCount,Client.class);
        clientList.forEach(client -> client.setSalary(ClientSettings.getSalarySample()));
        return clientList;
    }


    public static List<Worker> getWorkers(int workerCount) throws SQLException, ClassNotFoundException {
        List<Worker> workerList = administrator.read(Simulation.getClientSize(),workerCount,Worker.class);
        if(workerCount==1) WorkerThread.setJob(workerList.get(0),new MostEmployedJobSelector());
        else WorkerThread.setJobs(workerList);
        return workerList;
    }
```

*Ilustración 35: Clase SimulationInitializer (Inicializador de Simulables)*

## Controlador de inicialización de simulables

Es el encargado de inicializar los simulables del principio de la simulación, es decir los agentes que se generan al inicio de forma automática con el que poder empezar a simular correctamente. Para ello usará la clase **Inicializador de simulables** explicada en el apartado anterior. Además, se encarga de reiniciar los **Datos de la simulación** (SimulationData) y **los Datos de facturas de la simulación** (SimulationBillData) cuando se solicite reiniciar la simulación por parte del usuario.

```
public class SimulationInitializerController {
    public static List<Simulable> init(){
        initSimulables();
        SimulationFollowAdministrator.followRandomOptions();
        return SimulationInitializer.init();
    }

    private static void initSimulables(){
        try {
            int serviceCount = GeneralSettings.getServiceCount();
            SimulationDataController.getSimulationData().getCompanyList().addAll(SimulationInitializer.getServiceCompanies(serviceCount));
            int providerCount = GeneralSettings.getProviderCount();
            SimulationDataController.getSimulationData().getCompanyList().addAll(SimulationInitializer.getProviders(providerCount));
            int restaurantCount = GeneralSettings.getRestaurantCount();
            SimulationDataController.getSimulationData().getCompanyList().addAll(SimulationInitializer.getRestaurants(restaurantCount));
            int clientCount = GeneralSettings.getClientCount();
            SimulationDataController.getSimulationData().getClientList().addAll(SimulationInitializer.getClients(clientCount));
            int workerCount = GeneralSettings.getWorkerCount();
            SimulationDataController.getSimulationData().getClientList().addAll(SimulationInitializer.getWorkers(workerCount));
        } catch (SQLException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }


    public static void reset(){
        SimulationDataController.getSimulationData().reset();
        SimulationBillAdministrator.resetBills();
    }
}
```

*Ilustración 36: Clase SimulationInitializerController (Controlador de Inicialización de Simulables)*

## 7.2.4.4 Administración de simulables

### Committer de la simulación

Es el encargado de perpetrar (confirmar y realizar) los cambios solicitados por los simulables. Esta clase es usada directamente por el resto de los administradores de simulables, ya que cada uno tiene su propia instancia de esta clase para requerir sus servicios cada vez que sea necesario, para realizar un “commit” o confirmación de algún cambio.

```
public class SimulationCommitter extends EventGenerator {


    public void commitAddProvider(ComplexCompany company, Provider provider){
        company.addProvider(provider);
        provider.addClient(company);
        addEvent(new NewProviderCompanyEvent(company,provider));
        addEvent(new ClientAddedProviderEvent(provider,company));
    }

    public void commitRemoveProvider(ComplexCompany company, Provider provider){
        if(company.getProviders().contains(provider)){
            provider.removeClient(company);
            commitRemoveProviderClient(company, provider);
        }
    }

    private void commitRemoveProviderClient(ComplexCompany company, Provider provider) {
        company.removeProvider(provider);
        addEvent(new RemovedProviderCompanyEvent(company,provider));
        addEvent(new ClientRemovedProviderEvent(provider,company));
    }

    public void commitAddWorker(ComplexWorkerWithStaff company, Worker worker){
        company.addWorker(worker);
        addEvent(new WorkerHiredCompanyEvent(company,worker));
        addEvent(new HiredWorkerEvent(worker,company));
    }

    public void commitRemoveWorker(ComplexWorkerWithStaff company, Worker worker){
        company.removeWorker(worker);
        addEvent(new WorkerFiredCompanyEvent(company,worker));
        addEvent(new FiredWorkerEvent(worker,company));
    }
```

*Ilustración 37: Clase SimulationCommitter (Committer de la Simulación)*

## Controlador de Entrada/Salida de la simulación

Es administrador principal de las entradas y salidas de la simulación. Para ello, como explicamos anteriormente, hará uso de un **Committer de la Simulación**, para perpetrar estas entradas y salidas.

En cada ciclo de la Simulación, durante la fase de administración, este administrador será llamado por el **Controlador de Simulables** (que se explicará a continuación). Este se encargará de añadir, si es propicio y por cada tipo de simulable, un nuevo simulable de este tipo, como ya explicamos anteriormente durante el apartado de Bajas y Altas de simulables (apartado 7.2.3). Por cada simulable existe una probabilidad que varía según unos factores que afectan directamente al agente (para más información, apartado

7.2.3). Si esta probabilidad se cumple, entonces este controlador añadirá un nuevo agente de este tipo.

Además, durante la realización de los cambios solicitados por los simulables (que se explicará en la explicación del siguiente y último administrador), los cambios solicitados de Salida serán realizados por este, mediante la ayuda del **Committer de la Simulación**.

```
public class SimulationIOController extends EventGenerator {

    private List<Simulable> simulableList;
    private SimulationCommitter committer;

    public SimulationIOController(List<Simulable> simulableList, SimulationCommitter committer) {
        this.simulableList = simulableList;
        this.committer = committer;
    }

    public void manageSimulablesToAdd() {
        checkThereIsNewPerson();
        checkThereIsNewCompany();
    }

    private void checkThereIsNewPerson() {
        checkThereIsNewWorker();
        checkThereIsNewClient();
    }

    private void checkThereIsNewWorker() {
        if(!WorkerSettings.newWorker()) return;
        Simulable simulable = addWorker();
        if(simulable == null)return;
        addEvent(new NewClientEvent((Client)simulable));
        addSimulable(simulable);
    }

    private void checkThereIsNewClient() {
        if(!ClientSettings.newClient()) return;
        Simulable simulable = addClient();
        if(simulable == null)return;
        addEvent(new NewClientEvent((Client)simulable));
        addSimulable(simulable);
    }
```

*Ilustración 38: Clase Simulation I/O Controller (Controlador de Entrada/Salida de la Simulación)*

## Controlador de simulables

Es el director principal de la fase de administración, es llamado por el administrador principal de la simulación para organizar y realizar los cambios solicitados por los simulables y realizar las altas necesitadas.

Como explicamos en la sección del **Administrador Principal de la Simulación**, cada vez que un simulable quiere realizar un cambio, este lo solicita al administrador principal, para el cual manda al controlador de simulables a registrar la solicitud. Esta solicitud se registra mediante una serie de listas que administra y guarda el **Controlador de Simulables** mantener y actualizar los distintos tipos de cambios solicitados durante la

fase de acción de los simulables para posteriormente llevarlos a cabo en la fase de simulación, como la lista de empresas que solicitan quebrar en la fecha actual, la lista de fallecimientos de personas, la de jubilaciones, o la de cambios internos de alguna empresa, ya sea cambios en el personal, productos o servicios.

```
private Map<ComplexCompany,List<Simulable>> companiesAddingChanges;
private Map<ComplexCompany,List<Simulable>> companiesRemovingChanges;
private List<Worker> retiredWorkers;
private List<Client> deadClientList;
private List<Company> closedCompanyList;
```

*Ilustración 39: Listas de cambios solicitados*

En cada ciclo de simulación, este controlador se encarga de realizar 2 acciones: realizar los cambios solicitados por los simulables, y llamar al **Controlador de Entrada/Salida** para añadir los simulables necesarios.

Durante la primera acción lee cada uno de los cambios de las distintas listas y realiza los cambios mediante el uso del **Committer de la Simulación**. En caso de las solicitudes de salida, llama al **Controlador de Entrada/Salida** para realizar la salida. Tras realizar todos los cambios, vacía todas las listas de solicitudes de cambios.

```
public class SimulableController {

    private Map<ComplexCompany,List<Simulable>> companiesAddingChanges;
    private Map<ComplexCompany,List<Simulable>> companiesRemovingChanges;
    private List<Worker> retiredWorkers;
    private List<Client> deadClientList;
    private List<Company> closedCompanyList;
    private SimulationCommitter committer;
    private SimulationIOController simulationIOController;

    public SimulableController(SimulationCommitter committer) {
        this.committer = committer;
        companiesAddingChanges = new ConcurrentHashMap<>();
        companiesRemovingChanges = new ConcurrentHashMap<>();
        retiredWorkers = new CopyOnWriteArrayList<>();
        deadClientList = new CopyOnWriteArrayList<>();
        closedCompanyList = new CopyOnWriteArrayList<>();
    }

    public void initIO(SimulationIOController simulationIOController){
        this.simulationIOController = simulationIOController;
    }

    public void manageSimulation() {
        makeChanges();
        simulationIOController.manageSimulablesToAdd();
    }

    public void retire(Worker worker) {
        retiredWorkers.add(worker);
    }

    public void addSimulableForCompany(ComplexCompany company, Simulable simulable) {
        if(!companiesAddingChanges.containsKey(company)) companiesAddingChanges.put(company,new CopyOnWriteArrayList<>());
```

*Ilustración 40: Clase SimulableController (Controlador de Simulables)*

### 7.2.4.5 Ajustes

Todos los datos que afectan directamente a la simulación, ya que son accedidos durante la simulación, han sido implementado mediante una serie de ajustes con un valor por defecto, que puede ser cambiados por el usuario en el apartado de ajustes (que se explicará durante en el apartado del Entorno de Usuario de la Simulación, 7.3). Hay varios tipos de ajustes:

- Ajustes Generales: Controla el número de simulables iniciales de cada tipo con los que empezará la simulación.

```
public class GeneralSettings {


    private static GeneralData getGeneralDataSettings() {
        return SimulationDataController.getGeneralData();
    }

    public static int getClientCount() {
        return getGeneralDataSettings().getClientCount();
    }

    public static int getRestaurantCount() {
        return getGeneralDataSettings().getRestaurantCount();
    }

    public static int getProviderCount() {
        return getGeneralDataSettings().getProviderCount();
    }

    public static int getWorkerCount() {
        return getGeneralDataSettings().getWorkerCount();
    }

    public static int getServiceCount() {
        return getGeneralDataSettings().getServiceCount();
    }
}
```

*Ilustración 41: Clase GeneralSettings (Ajustes Generales)*

- Ajustes de Cliente: los ajustes que afectan directamente a los agentes que tengan el papel de clientes de la simulación, ya sea para obtener el sueldo que tendrá, la edad a partir de la cual, puede fallecer...

```
public class ClientSettings{

    public static final double PERCENTAGE_FOR_RESTAURANT = 0.148;
    public static final int DEATH_AGE = 75;
    private static double clientProbability = 1.0;


    private static ClientData getClientDataSettings() {
        return SimulationDataController.getClientData();
    }


    public static int getSalaryGroup(double salary) {
        List<Integer> salaryOptionList = new ArrayList<>(getClientDataSettings().getRestaurantGroup().keySet());
        if(salaryOptionList.size()==1)return salaryOptionList.get(0);
        return salaryOptionList.stream()
                .filter(salaryAuxOption -> salary<=salaryAuxOption)
                .findFirst().orElse(salaryOptionList.get(salaryOptionList.size() - 1));
    }

    public static Restaurant[] getRestaurantOptions(int salaryOption){
        int price = getPrices(salaryOption);
        return Simulation.getRestaurantListCopy().stream()
                .filter(Objects::nonNull)
                .filter(restaurant -> restaurant.getMaxPricePlate() <= price)
                .toArray(Restaurant[]::new);
    }

    private static int getPrices(int salaryOption) {
        return getClientDataSettings().getRestaurantGroup().get(salaryOption);
    }

    public static double getSalarySample() {
        double salary = Math.max(getClientDataSettings().getSalaryDistribution().sample(), getClientDataSettings().getMinSalary());
```

*Ilustración 42: Clase ClientSettings (Ajustes del Cliente)*

- Ajustes de Restaurantes: influyen directamente en el funcionamiento de los restaurantes, ya sea especificando el salario inicial de los trabajadores al principio de la simulación, o el número de mesas de los restaurantes… etc.

```
public class RestaurantSettings{

    private static final int ALPHA = 2;
    private static final int BETA = 4;
    private static final BetaDistribution numTablesDistribution = new BetaDistribution(ALPHA, BETA);
    private static final int MIN_TABLES = 4;
    private static final int MAX_TABLES = 50;
    private static final int WORKERS_MIN = 1;
    private static final Map<Job, Integer> lengthWorkerTable = new HashMap<>();
    public static final int EATINGS_PER_TABLE = 6;
    public static final double FINANCIAL_DIFFERENCE_PERCENTAGE = 1.25;
    private static double restaurantProbability = 1.0;

    static {
        getNumberOfWorkers();
    }

    private static void getNumberOfWorkers(){
        Integer[] lengthPerTenTable = {3,2,1,-1,-1};
        IntStream.range(0,lengthPerTenTable.length).boxed()
                .forEach(i -> lengthWorkerTable.put(Job.values()[i],lengthPerTenTable[i]));

    }

    private static RestaurantData getRestaurantDataSettings() {
        return SimulationDataController.getRestaurantData();
    }

    public static double getSalaryPerQuality(Worker worker) {
        return (worker.getQuality().getScore()/worker.getSalaryDesired())*10000;
    }

    public static double getInitialSocialCapital() {
        return getRestaurantDataSettings().getInitialSocialCapital();
```

*Ilustración 43: Clase RestaurantSettings (Ajustes de Restaurantes)*

- Ajustes de Proveedores: ajustes para los datos importantes de los proveedores, como el precio inicial de los productos, o la probabilidad de que el producto salga defectuoso… etc.

```
public class ProviderSettings{

    private static double providerProbability = 1.0;

    private static ProviderData getProviderDataSettings() {
        return SimulationDataController.getProviderData();
    }

    public static double getInitialSocialCapital() {
        return getProviderDataSettings().getInitialSocialCapital();
    }

    public static double getProductCost(Product product){
        return getProviderDataSettings().getProductSalePriceTable().get(product);
    }

    public static boolean isBadProduct(){
        return MathUtils.random(0,1000)<2;
    }

    public static boolean newProvider() {
        return MathUtils.calculateProbability((int)((100-getProviderRestaurantPercentage())*getProviderProbability()));
    }

    private static double getProviderRestaurantPercentage(){
        return ((double)Simulation.getProviderSize()/(double)(1+Simulation.getRestaurantSize()+Simulation.getProviderSize()))*100.0;
    }

    public static double getCloseLimit() {
        return getProviderDataSettings().getCloseLimit();
    }
```

*Ilustración 44: Clase ProviderSettings (Ajustes de Proveedores)*

- Ajustes de Servicios: son los ajustes relacionados con los servicios y las empresas de servicios, por ejemplo, el precio inicial de los servicios o el capital inicial de estas empresas. Este último dato lo tienen todas las empresas.

```
public class ServiceSettings {

    private static double serviceProbability = 1.0;

    private static ServiceData getServiceDataSettings() {
        return SimulationDataController.getServiceData();
    }

    public static double getInitialSocialCapital() {
        return getServiceDataSettings().getInitialSocialCapital();
    }

    public static double getPrice(Service service){
        return getServiceDataSettings().getServicePriceTable().get(service);
    }

    public static double getPriceChange() {
        return getServiceDataSettings().getPriceChange();
    }

    public static double getCloseLimit(){
        return getServiceDataSettings().getCloseLimit();
    }

    public static boolean newService() {
        return MathUtils.calculateProbability((int)((100 -  getServiceCompanyPercentage())*getServiceProbability()));
    }

    private static double getServiceCompanyPercentage() {
        return ((double)(Simulation.getServiceCompanySize()*5)/(double)(1+Simulation.getCompanyListCopy().size()))*100.0;
    }

    public static void setServiceProbability(double serviceProbability) {
        ServiceSettings.serviceProbability = serviceProbability;
```

*Ilustración 45: Clase ServiceSettings (Ajustes de Servicios)*

- Ajustes de Trabajadores: ajustes que afectan a los trabajadores de empresas. Por ejemplo, la edad de jubilación, o el porcentaje del sueldo que obtienen como pensión al jubilarse.

```
public class WorkerSettings{

    private static double workerProbability = 1.0;

    public static WorkerData getWorkerDataSettings(){
        return SimulationDataController.getWorkerData();
    }

    public static double reduceSalaryDesired(double salaryDesired) {
        return Math.max(getWorkerDataSettings().getMinSalary(),salaryDesired-WorkerSettings.getWorkerDataSettings().getSalaryDesiredChange(
    }

    public static boolean isInRetireAge(Worker worker) {
        return worker.getAge() >= WorkerSettings.getWorkerDataSettings().getRetireAge();
    }

    public static double getSalaryChange(){
        return getWorkerDataSettings().getSalaryChange();
    }

    public static double getPercentageRetirement() {
        return getWorkerDataSettings().getPercentageRetirement();
    }

    public static boolean newWorker() {
        return MathUtils.calculateProbability((int)((100- getUnemployedWorkersPercentage())*getWorkerProbability()));
    }

    public static double getUnemployedWorkersPercentage(){
        return ((double)Simulation.getUnemployedWorkers().size()/(double)(1+Simulation.getWorkerSize()))*100.0;
    }

    public static double getUnemployedWorkersPerJob(Job job){
        return ((double)Simulation.getUnemployedWorkers(job).size()/(double)(1+Simulation.getWorkerList(job).size()))*100.0;
```

*Ilustración 46: Clase WorkerSettings (Ajustes de Trabajadores)*

- Ajustes de Facturas: ajustes de las facturas que se generan en la simulación. Estos ajustes son muy poco editables por el usuario, ya que solo pueden ser editados ajustes afectados por ajustes de clientes en el caso de las facturas de comidas, como el número de platos que se pedirán a través de una distribución normal. El resto son ajustes de las facturas, los cuales no benefician su edición por parte del usuario, porque son datos muy complejos como el cálculo del precio de las comidas, o los conceptos que se pondrán en las facturas que es indiferente para el usuario.

```
public class BillSettings{

    private static final Map<String,String> conceptsTable = new HashMap<>();


    static {
        conceptsTable.put(EatingBill.class.getSimpleName(),"Bill of a eating.");
        conceptsTable.put(ProductPurchase.class.getSimpleName(),"Purchase of a product for the restaurant.");
        conceptsTable.put(ProductRefund.class.getSimpleName(),"Refund of a product in bad conditions.");
        conceptsTable.put(Payroll.class.getSimpleName(),"Payroll o a worker.");
        conceptsTable.put(ServiceBill.class.getSimpleName(),"Service for company.");
        conceptsTable.put(BuildingInversion.class.getSimpleName(),"Mortgage for company.");
    }

    private static ClientData getClientDataSettings() {
        return SimulationDataController.getClientData();
    }



    public static String getConcept(String billType){
        return conceptsTable.get(billType);
    }

    public static double calculatePrice(Restaurant restaurant, int peopleInvited){
        double mean = MathUtils.twoNumberMean(restaurant.getMinPricePlate(),restaurant.getMaxPricePlate());
        return getPriceOfPlate(restaurant, mean) * getPlateNumberSample() * (peopleInvited+1);

    }

    public static double getPlateNumberMean(){
        return getNormalDistribution().getMean();
    }
```

*Ilustración 47: Clase BillSettings (Ajustes de Facturas)*

Estos ajustes tienen un objeto que tiene todos los datos que serán usados para poder proporcionar los métodos de la clase que necesitarán durante la simulación. Las clases de datos de ajustes son las siguientes:

- Datos Generales
- Datos de Clientes
- Datos de Restaurantes
- Datos de Proveedores
- Datos de Servicios
- Datos de Trabajadores

```
public class ClientData {

    private DistributionData salary;
    private double minSalary;
    private Map<Integer,Integer> restaurantGroup;
    private MinMaxData invitedPeople;
    private MinMaxData numOfRestaurant;
    private DistributionData plateNumber;

    public ClientData(DistributionData salary, double minSalary, Map<Integer, Integer> restaurantGroup, MinMaxData invitedPeople, MinMaxData numOfRestaurant, DistributionData
        this.salary = salary;
        this.minSalary = minSalary;
        this.restaurantGroup = restaurantGroup;
        this.invitedPeople = invitedPeople;
        this.numOfRestaurant = numOfRestaurant;
        this.plateNumber = plateNumber;
    }
```

*Ilustración 48: Ejemplo de Clase de datos de Ajustes, Clase ClientData (Datos de Cliente)*

Como habíamos dicho anteriormente, las facturas no tienen muchas personalizaciones, y las disponibles provienen realmente de los datos de clientes, por eso no hay datos de facturas para los ajustes. Realmente, las clases de ajustes no tienen acceso a estos datos, sino a una clase intermedia que controla los cambios de usuario en los ajustes. Estas clases envuelven la clase de datos correspondiente con una interfaz que permite actualizar los datos fácilmente con otra nueva instancia de la clase que guarda los datos, además de proporcionar la opción de ponerlos  por defecto.

```
public class ClientDataSettings extends AdjustableSettingsData {

    private ClientData clientData;

    public ClientDataSettings() {
        super();
    }

    @Override
    public void init(Object data) {
        if(data instanceof ClientData)clientData = (ClientData) data;
    }

    @Override
    public void setDefault() {
        init(defaultSettings.getDefaultClientData());
    }

    public ClientData getClientData() {
        return clientData;
    }
}
```

*Ilustración 49: Ejemplo de clase con actualización de datos de ajuste, Clase ClientDataSettings (ajustes de datos de cliente)*

Estas clases de actualización de ajustes comparten una instancia estática de DefaultSettings (Ajustes por Defecto), es decir, que es única la instancia para todos, y que tiene todos los ajustes por defecto de la simulación. Esta clase al principio de la simulación accede a los datos por defecto que se encuentran guardados en la base de datos, esto se explicará con más detalle durante el apartado de la base de datos en la sección de la capa de datos (7.2.7.1).

```
public class DefaultSettings {

    private TableAdministrator administrator;
    private GeneralData defaultGeneralData;
    private ClientData defaultClientData;
    private RestaurantData defaultRestaurantData;
    private ProviderData defaultProviderData;
    private ServiceData defaultServiceData;
    private WorkerData defaultWorkerData;

    public DefaultSettings() {
        administrator = new SQLiteTableAdministrator();
        init();
    }

    public void init(){
        try {
            initGeneralData();
            initClientData();
            initRestaurantData();
            initProviderData();
            initServiceData();
            initWorkerData();
        } catch (SQLException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }

    public void initGeneralData() throws SQLException, ClassNotFoundException {
        defaultGeneralData = (GeneralData) administrator.read(0,1,GeneralData.class).get(0);
    }

    private void initClientData() throws SQLException, ClassNotFoundException {
        defaultClientData = (ClientData) administrator.read(0,1,ClientData.class).get(0);
        defaultClientData.setRestaurantGroup(initRestaurantGroups());
    }
```

*Ilustración 50: Clase DefaultSettings (Ajustes por defecto)*

Los cambios de ajustes los administra el SettingsBuilder (Constructor de Ajustes). Esta clase provee de una interfaz sencilla para actualizar los ajustes o ponerlos por defecto. La clase recibirá, en el caso de actualización los datos de los distintos ajustes, y se encargará de actualizar cada una de las clases que envuelven y permiten actualizar los datos de estos ajustes.

```
public class SettingsBuilder {

    public static void setDefault() {
        SimulationDataController.getGeneralSessionData().setDefault();
        SimulationDataController.getClientSessionData().setDefault();
        SimulationDataController.getRestaurantSessionData().setDefault();
        SimulationDataController.getProviderSessionData().setDefault();
        SimulationDataController.getServiceSessionData().setDefault();
        SimulationDataController.getWorkerSessionData().setDefault();
    }

    public static void build(GeneralData generalData, ClientData clientData, RestaurantData restaurantData,
                             ProviderData providerData, ServiceData serviceData, WorkerData workerData) {
        SimulationDataController.getGeneralSessionData().init(generalData);
        SimulationDataController.getClientSessionData().init(clientData);
        SimulationDataController.getRestaurantSessionData().init(restaurantData);
        SimulationDataController.getProviderSessionData().init(providerData);
        SimulationDataController.getServiceSessionData().init(serviceData);
        SimulationDataController.getWorkerSessionData().init(workerData);
    }


    public static GeneralData getGeneralData() {
        return SimulationDataController.getGeneralData();
    }


    public static ClientData getClientData() {
        return SimulationDataController.getClientData();
    }

    public static RestaurantData getRestaurantData() {
        return SimulationDataController.getRestaurantData();
    }

    public static ProviderData getProviderData() {
        return SimulationDataController.getProviderData();
```

*Ilustración 51: Clase SettingsBuilder (Constructor de Ajustes)*

## 7.2.5 Facturas

Uno de los objetivos principales de la simulación es la generación de facturas, para luego poder ser publicadas en el Datahub del otro módulo principal, y a su vez, el entorno de usuario pueda usarlas para mostrarlo las funcionalidades del proyecto. Todas las facturas generan eventos cuando se crean. Este apartado de eventos se explicará durante la sección 7.2.6.

Las facturas las genera un generador de estas, desarrollado internamente. Todas las facturas proceden de una misma clase (CFDIBill) que tiene todos los campos genéricos de las facturas, en el que cada tipo de esta rellena los campos acordes a las reglas establecidas, estos datos proceden de elementos genéricos que se reciben al generar la factura. Por ejemplo, las facturas de comidas que se generan cuando un cliente consume un restaurante, se generan a partir de tres datos: el restaurante que se consume, el cliente que lo consume, y el coste final de la comida. A partir de estos datos, la clase sabe cómo rellenar los distintos campos de la factura.

```java
public abstract class CFDIBill implements Event {
    protected int UUID;
    protected String street;
    protected Type type;
    protected Use use;
    protected String issuerName;
    protected int issuerRFC;
    protected String receiverName;
    protected int receiverRFC;
    protected double total;
    protected double taxRate;
    protected double subtotal;
    protected String currency;
    protected String concept;
    protected String date;

    public CFDIBill(int UUID,String street, Type type, Use use, String issuerName, int issuerRFC, String receiverName, int receiverRFC, do
        this.UUID = UUID;
        this.street = street;
        this.type = type;
        this.use = use;
        this.issuerName = issuerName;
        this.issuerRFC = issuerRFC;
        this.receiverName = receiverName;
        this.receiverRFC = receiverRFC;
        this.subtotal = subtotal;
        this.taxRate = (double)MathUtils.random(1,25)/100;
        this.total = (1+ this.taxRate)*subtotal;
        this.currency = "euro";
        this.concept = concept;
        this.date = TimeLine.getDate().toString();
    }
```

*Ilustración 52: Clase CFDIBill (Factura CFDI)*

Las facturas siguen el mismo formato que usan las facturas de CFDI reales [15]. Los campos de las facturas que se generan en esta simulación son los siguientes:

- UUID
- Localización
- Tipo
- Uso
- Nombre del emisor
- RFC del emisor
- Nombre del receptor
- RFC del receptor
- Subtotal
- Tasa de impuestos
- Total
- Moneda
- Concepto
- Fecha de Emisión

Al ser las facturas genéricas, el generador de facturas no conoce que tipo de factura está generando, ya que obtiene una CFDIBill con los campos ya rellenados, que luego esta simplemente usa para generar la factura en formato XML. Las facturas XML se guardan en la carpeta XMLFiles. Dentro de esta, hay una carpeta con el nombre cada tipo, y cada factura se guarda en la carpeta de su tipo.

```
public class CFDIBillGenerator extends EventGenerator implements BillGenerator {
    private static String uri= "./out/artifacts/RestaurantSimulator_war_exploded/xmlFiles/";
    private Document document;
    private CFDIBill bill;

    public static void setUri(String uri) {
        CFDIBillGenerator.uri = uri;
    }

    public void generateBill(CFDIBill bill){
        this.bill = bill;
        try {
            createBill();
            addEvent(bill);
        } catch (ParserConfigurationException | TransformerException e) {
            e.printStackTrace();
        }

    }

    private void createBill() throws ParserConfigurationException, TransformerException {
        getXMLDocument();
        appendData();
        saveXMLInFile();
        addBill();

    }

    private void getXMLDocument() throws ParserConfigurationException {
        DocumentBuilder documentBuilder = DocumentBuilderFactory.newInstance().newDocumentBuilder();
        document = documentBuilder.newDocument();
    }
```

*Ilustración 53: Clase CFDIBillGenerator (Generador de Facturas CFDI)*

Los distintos tipos de facturas que se generan son las siguientes:

- Facturas de comidas: son facturas de compras que generan los clientes al consumir comida en los restaurantes.
- Facturas de Productos: son facturas de productos que se compran a proveedores
- Devolución de Productos: son facturas que se generan cuando se produce la devolución del algún producto.
- Nómina: son facturas que se generan cuando se paga el sueldo a los trabajadores mensualmente.
- Servicio: facturas de servicios ofrecidos a un particular o empresa.
- Inversión de edificio o hipoteca: inversiones que se producen al pagar por un local que usar para realizar el negocio.

## 7.2.6 Eventos

Los **Eventos** son mensajes que se generan cuando se producen ciertas acciones dentro de la simulación que son provocados por uno o más simulables de la simulación. Estos mensajes se muestran al usuario como modo de informar o reportar al usuario de los acontecimientos que han ocurrido en la simulación. Esta es la manera de informar que he usado para este proyecto ya que es un proceso eminentemente cerrado e interno que usa este método como comunicación con el exterior, quedando fuera debido a falta de relevancia, la interfaz gráfica que muestre a estos elementos. El segundo módulo será el encargado de usar esos eventos para mostrarlos al usuario. Tiene un método para obtener el mensaje y otro para saber si ese evento está relacionado con algún simulable seguido. Esto último se explicará con más detalle durante el apartado de simulables seguidos dentro del apartado del Entorno de Usuario (7.3).

```
public interface Event {

    String getMessage();
    boolean isFollowed(List<Simulable> simulableList);

}
```

*Ilustración 54: Interfaz Evento (Event)*

Los eventos son de tres tipos:

- Eventos de Día: este evento se produce diariamente como su nombre indica. Indica el cambio de día en la simulación. Es el único evento que no involucra ningún simulable. El date de la simulación implementa
- Eventos de Factura o pagos: son eventos que traen consigo facturas de algún tipo o pagos que no generen factura como impuestos de empresas. Las facturas son de facto, eventos. Cada vez que se generan, se añaden también como eventos.
- Eventos de cambios de simulables: son acciones que realizan simulables que provocan algún cambio en las relaciones entre ellos. Por ejemplo, el despido o retiro de algún trabajador, la muerte de un cliente o el cierre de una empresa…. Etc.

Los eventos de simulables son eventos que no son implementadas por ningún elemento de la simulación, sino que son implementadas en una clase genérica de eventos más sofisticada, es una clase abstracta que trabaja con un tipo de dato genérico que es siempre un simulable. Cada implementación de esta funciona de forma genérica, es decir, todos los eventos que extienden del **Evento Genérico** tienen un simulable genérico e implementa el método isFollowed (apartado 7.3). A continuación, podemos ver la clase GenericEvent (Evento Genérico) para más detalles.

```java
public abstract class GenericEvent<Simulable> implements Event{

    protected Simulable simulable;

    public GenericEvent(Simulable simulable) {
        this.simulable = simulable;
    }

    public Simulable getSimulable() {
        return simulable;
    }

    @Override
    public boolean isFollowed(List<backend.model.simulables.Simulable> simulableList){
        return simulableList.stream()
                .anyMatch(this::isInBill);
    }

    private boolean isInBill(backend.model.simulables.Simulable simulableFollowed) {
        if(simulable instanceof backend.model.simulables.Simulable){
            backend.model.simulables.Simulable auxSimulable = (backend.model.simulables.Simulable) simulable;
            return auxSimulable.getNIF()==simulableFollowed.getNIF();
        }
        return false;
    }
}
```

*Ilustración 55: Clase GenericEvent (Evento Genérico)*

Estos eventos solo pueden generados por clases que hereden de la clase EventGenerator (Generador de Eventos). Las clases que pueden generar eventos son las siguientes:

- TimeLine (Línea de Tiempo): para generar los eventos de Día.
- Banco: para generar eventos de pagos de impuestos (eventos de pagos).
- SimulationIOController (Controlador de Entrada/Salida): para generar eventos de entrada de simulables.
- SimulationCommitter (Committer de la Simulación): para poder generar eventos de cambios de simulables y solicitudes de salida.
- CFDIBillGenerator (Generador de Facturas CFDI): cada vez que se genere una factura, añadirá como evento la propia factura (eventos de facturas).

```java
public abstract class EventGenerator {

    protected void addEvent(Event event) {
        EventController.addEvent(event);
    }

}
```

*Ilustración 56: Clase EventGenerator (Generador de Eventos)*

Los eventos que se generan son administrados por el **Controlador de Eventos**. Los generadores de eventos cuando añaden un evento, este lo manda directamente al controlador. Este contiene una lista de todos los eventos que se han producido. Cuando se añade, también es mostrado en la consola para poder ver los resultados cuando se ejecuta localmente con el **Interruptor del Simulador** (apartado 7.2.4.1). Además, esta lista de eventos se guarda en un fichero de datos. Este fichero solo guarda los eventos de la última simulación.

Por último, esta lista es solicitada por el websocket de eventos (puerto de eventos) para ser mostrados en el entorno, esto como es habitual, se explicará con detalle durante el apartado del Entorno de Usuario (7.3). El controlador de eventos solo enviará al puerto los eventos que seguían relevantes para el usuario (7.3). Tras ser proporcionados al websocket, este vacía la lista, para así tener solo los eventos pendientes por mostrar. La adición de estos eventos a lista que obtendrán en el websocket se realiza en el Threadpool de la simulación. Para más detalles sobre este Threadpool, consultar apartado 7.2.4.1 en el Interruptor del Simulador.

```
public class EventController {

    private static List<Event> eventList = new CopyOnWriteArrayList<>();

    public static void addEvent(Event event){
        if(isTheFirstDay()) return;
        System.out.println(event.getMessage());
        if(event.isFollowed(SimulationFollowAdministrator.getFollowedSimulables())) SimulatorThreadPool.executeTask(() -> addToWeb(event));
    }


    public static void addToWeb(Event event) {
        try {
            eventList.add(event);
        } catch (ConcurrentModificationException | IndexOutOfBoundsException e){
            SimulatorSwitcher.waitForOtherElements();
            addToWeb(event);
        }
    }

    private static boolean isTheFirstDay() {
        return TimeLine.isSameDate(new Date());
    }

    public static List<Event> getEvents(){
        List<Event> eventList = EventController.eventList;
        EventController.eventList = new CopyOnWriteArrayList<>();
        return eventList;
    }
}
```

*Ilustración 57: Clase EventController (Controlador de Eventos)*

## 7.2.7 Capa de datos

Durante esta sección, se explicará la organización de la base de datos el acceso a esta. El sistema de base de datos que uso en este proyecto es SQLITE Browser (10). La API de acceso a la base de datos es JDBC que permite la ejecución de operaciones sobre bases de datos desde el lenguaje de programación Java, independientemente del sistema operativo donde se ejecute o de la base de datos a la cual se accede, utilizando el lenguaje SQL del modelo de base de datos que se utilice [16].

A partir de esta API, he desarrollado una implementación propia que envuelve el uso de esta API mediante un conjunto de clases para leer, crear, actualizar y eliminar tablas y filas de forma automática y genérica para cualquier tabla. El motivo es la disminución de dependencias externas y la motivación de desarrollar un sistema completamente desde cero de forma autónoma.

### 7.2.7.1 Base de datos de la simulación

La base de datos está compuesta por 10 tablas, cada uno son accedidas para obtener datos importantes para el desarrollo de la simulación. Las tablas son las siguientes:

- Person (Persona): Es la tabla de la se extraen los clientes y trabajadores. Tiene 100.000 filas. A continuación, se pueden visualizar los campos.

| Person | | CREATE TABLE Person ( NIF integer P |
|---|---|---|
| NIF | integer | "NIF" integer |
| firstName | text | "firstName" text NOT NULL |
| lastName | text | "lastName" text NOT NULL |
| birthDate | text | "birthDate" text NOT NULL |
| gender | text | "gender" text NOT NULL |
| job | text | "job" text NOT NULL |
| country | text | "country" text NOT NULL |
| telephoneNumber | text | "telephoneNumber" text NOT NULL |
| email | text | "email" text NOT NULL |
| cardNumber | text | "cardNumber" text NOT NULL |

*Ilustración 58: Tabla Person*

- Restaurant (Restaurante): Tabla con los datos de todos los restaurantes de Gran Canaria extraidos de Tripadvisor.

| Restaurant | | CREATE TABLE Restaurant ( NIF integ |
|---|---|---|
| NIF | integer | "NIF" integer |
| name | text | "name" text NOT NULL UNIQUE |
| telephoneNumber | text | "telephoneNumber" text NOT NULL |
| street | text | "street" text NOT NULL |
| minPrice | integer | "minPrice" integer NOT NULL |
| maxPrice | integer | "maxPrice" integer NOT NULL |
| numberTables | integer | "numberTables" integer NOT NULL |

*Ilustración 59: Tabla Restaurant*

- Provider (Proveedor): De esta tabla se extraen los datos, tanto de los proveedores de products, como los proveedores de servicios.

| Provider | | CREATE TABLE Provider ( NIF integer P |
|---|---|---|
| NIF | integer | "NIF" integer |
| companyName | text | "companyName" text NOT NULL |
| creationDate | text | "creationDate" text NOT NULL |
| ownerName | text | "ownerName" text NOT NULL |
| street | text | "street" text NOT NULL |
| telephoneNumber | text | "telephoneNumber" text NOT NULL |

*Ilustración 60: Tabla Provider*

- Bill (Factura): En esta tabla se pueden guardar las facturas generadas si es necesario, ya que ya están generadas en archivos XML en el ordenador, y guardar miles de facturas por segundo en la base de datos puede provocar un cuello de botella a la hora de leer datos para añadir un nuevo simulable, ya que la conexión es única.

| Bill | | CREATE TABLE Bill ( UUID integer PRI |
|---|---|---|
| UUID | integer | "UUID" integer |
| street | text | "street" text NOT NULL |
| type | text | "type" text NOT NULL |
| use | text | "use" text NOT NULL |
| issuerName | text | "issuerName" text NOT NULL |
| issuerRFC | integer | "issuerRFC" integer NOT NULL |
| receiverName | text | "receiverName" text NOT NULL |
| receiverRFC | integer | "receiverRFC" integer NOT NULL |
| total | real | "total" real NOT NULL |
| taxRate | real | "taxRate" real NOT NULL |
| subtotal | real | "subtotal" real NOT NULL |
| currency | text | "currency" text NOT NULL |
| concept | text | "concept" text NOT NULL |
| date | text | "date" text NOT NULL |
| filePath | text | "filePath" text NOT NULL |
| fileName | text | "fileName" text NOT NULL UNIQUE |

*Ilustración 61: Tabla Bill*

- GeneralData (Datos Generales): Datos por defecto los ajustes generales (7.2.4.5). Tiene una sola fila con los datos por defecto.

| GeneralData | | CREATE TABLE GeneralData ( ID integ |
|---|---|---|
| ID | integer | "ID" integer |
| clientCount | integer | "clientCount" integer NOT NULL |
| restaurantCount | integer | "restaurantCount" integer NOT NULL |
| providerCount | integer | "providerCount" integer NOT NULL |
| serviceCount | integer | "serviceCount" integer NOT NULL |
| workerCount | integer | "workerCount" integer NOT NULL |

*Ilustración 62: Tabla GeneralData*

- ClientData (Datos de Cliente): Datos predeterminados para los ajustes de clientes (7.2.4.5). Tiene una sola fila con los datos por defecto.

| ClientData | | CREATE TABLE ClientData ( ID integer PRIM |
|---|---|---|
| ID | integer | "ID" integer |
| salaryMean | real | "salaryMean" real NOT NULL |
| salarySD | real | "salarySD" real NOT NULL |
| minSalary | real | "minSalary" real NOT NULL |
| invitedPeopleMin | integer | "invitedPeopleMin" integer NOT NULL |
| invitedPeopleMax | integer | "invitedPeopleMax" integer NOT NULL |
| numOfRestaurantMin | integer | "numOfRestaurantMin" integer NOT NULL |
| numOfRestaurantMax | integer | "numOfRestaurantMax" integer NOT NULL |
| plateNumberMean | real | "plateNumberMean" real NOT NULL |
| plateNumberSD | real | "plateNumberSD" real NOT NULL |

*Ilustración 63: Tabla ClientData*

- RestaurantData (Datos de Restaurante): Datos por defecto de los ajustes de restaurantes (7.2.4.5). Tiene una sola fila con los datos por defecto.

| RestaurantData | | CREATE TABLE RestaurantData ( ID intege |
|---|---|---|
| ID | integer | "ID" integer |
| initialSocialCapital | real | "initialSocialCapital" real NOT NULL |
| lengthContractMin | integer | "lengthContractMin" integer NOT NULL |
| lengthContractMax | integer | "lengthContractMax" integer NOT NULL |
| priceChange | real | "priceChange" real NOT NULL |
| capacity | real | "capacity" real NOT NULL |
| closeLimit | real | "closeLimit" real NOT NULL |

*Ilustración 64: tabla RestaurantData*

- ProviderData (Datos de Proveedor): En esta tabla se guardan los datos por defecto de los proveedores (7.2.4.5). Tiene una sola fila con los datos por defecto.

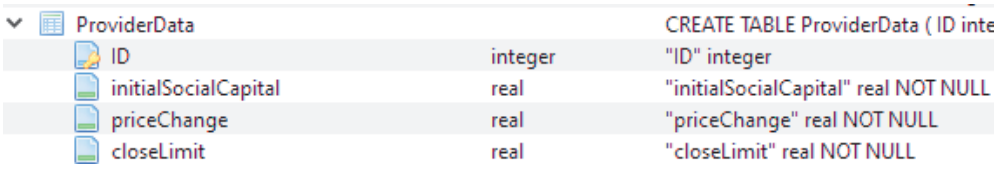



*Ilustración 65: ProviderData*

- ServiceData (Datos de Servicio): Datos por defecto para los servicios y sus empresas que lo ofrecen (7.2.4.5). Tiene una sola fila con los datos por defecto.

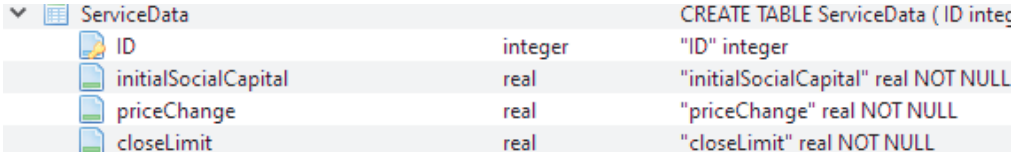



*Ilustración 66: Tabla ServiceData*

- WorkerData (Datos de Trabajador): Datos predeterminados de los trabajadores (7.2.4.5). Tiene una sola fila con los datos por defecto.

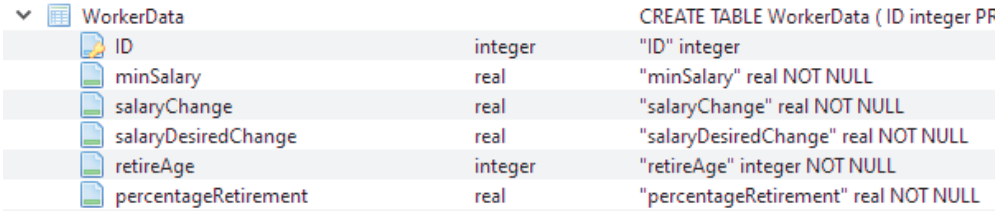



*Ilustración 67: Tabla WorkerData*

## 7.2.7.2 Paquete de acceso a datos

Esta implementación propia me permita acceder a cualquier tabla de la base de datos con una misma implementación. El funcionamiento de este paquete radica en ocultar la implementación de la API y con ello, la necesidad de mandar instrucciones SQL directamente, gracias a una interfaz genérica.

## Elementos básicos de bases de datos

Para poder llevar esta implementación a cabo, y que la interfaz sea genérica y oculte los detalles de la API, es necesario crear ciertos elementos propios de las bases de datos que sepa administrar el paquete de la implementación. Los elementos son los siguientes:

- DataType o tipo de Dato: es una clase con los tipos de datos que se usan en esta simulación de SQLITE, estos son integer (entero), real y text (texto).
- Restricción: esta clase especifica el tipo de restricción que tendrá el dato que referencie. Los usados en esta simulación son Primary Key (Clave Primaria), not null (no nulo) y not null unique (no nulo único).
- Field o campo: es un campo de la tabla. Está compuesto por un tipo de dato y una restricción.
- Header o cabecera: es la cabecera de la tabla, con los todos los datos importantes de la tabla. Contiene el nombre de la tabla, un mapa con clave el nombre del campo y valor el propio campo y el valor inicial de la clave primaria.

```
public class Header {
    private String name;
    private Map<String, Field> parameters;
    private String primaryKeyName;
    private int initialPrimaryKeyValue;

    public Header(String name, Map<String, Field> parameters) {
        this.name = name;
        this.parameters = parameters;
        primaryKeyName = getPrimaryKey();
    }

    private String getPrimaryKey() {
        return parameters.keySet().stream()
                .filter(fieldName -> parameters.get(fieldName).getRestriction().equals(Restriction.PRIMARY_KEY))
                .findFirst()
                .orElseGet(this::searchPrimaryKey);
    }

    private String searchPrimaryKey() {
        return parameters.keySet().stream()
                .findFirst()
                .map(this::addPrimaryKey)
                .orElse("");
    }

    private String addPrimaryKey(String fieldName) {
        parameters.get(fieldName).setRestriction(Restriction.PRIMARY_KEY);
        return fieldName;
    }

    public String getName() {
        return name;
    }

    public String getPrimaryKeyName() {
        return primaryKeyName;
    }
```

*Ilustración 68: Clase Header (Cabecera)*

- Selector: los selectores sirven para buscar datos en la base de datos, en este paquete se proporcionan 3 tipos, de igualdad, “like” y order by. El primer tipo sirve para buscar filas que tengan el campo proporcionado con valor igual al valor proporcionado (edad = 25), el segundo es similar, pero sirve para buscar filas que contengan ese valor (nombre like “Al”), y el último sirve para ordenar las filas finales, ascendente o descendente. Se puede usar tantos como se quieran.

```
public class EqualSelector implements Selector {

    private String fieldName;
    private String value;

    public EqualSelector(String fieldName, String value) {
        this.fieldName = fieldName;
        this.value = value;
    }

    @Override
    public String getInstruction() {
        return " " + fieldName + " = " + value;
    }

    @Override
    public String getFieldName() {
        return fieldName;
    }

    @Override
    public Object getValue() {
        return value;
    }
}
```

*Ilustración 69: Clase equalSelector (Selector de igualdad)*

- Fila: es una fila de una tabla, una lista con los datos pertenecientes a una fila de una tabla.

```
public class Row {
    private List<Object> parameters;

    public Row(List<Object> parameters) {
        this.parameters = parameters;
    }

    public List<Object> getParameters() {
        return parameters;
    }

}
```

*Ilustración 70: Clase Row (Fila)*

## Controladores

Son los intermediarios directos entre la base de datos y la simulación. Permiten la lectura, creación, eliminación, actualización e inserción en tablas. Los controladores son los siguientes:

- Conector: conector de la base de datos para poder interactuar con este, este conector es único y no se pueden crear varios. Todos los demás controladores los usan.

```
public class SQLiteDatabaseConnector extends DatabaseConnector {

    public static final String SQLITE_CLASS = "org.sqlite.JDBC";

    @Override
    public Connection connect() throws ClassNotFoundException, SQLException {
        if (connection == null){
            Class.forName(SQLITE_CLASS);
            connection = DriverManager.getConnection(url);
        }
        return connection;
    }

    @Override
    public void disconnect() throws SQLException {
        connection.close();
        connection = null;
    }
}
```

*Ilustración 71: Clase SQliteDatabaseConnector (Conector de la base de datos)*

- Creador de tablas: permite crear y destruir tablas en la base de datos de la simulación. Se proporciona una header o cabecera para poder crearla o destruirla.

```java
public class SQLiteTableCreator extends DatabaseController implements TableCreator {

    public void createTable(Header header) throws SQLException, ClassNotFoundException {
        createTable(header.getName(),header.getFields());
    }

    @Override
    public void createTable(String headerName, Map<String, Field> parameters) throws SQLException, ClassNotFoundException {
        init(headerName);
        Statement statement = getStatement();
        statement.execute(createTable(parameters));
        statement.close();
    }

    private String createTable(Map<String, Field> parameters) {
        String result = getParameters(parameters);
        result = result.substring(0,result.length()-2);
        System.out.println("CREATE TABLE IF NOT EXISTS "+ actualHeaderName +" (\n" + result + ");");
        return "CREATE TABLE IF NOT EXISTS "+ actualHeaderName +" (\n" + result + ");";
    }

    private String getParameters(Map<String, Field> parameters) {
        return parameters.keySet().stream()
                .map(parameterName -> getParameter(parameters.get(parameterName),parameterName))
                .reduce(String::concat).orElse("");
    }


    private String getParameter(Field field, String parameterName){
        return " " + parameterName + " " + field.getDataType() + " " + field.getRestriction() + ",\n";
    }

    @Override
    public void dropTable(String headerName) throws SQLException, ClassNotFoundException {
        init(headerName);
        Statement statement = getStatement();
        statement.execute(dropTable());
        statement.close();
```

*Ilustración 72: Clase SQLiteTableCreator (Creador de tablas)*

- Insertador de Filas: permite insertar filas en las tablas. Se proporciona en nombre de la tabla y la fila usando el elemento explicado.

```
public class SQLiteTableInsert extends DatabaseController implements TableInsert {


    @Override
    public void insert(String headerName, Row parameters) throws SQLException, ClassNotFoundException {
        if (notExist(headerName)) return;
        init(headerName);
        PreparedStatement preparedStatement = getPreparedStatement(prepareHeader());
        insertValues(parameters.getParameters(),preparedStatement);
        preparedStatement.execute();
        preparedStatement.close();
    }

    private String prepareHeader(){
        final String[] firstSection = {"insert into " + actualHeaderName +" ("};
        final String[] secondSection = {") values ("};
        concatParameters(firstSection, secondSection);
        firstSection[0] = firstSection[0].substring(0, firstSection[0].length()-2);
        secondSection[0] = secondSection[0].substring(0, secondSection[0].length()-2);
        return firstSection[0] +secondSection[0] +")";
    }


    private void concatParameters(String[] firstSection,String[] secondSection){
        Header header = getActualHeader();
        header.getFields().keySet().forEach(parameter -> {
            firstSection[0] += concatParameter(parameter);
            secondSection[0] += concatParameter("?");
        });
    }


    private String concatParameter(String parameter) {
        return parameter + ", ";
    }
```

*Ilustración 73: Clase SQLiteTableInsert (Insertador de filas)*

- Eliminador de Filas: permite eliminar filas de una tabla. En este caso, al solo necesitarse el delete all (eliminar todas), solo fue implementado este método, pero la estructura permite añadir nuevas funcionalidades al paquete fácilmente. Se proporciona el nombre de la tabla.

```
public class SQLiteRowDeleter extends DatabaseController implements RowDeleter {

    private String headerName;

    @Override
    public void deleteAll(String headerName) throws SQLException, ClassNotFoundException {
        if (notExist(headerName)) return;
        init(headerName);
        PreparedStatement preparedStatement = getPreparedStatement(getSentence());
        preparedStatement.execute();
        preparedStatement.close();
    }

    private String getSentence() {
        return "Delete from " + actualHeaderName + ";";
    }
}
```

*Ilustración 74: Clase SQLiteRowDeleter (Eliminador de Filas)*

- Actualizador de Filas: permite actualizar campos de filas de una de las tablas. Se proporciona el nombre de la tabla, unos selectores de igualdad para saber los campos que hay que actualizar y una lista de selector para proporcionar las condiciones de actualización.

```
public class SQLiteRowUpdater extends DatabaseController implements RowUpdater {

    public void updateRow(String headerName, EqualSelector equalSelector, Selector... selectors) throws SQLException, ClassNotFoundException {
        updateRow(headerName,new EqualSelector[]{equalSelector},selectors);
    }

    @Override
    public void updateRow(String headerName, EqualSelector[] equalSelectors, Selector... selectors) throws SQLException, ClassNotFoundException {
        if (notExist(headerName)) return;
        init(headerName);
        PreparedStatement preparedStatement = getPreparedStatement(getQuery(equalSelectors,selectors));
        prepareUpdate(equalSelectors, preparedStatement);
        preparedStatement.executeUpdate();
    }

    public void prepareUpdate(EqualSelector[] equalSelectors, PreparedStatement preparedStatement) {
        IntStream.range(0,equalSelectors.length).boxed()
                .forEach(pos -> {
                    try {
                        insertValue(equalSelectors[pos].getValue(),preparedStatement,pos);
                    } catch (SQLException e) {
                        e.printStackTrace();
                    }
                });
    }

    private String getQuery(EqualSelector[] equalSelectors, Selector... selectors) {
        String query = "update " + actualHeaderName + " set";
        query += getFieldToChange(equalSelectors);
        return checkQuery(query + " where"+ getAndSelectors(selectors));
    }
}
```

*Ilustración 75: Clase SQLiteRowUpdater (Actualizador de Filas)*

- Selector de filas: permite seleccionar filas y obtener los datos en la simulación para su posterior uso. Se proporciona el nombre de la tabla y un rango de valores (inicio y final) con respecto a los valores de la clave primaria.

```
public class SQLiteTableSelector extends DatabaseController implements TableSelector {
    private int fromID;
    private int toID;

    @Override
    public List<Row> read(String headerName, int page) throws SQLException, ClassNotFoundException {
        init(headerName);
        int fromID = getActualHeader().getInitialPrimaryKeyValue()+(page-1)* DatabaseManager.getPageLength();
        return read(headerName,fromID,fromID+ DatabaseManager.getPageLength());
    }

    @Override
    public List<Row> read(String headerName, int page, Selector...selectors) throws SQLException, ClassNotFoundException {
        init(headerName);
        int fromID = getActualHeader().getInitialPrimaryKeyValue()+(page-1)* DatabaseManager.getPageLength();
        return read(headerName,fromID,fromID+ DatabaseManager.getPageLength(),selectors);
    }

    @Override
    public List<Row> read(String headerName,int fromID, int toID) throws SQLException, ClassNotFoundException {
        if (initReader(headerName, fromID, toID)) return new LinkedList<>();
        ResultSet resultSet = getResultSet(getSelect());
        return getRows(resultSet);
    }

    public boolean initReader(String headerName, int fromID, int toID) throws ClassNotFoundException, SQLException {
        if (notExist(headerName)) return true;
        this.fromID = fromID;
        this.toID = toID - 1;
        init(headerName);
        return false;
    }

    @Override
    public List<Row> read(String headerName, int fromID, int toID, Selector ...selectors) throws SQLException, ClassNotFoundException {
        if (notExist(headerName)) return new LinkedList<>();
        this.fromID = fromID;
        this.toID = toID-1;
```

*Ilustración 76: Clase SQLiteTableSelector (Selector de filas)*

## Builders

Para la selección e inserción de filas (Entrada/Salida) se utiliza un traductor que las traduce a objetos de la simulación y viceversa. Hay un traductor o constructor por cada objeto que se quiere guardar en la simulación, es decir, uno para clientes, restaurantes, proveedores… etc. Estos constructores se llaman builders. A continuación, tenemos la clase genérica de un builder y un ejemplo de su implementación.

```java
public abstract class Builder<T> {
    public List<T> buildList(List<Row> rows) {
        return rows.stream()
                .map(this::build)
                .collect(Collectors.toCollection(LinkedList::new));
    }

    public T build(Row row){
        return getItem(row.getParameters().toArray());
    }

    public  Row buildRow(T object){
        return new Row(getRow(object));
    }

    protected abstract List<Object> getRow(T object);

    public abstract String getHeader();

    protected abstract T getItem(Object[] parameters);
}
```

*Ilustración 77: Clase Builder genérica*

```java
public class ClientBuilder extends Builder<Client> {

    @Override
    public String getHeader() {
        return "Person";
    }

    @Override
    protected List<Object> getRow(Client client){
        return Arrays.asList(new Object[]{client.getNIF()
                ,client.getFirstName(),client.getLastName(),client.getBirthDate()
                ,client.getGender(),client.getJob(),client.getCountry()
                ,client.getTelephoneNumber(),client.getEmail(),client.getCardNumber()});
    }


    @Override
    protected Client getItem(Object[] parameters) {
        return new Client(new PersonalData((int)parameters[0]
                ,(String) parameters[1],(String) parameters[2],(String)parameters[3]
                ,(String) parameters[4],(String) parameters[5],(String) parameters[6]
                ,(String) parameters[7],(String) parameters[8],(String) parameters[9]));

    }
}
```

*Ilustración 78: Ejemplo de builder, builder de cliente*

Además, hay un controlador de builders que tiene un mapa de builder para que cuando se quiera obtener o insertar datos de alguna entidad de la simulación (cliente, restaurante…), el controlador te proporcione el builder adecuado sin conocer que implementación es, utilizando únicamente los métodos genéricos para traducción de fila a objeto y viceversa. A partir de la Class (clase) de la entidad (Client, Provider, Restaurant…), se obtiene el builder.

```
public class BuilderController {

    private static final Map<Class,Builder> builderTable = new HashMap<>();

    static {
        builderTable.put(XMLBill.class, new BillBuilder());
        builderTable.put(Client.class, new ClientBuilder());
        builderTable.put(Restaurant.class, new RestaurantBuilder());
        builderTable.put(Provider.class, new ProviderBuilder());
        builderTable.put(ServiceCompany.class, new ServiceCompanyBuilder());
        builderTable.put(Worker.class, new WorkerBuilder());
        builderTable.put(GeneralData.class, new GeneralDataBuilder());
        builderTable.put(ClientData.class, new ClientDataBuilder());
        builderTable.put(RestaurantData.class, new RestaurantDataBuilder());
        builderTable.put(ProviderData.class, new ProviderDataBuilder());
        builderTable.put(ServiceData.class, new ServiceDataBuilder());
        builderTable.put(WorkerData.class, new WorkerDataBuilder());
    }

    public static Builder getBuilder(Class object){
        return builderTable.get(object);
    }
}
```

*Ilustración 79: Clase BuilderController (Controlador de Builders)*

## Manager de la base de datos

Guarda las cabeceras de todas las tablas que hay en la simulación para mejorar el rendimiento y la facilidad de uso del paquete. Además, proporciona métodos de paginación para el acceso a los datos.

```
public class DatabaseManager {

    public static final int LIST_LIMIT = 1000;
    private static List<Header> headers = new LinkedList<>();
    private static final int PAGE_LENGTH = 30;

    static {
        createRestaurantTable();
        createProviderTable();
        createPersonTable();
        createBillTable();
        createGeneralDataTable();
        createClientDataTable();
        createRestaurantDataTable();
        createProviderDataTable();
        createServiceDataTable();
        createWorkerDataTable();
    }

    private static void createRestaurantTable() {
        Map<String, Field> parameters = new LinkedHashMap<>();
        parameters.put("NIF",new Field(Restriction.PRIMARY_KEY, DataType.integer));
        parameters.put("name",new Field(Restriction.NOT_NULL_UNIQUE, DataType.text));
        parameters.put("telephoneNumber",new Field(Restriction.NOT_NULL, DataType.text));
        parameters.put("street",new Field(Restriction.NOT_NULL, DataType.text));
        parameters.put("minPrice",new Field(Restriction.NOT_NULL, DataType.integer));
        parameters.put("maxPrice",new Field(Restriction.NOT_NULL, DataType.integer));
        parameters.put("numberTables",new Field(Restriction.NOT_NULL, DataType.integer));
        headers.add(new Header("Restaurant",parameters));
        headers.get(0).setInitialPrimaryKeyValue(RestaurantNIFCreator.getInitialValue());
    }

    private static void createProviderTable() {
        Map<String, Field> parameters = new LinkedHashMap<>();
        parameters.put("NIF",new Field(Restriction.PRIMARY_KEY, DataType.integer));
        parameters.put("companyName",new Field(Restriction.NOT_NULL, DataType.text));
        parameters.put("creationDate",new Field(Restriction.NOT_NULL, DataType.text));
        parameters.put("ownerName",new Field(Restriction.NOT_NULL, DataType.text));
```

*Ilustración 80: Clase DatabaseManager (Manager de la base de datos)*

## Administrador de tablas

Es una clase que engloba todos los controladores en una clase que proporciona todos los servicios de todos los controladores desde un mismo punto. Además, mediante el uso del controlador de builders permite acceder a la base de datos sin conocer ningún elemento o implementación del paquete. Cada elemento de la simulación que requiere de acceder a la capa de datos tiene una instancia de esta clase, como el **Inicializador de Simulables** (Apartado 7.2.4.3).

```
public class SQLiteTableAdministrator extends DatabaseController implements TableAdministrator {
    @Override
    public List read(int initialValue ,int count,Class simulableClass) throws SQLException, ClassNotFoundException {
        Builder builder = initBuilder(simulableClass);
        return builder.buildList(getRows(getModuleOfInitialValue(initialValue), count));
    }

    @Override
    public List readPage(int page, Class simulableClass) throws SQLException, ClassNotFoundException {
        Builder builder = initBuilder(simulableClass);
        return builder.buildList(getRows(page));
    }

    private int getModuleOfInitialValue(int initialValue) throws SQLException, ClassNotFoundException {
        return getActualHeader().getInitialPrimaryKeyValue() + (initialValue % readCount());
    }

    private List<Row> getRows(int initialValue, int count) throws SQLException, ClassNotFoundException {
        return new SQLiteTableSelector().read(actualHeaderName, initialValue, initialValue+count);
    }

    private List<Row> getRows(int page) throws SQLException, ClassNotFoundException {
        return new SQLiteTableSelector().read(actualHeaderName, page);

    }

    @Override
    public int readCount(Class simulableClass) throws SQLException, ClassNotFoundException {
        Builder builder = initBuilder(simulableClass);
        return readCount();
    }
    private int readCount() throws SQLException, ClassNotFoundException {
        return new SQLiteTableSelector().readCount(actualHeaderName);
    }

    @Override
    public void save(Object object) throws SQLException, ClassNotFoundException {
        Builder builder = initBuilder(object.getClass());
```

*Ilustración 81: Clase SQLiteTableAdministrator (Administrador de Tablas)*

## Resultado del paquete desarrollado

Como resultado, se obtiene una interfaz propia, sencilla y genérica de acceso de datos que esconde y separa completamente la implementación del resto de la simulación. Los elementos de la simulación que quieran acceder a los datos solo deben tener una instancia del **Administrador de Tablas** y a la hora de obtener un simulable solo tienen que proporcionar la clase de este simulable, el valor inicial (que procede del NIF del simulable como vimos anteriormente o es solo una fila en el caso de los ajustes) y la cantidad de simulables que se quiere de ese tipo.

### 7.2.7.2 Otras opciones de acceso a datos

Todas las formas de acceso a datos, incluido en la base de datos, contienen una interfaz genérica de acceso, pero con implementaciones distintas. Además de la lectura de la base de datos existen dos formas de acceso adicionales que se han usado durante el desarrollo de este proyecto en versiones anteriores, lecturas de archivos CSV, y lectura directa de internet (Web Scrapping).

- CSV: los datos de clientes-trabajadores y proveedores-servicios inicialmente eran leídos de archivos CSV (Comma-Separated Values) en el que cada fila es una instancia del simulable. Este archivo provino del generador de datos aleatorios usado (6).
- Web Scrapping: mediante la librería de Jsoup(5), se obtenía los datos al principio de los restaurantes. Esta clase leía los datos de restaurantes de la cantidad solicitada a través de la lectura directa de la página web de Tripadvisor, concretamente los restaurantes de Gran Canaria. Posteriormente, se prefirió guardarlos directamente todos los restaurantes de la web en la base datos debido a la baja velocidad de lectura que tiene el Web Scrapping.

## 7.2.8 Preparación de simulables

Tras la lectura y creación de los simulables, estos no están todavía preparados para entrar en la simulación. Estos requieren una serie de necesidades adicionales que cubrir. Estas serán proporcionadas por los preparadores de los simulables. Estos preparadores son llamados por el **Inicializador de Simulables**. Los preparadores son clases funcionales, es decir, que siguen la metodología de la programación funcional. Estos mediante la lista de simulables del tipo con alguna necesidad que cubrir, se prepara paralelamente en un hilo aparte. Los inicializadores son los siguientes:

- Inicializador de Rutinas: inicializas las rutinas de todos los clientes que se van a preparar.
- Inicializador de trabajadores: selecciona trabajo a cada uno de los trabajadores.
- Inicializador de provisión de personal: permite buscar un personal para cubrir inicialmente las necesidades de personal de un restaurante recién abierto.
- Inicializador de personal: selecciona trabajadores para trabajar en el restaurante que se va a inaugurar.
- Inicializador de productos: selecciona el producto que va a vender cada uno de los proveedores a preparar.
- Inicializador de provisión de productos: añade los proveedores necesarios a las empresas que quieren entrar en la simulación.
- Inicializador de servicios: selecciona el servicio que se ofrecerá por esta compañía.

- Inicializador de provisión de servicios: añade los servicios necesarios a las empresas que quieren entrar en la simulación.

### 7.2.9 Resumen

Este módulo es el núcleo del proyecto, el punto central en el que se generan todos los procesos principales en un entorno financiero. Este núcleo es independiente y sin ninguna interfaz de usuario, ya que es el objetivo del segundo módulo. El núcleo funciona de forma autónoma, ya que solo necesita ser inicializado, y este generará los agentes correspondientes para arrancar la simulación y que cada uno de los participantes participe en este entorno con el rol o papel dado para generar las facturas que se requieren.

En la siguiente sección se explicará el siguiente módulo encargado de mostrar al usuario los eventos o acciones que se están produciendo internamente en la simulación, además de poder interactuar con la simulación y cambiar los ajustes principales de esta para tener unos resultados u otros, con el objetivo de facilitar el uso para el usuario.

## 7.3 Entorno de usuario de la simulación

### 7.3.1 Introducción al entorno

La simulación hasta ahora no ha mostrado ninguna interfaz para mostrar la simulación y los eventos que se producen al usuario. Este es el objetivo de esta sección, mostrar de cierto modo la simulación al usuario.

El entorno se ha desarrollado en web con una arquitectura cliente-servidor, en el que el servidor usa directamente la simulación, ejecutándola e interactuando acorde a las instrucciones solicitadas desde el cliente.

En cuanto al servidor, se ha usado el Tomcat EE Server con el lenguaje en Java [17]. Este servidor funciona de forma autónoma a partir de un Servlet o archivo JSP, como archivo inicial que ejecutar, siendo en este caso, un Servlet. El servidor está compuesto por dos Servlet, cuatro websockets y una serie de comandos que controlan la simulación a partir de las instrucciones mandadas desde el cliente.

En cuanto al cliente, se ha desarrollado usando archivos JSP, JavaScript y CSS. Los archivos JSP se usan para mostrar el contenido al alcance del usuario, los archivos JavaScript para controlar los procesos que se producen en el cliente y los archivos CSS

para generar una vista atractiva al usuario final. Para el control de estos procesos se usa la librería Jquery [9] que es compatible con el uso de archivos JSP [18].

## 7.3.2 Servidor

### 7.3.2.1 Servlets

#### Inicializador de la simulación

Inicializa las URIs de los distintos archivos como la base de datos, los archivos CSV o la carpeta donde están las facturas. Además, vacía las facturas de la base de datos si hay, instancia los **Datos de la Simulación** y guarda los datos de los ajustes en la sesión de la web. Por último, carga la vista principal del cliente, que se explicará en la sección del cliente (7.3.3).

```
public class InitSimulationServlet extends HttpServlet {
    protected void doPost(HttpServletRequest request, HttpServletResponse response){
        initUris();
        SimulationBillAdministrator.resetBills();
        SimulationDataController.initSimulationData();
        SettingsBuilder.setCurrentSettingsToSession(request);
        FrontControllerUtils.setQuickSettings(request);
        forwardToMainPage(request, response);
    }

    private void initUris() {
        SimulatorSwitcher.setUriClient(getServletContext().getRealPath("/CSVFiles/Clients.csv"));
        SimulatorSwitcher.setUriProvider(getServletContext().getRealPath("/CSVFiles/Providers.csv"));
        CFDIBillGenerator.setUri(getServletContext().getRealPath("/xmlFiles")+"/");
        SQLiteDatabaseConnector.setUri("jdbc:sqlite:" + getServletContext().getRealPath("/Simulator.db"));
    }

    private void forwardToMainPage(HttpServletRequest request, HttpServletResponse response) {
        try {
            getServletContext().getRequestDispatcher("/index.jsp").forward(request, response);
        } catch (ServletException | IOException e) {
            e.printStackTrace();
        }
    }
```

*Ilustración 82: Clase InitSimulationServlet (Inicializador de la Simulación)*

## FrontController

Corresponde a un patrón de diseño conocido, que consiste en un usar un Servlet como el receptor central de peticiones HTTP en el que se me manda el comando a solicitar a través de la request de la petición. A partir de ahí, todas las acciones del cliente hacen una petición al mismo Servlet con el comando a realizar y por lo que se elige este directamente en el cliente.

```
public class FrontControllerServlet extends HttpServlet {

    @Override
    public void doGet(HttpServletRequest request, HttpServletResponse response){
        doPost(request,response);
    }

    @Override
    public void doPost(HttpServletRequest request, HttpServletResponse response){
        FrontCommand command = getCommand(request);
        command.init(getServletContext(), request, response);
        command.process();
    }

    private FrontCommand getCommand(HttpServletRequest request) {
        try {
            return (FrontCommand) getCommandClass(request).newInstance();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return null;
    }

    private Class getCommandClass(HttpServletRequest request) {
        String command = request.getParameter("command");
        try {
            return Class.forName("backend.server.commands."+ FrontControllerUtils.getFolder(command) + command);
        } catch (ClassNotFoundException e) {
            return UnknownCommand.class;
        }
    }
}
```

*Ilustración 83: Clase FrontControllerServlet (FrontController)*

El FrontController usa un tipo de comando llamado FrontCommand, que es una clase abstracta de la que todos los comandos que se usarán heredan de esta clase implementando el método process(), que como su nombre indica, procesa el comando elegido.

```java
public abstract class FrontCommand {
    protected ServletContext context;
    protected HttpServletRequest request;
    protected HttpServletResponse response;
    public void init(ServletContext context, HttpServletRequest request, HttpServletResponse response){
        this.context = context;
        this.request = request;
        this.response = response;
    }
    abstract public void process();
    public void forward(String target){
        try {
            context.getRequestDispatcher(target).forward(request, response);
        } catch (ServletException | IOException e) {
            e.printStackTrace();
        }
    }

    protected int getAbsoluteIntParameter(String name){
        return Math.abs(Integer.parseInt(request.getParameter(name)));
    }

    protected double getAbsoluteDoubleParameter(String name){
        return Math.abs(Double.parseDouble(request.getParameter(name)));
    }

    protected int getIntParameter(String name){
        return Integer.parseInt(request.getParameter(name));
    }

    protected double getDoubleParameter(String name){
        return Double.parseDouble(request.getParameter(name));
    }

    protected String getStringParameter(String name){
        return request.getParameter(name);
    }

    protected void setToRequest(String name, Object value) {
        request.setAttribute(name, value);
    }
}
```

*Ilustración 84: Clase Abstracta FrontCommand*

### 7.3.2.2 Comandos

Los tipos de comandos que se pueden encontrar en este proyecto son los siguientes:

- UnknownCommand: este comando se ejecuta cuando no se ha encontrado ninguno con el nombre enviado en la petición. Se muestra una vista indicando el error producido.
- StartCommand/RestartCommand: empieza, termina o reinicia la simulación.
- Comando de paginación: comando abstracto que extiende del Frontcommand proporcionando una interfaz a implementar para paginar una lista de datos que se mostrarán posteriormente en una tabla. Se utiliza para mostrar en tablas

paginadas los distintos simulables, por lo que cada uno de ellos tiene una implementación de este comando para mostrarse en el cliente con su vista adecuada.

- SearchCommand: permite buscar simulables, mostrándolos directamente en una tabla como resultado de la petición.
- FollowCommand/UnfollowCommand: te permite seguir o dejar de seguir simulables. Esto se explicará durante la explicación de las funcionalidades en la sección del cliente (7.3.3).
- DownloadCommand: te permite descargar una factura seleccionada. Se proporcionada el UUID de la factura (ID) para luego proporcionar al cliente el archivo XML de la factura.
- Comando de Ajustes: son una larga lista de comandos para controlar los ajustes, ya sea mediante la actualización de los ajustes, el guardado de estos, la cancelación de cambios, poner los ajustes por defecto… etc.

```
public class StartCommand extends FrontCommand {
    @Override
    public void process() {
        SimulatorSwitcher.startStop(true);
    }
}
```

*Ilustración 85: Ejemplo de comando, StartCommand*

### 7.3.2.3 Websockets

En esta simulación se utilizan un total de cuatro websockets, que se utilizan para mandar información de forma asíncrona desde el servidor al cliente, concretamente desde la clase java del websocket hasta el archivo de Javascript que se encarga de administrar estos datos recibidos a través de “Workers” (trabajadores), aunque esto se explicará más detalladamente en el apartado del Cliente (7.3.3).

Los websockets que nos encontramos en la simulación son los siguientes:

- Websocket de eventos: este se encarga de mandar los eventos pendientes por mostrar en el cliente. De los eventos pendientes, solo se mandan los eventos relacionados con los simulables seguidos en el cliente (para más detalles, apartado 7.3.3).
- Websocket de personas: se encarga de mandar los datos actualizados de las personas seguidas por el cliente.
- Websocket de empresas: al igual que el anterior, se encarga de mandar los datos actualizados, pero esta vez de las empresas seguidas por el cliente.
- Websocket contador de simulables: envía datos actuales del número de simulables de cada tipo (Cliente, restaurante, proveedor…).

```
@ServerEndpoint("/eventSocketEndpoint")
public class EventWebSocket {

    private static final String ERROR = "Error sending the events";

    @OnOpen
    public void onOpen(){
        System.out.println("Event socket opened....");
    }

    @OnClose
    public void onClose(){
        System.out.println("Event socket close....");
    }

    @OnMessage
    public String onMessage(String message){
        StringBuilder events = new StringBuilder();
        EventController.getEvents()
                .forEach(event -> events.append("<p>").append(event.getMessage()).append("</p>"));
        return events.toString();
    }

    @OnError
    public void onError(Throwable e){
        System.out.print(ERROR);
    }
}
```

*Ilustración 86: Ejemplo de websocket, websocket de eventos*

## 7.3.2.4 Buscador

En el servidor se ha desarrollado un buscador que a partir de unos datos de búsqueda que se especificaran en las funcionalidades del apartado del cliente (7.3.3). El buscador procesa los datos realizando la búsqueda con los comparadores adecuados que filtran cuales simulables se eligen y cuáles no. Por último, devuelve la lista de simulables que cumplen con los requerimientos de la búsqueda. A continuación, se muestra la interfaz Search (búsqueda) que por cada tipo de búsqueda que se añade se requiere una implementación de esta interfaz.

```
public interface Search<Filterable> {

    static List<Client> searchPeople(Predicate<Client> filter){
        return Simulation.getClientListCopy().stream()
                .filter(client -> !SimulationFollowAdministrator.getFollowedSimulables().contains(client))
                .filter(filter).collect(Collectors.toCollection(LinkedList::new));
    }

    static List<Company> searchCompanies(Predicate<Company> filter){
        return Simulation.getCompanyListCopy().stream()
                .filter(company -> !SimulationFollowAdministrator.getFollowedSimulables().contains(company))
                .filter(filter).collect(Collectors.toCollection(LinkedList::new));
    }

    Predicate<Filterable> search(String searchText);
}
```

*Ilustración 87: Interfaz Search (Busqueda)*

## 7.3.3 Cliente

### 7.3.3.1 Vistas

Las vistas disponibles para ser accedidas por parte del cliente son las siguientes:

- Vista principal: es la interfaz más importante del entorno de usuario. Aquí es donde se visualiza la simulación y se interactúa con esta directamente. Permite controlar la ejecución de la simulación, mostrar los eventos y buscar y seguir simulables. Todas estas funcionalidades y las de las siguientes vistas se explicarán a continuación en el apartado de funcionalidades (7.3.3.2). Arriba se encuentra el navegador principal con el acceso al resto de vistas.

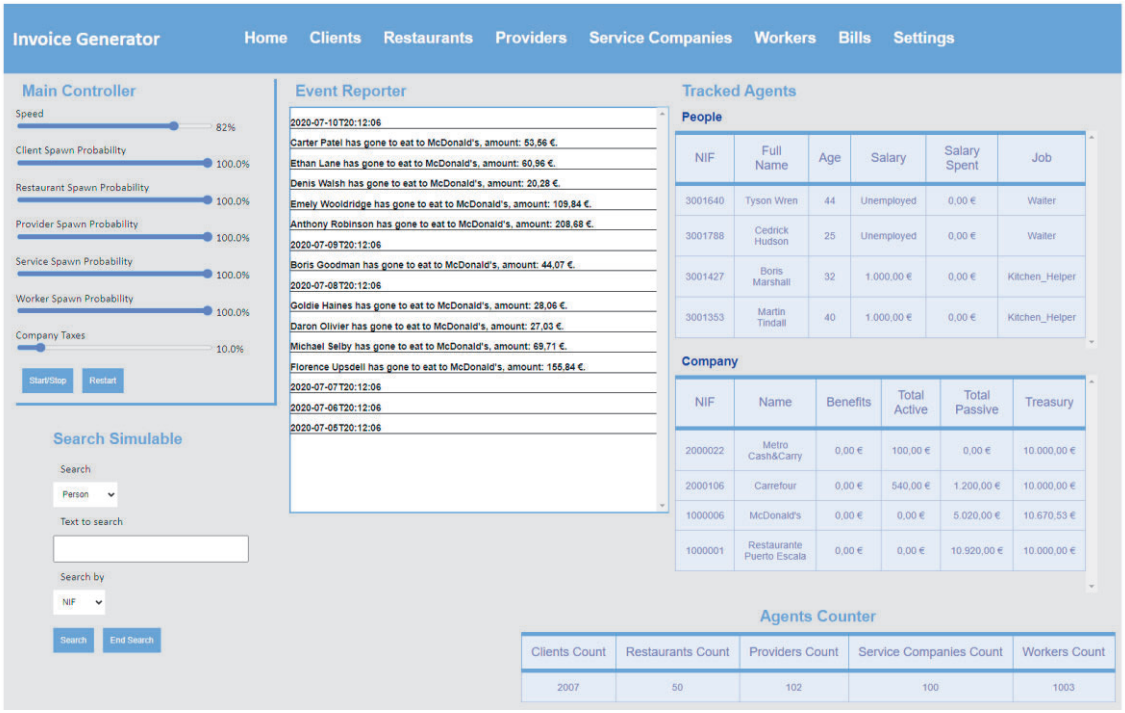



*Ilustración 88: Vista principal de la web*

- Vista de clientes: es la vista donde se pueden ver todos los clientes de la simulación y todos los datos de estos. Esta ordenado por el NIF y paginado en el que cada página se encuentran 30 clientes. En el resto, la paginación y ordenación es la misma.

Invoice Generator Home Clients Restaurants Providers Service Companies Workers Bills Settings

Clients

Current Date: 2020-06-04T22:52:47

Page 1

| NIF | First Name | Last Name | Birth Date | Gender | Job | Salary | Salary Spent | Country | Telephone Number | Email | Card Number |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 3000000 | Kurt | Latham | 5/8/1968 | Male | Front Desk Coordinator | 998,67 € | 105,14 € | Montenegro | 4-272-505-6226 | Kurt_Latham9227@corti.com | 3150-0231-3184-7012 |
| 3000001 | Johnny | Rodgers | 4/21/1947 | Male | Banker | 723,99 € | 107,15 € | Romania | 7-827-043-2626 | Johnny_Rodgers7829@tonsy.org | 2811-1776-8442-8671 |
| 3000002 | Angel | Drake | 6/28/1944 | Female | Staffing Consultant | 1.001,01 € | 148,15 € | Bosnia and Herzegovina | 4-043-842-1711 | Angel_Drake1759@gmail.com | 6147-4650-6506-5564 |
| 3000003 | Shannon | Pitt | 6/2/1981 | Female | Production Painter | 3.450,78 € | 510,72 € | Netherlands | 8-878-507-2381 | Shannon_Pitt6849@bulaffy.com | 1706-3352-7560-6843 |
| 3000004 | Christine | Talbot | 2/9/1952 | Female | Stockbroker | 2.843,79 € | 282,46 € | Vietnam | 0-821-720-5070 | Christine_Talbot5889@supunk.biz | 7810-3188-0380-4748 |
| 3000005 | Shay | Hunter | 12/12/1994 | Female | Software Engineer | 1.213,41 € | 179,59 € | Nepal | 3-807-536-0427 | Shay_Hunter7921@acrit.org | 1808-5312-3566-5260 |
| 3000006 | Gwen | Tennant | 9/29/1963 | Female | Doctor | 648,09 € | 95,92 € | Hungary | 5-343-214-7402 | Gwen_Tennant4389@gembat.biz | 5440-3300-0080-0437 |
| 3000007 | Anthony | Wright | 7/6/1982 | Male | Food Technologist | 1.255,78 € | 185,86 € | Bhutan | 4-276-668-8570 | Anthony_Wright6423@malfthy.com | 3154-2421-5000-0862 |
| 3000008 | Julian | Bell | 10/15/1943 | Male | Cashier | 620,64 € | 91,85 € | Slovenia | 2-018-477-3851 | Julian_Bell2583@womeona.net | 2628-1765-7257-1255 |
| 3000009 | John | Ripley | 9/15/1953 | Male | Call Center Representative | 2.480,05 € | 367,05 € | Kazakhstan | 2-260-115-1437 | John_Ripley9408@nimogy.biz | 3574-7413-4263-2605 |
| 3000010 | Martha | Jenkins | 7/14/1958 | Female | Electrician | 2.915,91 € | 431,55 € | Switzerland | 4-286-274-5050 | Martha_Jenkins8419@supunk.biz | 4416-0867-3445-7550 |

*Ilustración 89: Vista de clientes*

- Vista de restaurantes: la misma vista que los clientes, pero de restaurantes.

Invoice Generator Home Clients Restaurants Providers Service Companies Workers Bills Settings

Restaurants

Current Date: 2020-06-04T22:52:47

Page 1

| Company NIF | Company Name | Score | Table Number | Minimum Price Per Plate | Maximum Price Per Plate | Number Of Workers | Zipcode | Telephone Number | Profits | Losses | Sales | Social Capital | Total Active | Total Passive | Treasury |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1000000 | el patio de lola | 3,75 | 20 | 15,00 € | 25,00 € | 12 | 37925 | +34 828 12 63 37 | 914,92 € | 12.920,00 € | 914,92 € | 10.000,00 € | 0,00 € | 12.920,00 € | 10.914,92 € |
| 1000001 | Restaurante Puerto Escala | 2,70 | 15 | 13,00 € | 35,00 € | 10 | 35573 | +34 928 56 06 90 | 0,00 € | 11.120,00 € | 0,00 € | 10.000,00 € | 0,00 € | 11.120,00 € | 10.000,00 € |
| 1000002 | El Salsete | 3,08 | 22 | 15,00 € | 50,00 € | 13 | 39681 | +34 928 77 82 55 | 42,26 € | 13.720,00 € | 42,26 € | 10.000,00 € | 0,00 € | 13.720,00 € | 10.042,26 € |
| 1000003 | Whaoo | 2,79 | 23 | 12,00 € | 44,00 € | 14 | 38029 | | 0,00 € | 14.720,00 € | 0,00 € | 10.000,00 € | 0,00 € | 14.720,00 € | 10.000,00 € |
| 1000004 | Aguaviva Restaurant | 2,86 | 12 | 8,00 € | 24,00 € | 7 | 39530 | | 0,00 € | 7.820,00 € | 0,00 € | 10.000,00 € | 0,00 € | 7.820,00 € | 10.000,00 € |
| 1000005 | Chino Royal | 2,61 | 29 | 12,00 € | 20,00 € | 18 | 35120 | | 0,00 € | 18.820,00 € | 0,00 € | 10.000,00 € | 0,00 € | 18.820,00 € | 10.000,00 € |
| 1000006 | McDonald's | 3,75 | 7 | 6,00 € | 20,00 € | 4 | 36800 | +34 928 56 03 43 | 104,77 € | 5.220,00 € | 104,77 € | 10.000,00 € | 0,00 € | 5.220,00 € | 10.104,77 € |
| 1000007 | The Great Wall | 2,40 | 8 | 7,00 € | 36,00 € | 5 | 36597 | +34 928 16 34 04 | 0,00 € | 6.220,00 € | 0,00 € | 10.000,00 € | 0,00 € | 6.220,00 € | 10.000,00 € |
| 1000008 | La Taberna de El Monje | 2,50 | 20 | 5,00 € | 15,00 € | 12 | 38677 | +34 928 31 01 85 | 0,00 € | 12.920,00 € | 0,00 € | 10.000,00 € | 0,00 € | 12.920,00 € | 10.000,00 € |
| 1000009 | La Bodega de Perdomo | 3,00 | 10 | 15,00 € | 20,00 € | 6 | 35593 | +34 928 37 30 24 | 255,93 € | 7.020,00 € | 255,93 € | 10.000,00 € | 0,00 € | 7.020,00 € | 10.255,93 € |
| 1000010 | Restaurante Costa del Burrero | 2,17 | 11 | 8,00 € | 28,00 € | 6 | 37437 | +34 678 36 44 50 | 0,00 € | 7.020,00 € | 0,00 € | 10.000,00 € | 0,00 € | 7.020,00 € | 10.000,00 € |

*Ilustración 90: Vista de restaurantes*

- Vista de proveedores


| Company NIF | Company Name | Product | Product Price | Owner Name | Creation Date | Zipcode | Telephone Number | Profits | Losses | Sales | Social Capital | Total Active | Total Passive | Treasury |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2000100 | Erickson | Vegetable | 150,00 € | Nate Jeffery | 7/13/2565 | 39603 | 5-824-085-5380 | 150,00 € | 1.600,00 € | 0,00 € | 10.000,00 € | 150,00 € | 1.600,00 € | 10.000,00 € |
| 2000101 | Team Guard SRL | Meat | 160,00 € | Alexander Neville | 12/23/1367 | 36690 | 2-076-000-1142 | 0,00 € | 1.700,00 € | 0,00 € | 10.000,00 € | 0,00 € | 1.700,00 € | 10.000,00 € |
| 2000102 | Facebook | Fish | 160,00 € | Mason Ryan | 5/29/8845 | 38096 | 5-304-664-7562 | 160,00 € | 1.700,00 € | 0,00 € | 10.000,00 € | 160,00 € | 1.700,00 € | 10.000,00 € |
| 2000103 | Facebook | Egg | 80,00 € | Rosie Khan | 3/21/1986 | 37717 | 5-448-136-1520 | 0,00 € | 900,00 € | 0,00 € | 10.000,00 € | 0,00 € | 900,00 € | 10.000,00 € |
| 2000104 | Metro Cash&Carry | Legume | 80,00 € | Erin Phillips | 4/21/3630 | 38339 | 3-110-676-8878 | 0,00 € | 900,00 € | 0,00 € | 10.000,00 € | 0,00 € | 900,00 € | 10.000,00 € |
| 2000105 | ExxonMobil | Fruit | 100,00 € | Marvin Allington | 5/7/6013 | 35000 | 4-163-632-1426 | 0,00 € | 1.100,00 € | 0,00 € | 10.000,00 € | 0,00 € | 1.100,00 € | 10.000,00 € |
| 2000106 | Carrefour | Others | 90,00 € | Emmanuelle Gilmore | 8/10/3090 | 35768 | 5-350-447-4333 | 0,00 € | 1.000,00 € | 0,00 € | 10.000,00 € | 0,00 € | 1.000,00 € | 10.000,00 € |
| 2000107 | Telekom | Egg | 80,00 € | Chad Hunter | 5/30/0543 | 39790 | 1-646-034-1146 | 0,00 € | 900,00 € | 0,00 € | 10.000,00 € | 0,00 € | 900,00 € | 10.000,00 € |
| 2000108 | Leadertech Consulting | Vegetable | 150,00 € | Sara Andrews | 4/30/3217 | 37286 | 5-655-816-8858 | 0,00 € | 1.600,00 € | 0,00 € | 10.000,00 € | 0,00 € | 1.600,00 € | 10.000,00 € |
| 2000109 | Vodafone | Fruit | 100,00 € | Sabina Windsor | 4/18/9094 | 36990 | 6-348-545-8225 | 200,00 € | 1.100,00 € | 0,00 € | 10.000,00 € | 200,00 € | 1.100,00 € | 10.000,00 € |
| 2000110 | Coca-Cola Company | Meat | 160,00 € | Bryon Shields | 9/4/8216 | 35739 | 3-508-082-0071 | 0,00 € | 1.700,00 € | 0,00 € | 10.000,00 € | 0,00 € | 1.700,00 € | 10.000,00 € |
| 2000111 | Coca-Cola Company | Meat | 160,00 € | Peyton Skinner | 1/8/2927 | 37175 | 2-386-607-5603 | 0,00 € | 1.700,00 € | 0,00 € | 10.000,00 € | 0,00 € | 1.700,00 € | 10.000,00 € |

*Ilustración 91: Vista de proveedores*

- Vista de empresas de servicios


| Company NIF | Company Name | Service | Service Price | Owner Name | Creation Date | Zipcode | Telephone Number | Profits | Losses | Sales | Social Capital | Total Active | Total Passive | Treasury |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2000000 | Biolife Grup | Cleaning | 100,00 € | Henry Button | 8/19/1095 | 39606 | 1-710-074-3636 | 0,00 € | 0,00 € | 0,00 € | 10.000,00 € | 0,00 € | 0,00 € | 10.000,00 € |
| 2000001 | Leadertech Consulting | Transport | 100,00 € | Anabelle Jacobs | 8/28/1479 | 37930 | 0-101-060-2201 | 500,00 € | 0,00 € | 0,00 € | 10.000,00 € | 500,00 € | 0,00 € | 10.000,00 € |
| 2000002 | Apple Inc. | Transport | 100,00 € | Carter Tobin | 8/6/8631 | 37389 | 1-652-478-0408 | 1.100,00 € | 0,00 € | 0,00 € | 10.000,00 € | 1.100,00 € | 0,00 € | 10.000,00 € |
| 2000003 | ExxonMobil | Transport | 100,00 € | Boris White | 3/10/0343 | 36704 | 6-175-848-0113 | 200,00 € | 0,00 € | 0,00 € | 10.000,00 € | 200,00 € | 0,00 € | 10.000,00 € |
| 2000004 | CarMax | Transport | 100,00 € | Crystal Gilmore | 10/3/3346 | 38444 | 3-206-571-5034 | 500,00 € | 0,00 € | 0,00 € | 10.000,00 € | 500,00 € | 0,00 € | 10.000,00 € |
| 2000005 | CarMax | Cleaning | 100,00 € | Elise Lewis | 8/27/3689 | 35717 | 6-205-283-1871 | 100,00 € | 0,00 € | 0,00 € | 10.000,00 € | 100,00 € | 0,00 € | 10.000,00 € |
| 2000006 | Boeing | Transport | 100,00 € | Miley Brooks | 11/1/5086 | 36472 | 8-432-342-7102 | 1.200,00 € | 0,00 € | 0,00 € | 10.000,00 € | 1.200,00 € | 0,00 € | 10.000,00 € |
| 2000007 | Biolife Grup | Cleaning | 100,00 € | Mackenzie Shaw | 9/11/7832 | 37474 | 3-248-265-0426 | 100,00 € | 0,00 € | 0,00 € | 10.000,00 € | 100,00 € | 0,00 € | 10.000,00 € |
| 2000008 | Apple Inc. | Transport | 100,00 € | Jolene Walton | 7/22/5661 | 35589 | 3-287-584-6314 | 500,00 € | 0,00 € | 0,00 € | 10.000,00 € | 500,00 € | 0,00 € | 10.000,00 € |
| 2000009 | Carrefour | Cleaning | 100,00 € | Benjamin Clarke | 2/8/0310 | 38871 | 7-730-656-2037 | 100,00 € | 0,00 € | 0,00 € | 10.000,00 € | 100,00 € | 0,00 € | 10.000,00 € |
| 2000010 | Areon Impex | Cleaning | 100,00 € | Angel Speed | 10/25/2211 | 37009 | 2-042-285-1754 | 300,00 € | 0,00 € | 0,00 € | 10.000,00 € | 300,00 € | 0,00 € | 10.000,00 € |
| 2000011 | UPC | Cleaning | 100,00 € | Mike Waterson | 5/16/0685 | 39006 | 4-704-474-5561 | 100,00 € | 0,00 € | 0,00 € | 10.000,00 € | 100,00 € | 0,00 € | 10.000,00 € |
| 2000012 | Leadertech Consulting | Cleaning | 100,00 € | Harvey Stanton | 5/27/4174 | 39827 | 7-568-054-7853 | 400,00 € | 0,00 € | 0,00 € | 10.000,00 € | 400,00 € | 0,00 € | 10.000,00 € |

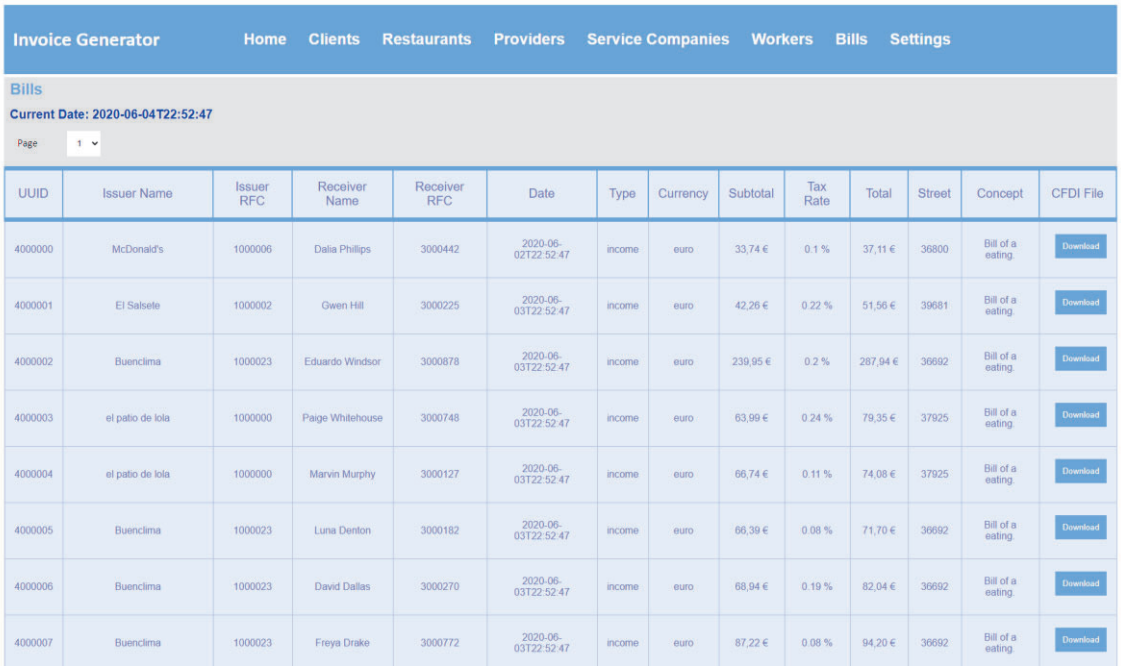
*Ilustración 92: Vista de empresas de servicios*

- Vista de trabajadores


| NIF | First Name | Last Name | Birth Date | Gender | Job | Salary | Contract's Expire Date | Restaurant | Quality | Country | Telephone Number | Email | Card Number |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3001000 | Alessandra | Gordon | 6/20/1977 | Female | Waiter | 800,00 € | 2021-00-09T22:51:09 | el patio de lola | HIGH | Uganda | 0-263-887-8660 | Alessandra_Gordon6720@liret.org | 0224-1053-7675-1005 |
| 3001001 | Leroy | Brock | 4/17/1994 | Male | Kitchen_Helper | 1.000,00 € | 2020-11-24T22:51:09 | el patio de lola | VERY LOW | Estonia | 5-312-633-0333 | Leroy_Brock3107@iatim.tech | 2641-5777-7738-4168 |
| 3001002 | Nicholas | Mccall | 11/18/1993 | Male | Cooker | 1.500,00 € | 2020-09-22T22:51:09 | el patio de lola | VERY HIGH | India | 4-106-514-5627 | Nicholas_Mccall4581@ubusive.com | 3786-5621-2618-0333 |
| 3001003 | Peter | Pitt | 12/14/1974 | Male | Maitre | Unemployed | Unemployed | Unemployed | LOW | Guinea | 2-473-408-8602 | Peter_Pitt7280@vetan.org | 4371-3875-7028-6576 |
| 3001004 | Javier | Bullock | 2/13/1972 | Male | Chef | Unemployed | Unemployed | Unemployed | VERY HIGH | Saint Vincent and the Grenadines | 0-806-300-7616 | Javier_Bullock2411@liret.org | 7354-4546-3026-8012 |
| 3001005 | Chloe | Palmer | 2/25/1958 | Female | Waiter | 800,00 € | 2021-02-15T22:51:09 | el patio de lola | VERY HIGH | Zambia | 2-788-506-8317 | Chloe_Palmer483@twipet.com | 6236-7704-8755-5288 |
| 3001006 | Rosie | Huggins | 4/5/1962 | Female | Waiter | 800,00 € | 2021-05-24T22:51:09 | el patio de lola | VERY HIGH | New Zealand | 1-885-038-2688 | Rosie_Huggins2172@infotech44.tech | 5022-8220-7784-4665 |

*Ilustración 93: Vista de trabajadores*

- Vista de facturas: además de características ya nombradas, también se pueden descargar, para más detalles (7.3.3.2).




| UUID | Issuer Name | Issuer RFC | Receiver Name | Receiver RFC | Date | Type | Currency | Subtotal | Tax Rate | Total | Street | Concept | CFDI File |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 4000000 | McDonald's | 1000006 | Dalia Phillips | 3000442 | 2020-06-02T22:52:47 | income | euro | 33,74 € | 0.1 % | 37,11 € | 36800 | Bill of a eating. | Download |
| 4000001 | El Salsete | 1000002 | Gwen Hill | 3000225 | 2020-06-03T22:52:47 | income | euro | 42,26 € | 0.22 % | 51,56 € | 39681 | Bill of a eating. | Download |
| 4000002 | Buenclima | 1000023 | Eduardo Windsor | 3000878 | 2020-06-03T22:52:47 | income | euro | 239,95 € | 0.2 % | 287,94 € | 36692 | Bill of a eating. | Download |
| 4000003 | el patio de lola | 1000000 | Paige Whitehouse | 3000748 | 2020-06-03T22:52:47 | income | euro | 63,99 € | 0.24 % | 79,35 € | 37925 | Bill of a eating. | Download |
| 4000004 | el patio de lola | 1000000 | Marvin Murphy | 3000127 | 2020-06-03T22:52:47 | income | euro | 66,74 € | 0.11 % | 74,08 € | 37925 | Bill of a eating. | Download |
| 4000005 | Buenclima | 1000023 | Luna Denton | 3000182 | 2020-06-03T22:52:47 | income | euro | 66,39 € | 0.08 % | 71,70 € | 36692 | Bill of a eating. | Download |
| 4000006 | Buenclima | 1000023 | David Dallas | 3000270 | 2020-06-03T22:52:47 | income | euro | 68,94 € | 0.19 % | 82,04 € | 36692 | Bill of a eating. | Download |
| 4000007 | Buenclima | 1000023 | Freya Drake | 3000772 | 2020-06-03T22:52:47 | income | euro | 87,22 € | 0.08 % | 94,20 € | 36692 | Bill of a eating. | Download |

*Ilustración 94: Vista de facturas*

- Vista de ajustes generales: es la primera vista mostrada al seleccionar el apartado de ajustes en el cliente. Aquí se pueden editar los ajustes generales explicados en el apartado de ajustes (7.2.4.5). El resto de los ajustes es accesible a través de su propio navegador y cada una ajusta los datos que les corresponden.

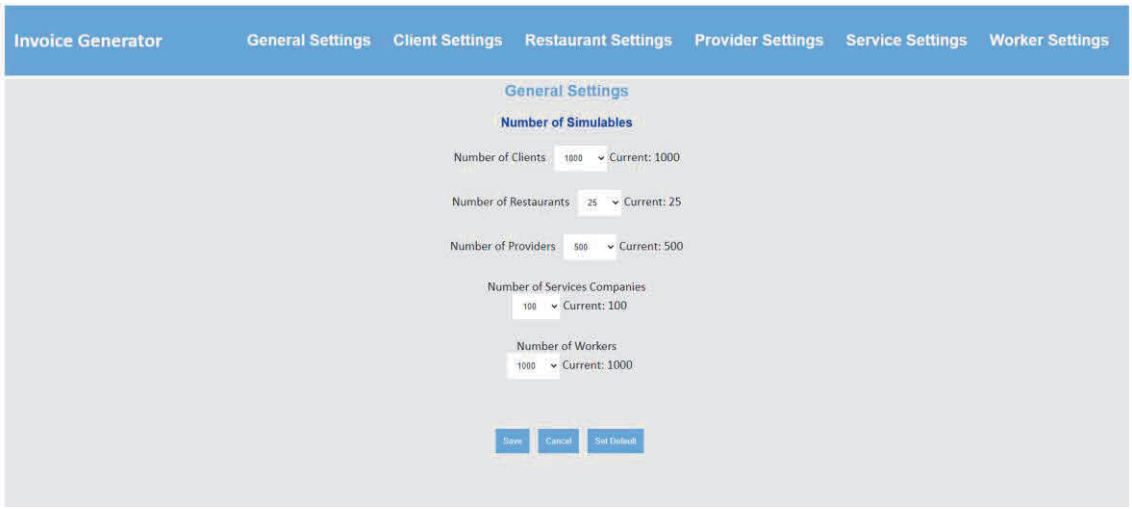



*Ilustración 95: Vista de ajustes generales*

- Vista de ajustes de clientes

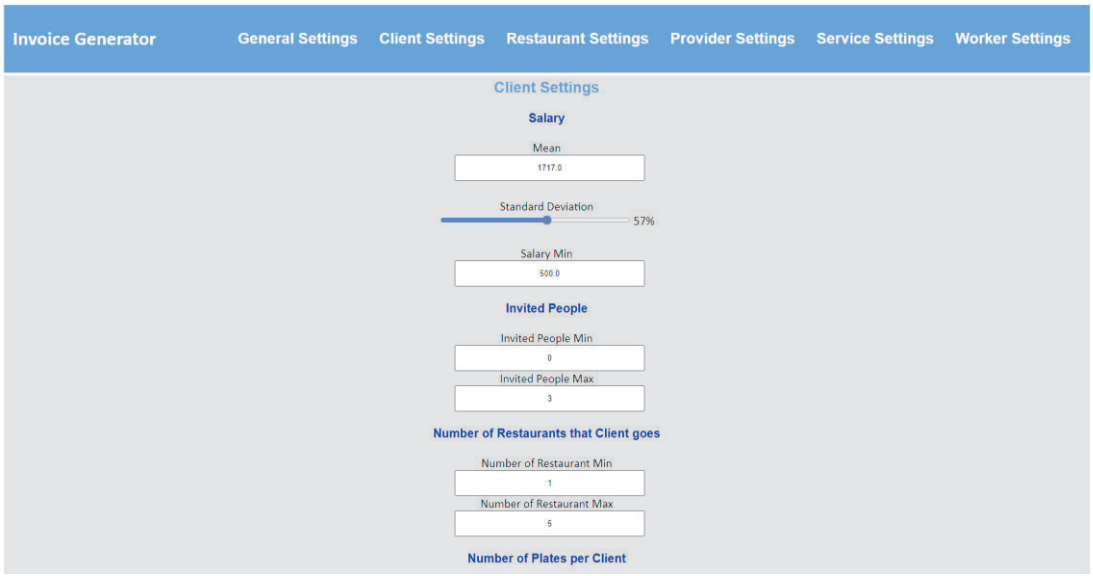



*Ilustración 96: Vista de ajustes de clientes*

- Vista de ajustes de restaurantes

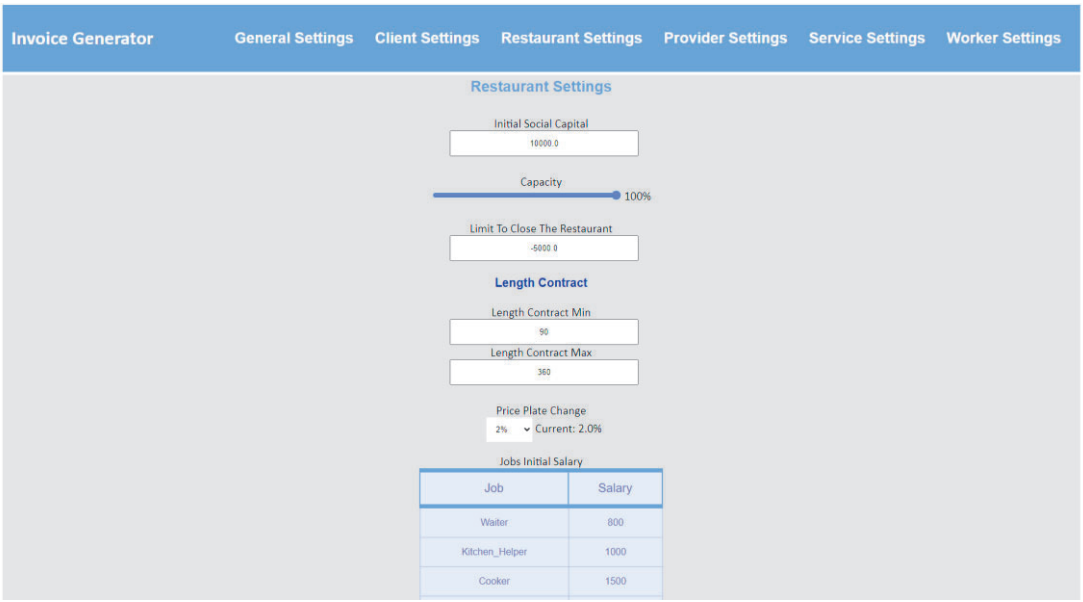



*Ilustración 97: Vista de ajustes de restaurantes*

- Vista de ajustes de proveedores

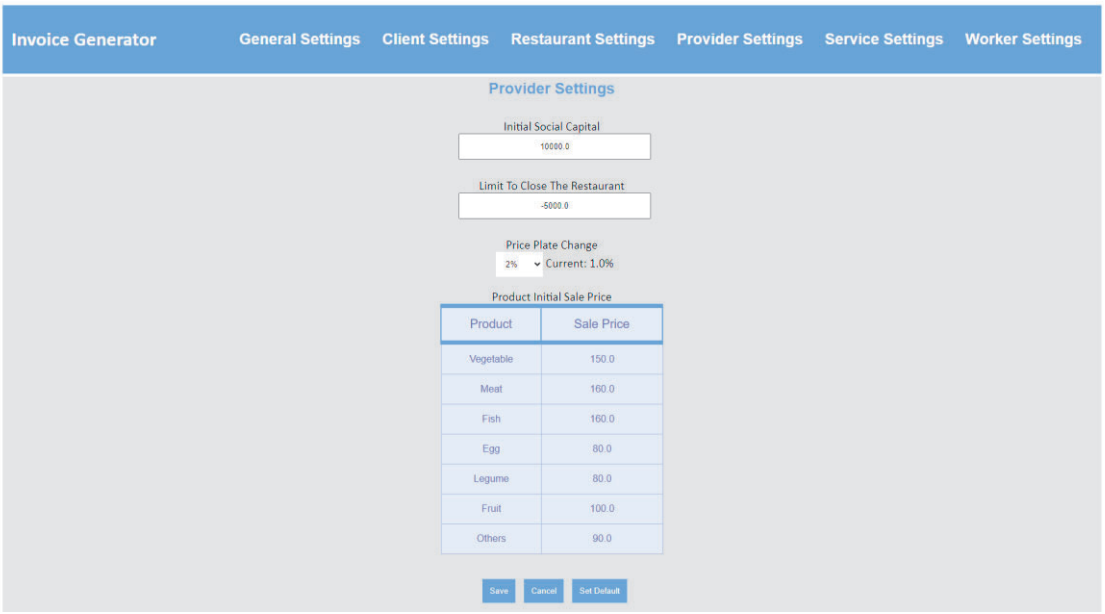



*Ilustración 98: Vista de ajustes de proveedores*

- Vista de ajustes de servicios

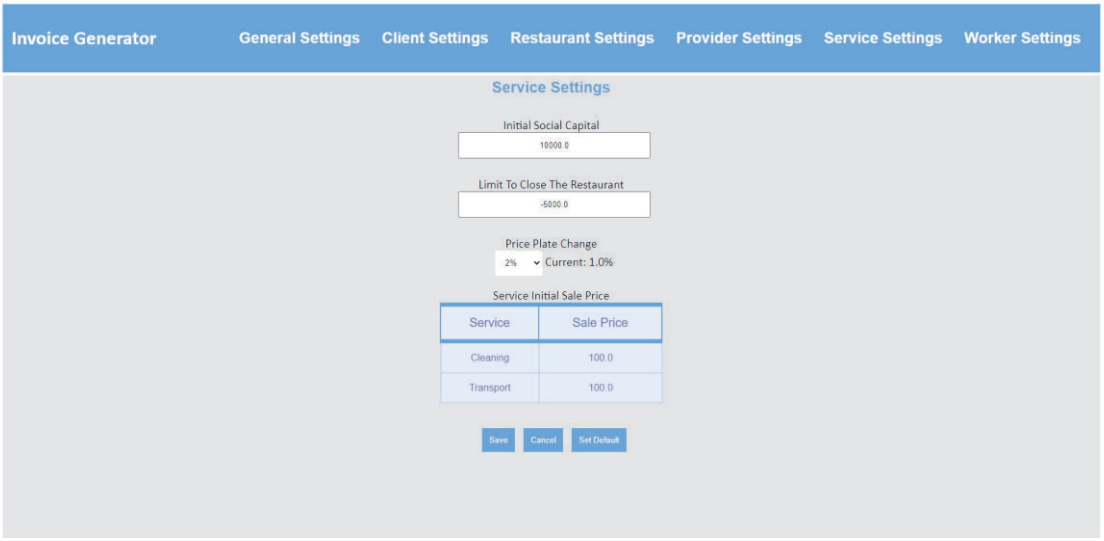



*Ilustración 99: Vista de ajustes de servicios*

- Vista de ajustes de trabajadores

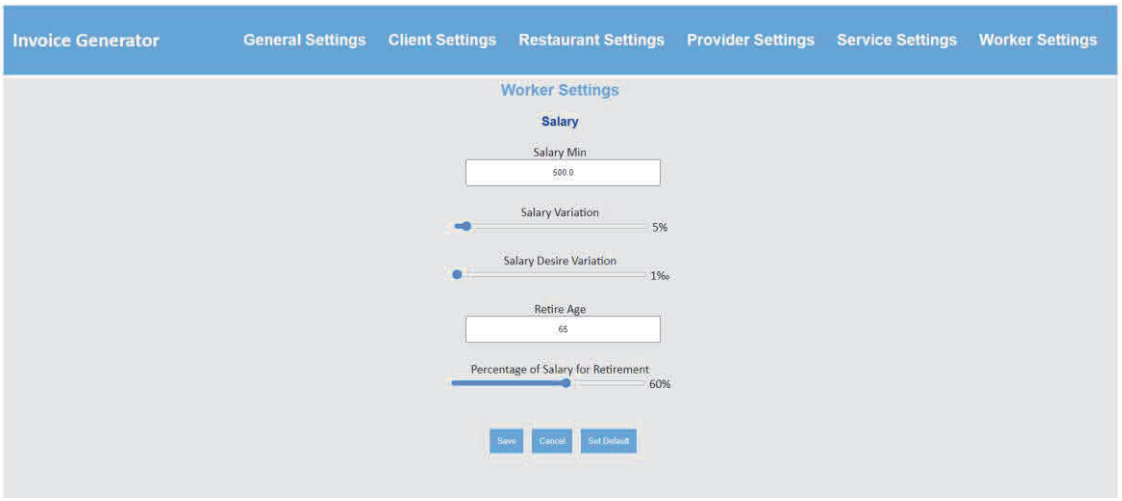



*Ilustración 100: Vista de ajustes de trabajadores*

## 7.3.3.2 Funcionalidades para el usuario

### Visualización de eventos

Esta funcionalidad se encuentra en la vista principal. Es la funcionalidad más importante del cliente, ya que es medio principal de comunicación entre la simulación y el usuario. Esto se produce a través de un panel que muestran eventos o mensajes de los acontecimientos ocurridas dentro del entorno de la simulación. Las acciones que se muestran son producidas o están relacionadas por alguno de los agentes que se están siguiendo en la tabla de agentes seguidos, esta última funcionalidad se explicará a continuación.

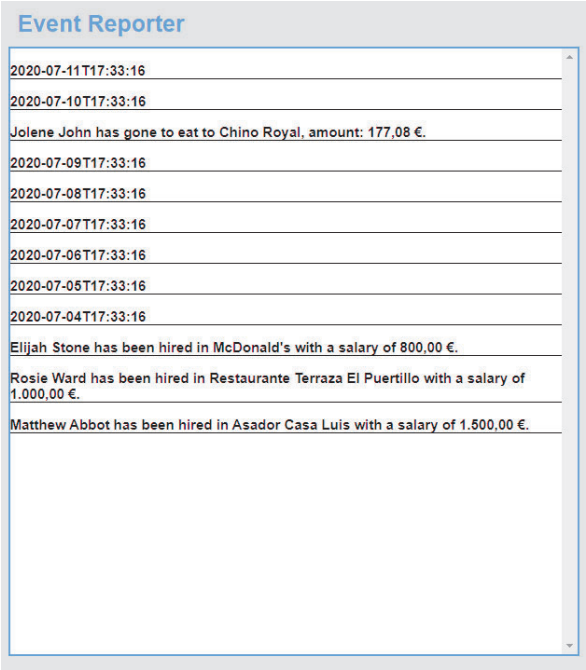



*Ilustración 101: Reporte de eventos*

## Seguimiento de simulables

Si se mostraran todos los eventos que se producen realmente en la simulación, el usuario no podría leerlos adecuadamente. Esto se debe a que, al haber 2000 personas y 250 empresas por defecto al inicio de la simulación, se producirán un elevado número de eventos diariamente y, por lo tanto, un gran número de eventos que mostrar. Para ello, he optado por desarrollar una funcionalidad que se encuentra en la vista principal y que permite al usuario centrarse en ciertos agentes de la simulación a los que le interese “seguir sus pasos” y obtener información constante acerca del estado que se encuentran y las acciones que realizan diariamente. Como implementación, decidí mostrar 2 tablas, una de personas y otra de empresas con los simulables o agentes que son seguidos por el usuario. Al hacer clic en ellos se eliminan de la tabla y se dejan de seguir.

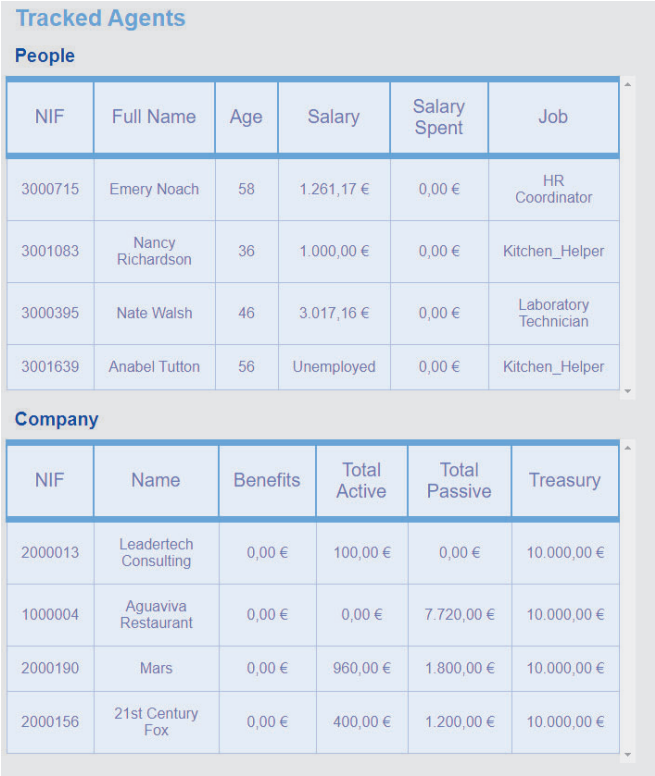



*Ilustración 102: Tablas de agentes seguidos*

Para poder añadir agentes a las tablas, se usa un buscador que se encuentra abajo a la izquierda también en la vista principal. Este buscador permite buscar por persona o empresa, se especifica el texto de búsqueda y el filtro de búsqueda que depende de si es persona o empresa. Los filtros son los siguientes:

- NIF: filtro para buscar por NIF tanto personas como empresas.
- Nombre: para buscar por nombre tanto en personas como empresa.
- Salario: para buscar personas por su salario.
- Trabajo: para buscar personas por el trabajo que desempeñan.
- Beneficios: para buscar empresas por el beneficio proporcionado.
- Tesorería o patrimonio neto: para buscar empresas a través del capital actual de la empresa.

Tras la búsqueda se muestra una tabla debajo con los 30 primeros resultados obtenidos. Al hacer clic sobre ellos se añaden automáticamente a lista de simulables seguidos.

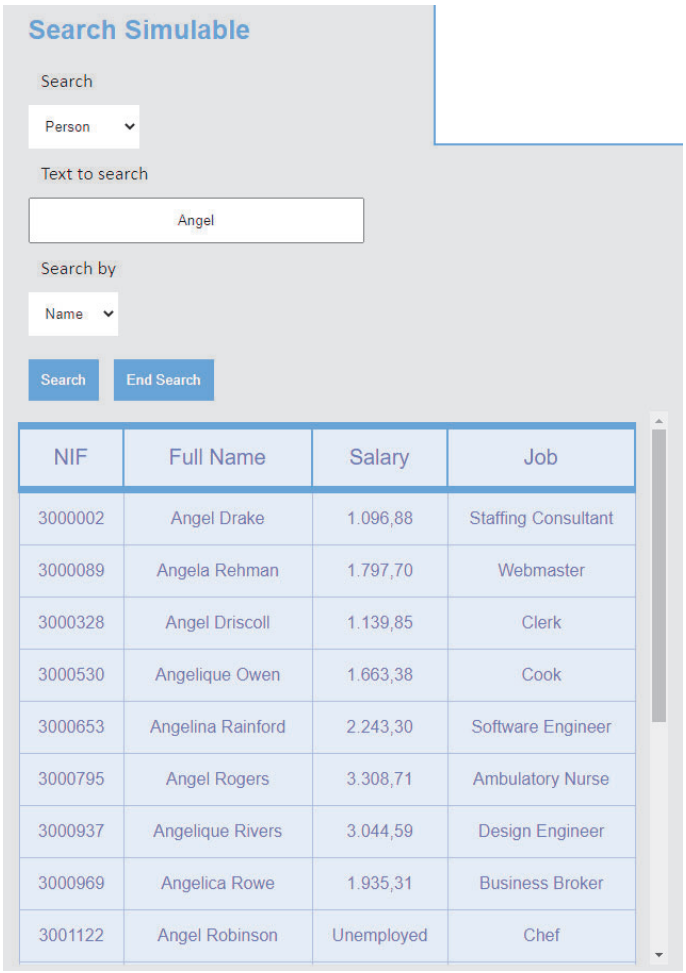



*Ilustración 103: Buscador de simulables*

## Mando de control

Esta funcionalidad se encuentra en la vista principal y permite controlar de forma rápida y sencilla el estado de la simulación y otros aspectos muy importantes. A través de botones de rango que permiten editar datos de ajustes que son porcentajes que afectan en gran medida a la simulación. Los controles que se pueden editar son los siguientes:

- Velocidad: cambia la velocidad de la simulación, es decir, el tiempo que se transcurre entre un día y otro.
- Probabilidades de aparición: Es un porcentaje que tienen cada uno de los agentes de la simulación que afectan a su probabilidad de aparición. Este porcentaje se añade al calculado con los factores de interés de cada uno de los simulables, explicados en el apartado de bajas y altas de simulables (7.2.3). Cuando el porcentaje es 100%, significa que no disminuye en absoluto el porcentaje calculado a través de los factores. Si el porcentaje es 0%, nunca aparecerán simulables de ese tipo concreto.
- Impuestos: aumenta o disminuye el porcentaje de beneficios que se pagarán al estado. Para más detalles sobre el pago de impuestos, consultar apartado de Empresas (7.2.2.1) y de Banco (7.2.2.3).

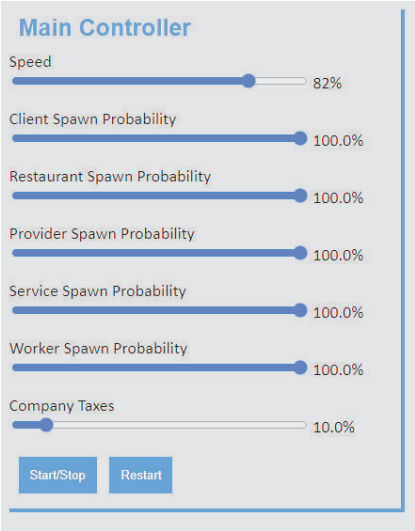



*Ilustración 104: Mando de control*

Además, se pueden visualizar abajo a la derecha el número de simulables de cada tipo en una tabla que se actualiza constantemente.


*Ilustración 105: Contador de agentes*

## Vista de simulables

Como habíamos explicado en el apartado de Vistas (7.3.3.1), se pueden ver en sus vistas correspondientes la tabla con todos los simulables de ese tipo. Estas se muestran de treinta en treinta por páginas. Al introducir una nueva página, se manda esta página al servidor y a partir de ahí se muestran los simulables correspondientes a esta página.

Además, se permiten descargar la factura cuando se solicita en la vista de facturas correspondientes. Cuando se solicita, se manda el UUID de la factura para obtener la factura correspondiente en el servidor y se ejecuta el comando de descarga de facturas.

### Ajustes

En la vista de ajustes se han podido ver las distintas vistas de cada apartado de ajustes. Estos ajustes se pueden editar constantemente. Cuando se editan se actualizan automáticamente en la sesión del cliente web. Además, se pueden guardar, cancelar los cambios o establecer como predeterminado:

- Guardar: en el caso de que el cliente quiera guardar los ajustes, el servidor obtiene de la sesión los ajustes actualizados en la sesión y los establece como los actuales en la simulación.
- Cancelar los cambios: se actualizan los ajustes de la sesión con los actuales de la simulación, con el objetivo de cancelar los cambios realizados por el cliente.
- Establecer como predeterminado: se establecen los ajustes de la simulación con los ajustes predeterminados en la base de datos, como vimos en el apartado de Ajustes de la sección de Administración (7.2.4.5).

## 7.3.3.3 Administrador del cliente

Todas las funcionalidades explicadas anteriormente son llevadas por un administrador. Este administrador es un archivo JavaScript, que mediante la librería de Jquery (8), realiza las tareas para poder llevar a cabo las tareas correspondientes para realizar las funcionalidades mostradas. Este archivo es el encargado en la mayor parte de los casos de realizar todos los cambios que no requieran de servidor y de todas las peticiones HTTP al servidor con los datos adecuados, para luego obtener la respuesta y mostrarla correctamente al usuario.

Asimismo, es el encargado de activar y desactivar los Workers (Trabajadores) en función de la situación (normalmente si la simulación está en ejecución o no). Los Workers de JavaScript trabajan asíncronamente en un hilo aparte donde ejecutan la tarea que le encomiendan y devuelven el resultado, esto permite actualizar datos que se necesiten consultar constantemente como el envío de eventos, y que no afecte al hilo principal del usuario. Cada Worker es encargado de conectarse a un websocket, de los explicados en el apartado de Websockets (7.3.2.3) Cada Worker, al recibir la información, la manda directamente al administrador y ésta la procesa y la muestra al usuario.

```
let eventWorker;
let personWorker;
let companyWorker;
let simulableCountWorker;
let isRunning = false;

function showResult() {
    document.getElementById("divSearchTable").style.display="block";
}

function hideResult() {
    document.getElementById("divSearchTable").style.display="none";
}

function showTables() {
    document.getElementById("followedSimulables").style.display="block";

}

function hideTables() {
    document.getElementById("followedSimulables").style.display="none";
}

function changeOptions(){
    let val = $('#simulableOptions').val();
    if (val === "person") {
        $("#searchOptions").html("<option value='NIF'>NIF</option>" +
            "<option value='Name'>Name</option>" +
            "<option value='Job'>Job</option>" +
            "<option value='Salary'>Salary</option>");
    } else {
        $("#searchOptions").html("<option value='NIF'>NIF</option>" +
            "<option value='Name'>Name</option>" +
            "<option value='Benefits'>Benefits</option>" +
            "<option value='Treasury'>Treasury</option>");
    }
}

function receiveData() {
    if(isRunning)startWorkers();
    else stopWorkers();
}

function startWorkers() {
    startEventWorker();
    startSimulableWorkers();
}
```

*Ilustración 106: Clase Administrador de JavaScript*

# 8. Conclusiones

## 8.1 Resultado obtenido

### 8.1.1 Del proyecto

El resultado que se ha obtenido es un proyecto compuesto por dos módulos: el núcleo de procesos, y la interfaz que muestra al usuario los procesos internos. El núcleo cumple por si solo y de forma independiente con los objetivos del proyecto: generar facturas realistas a través de un entorno financiero. El objetivo del segundo módulo es mostrar los eventos producidos internamente en el núcleo sin interferir directamente en la simulación ni en su rendimiento, además de proporcionar una interfaz de interacción con esta de forma directa.

El resultado obtenido del primer módulo corresponde con los objetivos iniciales proporcionando adicionalmente, una arquitectura adecuada y correctamente desarrollada para ampliar fácilmente el proyecto manteniendo la arquitectura actual, y añadiendo, sin demasiadas dificultades, nuevos elementos a la simulación. Además, proporciona un sistema genérico y eficiente de ejecución de procesos de los distintos simulables de forma simultánea en el mismo entorno de la simulación.

En el caso del segundo módulo, cumple también con los objetivos, ya que proporciona una interfaz sencilla y clara para mostrar los eventos que se producen en la simulación, además de proporcionar una gran variedad de ajustes e interacciones para poder adecuar las necesidades a las situaciones requeridas o para adaptarse al estado actual del mundo como, por ejemplo, reduciendo el porcentaje de ocupación de los restaurantes debido a una pandemia.

### 8.1.2 A Nivel personal

A nivel personal, este proyecto ha supuesto un gran avance a nivel organizativo, ya que he tenido que organizarme para hacer un proyecto a largo plazo, el más largo hasta ahora. Este proyecto empezó en octubre y terminó en julio, 9 meses de trabajo constante para desarrollar este proyecto, lo que ha conllevado tener que organizarme adecuadamente para cumplir con los objetivos de las distintas iteraciones de forma completamente autónoma.

Además, en este proyecto he sido el propio director y he tomado todas las decisiones, con el apoyo y el consejo de mi tutor, pero, al fin y al cabo, es un proyecto desarrollado de forma independiente, tomando la mayoría de las decisiones necesarias para continuar con el proyecto y para solucionar los diversos problemas que surgían.

Ha sido un arduo trabajo que ha llevado muchas horas pero que ha tenido sus frutos donde he podido aprender a ser más independiente, organizativo y profesional.

### 8.1.3 A Nivel laboral

A nivel laboral también ha supuesto un enorme avance en lo relacionado al conocimiento de la profesión, tanto en el uso de nuevas tecnologías, como en el desarrollo de una arquitectura adecuada para la escalabilidad de la simulación. Además, he mejorado enormemente mi eficacia a la hora de programar para desarrollar un proyecto grande que cumpla con el objetivo.

Asimismo, he podido usar mi imaginación y gustos para añadir los elementos y aspectos que me parecían interesantes y necesarios para mejorar el realismo del proyecto, de forma adicional al objetivo inicial de generación de facturas y que tenga un valor mayor como proyecto independiente.

Adicionalmente, he desarrollado una visión de futuro a nivel arquitectónico y de programación para prever de futuros problemas antes de tenerlos y solucionarlos con tiempo, además de desarrollar el proyecto en base a unos fundamentos que luego me permitirán desarrollar de forma fluida posibles aspectos futuros que añadir al proyecto, sin tener que cambiar secciones del código ya desarrollado anteriormente, debido a que se desarrollaron teniendo en cuenta todos los aspectos futuros que consideré oportunos.

Por último, he podido desarrollar por primera vez un proyecto de ciertas dimensiones, lo que me ha ayudado a entender ciertos aspectos importantes en la programación, como el trabajo conjunto con otros compañeros en el futuro, la claridad en el código y desarrollar una buena arquitectura, poco rígida y bien definida, que este separada correctamente en módulos para mejorar su escalabilidad, ya que este proyecto no es realmente tan grande cuando lo comparamos con los proyectos que se desarrollan en el negocio. Por ello, he podido comprobar que los aspectos más importantes para que un proyecto de gran tamaño funcione, es la escalabilidad, la claridad en la comunicación y la definición de una arquitectura estable.

## 8.2 Aportaciones

Este proyecto aporta un generador sintético de datos financieros que puede ser usado para el desarrollo de otros proyectos financieros de Big Data, como el realizado en este proyecto conjunto. Además, permite simular distintos puntos de vistas o situaciones en las que se puede encontrar el entorno financiero, adaptando los ajustes a las nuevas necesidades, con lo que se puede obtener resultados de simulaciones reales que tengan en cuentas estas necesidades. Como resultado, se pueden hacer estudios razonables de la situación que podría encontrarse la economía en un futuro con esas características o necesidades, causadas por alguna pandemia o problema a nivel mundial como el Coronavirus actual.

Además, se aporta una arquitectura bien desarrollada para poder ser adaptada a nuevos elementos y eventos que afecten a la simulación, con lo que poder incrementar el realismo de este indefinidamente. Es un proyecto escalable indefinidamente, ya que el entorno empresarial y la economía es un sector inmenso y complicado que se puede reflejar adecuadamente a través de iteraciones que incrementen gradualmente el realismo con la economía real.

## 8.3 Trabajos futuros

Unas de las ventajas que tiene este proyecto, es su infinita escalabilidad, debido al tamaño y a la complejidad del sector. Por ello, se pueden extender el proyecto en una gran variedad de direcciones. Podemos destacar las siguientes mejoras:

- **Inversiones**: Un aspecto importante en la economía que carece actualmente el proyecto y que sería interesante añadir son las inversiones. Estas inversiones se producen para mejorar la calidad del producto o servicio ofrecido, como la compra de nuevo locales, mejoras de la infraestructura a usar… etc.
- **Extensión a nuevos sectores**: este proyecto se ha centrado en el sector de la restauración debido a la complejidad del sector de la economía y negocios. Un gran avance sería extender este proyecto a sectores relacionados a la restauración, como el sector turístico o el de la construcción.
- **Añadir más interacciones y factores**: este proyecto se ha centrado en este punto añadiendo el máximo de realismo a esta simulación. Aun así, se puede seguir numerosas mejoras como la adición de factores como los días festivos o las vacaciones, así como añadir un gobierno y una hacienda que controle la economía como un agente más.
- **Mejorar la comunicación Simulación-Usuario**: actualmente en el proyecto, la simulación se comunica con el usuario a través del entorno de usuario, usando un panel que muestra en texto todos los eventos que se han producido

relacionados con los agentes que se siguen. Además, se muestran otros datos importantes como el número de agentes de cada tipo. Sin embargo, se puede mejorar la simulación añadiendo más interacciones directas con simulables concretos que se seleccionen o añadir nuevos datos genéricos que se muestren al usuario, como el porcentaje de paro de los trabajadores o el precio medio de un producto.

# 9. Bibliografía

[1] D. Carbajo Vasco. “La factura electrónica y su legislación desde el punto de vista fiscal”. *Crónica Tributaria,* nº 121, pp. 11-30, 2006

[2] F. Piedra Herrera, F. (dir.) *Contabilidad Financiera volumen II*, Madrid: Delta Publicaciones Universitarias, 2008

[3] McGraw-Hill, «La facturación», ultimo acceso: 2 de mayo de 2020

https://www.mheducation.es/bcv/guide/capitulo/8448614194.pdf

[4] Wikipedia, «CFDI», ultimo acceso: 3 de mayo de 2020

https://es.wikipedia.org/wiki/CFDI

[5] Wikipedia, «Fintech», ultimo acceso: 5 de mayo de 2020

https://es.wikipedia.org/wiki/Tecnolog%C3%ADa_financiera

https://www.condusef.gob.mx/Revista/index.php/usuario-inteligente/educacion-financiera/763-que-son-las-fintech

[6] Jonathan Hedley, «Librería Jsoup», [ultimo acceso: 25 de octubre de 2019]

https://jsoup.org/

[7] Whiteboard Technologies, «Generador de datos Aleatorios», [ultimo acceso: 29 de marzo de 2020]

https://www.onlinedatagenerator.com/

[8] TripAdvisor LLC, «Tripadvisor», [ultimo acceso: 30 de octubre de 2019]

https://www.tripadvisor.es/Restaurants-g187471-Gran_Canaria_Canary_Islands.html

[9] The jQuery Foundation, «JQuery», [ultimo acceso: 12 de junio de 2020]

https://jquery.com/

[10] DigitalOcean LLC, «DB Browser for SQLite», [ultimo acceso: 27 de noviembre de 2019]

https://sqlitebrowser.org/

[11] Danielle Delgado, «Arquitectura Publisher/Subscriber, [ultimo acceso: 3 de julio de 2020]

http://arquitecturasomos4.blogspot.com/2014/12/publish-subscriber.html

[12] Wikipedia, «Sistema Multi-Agente», [ultimo acceso: 3 de julio de 2020]

https://es.wikipedia.org/wiki/Sistema_multiagente

[13] Wikipedia, «Bytecode Java», [ultimo acceso: 3 de julio de 2020]

https://es.wikipedia.org/wiki/M%C3%A1quina_virtual_Java

[14] Oracle, «ThreadPoolExecutor», [ultimo acceso: 23 de junio de 2020]

https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ThreadPoolExecutor.html

[15] Ana Karen Uicab Cardeña, «Ejemplos CFDI», [ultimo acceso: 15 de junio de 2020]

https://facturador.zendesk.com/hc/es/articles/115012733308-Ejemplo-de-CFDI-Emisi%C3%B3n-3-3

[16] IBM, «JDBC», [ultimo acceso: 26 de junio de 2020]

https://www.ibm.com/support/knowledgecenter/es/SSGU8G_11.70.0/com.ibm.jdbc_pg.doc/ids_jdbc_011.htm

[17] The Apache Software Foundation, «Tomcat», [ultimo acceso: 1 de julio de 2020]

https://tomee.apache.org/

[18] Ricardo Moya, «Tutorial Básico Ajax con JSP», [ultimo acceso: 20 de mayo de 2020]

https://jarroba.com/ajax-con-jsp-y-servelts/

# 10. Índice de imágenes