



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADO EN INGENIERÍA DEL SOFTWARE

Curso Académico 2017/2018

Trabajo Fin de Grado

**Arquitectura, diseño y configuración de un
simulador de sistemas de bicis compartidas**

Autor: Carlos Ruiz Ballesteros

Tutor: Holger Billhardt

*Se lo dedico a mi padre y a mi madre
por su apoyo y dedicación
en todo lo que he hecho
a Tao y a Sandra por tan buenos momentos
y a todas las personas que,
sin a veces darme cuenta,
han estado ahí*

Resumen

Imaginemos por un momento que necesitamos ir de un sitio a otro y disponemos de una estación de bicis cerca. En ese momento nos acordamos de la aplicación que instalamos el otro día que nos recomendó un amigo, para obtener descuentos y premios utilizando el sistema de bicis.

Al abrir la aplicación, nos damos cuenta de que se está dando una competición en la que el que deja la bici en zonas clave (estaciones con falta de bicis), recibe una puntuación en función del numero de bicis que necesita la estación para estar equilibrada. El que más puntos gana a lo largo de la semana, se lleva una entrada para el próximo partido de fútbol de nuestro equipo preferido, el segundo recibe descuentos en el sistema y el tercero consigue un descuento para una cena en un restaurante conocido, así que decidimos participar.

En otra parte, otro usuario ha conseguido una participación en un sorteo por haber cogido la bici en una zona congestionada y otro usuario acaba de recibir un descuento por haberla dejado cerca de un estadio de fútbol, en el cual está a punto de terminar el partido que se está disputando. El arbitro pita el final, y muchos de ellos van a la estación que hay cerca de la salida del estadio. La gran mayoría han conseguido coger su bici e irse a sus casas, y no han pasado ni 2 minutos y ya hay alguien devolviendo una, y acaba de conseguir la mayor puntuación del día para otra competición.

Mientras sucede eso nosotros hemos llegado a nuestro destino, era una zona con pocas bicis, pero no muchas, por lo que no hemos recibido puntos, pero hemos entrado en la participación en un sorteo de 2 entradas de cine gratis.

Si bien esto puede sonar un poco ridículo ya que esto no sucede ahora mismo en la realidad, pero son este tipo de situaciones y sistemas los que queremos probar para comprobar su viabilidad.

Esto es muy difícil de probar directamente en el sistema, ya que habría que implementar mucha lógica, servidores e inversión en premios, sorteos y descuentos, por lo que puede que es necesario probar la viabilidad de ciertos sistemas de recomendación e incentivación.

Pero esto no es imposible de testear su viabilidad si para ello desarrollamos un **simulador** capaz de probar estos sistemas de recomendación y algoritmos de balanceo para mantener el sistema de bicis activo la mayor parte del tiempo para todos los usuarios en todas las estaciones.

En eso se centra este trabajo, en la implementación de un simulador capaz de simular de la manera más realista posible, un sistema de bicis compartida, con el que simular situaciones de la vida real, con diferentes tipos de usuarios, sistemas de recomendaciones, incentivos... Para probar la posible efectividad y balanceo en los sistemas de bicis compartidas.

Se crearán archivos de configuración para crear situaciones con estaciones y ciudades reales. Posteriormente tras esta simulación se podrán analizar los datos obtenidos y visualizar para comprobar su eficacia. Incluso se creará una interfaz de usuario para facilitar la creación de estas situaciones reales.

Estos sistemas de recomendaciones serán externos al simulador, pudiendo este integrar cualquier sistema de recomendaciones siempre que cumpla una interfaz de comunicación. Veremos paso a paso el diseño de un software grande y complejo, que nos ayudara a probar algo que tan difícil es de hacer a gran escala en la realidad.

Contenido

Resumen	IV
1 Introducción	1
1.1 Motivación	1
1.2 Contexto	3
1.3 Objetivos.....	4
1.4 Metodología.....	5
2 Descripción informática	9
2.1 Especificación de requisitos.....	9
2.1.1 Perspectiva general del producto	9
2.1.2 Definición de acrónimos y abreviaturas.	9
2.1.3 Tipos de usuario:.....	10
2.1.4: Requisitos funcionales	10
2.1.5: Requisitos no funcionales.....	13
2.2 Análisis	13
2.2.1 Núcleo.....	14
2.2.2. Usuarios y sistema de recomendaciones.	16
2.2.3. Configuración.....	18
2.2.4 Generación de usuarios con distribución de Poisson.....	21
2.3 Diseño	26
2.3.1 Arquitectura general.....	26
2.3.2 Simulador basado en eventos.....	27
2.3.3 Arquitectura detallada	29
2.4 Implementación	34
2.4.1 Tecnologías	34
2.4.2 Lógica del simulador (Backend)	36
2.4.3 Carga de mapas y cálculo de rutas	43
2.4.4 Formularios dinámicos configuración(Frontend)	45
3 Evaluación.....	47
3.1 GUI	47
3.2 Prueba simulador.....	50
4 Conclusiones.....	53

4.1 Lecciones aprendidas	53
4.2 Líneas futuras.....	54
Apéndice A.....	57
Apéndice B.....	63
Apéndice C.....	67
Apéndice D.....	71
Apéndice E	76
5 Referencias	77

1 Introducción

En este capítulo se darán a conocer las motivaciones que llevaron a la realización de este proyecto, el contexto de la misma y los objetivos.

1.1 Motivación.

El proyecto consiste en la realización de un simulador de sistemas de bicis compartidas en entornos urbanos, como por ejemplo BiciMad¹ en Madrid, o Vélib² en Francia.

La necesidad de alentar a las personas a utilizar vías alternativas de transporte a las comúnmente usadas es cada vez más necesario. El aumento de la población en las grandes ciudades, el aumento de CO₂ son hechos que nos obligan a pensar como cambiar y/o mejorar nuestros medios de transporte público.

Los sistemas de bicis compartidas instaladas en las grandes ciudades son una muy buena opción a la hora de buscar alternativas de movilidad. Estos sistemas permiten a los ciudadanos moverse por distintos puntos de la ciudad alquilando una bici de cualquiera de las estaciones disponibles y devolviéndola en otra estación diferente.

Pero el problema no solo se centra en promover el uso de estos sistemas, sino que va más allá. Uno de los más importantes es el balanceo de bicicletas entre estaciones. Algunas situaciones que pueden producir este desequilibrio son:

- Eventos especiales.
- Horas punta.
- Aglomeraciones inesperadas.

Para evitar estos problemas es necesario tener un control sobre los recursos disponibles, manejarlos y distribuirlos de la manera más eficiente posible. Llevar a cabo nuevas ideas en un sistema de bicis y probarlo en la realidad, puede ser muy tedioso y costoso.

¹ BiciMad: <https://www.bicimad.com/>

² Vélib: <https://www.velib-metropole.fr/>

Es por esto por lo que surge la necesidad de un simulador, con el que podamos poner a prueba todos estos tipos de sucesos, crear distintos tipos de usuarios, probar sistemas de recomendaciones o de incentivos, para que los usuarios puedan ayudar a balancear el sistema, etc.

Un simulador es útil para el estudio de estos problemas donde se pueda probar en cualquier ciudad del mundo distintos sistemas, algoritmos e ideas posibles de implementar para incentivar su uso, mejorar el sistema, o incluso llevar los sistemas de bicis compartidas a otras ciudades.

Por otro lado, no solo está la posibilidad de utilizar el simulador como una herramienta para sacar conclusiones respecto al funcionamiento del sistema en sí, sino que además cabe la posibilidad de analizar el comportamiento de diferentes tipos de usuario en este tipo de sistemas. Para ello es necesario que se pueda añadir a nivel de código usuarios para qué cualquier persona con conocimientos de programación pueda implementar sus propios algoritmos de comportamiento dentro del simulador.

Adicionalmente, no debería ser un simulador único e invariable, sino que debe ser posible su modificación y adaptación a las necesidades de cada investigación, pero todo dentro del ámbito de los sistemas de bicis compartidas.

1.2 Contexto

Para definir las diferentes partes de nuestro simulador, es necesario tener claro una visión general de la infraestructura física del sistema de bicis en la realidad.

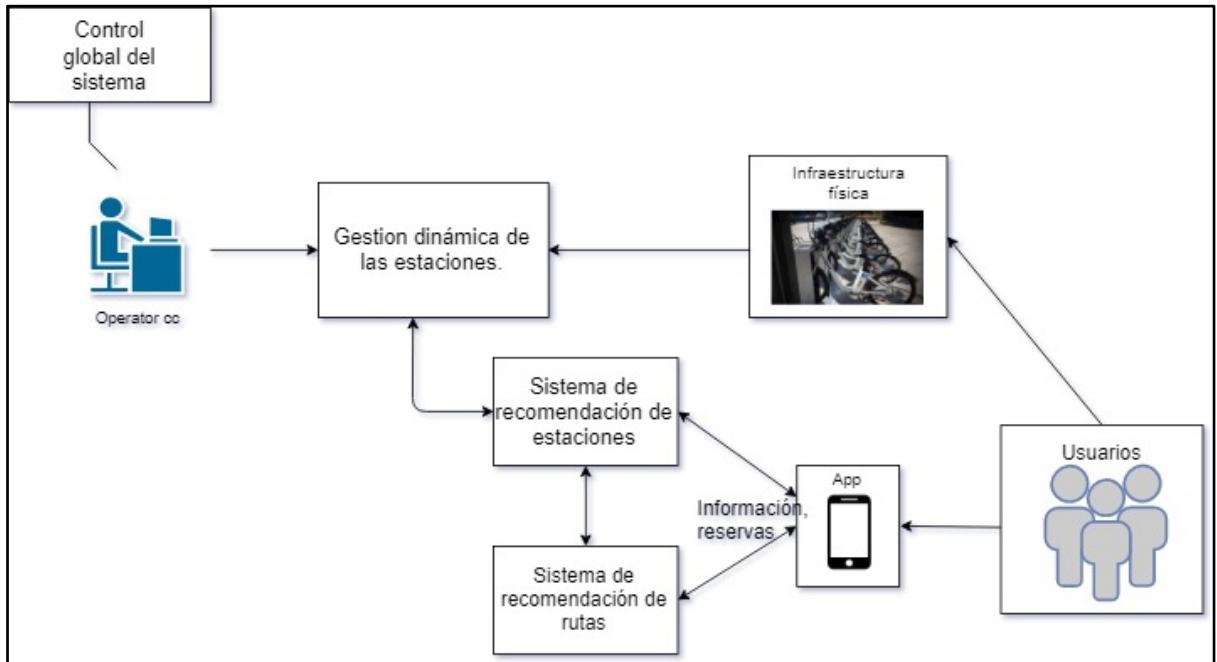


Figura 1. Sistema de bicis compartidas.

Podemos distinguir claramente en la *Figura 1* tres partes principales dentro del sistema de bicis compartidas:

- Infraestructura física (estaciones y bicis)
- Usuarios con Smartphone y App del sistema
- Sistema de recomendaciones³.

Los usuarios hacen uso de la infraestructura cuando cogen o dejan una bici y también hacen uso del sistema de recomendaciones a través de la App, para reservar una bici, un hueco, ver el estado de la estación, o quizás hacer caso a una de las sugerencias de la App.

³ Nótese en la Figura 1 que el sistema de recomendaciones se basa en la información global de los recursos del sistema.

Vemos que hay una interacción continua, entre usuarios, infraestructura física y no solo eso, si no que el sistema de recomendaciones puede influenciar en la decisión final del usuario.

En este simulador hemos participado varias personas en su desarrollo hasta el día de la fecha de publicación de esta memoria. Se pueden distinguir varias zonas importantes del desarrollo del simulador, aunque parte de este desarrollo ha sido rea- lizado de forma conjunta, debido a la necesidad de tener una base común y están- dares definidos para nuestro simulador.

Con esta visión global del sistema, debemos centrarnos en que objetivos necesitamos antes de analizar cómo tiene que ser nuestro simulador, que queremos probar, de que información disponemos, etc.

1.3 Objetivos

El propósito de este TFG es bastante variado. Antes de plantear que objetivos perseguir, hay que analizar que necesidades va a tener nuestro simulador a futuro, y con qué fin se va a utilizar en términos globales además de plantear una arqui- tectura versátil y adaptable a cambios. Se puede ver en más detalle los requisitos de nuestro TFG en la sección 2.1.

Podríamos distinguir varios objetivos si hablamos de la totalidad del simulador:

- Probar algoritmos de balanceo y predicción de demanda en un sistema de bicicletas compartido.
- Implementar sistemas de recomendaciones e incentivos para optimizar la demanda.
- Plantear nuevas distribuciones de estaciones sobre una ciudad.

Al igual que antes hemos analizado los objetivos globales, vamos ahora a desglosar los objetivos concretos que debe cumplir nuestro trabajo:

- Recrear infraestructuras reales en ciudades reales
- Generar usuarios en cualquier punto de la ciudad.
- Generación de usuarios versátil y que puedan seguir distribuciones (Pois- son), un máximo de usuarios o un rango de tiempo.
- Poder definir qué tipos de usuarios queremos en nuestro sistema y para- metrizarlos para que puedan tener comportamientos variados.
- Una interfaz de usuario capaz de realizar esta configuración.

- Creación de distintos tipos de usuario, con diferentes comportamientos que respondan de forma distinta a los eventos dados y las recomendaciones.
- Simulación realista con respecto a la realidad.
- Análisis de los datos, para probar los diferentes algoritmos.

Todos estos objetivos tienen algo en común: nada podría simularse sin un primer estado del sistema. Necesitamos una **configuración previa** que nos permita realizar todas esas pruebas.

En general el simulador tiene que ser capaz de recrear situaciones reales, en entornos reales, con infraestructuras existentes o que puedan existir en el mundo real. Es decir, uno de los objetivos primordiales es dotar al simulador de dinamismo a la hora de configurar las simulaciones que se desean realizar.

Además, esta configuración tiene que poder ser generada por cualquier software independientemente del simulador, añadiendo la posibilidad de que nuevos desarrolladores puedan crear sus propias herramientas que generen casos para la simulación y ofreciendo una interfaz agradable para la realización de simulaciones.

Dentro de todas estas posibilidades, el módulo que nos atañe es el de **configuración**⁴. Además del sistema de recomendación de rutas y algunos patrones de diseños aplicados al simulador, para una fácil modificación, adición y mantenibilidad del simulador.

En este TFG solo vamos a comentar las partes antes mencionadas. Sobre la implementación y temas de arquitectura y diseño, véanse las secciones 2.2 y 2.3.

1.4 Metodología.

Este software se ha realizado en un grupo de varias personas, por lo que necesitamos de una metodología para organizarnos. Podríamos considerar que estamos utilizando Scrum [1], pero para los más puristas en cuanto a metodologías software no sería considerado como tal, ya que utilizamos una estructura organizativa horizontal, que suele ser más propio de empresas que venden su propio producto software o, como es el caso, en desarrollos de software para investigación. El equipo de desarrollo tiene un contacto directo con el cliente (que serían nuestros tutores de proyecto) y hay casi una comunicación total día a día con ellos, sin roles intermediarios

⁴ En la sección 2.2 se analizan las diferentes partes del simulador.

de por medio⁵. Sin embargo, sí que se tienen reuniones cada semana en el equipo para ver cómo avanza el proyecto, retrospectivas, prototipos, integración, pruebas... No obstante, como no aplicamos todas las reglas de Scrum, consideraremos que el desarrollo se está realizando con una metodología iterativa e incremental tal y como se muestra en la Figura 2.

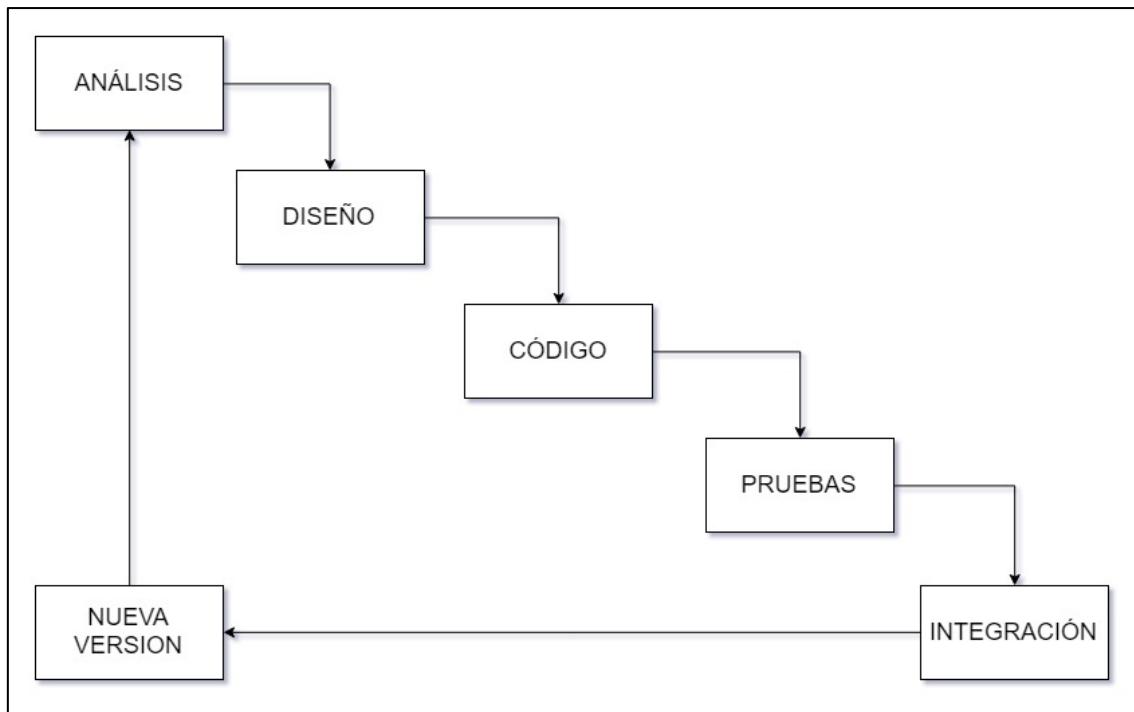


Figura 2: Ciclo iterativo e incremental

En este modelo, primero se realiza un análisis de los requisitos que se van a necesitar para cada iteración. Después del desarrollo de estos, se hacen pruebas y para finalizar se integra con el resto sistema.

Cada 2 semanas realizamos una iteración donde se realizan todos los pasos comentados anteriormente, donde todo el equipo de desarrollo decide qué requisitos son más críticos e importantes, bugs, y releases. Esto junto a una serie de herramientas online como Trello⁶, podemos gestionar que tareas está realizando cada uno.

⁵ En una aplicación de la metodología de Scrum pura, existirá un responsable dentro, encargado de hacer comunicar las cosas entre el dueño del producto y el equipo de desarrollo que se asegura de que Scrum es bien aplicado

⁶ Es un tablero online donde se pueden crear, asignar y clasificar tareas, de tal modo que todo el equipo tiene una visión global del estado actual de desarrollo del software que se está creando.

Un desarrollo iterativo e incremental ofrece varias ventajas con los modelos aplicados con anterioridad a este como el modelo de cascada, con la entrega de un software usable a mitad de desarrollo que ofrece al cliente poder aprovechar desde el principio funcionalidad más importante. En un modelo en cascada cada fase del proceso debe ser finalizada (firmada) para pasar a la siguiente fase, y así hasta llegar a una versión final del producto. Sin embargo, el desarrollo de un software no es lineal y esto crea dificultades con esta metodología. [2]

Cada cierto tiempo realizamos una reléase. Utilizamos un formato de versiones semántico [3] del tipo X.Y.Z donde X, Y y Z son números enteros mayores que 0.

X se corresponde a la versión mayor (cambios grandes que modifican parte o gran parte de la funcionalidad). Y se corresponde a la versión menor (pequeños cambios pequeños, corrección de bugs) y Z, que son micro versiones (parches, pequeños bugs críticos...).

2 Descripción informática

2.1 Especificación de requisitos.

Antes de comenzar las iteraciones y primeros prototipos de nuestro proyecto debemos tener una especificación de requisitos clara y concisa. Vamos a seguir algunas de las recomendaciones del estándar IEEE830 [4] para ello. En un desarrollo iterativo e incremental ágil debemos tener muy en cuenta que los requisitos puedan ser lo más modificables posibles.

2.1.1 Perspectiva general del producto

Para examinar y definir en detalle las distintas especificaciones de nuestro simulador, debemos introducir un poco las necesidades globales del simulador en general.

Necesitamos de un software que sea capaz a partir de un estado inicial del sistema de simular situaciones reales y ser capaz de visualizar y dar una serie de datos correspondientes a la simulación. El simulador tiene que mostrar estos datos de la simulación a través de un histórico que posteriormente se utilizará para analizar los resultados del sistema de recomendaciones y los algoritmos implementados.

En resumen, el simulador tiene que ser capaz de a partir de unos datos de configuración iniciales, generar un histórico con los resultados de la simulación para probar distintos algoritmos de balanceo en este tipo de sistemas.

La parte de nuestro TFG es la de configuración que será diseñado con el objetivo de inicializar el SBC utilizando la ubicación de una ciudad real, poder generar usuarios en zonas especificadas, parametrizar valores globales de la simulación, ubicar estaciones en cualquier punto y parametrizar también qué tipos de usuarios se van a utilizar en la simulación, en qué zonas y con qué distribución aparecerán.

Pero no solo la **configuración**, sino también la separación en módulos del simulador (2.4.2), la aplicación de ciertos **patrones de diseño** y la implementación de la parte gráfica de **configuración y simulación**.

2.1.2 Definición de acrónimos y abreviaturas.

En esta sección se describirán los términos y abreviaturas utilizados para la especificación de requisitos, así como lo exige el IEEE830

- **Sistema de bicis compartidas (SBC):** Infraestructura (estaciones, bicis).
- **Sistema de recomendaciones (SDR):** Parte del simulador encargado de recomendar a los *usuarios* estaciones con la finalidad de *balancear* el sistema.
- **Configuración:** Siempre hace referencia a la parte de configuración de nuestro proyecto.
- **Usuario simulado (US):** Agente simulado que interactúan dentro del sistema de bicis compartidas y que pueden hacer uso del sistema de recomendaciones.
- **Alquilar:** Acción que realizan los usuarios simulados al coger una bici.
- **Reservar:** Acción que realizan los usuarios al reservar un hueco o una bici antes de llegar a una estación.
- **Histórico:** Resultado de una simulación.
- **Interfaz de usuario (GUI).**

2.1.3 Tipos de usuario:

Solo tendríamos un tipo de usuario, que serían los investigadores, profesores o cualquier persona que quiera hacer uso de nuestro simulador. Podrán o no tener conocimientos de programación, pero los usuarios con conocimientos de desarrollo podrán modificar y crear usuarios de una forma mucho más precisa dentro del código fuente.

2.1.4: Requisitos funcionales

En esta sección se expondrán a grandes rasgos los requisitos funcionales del sistema. Nótese que estos requisitos han ido cambiando a lo largo del desarrollo iterativo e incremental. Estos requisitos sólo entran dentro de nuestro TFG:

Requisito funcional 1

Fichero de configuración global del SBC: La simulación deberá partir de una serie de parámetros globales en un fichero de texto. Partiremos de las siguientes necesidades, aunque puedan cambiar, quitarse o añadirse según el progreso y uso del simulador. El fichero de configuración deberá especificar:

- Un parámetro con el cual poder realizar siempre las mismas simulaciones de tal modo que los sucesos aleatorios sean los mismos en el momento en que se ejecuten.
- Parámetro de tiempo total de simulación.

- Un área donde sucederá la simulación y así mismo hacer uso de un mapa real en la tecnología que se desee.

Requisito funcional 2

Fichero de configuración de estaciones: Se podrá mediante un fichero de configuración, disponer para la simulación de un conjunto finito de estaciones. El archivo de configuración deberá especificar:

- Las bicis disponibles en las estaciones.
- Número total de huecos en cada estación
- Punto geográfico de su ubicación real o ficticia.

Requisito funcional 3.1

Fichero de configuración de entrada de los US en el SBC: La configuración deberá proporcionar un mecanismo con el cual se puedan generar usuarios en distintos puntos geográficos. De momento los más importantes son:

- Distribución exponencial (Poisson)
- Único US.

Además, pueden ser de cualquier tipo, distribución o regla, simplemente deben generar usuarios para la simulación y si es posible, deberán disponer de un mecanismo por el cual los usuarios puedan ser generados en un punto y radio dados.

Requisito funcional 3.2

Configuración de usuarios independiente: La configuración de entrada de los US en el SBC debe ser independiente del simulador, es decir, deberá generar un fichero con los usuarios que van a aparecer en la simulación.

Por lo tanto, habrá dos ficheros de configuración:

- Fichero de configuración con puntos de entrada y distribuciones.
- Fichero de configuración con usuarios generados según el fichero con puntos de entrada.

Los US podrán o no tener parámetros de configuración que modifiquen su comportamiento de facto. Los parámetros dependerán del tipo de US que se quiera implementar.

Requisito funcional 3.3

Generador de usuarios: Del requisito anterior podemos deducir que deberá haber un generador de US que reciba el fichero de configuración con puntos de entrada y distribuciones y nos genere un fichero de configuración con US siguiendo dichas distribuciones o reglas definidas.

Requisito funcional 4

Procesador de la configuración: En el simulador deberá haber un procesador para las configuraciones descritas que sea capaz de preparar todo el sistema para su correspondiente ejecución.

Requisito funcional 5

Gestor de rutas: Los US deberán tomar rutas reales y decisiones basándose en el mapa y la situación del SBC. Estas rutas serán posteriormente guardadas en el histórico para su visualización.

Requisito funcional 5.1

Herramientas GUI: La GUI deberá proporcionar las siguientes herramientas:

- Crear y cargar configuraciones
- Visualizar históricos
- Analizar y exportar datos.

Requisito funcional 5.2

Crear y cargar configuraciones: La herramienta encargada de la creación y carga de configuraciones en la GUI, deberá poder crear configuraciones a través de un mapa y además para hacerlo accesible también se podrá a través de botones que den opción a ello.

Requisito funcional 5.3

Crear y cargar configuraciones: Los elementos de la configuración que se vayan añadiendo, deberán verse en un mapa y en una vista en forma de árbol para que sea accesible.

2.1.5: Requisitos no funcionales

Requisito no funcional 1

Interfaz de usuario dinámica: Al añadir o quitar parámetros de configuración a los usuarios, añadir o quitar tipos de usuarios, los formularios de la GUI para configurar la simulación deben ser lo más dinámicos posibles, para agilizar el desarrollo.

Requisito no funcional 2

El simulador debe ser multiplataforma, pudiendo así utilizarlo y desarrollarlo en las plataformas de GNU/Linux, Windows y MacOs.

Requisito no funcional 3

El diseño del simulador y el código debe ser lo más sencillo posible y aportar facilidades a la hora de añadir, modificar o alterar implementaciones de usuarios y de parámetros de configuración, así como de métodos de generación de usuarios.

Requisito no funcional 4

Tanto el formato del fichero de configuración como el del histórico deben ser independientes, es decir, la configuración podrá ser creada y el histórico leído por otro software independientemente del simulador. Se utilizará el formato JSON y deberá contar de un sistema para la verificación de datos.

Más adelante en la sección 2.2 veremos cómo enfocar estos requisitos en la parte de configuración y en la sección Diseño y su correspondiente diseño e implementación.

2.2 Análisis

Para ir formando poco a poco las partes de nuestro simulador, vamos a partir de lo siguiente:

- 1 En el sistema de bicis solo se pueden realizar ciertas **acciones**.
- 2 Los usuarios pueden elegir entre esas **acciones** posibles dependiendo de su **estado**, que tipo de usuario sea, y el **sistema de recomendaciones**.

- 3 El sistema de bicis se encuentra en un lugar en específico y, si se tienen datos anteriores del sistema, se pueden deducir distribuciones de aparición de los usuarios en el sistema. Por lo cual contamos con un **estado inicial**.
- 4 Cada persona que utiliza el sistema es distinta, el uso del SBC es diferente para cada tipo de usuario.

Partiendo de esta pequeña analogía, con los puntos anteriormente citados vamos a ir mencionando las distintas partes de nuestro simulador que se corresponden con cada punto anterior en el SBC real:

- 1 **Núcleo:** Si sólo podemos realizar ciertas acciones, aquí definiremos las reglas de nuestro sistema de bicis, y las interacciones de los usuarios con la infraestructura. En la sección 2.2 y 2.3 se explica cómo se definen estas reglas (simulador basado en eventos).
- 2 **Sistema de recomendaciones:** Con esto le damos la posibilidad al usuario de elegir entre las opciones que más le favorezcan a él o al sistema.
- 3 **Configuración - Inicio del Simulador:** Antes de poner en marcha nuestro simulador, necesitamos definir la infraestructura y cómo van a aparecer los usuarios dentro de él. Es por ello por lo que, para definir el estado inicial, necesitamos un módulo de configuración.
- 4 **Implementaciones de usuarios:** Como cada persona del SBC en el mundo real es diferente, necesitamos hacer que nuestra simulación sea lo más realista posible e implemente diferentes tipos de US.

A continuación vamos a explicar de forma sintetizada el modelo que se ha seguido para cada una de las partes de nuestro simulador.

2.2.1 Núcleo.

Como hemos mencionado con anterioridad, en el núcleo definiremos las reglas de nuestro sistema. A fin de poder definir estas reglas a nivel teórico, necesitamos definir en un diagrama de flujo las decisiones posibles del usuario dentro del sistema:

Este es el flujo de eventos que deberá seguir cualquier usuario del sistema. Las decisiones que tomará el usuario para realizar unas acciones u otras vendrán determinadas por el estado del SBC y por las implementaciones concretas de los usuarios implementados. En la sección 2.3.2 veremos que toda esta lógica será implementado como un simulador por eventos discretos.

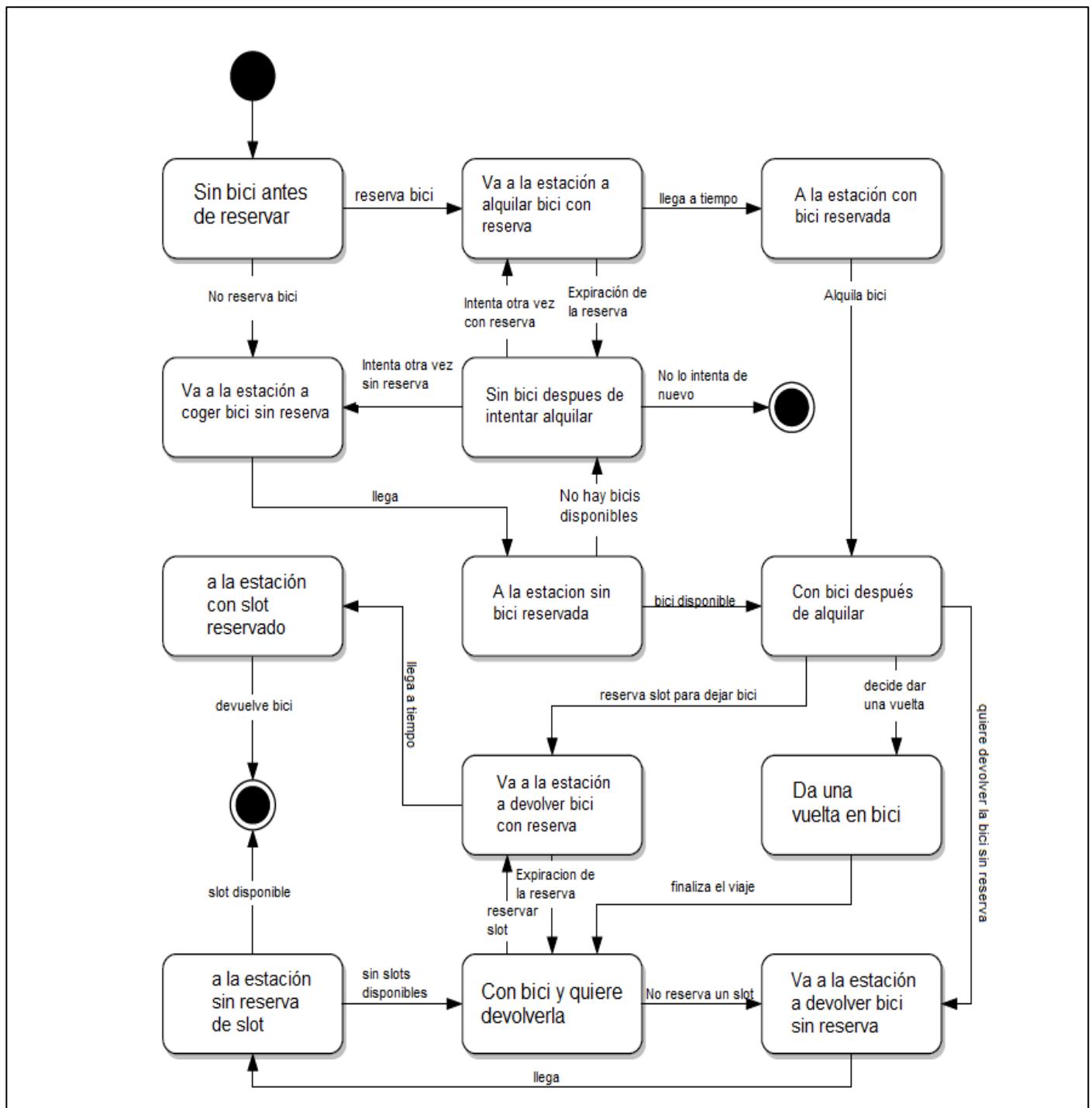


Figura 3 - Flujo de decisiones de un US

2.2.2. Usuarios y sistema de recomendaciones.

Los US interactuarán con el SBC en el núcleo, pero ¿en base a que tomarán decisiones? Si nos fijamos en el flujo de decisiones, muchas de los posibles estados por los que puede pasar el US dependen de si reserva, no reserva, si decide volver a intentarlo después de un intento fallido, a que estación decide ir...

Si pensamos por un momento en una persona utilizando el SBC en la realidad, vemos que su información del sistema es limitada sin el uso de un dispositivo smartphone. Pero si el usuario dispusiera de uno y de una aplicación que le proporcionara información, puede tener un amplio conocimiento actual del SBC y además seguir consejos, recomendaciones de la información proporcionada, etc.

Por lo tanto, los US solo pueden obtener información del sistema por una vía. Esta vía hará uso del Sistemas de recomendaciones (SDR), para cada tipo de usuarios, que les ayude a tomar decisiones y sean además beneficiosas para el SBC.

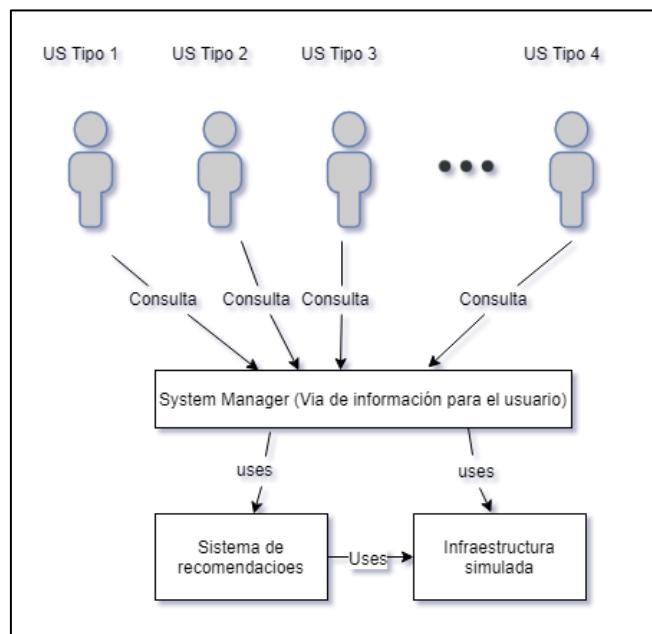


Figura 4 Diagrama Usuarios simulados utilizando la información del SDR y la infraestructura

En la Figura 4 vemos un diagrama en el que se explica cómo los US podrían tomar decisiones en el sistema, según el momento de la simulación en el que se encuentren. Cabe destacar de esta figura dos cosas en cuanto a los US:

- Tipo de usuario
- Consultas de los usuarios al System Manager

- System Manager puede ser usado externamente tanto para usar el SDR⁷ como para ver información actual de la infraestructura de bicis.

El System Manager bien podría ser un intermediario que le proporciona al usuario información del sistema, según la vaya necesitando, tanto del estado del SBC o recomendaciones del SDR.

Supongamos una situación en la que un usuario tiene que decidir dar una vuelta o no. (Esta decisión se puede ver en la *Figura 3*). Y supongamos además tres tipos de usuarios, un usuario altruista (TA), un usuario deportista (TD) y un usuario ocasional (TO).

Supongamos por un momento que el sistema de recomendaciones funciona. En un primer instante, el usuario TA decide ir a la estación con más bicis para posteriormente devolverla en la estación que menos bicis tiene equilibrando así el sistema. El usuario TD estará dispuesto a dar una vuelta tras coger la bici y no ir a una estación en concreto mientras que el usuario TO no querrá hacer eso e irá directamente a la estación más cercana. Al final el usuario TD dejará la bici en una estación lejos de su casa para poder hacer más ejercicio en la vuelta.

¿Qué factores determinan las decisiones de nuestros usuarios?

Usuario TA:

- Decide ir a estaciones para equilibrar el sistema. Es decir, su tipo, **el tipo de usuario que es, le ha hecho tomar esa decisión**. El SDR en este caso no influye en esa decisión
- Va a las estaciones en peor situación dentro del sistema. **Es el sistema de recomendaciones a través del System Manager, el que le da esa información y la toma como correcta para cumplir su cometido.**

Usuario TD:

- Decide dar una vuelta, por el tipo de usuario que es. Aquí no es el SDR el que influencia al usuario.
- Decide dejar la bici en una estación lejos de su casa por lo que estará usando el System Manager para conocer cual es la estación mas alejada de su hogar.

⁷ Hay que aclarar que el sistema de recomendaciones puede ser un modulo dentro del simulador que infiera recomendaciones en base a la información que posee, un servicio externo, cualquier tipo de software siempre que cumpla su cometido y tenga acceso a la infraestructura del sistema en todo momento.

En este caso puede, o no, ser una recomendación del sistema. Puede que el usuario solo haya consultado cierta información actual del SBC. Es por eso por lo que a veces el usuario puede tomar decisiones en base al estado global de la infraestructura física a través del System Manager como se ve en la Figura 4.

Usuario TO:

- Decide no dar una vuelta con la bici. Esa decisión es implícita dentro de este tipo de usuario.
- Decide ir a la estación más cercana. Para ello utilizará el SDR.

Como podemos observar, las decisiones de los usuarios se ven influenciadas directamente por el tipo de usuario que es y por el SDR o el estado del SBC en ese momento.

Pero, sin embargo, esta forma de decidir de los usuarios es una analogía muy acertada si la comparamos con la realidad. Es lógico pensar que cada tipo de usuario reaccionara dentro del sistema de una u otra forma, pero podríamos **modificar ese sistema de recomendaciones para que el usuario decida, en última instancia, tomar una decisión que beneficie al SBC por completo**. Tal y como se mencionaba en la sección 1.1, una de las motivaciones es esa, encontrar formas de hacer que los usuarios cambien su decisión por una decisión buena tanto para el usuario como para el sistema, con el uso de incentivos.

2.2.3. Configuración

En los requisitos 1, 2 y 3 se puede deducir que necesitamos varios archivos de configuración. Estas configuraciones deberán poder ser legibles y modificables a nivel de texto, por lo que utilizaremos la notación JSON. Los ficheros serían los siguientes.

- Fichero de configuración de estaciones:
- Fichero de configuración global.
- Fichero de puntos de entrada de generación de usuarios.
- Fichero de US independientes, proveniente del fichero anterior.

Como se puede ver tenemos dos ficheros para los usuarios, uno de puntos de entrada y otro de usuarios independientes. Esto se debe a una decisión de diseño que es explicada con profundidad en el apartado 2.3.3. Esta decisión se basa en la necesidad de tener un simulador que simplemente reciba usuarios individuales y no puntos

de entrada que estos tengan que ser procesados antes por el simulador. Se detalla en el apartado mencionado anteriormente la necesidad de un módulo generador de usuarios.

A continuación, se muestran los datos que debe contener cada uno de los ficheros de configuración. En el

Apéndice A, se muestra un ejemplo de configuración y sus esquemas correspondientes de verificación para una mayor clarificación.

El contenido del archivo de **configuración de las estaciones** de momento es bastante simple:

- Posición geográfica de la estación.
- Cantidad de bicis.
- Capacidad de bicis.

Por otro lado, el contenido del **archivo de configuración de parámetros globales** contendrá lo siguiente:

- Tiempo máximo de reservas.
- Tiempo total de la simulación
- Semilla (Para generar los mismos eventos aleatorios)
- Cuadro delimitador (Para delimitar la zona de la simulación).
- Mapa: Recurso con el mapa en cuestión.
- Directorio del histórico: Dónde se van a guardar los resultados de la simulación.

Sobre todos estos parámetros, hay que destacar la semilla. Los sucesos aleatorios que suceden dentro del simulador son en realidad “pseudoaleatorios” ya que parten de un primer valor (semilla), a través del cual una secuencia de números aleatorios es la misma siempre que partan de ese mismo valor. Esta **semilla** es una parte importante de nuestra configuración ya que de ella depende que, de una simulación, se puedan obtener la misma aleatoriedad de una simulación a otra para poder realizar pruebas.

Por otro lado, es importante en la definición de los puntos de entrada de los usuarios como podemos definir la manera en que queremos que estos aparezcan en el mapa y de qué forma. Necesitamos definir un concepto genérico y flexible. Definiremos a lo largo de este documento a este concepto como **Entry Point**.

Definimos como **Entry Point** un punto geográfico del cual aparecerán usuarios de una forma determinada a lo largo del tiempo. Un Entry Point puede tener cualquier tipo de propiedades, y puede aparecer utilizando distribuciones.

En nuestro caso en particular nos interesa que los usuarios sean generados en dichos puntos por una distribución de Poisson y que estén distribuidos dado un radio de forma uniforme en el área abarcado por éste. Un **Entry Point** de estas características podría tener las siguientes propiedades:

- Posición geográfica
- Radio de aparición
- Tipo de distribución y parámetro lambda
- Instante inicial y final.
- Tipo de usuario.

También nos puede interesar un usuario único en un determinado instante de tiempo que podría tener las siguientes propiedades:

- Posición geográfica
- Instante de aparición
- Tipo de usuario.

El objetivo de estos dos pequeños ejemplos es dejar claro que **Entry Point** es un concepto genérico de entrada y pueden variar sus parámetros.

2.2.4 Generación de usuarios con distribución de Poisson.

Una vez tenemos los Entry Points definidos con distribuciones de Poisson necesitamos encontrar un modo de generar nuestros usuarios individualmente en otro fichero, con los instantes de aparición en los que deberían aparecer. Necesitamos computar de alguna manera los instantes de aparición de cada usuario. Para ello primero debemos recordar como es una función exponencial. En las distribuciones exponenciales tenemos la siguiente función de distribución de probabilidad.

$$f(x) = 1 - e^{-\lambda*x} \quad (1)$$

Para poner un ejemplo imaginemos que un usuario aparece cada cinco segundos. Si aparece cada segundo su valor $\lambda = \frac{1}{5}$.

Con esta función lo que obtendríamos es la probabilidad de que un usuario aparezca dado un instante de tiempo x , pero no es suficiente para nuestro problema, podríamos acercarnos a una posible solución si dado un intervalo de tiempo V calculamos la probabilidad de que aparezca un usuario en dicho intervalo, pero si los intervalos

de tiempo son del orden de 1 segundo por ejemplo, el coste computacional puede ser bastante grande [5].

Sin embargo, el doctor Donald Knuth describe una forma de generar estos valores. [6]. Según explica Knuth, siempre que tengamos una distribución continua y su función de distribución de probabilidad $F(x)$ cumpla que:

$$F(x_1) \leq F(x_2), \text{ si } x_1 < x_2; \quad (2)$$

$$F(-\infty) = 0, F(+\infty) = 1 \quad (3)$$

Entonces si $F(x)$ es continua y estrictamente creciente (1), como todos los valores de $F(x)$ toma todos los valores del cero al uno, existirá una función inversa $F^{-1}(y)$ donde $0 < y < 1$.

Con esta función inversa, podríamos calcular un valor aleatorio X por medio de una variable aleatoria de distribución uniforme $u = U(0,1)$,

$$X = F^{-1}(u) \quad (4)$$

Si hacemos la inversa de la distribución de probabilidad exponencial, obtenemos un posible valor de la distribución:

$$X = \frac{-\ln(u)}{\lambda} \quad (5)$$

Donde X serían los segundos que quedan para que aparezca de nuevo un evento, que en nuestro caso sería la aparición de un usuario nuevo.

Gráficamente lo que estamos haciendo es a partir del valor de μ (entre 0 y 1), es calcular su valor correspondiente en tiempo, que falta para el siguiente suceso.

Por ejemplo, si el valor de μ es 0.42 el valor que devolverá es de 2,64 segundos para un $\lambda = \frac{1}{5}$

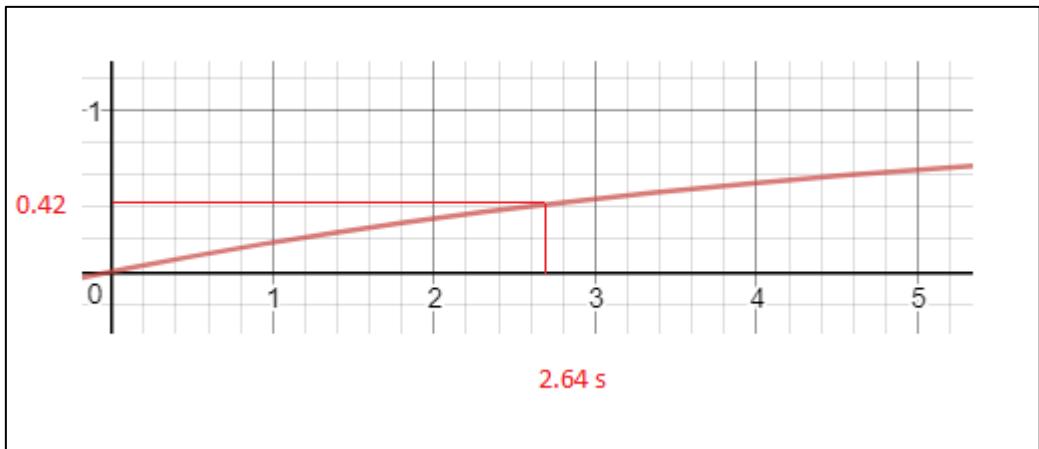


Figura 5 - Ejemplo gráfico de

Ya solo nos quedaría definir el algoritmo para la generación de usuarios siguiendo una distribución de Poisson.

Las variables de entrada de nuestro algoritmo son las siguientes:

- p = Punto geográfica de los entry points.
- r = Radio de aparición de los usuarios.
- e = Tiempo total de generación de usuarios.
- m = Número máximo de usuarios a generar.
- ut = Tipo de usuario. (Usuarios con distinto comportamiento)
- λ = Valor lambda de la distribución de Poisson.

Si no se recibe r , todos los usuarios serán generados en el punto p .

```

POISSON-USER-GENERATOR( $p, r, e, m, ut, \lambda$ ):
1   let L be a new List
2
3    $ct = 0$  //current time
4    $uc = 0$  //users counter
5
6   while (( $ct < e$ ) and ( $uc < m$ ))
7      $uc = uc + 1$ 
8
9     if ( $r > 0$ )
10        $up = RandomPositionInCircle(p, r)$       // User position
11     else
12        $up = p$ 
13
14      $t = -\ln(Random(0,1))/\lambda$     //Apparition time
15      $ct = ct + t$ 
16      $u = NewUser(up, ut, ct)$ 
17      $L.add(u)$ 
18   return L

```

En la línea 14 aplicamos la fórmula deducida en (5). Por cada iteración del bucle (lineas 6-17) se calcula un instante de aparición.

El algoritmo implementado en el proyecto es mucho más completo ya que aquí no se tienen en cuenta muchos más parámetros que puede recibir un Entry Point. De hecho, este algoritmo sería fácilmente parametrizable, como por ejemplo añadir rangos de tiempo de aparición para que un tipo de usuario aparezca a determinadas horas del día en la simulación. El algoritmo se puede ver implementado en el Apéndice B.

2.3 Diseño

A fin de explicar el diseño final obtenido vamos a partir desde un concepto básico y se irán añadiendo partes a la arquitectura analizando las necesidades y objetivos del software en cuestión en cada una de las partes de ésta.

Como hemos mencionado a lo largo de este trabajo, el objetivo principal es tener un simulador de bicis compartidas capaz de, a partir de unos archivos de configuración (estado inicial del sistema), simule una situación real utilizando distintos tipos de usuarios implementados y de la forma más realista posible, utilizando distintas estrategias de balanceo, incentivos y algoritmos para probar la eficacia de estos en un entorno real. Dicha simulación da como resultado un histórico que posteriormente se utilizará para analizar los resultados, y también para poder visualizar la situación en un entorno gráfico.

2.3.1 Arquitectura general

Con este apartado se pretende dar una visión global de la arquitectura antes de ir detallando las distintas partes más a fondo.

Una primera vista de la arquitectura sería la siguiente:



Figura 6 - Vista básica del simulador

A un nivel muy básico necesitamos tener estas cuatro partes diferenciadas:

- Archivos de configuración: En un principio partimos de tres archivos:

- Estaciones: Punto geográfico de las estaciones, numero de bicis, huecos...
- Entry Points: Puntos de entrada de los usuarios...
- Parametros globales: Tiempo total de la simulación, semilla...
- Simulador: De momento contendrá toda la lógica del simulador y sus interfaces de comunicación con el SDR, a partir de la sección 2.3.3 se explica con más detalle. Para leer la sección 2.3.3 es necesario saber que el simulador está basado en una lógica de eventos.
- SDR (Sistema de recomendaciones): Algoritmos de recomendación de rutas, incentivos para los usuarios. Este será el medio de consulta para los usuarios.
- Histórico: Resultado de las simulaciones que posteriormente se analizarán.

Para examinar la arquitectura más en detalle, en los siguientes apartados vamos a ir desglosando esta arquitectura en partes más complejas a partir de la sección 2.3.3.

2.3.2 Simulador basado en eventos.

Existen dos tipos de simuladores:

- Simulador basado en eventos discretos.
- Simulador de tiempo continuo.

En un simulador de tiempo continuo el estado del **sistema cambia en cada instante de tiempo**, mientras que, en un simulador basado en eventos, **el tiempo varía en el instante en el que se ha producido dicho evento**

En nuestro desarrollo vamos a utilizar la lógica de un simulador basado en eventos. Este tipo de simuladores definen un conjunto finito de eventos. Estos eventos ocurren cada vez que hay un cambio en el sistema y pueden originar nuevos eventos.

Para la ejecución de un simulador basado en eventos es necesario una cola de prioridad ordenada en función del instante de tiempo, en la que se irán insertando los eventos que tienen que ejecutarse. El primer elemento será siempre el siguiente cambio de estado en la simulación.

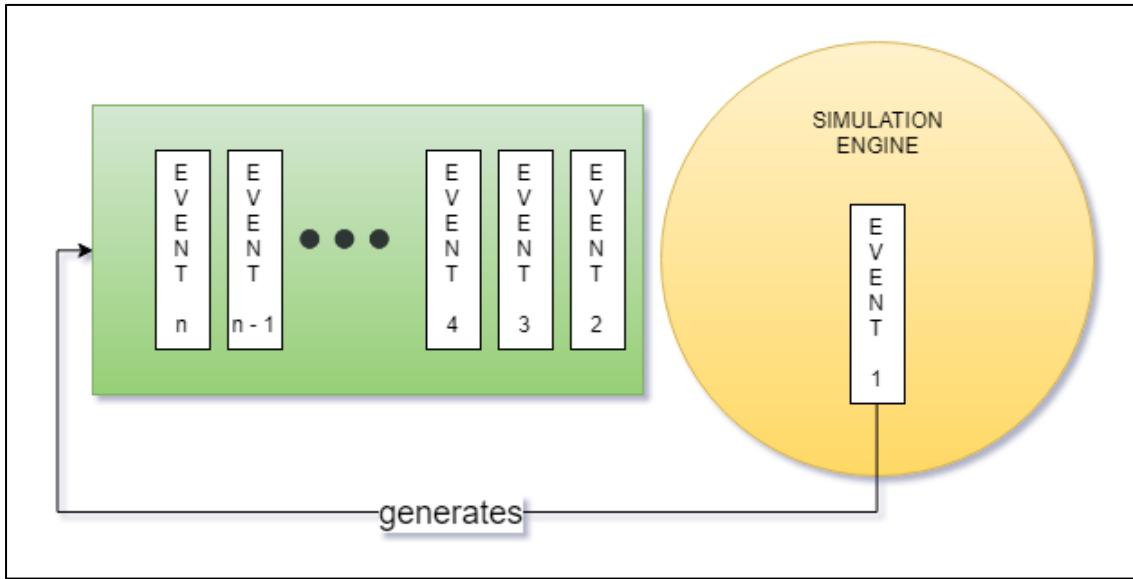


Figura 7 - Simulador basado en eventos

En la Figura 7 vemos una ejecución básica del motor de nuestro simulador. El primer evento en ejecutarse es el primero de la cola. Cada evento al ejecutarse puede generar nuevos eventos y estos son insertados en la cola.

Para definir el comportamiento de nuestro simulador, hemos definido el siguiente conjunto de eventos a partir del flujo de eventos de la *Figura 3*:

- Usuario aparece
- Usuario llega a la estación con reserva
- Usuario llega a la estación sin reserva
- *Timeout⁸* de reserva de bici.
- *Timeout de reserva de hueco.*
- Usuario finaliza vuelta en bici
- Usuario llega a la estación para dejar bici con reserva.
- Usuario llega a la estación para dejar bici sin reserva.

Se puede encontrar una explicación más detallada del diseño e implementación del simulador en el TFG de Sandra Timón Mayo [7].

⁸ Momento en el que una reserva pierde su validez debido a un tiempo máximo alcanzado

2.3.3 Arquitectura detallada

Dentro de todo el conjunto de software para realizar las simulaciones tendremos una parte que se encargará de cargar las configuraciones, como se puede ver en la siguiente figura:

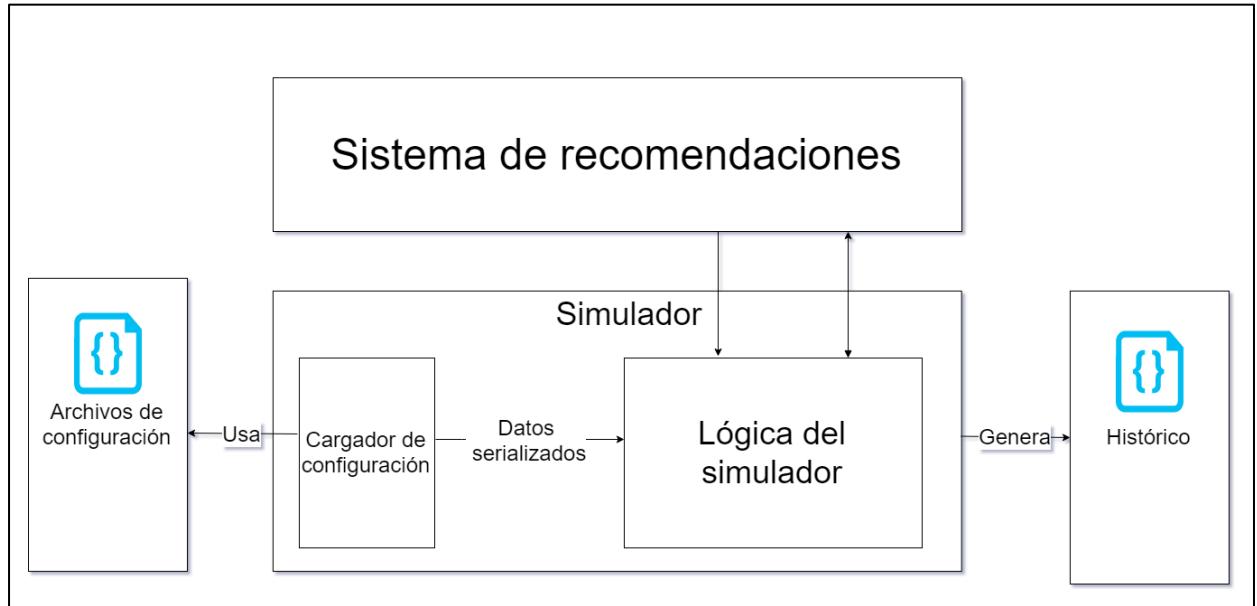


Figura 8 - Vista básica del simulador con carga de configuración

La Figura 8 muestra una parte importante de nuestra arquitectura y es el **cargador de configuración**. Éste se encargará de convertir los datos expresados en los archivos a objetos serializados en el simulador para que puedan ser utilizados en éste.

En la parte **Lógica del simulador** estaría toda la parte relacionada con cómo los usuarios interactúan dentro del simulador, cómo éstos interactúan dentro de la infraestructura del SBC. Esta **lógica de simulador** es la siguiente que vamos a desglosar en nuestra arquitectura.

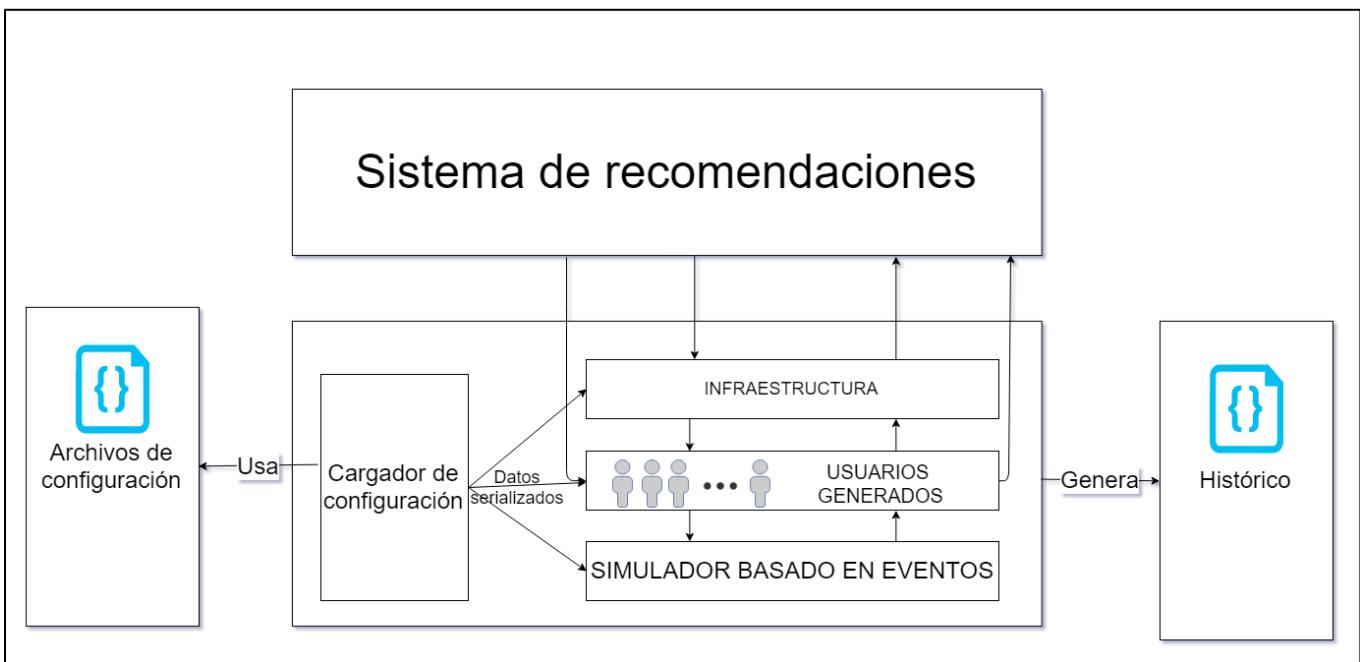


Figura 9 - Simulador con arquitectura interna del simulador

En la Figura 9 se puede ver con más detalle el comportamiento de la **lógica del simulador**. Como podemos ver lo que hace el **cargador de configuración** es interpretar los datos de los archivos de configuración para hacer funcionar nuestro simulador basado en eventos. El **simulador basado en eventos** se encargará de controlar los diferentes sucesos de la simulación para que los usuarios a su vez actúen en base al evento que les corresponde. Estos **usuarios generados(US)** interactúan con la infraestructura (cogiendo bicis, dejándolas, reservando...), y también pueden hacer uso del **sistema de recomendaciones**.

La arquitectura de la Figura 9 iba a ser la arquitectura principal de nuestro proyecto, sin embargo, surgió la necesidad de añadir un pequeño módulo externo que se encargue de generar los usuarios.

En un principio los archivos de configuración que íbamos a utilizar fueron los siguientes:

- Parámetros globales
- Estaciones
- Entry Points

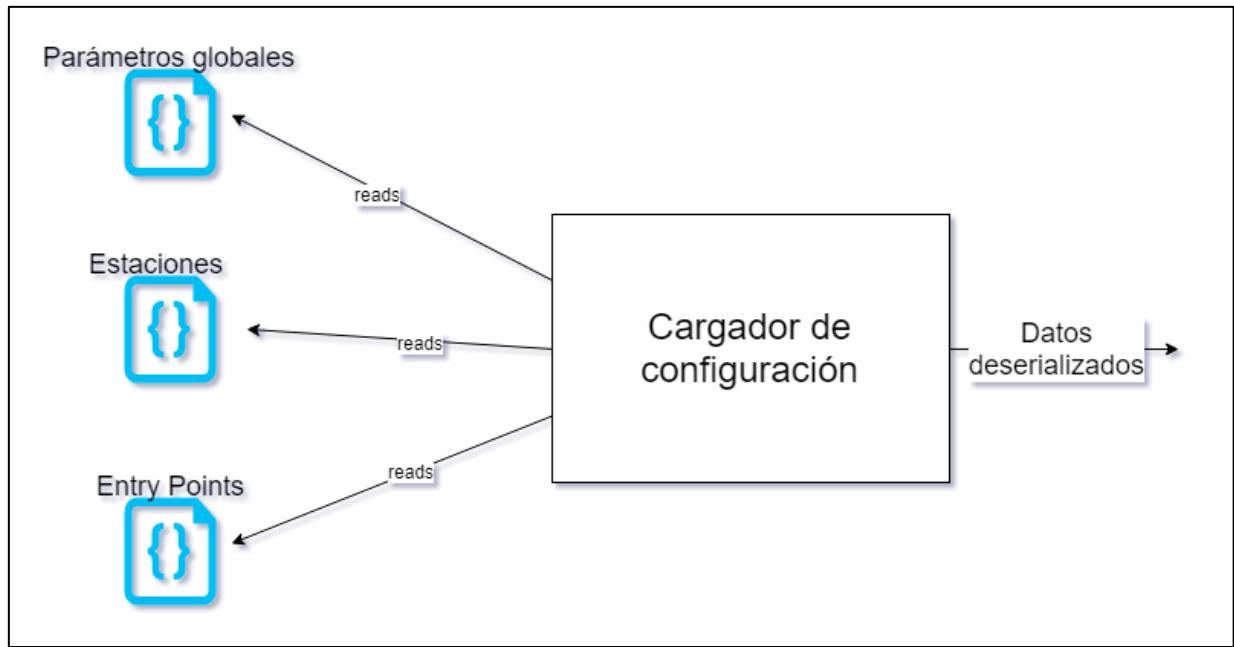


Figura 10 - Primera idea para la configuración

Parece apropiado pensar que esos tres ficheros de configuración son suficientes, sin embargo, surgió la necesidad de independizar la generación de usuarios. Si sólo utilizamos un archivo de configuración con los **entry points**, nuestro simulador depende del concepto de **entry point**, por lo que hacemos al simulador dependiente de éste. ¿Y si un tercero quiere generar los usuarios a su modo de forma individual? Tendría que estudiar cómo funciona un Entry Point, y crearlos en base a este concepto. Eso limita de algún modo los usuarios que podemos definir para el sistema. Un Entry Point nos ayuda a definir apariciones de usuarios de una forma más cómoda, pero puede darse el caso de que un desarrollador o investigador quiera generar los usuarios de otra manera.

Imaginemos por un momento que el simulador dispone sólo de esos tres archivos de configuración en el sistema. En ese caso, el simulador antes de arrancar deberá generar los usuarios y después arrancar la simulación. Sin embargo, ¿Qué pasaría si el simulador en vez de recibir **entry points**, recibiera los usuarios de forma individual en el punto y el instante exacto en el que se quiere que aparezcan? Es en este punto donde surge la necesidad de definir un nuevo módulo en nuestra arquitectura, un **generador de usuarios**. En la Figura 11 se puede ver el modelo final de la configuración.

¿Qué ganamos con separar la generación de usuarios del simulador?

1. Un tercero puede desarrollar un software cualquiera independiente, que genere estos usuarios en cualquier lenguaje.

2. Nuestro simulador se centra exclusivamente en tratar las acciones de los usuarios y no su generación.
3. Más modularidad y menos dependencia del simulador.

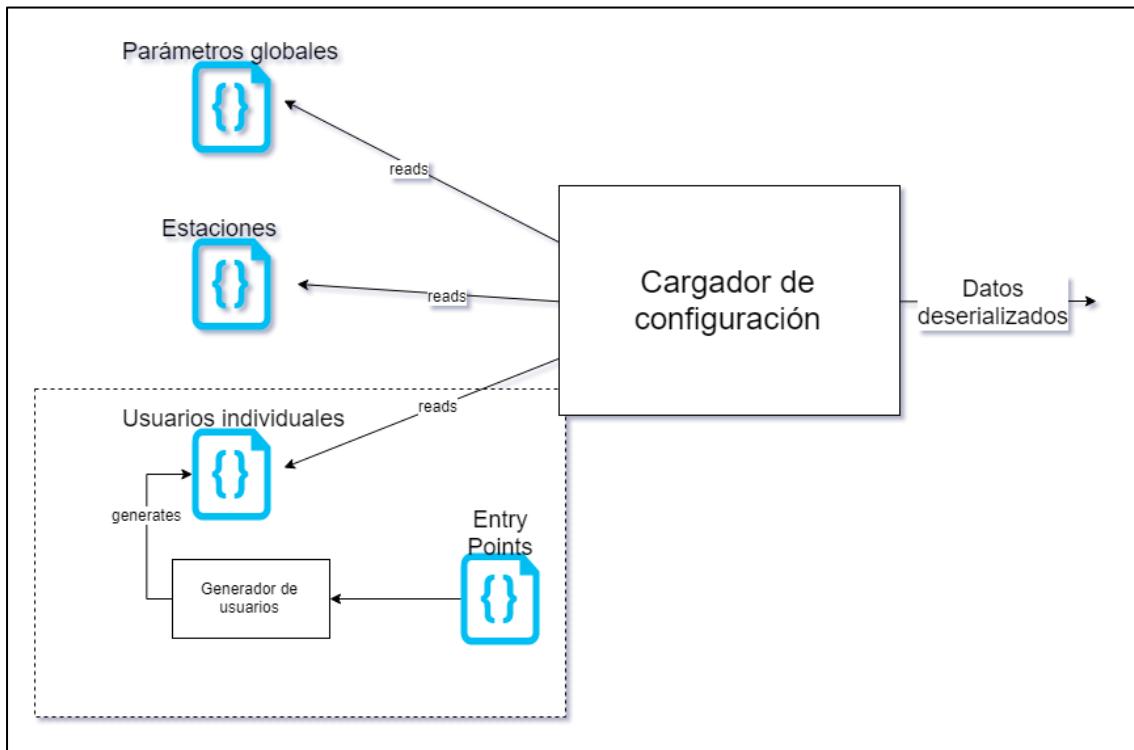


Figura 11 - Cargador de configuración con generados de usuarios

Por otro lado, la generación de los históricos es también muy importante. El **simulador basado en eventos** deberá de algún modo escribir en un fichero lo que ha sucedido en cada uno de los eventos. Para ello vamos a añadir un módulo más a nuestra arquitectura, el **generador de históricos**. Se habla más en profundidad de esta parte en el proyecto final de grado de Tao Cumplido [8].

En general este módulo se encargará de escuchar cada evento del simulador y escribir el resultado en uno o varios ficheros.

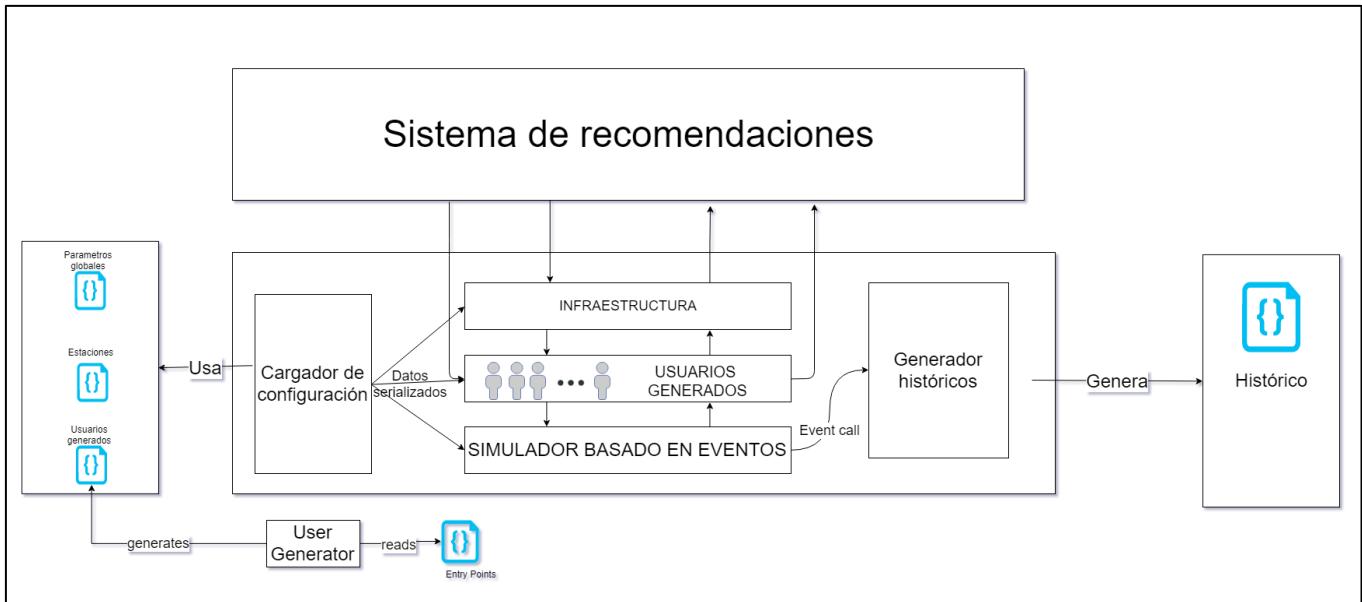


Figura 12 - Arquitectura final

El objetivo de comenzar explicando con una arquitectura abstraída como la Figura 6, era ir explicando paso por paso, como la arquitectura ha ido cambiando y a su vez introducir la arquitectura final de la Figura 12. Hemos intentado que el diseño sea lo más simple, flexible, útil y modularizado para una fácil implementación posterior.

A fin de cumplir con los requisitos de interfaz de usuario número 5.1, 5.2 y 5.3 necesitamos también disponer de una interfaz de usuario que nos permita crear simulaciones de una forma interactiva sin necesidad de escribir los archivos de configuración en un archivo de texto.

Por ello tenemos que añadir a nuestra arquitectura una parte más con la que llamar a nuestro simulador para:

- Crear / Cargar configuraciones.
- Simular a partir de los archivos de configuración creados.
- Visualizar históricos.
- Hacer análisis en base a los históricos.

En la Figura 13 que se ve a continuación se añaden a la arquitectura los módulos encargados de esta tarea dentro de la interfaz de usuario.

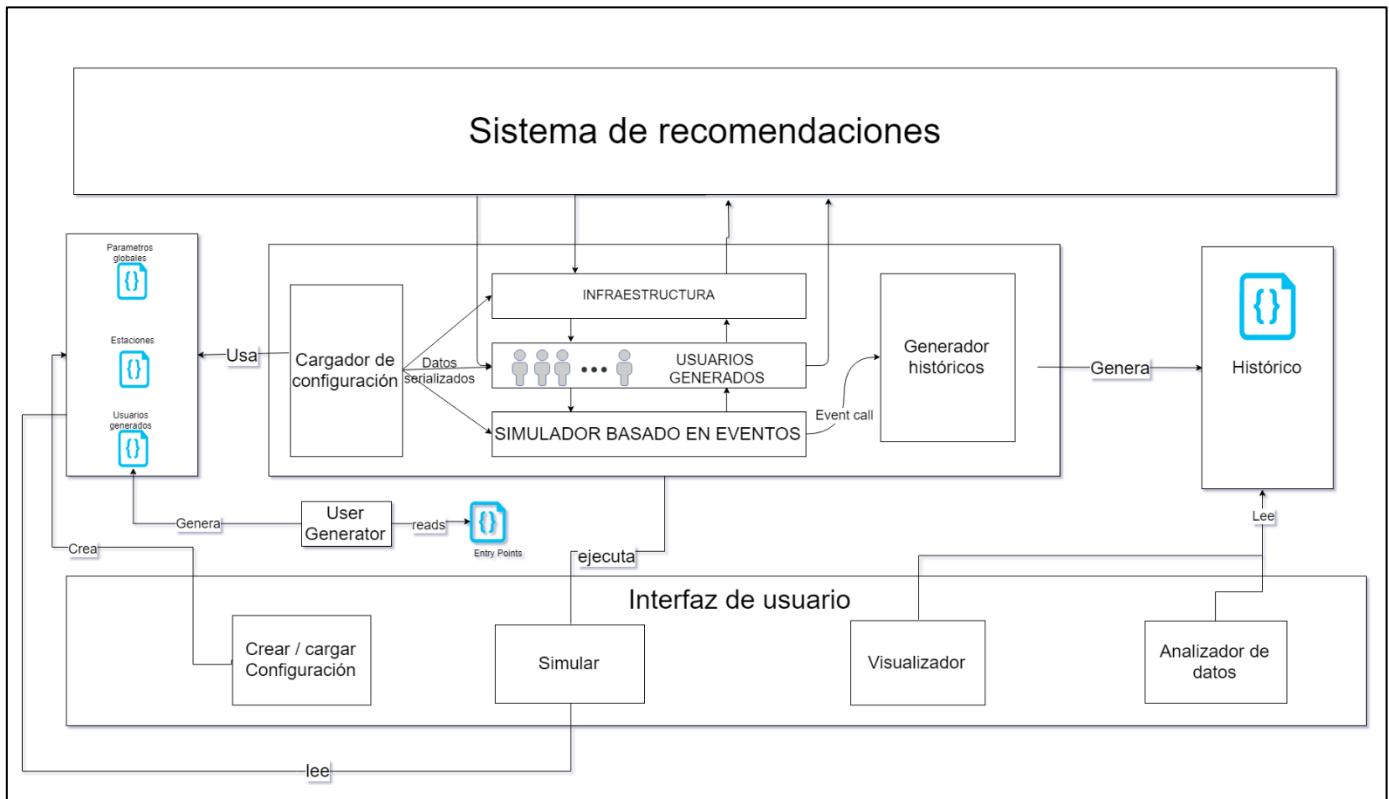


Figura 13 - Arquitectura final con interfaz de usuario

2.4 Implementación

2.4.1 Tecnologías

En cuanto a control de versiones hemos utilizado Git y la plataforma GitHub como repositorio remoto. Hemos utilizado la mayoría de herramientas que ofrece Github también para notificar errores y comentar mejoras en línea.

Uno de los objetivos que hemos tenido como desarrolladores es crear un simulador donde cada una de las partes sea lo más independiente posible. Permitiendo así que, si un módulo no gusta, o la interfaz no se adapta a las necesidades del proyecto, esta se pueda reemplazar, adaptar, cambiar de tecnología, etc...

En la Figura 14 hemos separado los distintos proyectos y tecnologías por colores. En color naranja tenemos el simulador, y en color azul la interfaz de usuario (**backend-simulator** y **frontend-simulator** respectivamente a partir de este apartado). Las partes de la arquitectura de color verde son partes que actualmente existen y están en Java implementadas, pero pueden utilizar cualquier lenguaje en un futuro, ya que son módulos externos que se comunican con el backend. Esto le da mucha flexibilidad a nuestro software.

¿Por qué los llamamos backend y frontend? Porque realmente van a tener este rol a nivel de implementación. Son proyectos en lenguajes diferentes y solo tienen en común los datos que comparten, que serían los archivos de configuración y los históricos. Además, el frontend se encarga de hacer las llamadas al backend para realizar las simulaciones.

Uno de los requisitos de nuestro proyecto es que sea multiplataforma. Es por ello por lo que en el backend hemos utilizado **Java** y para el frontend **Electron** y **TypeScript**.

Por un lado, Java es un lenguaje ya muy maduro que lleva muchos años en la cúspide de los mejores lenguajes empresariales. Un lenguaje muy conocido, demandando, potente y multiplataforma. Su comunidad es muy grande y hay una gran cantidad de librerías y frameworks disponibles.

Por otro lado, en el frontend hemos utilizado Electron, una tecnología que permite desarrollar aplicaciones de escritorio con tecnologías web. Teniendo en cuenta la cantidad de herramientas de visualizaciones que hay para visualizar datos, mapas, crear interfaces de usuario atractivas, pensamos que ésta era la mejor opción.

Electron es una combinación de HTML5, CSS y JavaScript para aplicaciones de escritorio. Es bien sabido que JavaScript no escala muy bien y debido a su naturaleza dinámica, a veces es complicado mantener un orden y estándar que otros desarrolladores puedan entender. Con el fin de crear un proyecto maduro y escalable decidimos utilizar TypeScript, que utilizado de la forma correcta tiene un aspecto similar a Java por lo que no es difícil adaptar a gente nueva a esta tecnología si vienen de Java.

TypeScript no deja de ser JavaScript, solo que añadiéndole tipado estático y la posibilidad de crear clases y objetos. Como framework para la visualización hemos decidido utilizar Angular (no confundir con AngularJS), que es una tecnología bastante conocida para la creación de aplicaciones web.

A esta arquitectura hay que añadir dos herramientas más. **Un generador de esquemas** y **un validador de archivos JSON**. Con el **generador de esquemas** definiremos la estructura de nuestros datos. Los esquemas los utilizaremos para la validación de los archivos JSON con el **validador**.

Estos dos módulos son difíciles de representar en la arquitectura presentada, ya que son muy independientes del proyecto. El validador de archivos JSON es utilizado por el backend y el simulador de usuarios.

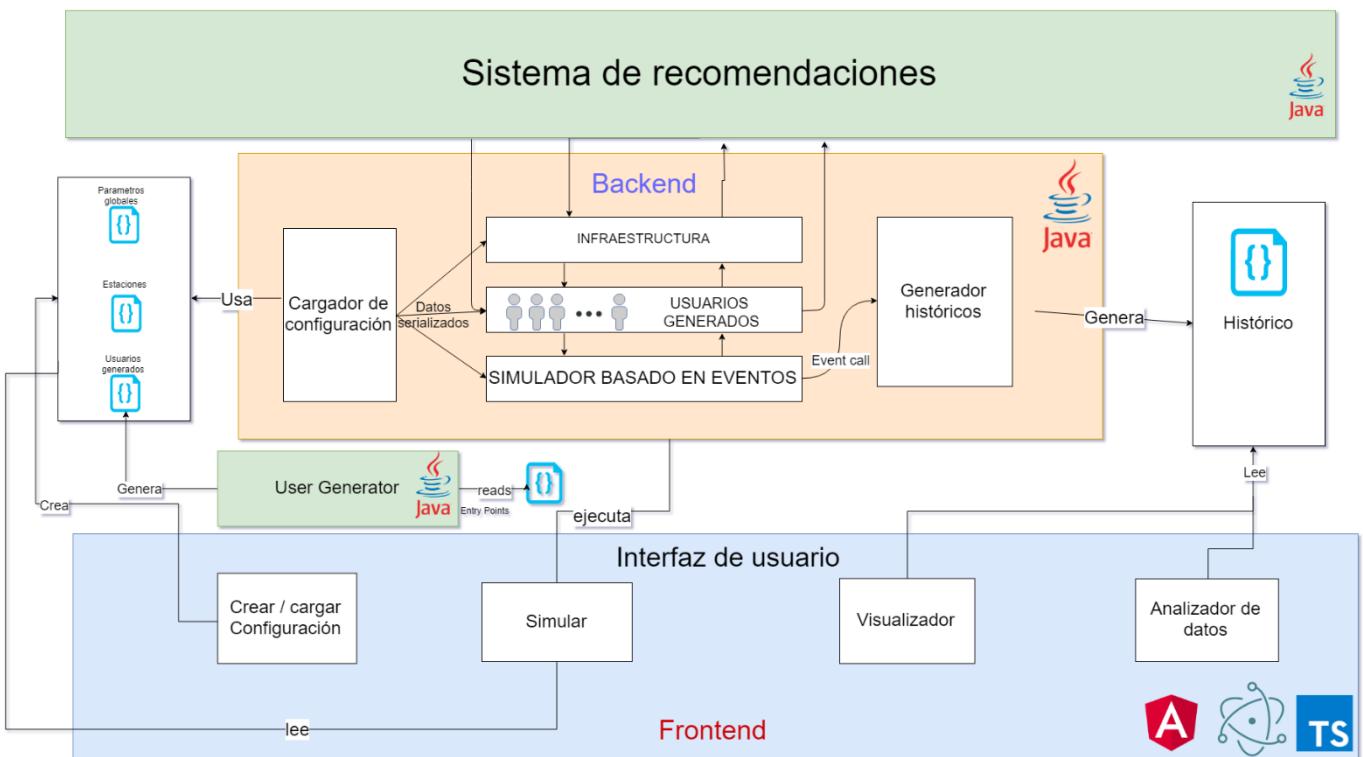


Figura 14 - Arquitectura final con tecnologías

2.4.2 Lógica del simulador (Backend)

En un principio, enfocamos el proyecto con una estructura monolítica, donde cada módulo era un paquete. Pensamos que este punto de partida era el correcto, pero en medio del desarrollo surgió la necesidad de implementar un generador de usuarios externos.

Entonces surgió la idea de implementar nuestro proyecto en módulos, para facilitar la adición de nueva funcionalidad y tener menos dependencias en nuestro código. A esto hay que añadir la necesidad de que el simulador no generase internamente los usuarios, si no que los recibiera de forma externa en un fichero. Como gestor de dependencias y herramienta de empaquetado hemos utilizado Maven, el cual permite crear proyectos modulares.

Figura 15 se pueden ver los diferentes módulos del backend.

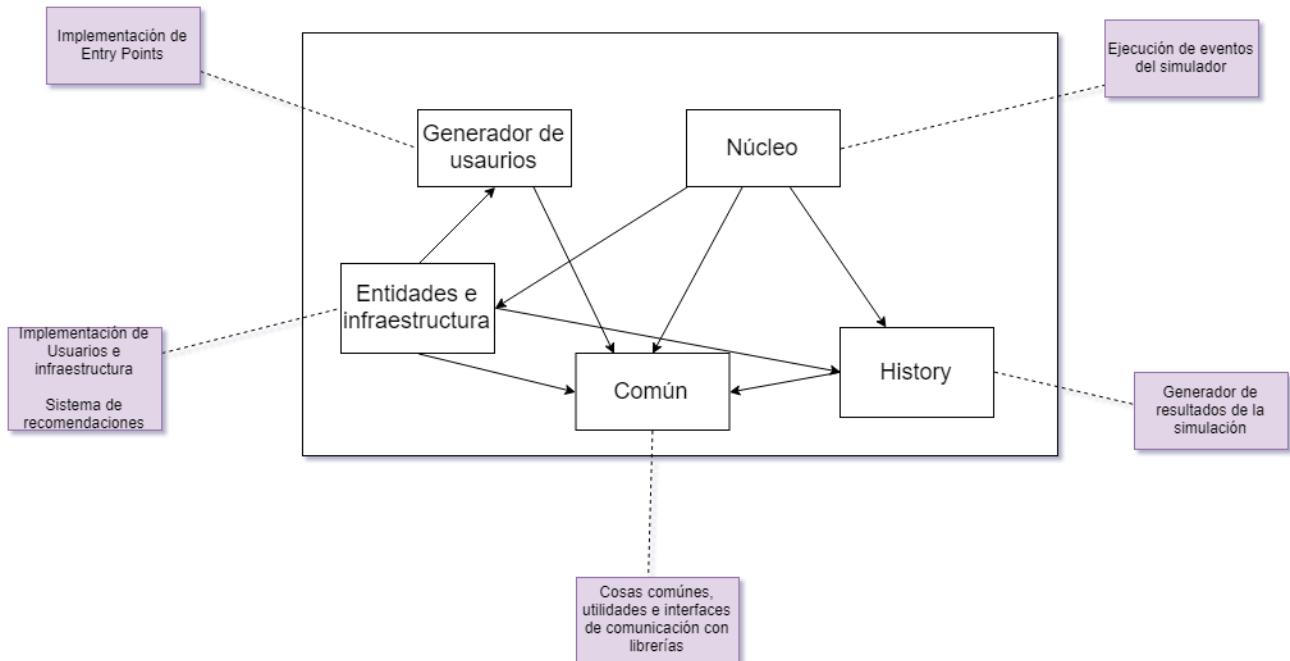


Figura 15 - Modulos backend-simulator

A primera vista puede parecer muy diferente con lo definido en la arquitectura de la Figura 14, pero eso se debe a que esta arquitectura sólo define como se organizan y comunican las partes de nuestro simulador.

A continuación, vamos a explicar todos los módulos de uno en uno:

- **Común:** Este módulo incluye todas las utilidades e interfaces necesarias para la comunicación entre los módulos. Algunas de las utilidades interesantes implementadas que merece la pena mencionar son las siguientes:
 - **GeoPoint:** Clase que implementa muchos de los métodos necesarios para calcular distancias entre puntos geométricos. Es utilizada dentro del simulador como una forma de representación de los puntos geográficos.
 - **GraphManager, GraphHopperIntegration y GeoRoute:** Véase la sección 2.4.3
 - **Debug Logger:** Utilidad creada para depurar los usuarios implementados. Esta utilidad es realmente útil para ver errores en las implementaciones de los usuarios
- **Núcleo:** Contiene todo lo referente a los eventos del simulador, y el motor de ejecución de la cola de eventos. En este módulo están definidos los

módulos, la cola de eventos, y el cargador de los archivos de configuración. Los eventos siguen la jerarquía de clases de la *Figura 17*. La clase EventUser contiene los métodos necesarios para la realización de reservas. El evento EventUserAppears es el primero en crearse por cada usuario leído del archivo de configuración. Una de las partes del núcleo se encarga de leer los usuarios del archivo de configuración y meter todos los eventos en la cola (Veáse Figura 7). Una explicación más detallada del núcleo se encuentra en el trabajo final de Sandra Timón Mayo [7].

- **Generador de usuarios:** Este módulo contiene las clases con los entry points definidos. Para facilitar la implementación de nuevos entry points, se ha utilizado el patrón factoría, como se puede ver en la Figura 16.

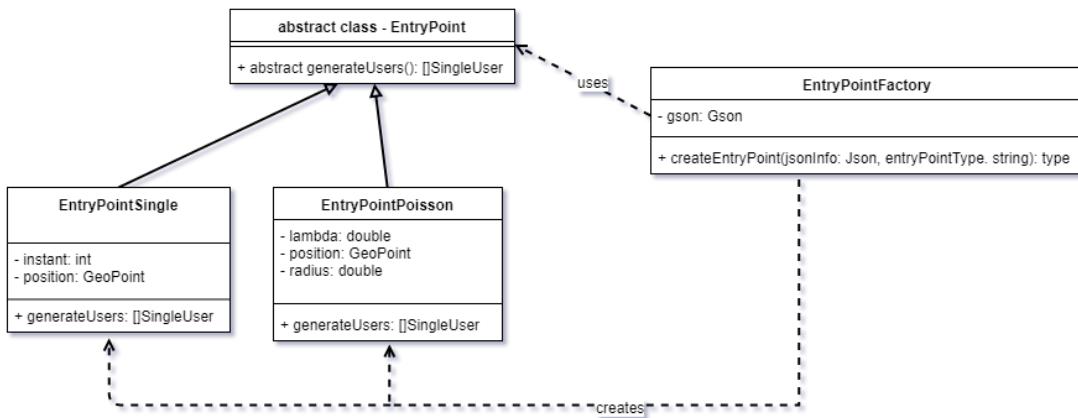


Figura 16 - Entry Point Factory

En la clase EntryPointFactory podemos ver un atributo de la clase Gson.⁹ Gson⁹ es una librería que nos permite convertir los archivos JSON en instancias de clases definidas con los mismos datos que el fichero. Utilizamos esta flexibilidad que nos proporciona Gson para crear una factoría de Entry Points. En el Apéndice B se muestra la implementación de **EntryPointPoisson.java** que representa el algoritmo implementado en Java de la sección 2.2.4

⁹ Gson - <https://github.com/google/gson>

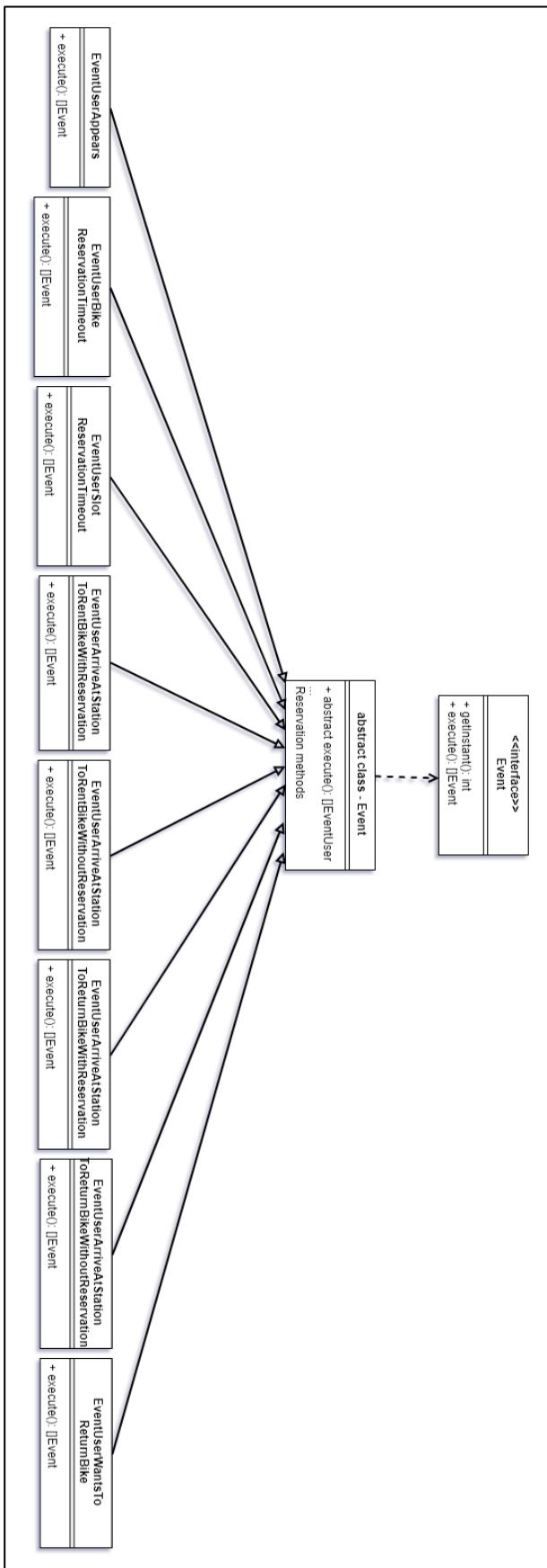


Figura 17 - Jerarquía de clases de los eventos

- **Entidades e infraestructura:** En este módulo se definen todas las entidades de la simulación. Consideramos como entidad todo lo que cambia por cada evento que sucede. Entidades en nuestro sistema son: **usuarios**, **estaciones**, **reservas y bicis**. Una reserva es una entidad ya que puede cambiar su estado, al igual que una estación cuando un usuario coge o deja una bici. En la figura se muestra el diagrama de clases de las entidades:

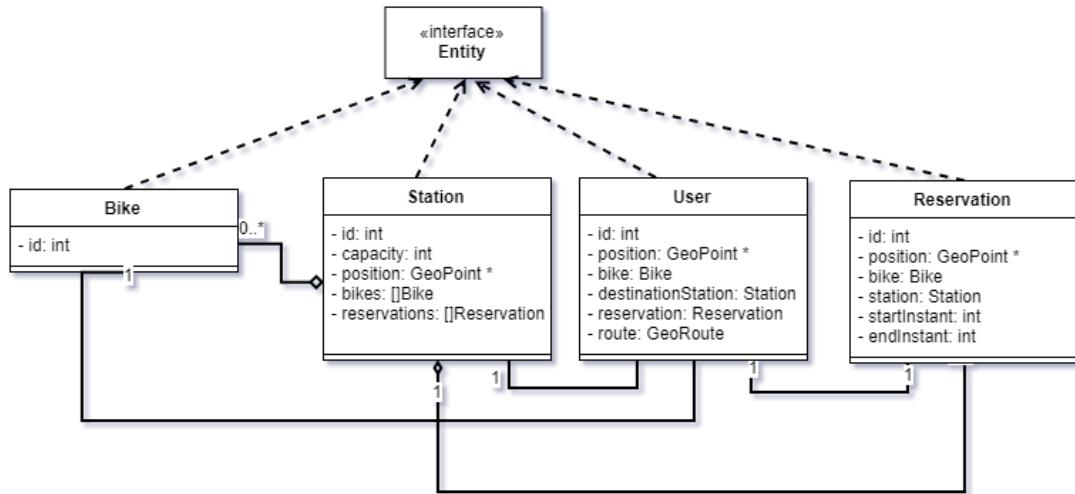


Figura 18 - Diagrama UML de entidades e infraestructura

Aquí también están definidas las implementaciones de cada usuario. Al igual que con los Entry Points, tenemos una factoría definida que nos facilita la implementación de nuevos usuarios (Figura 19). Estos usuarios heredan todos de una clase abstracta **User**, que define todos los métodos abstractos que debe tener un usuario al interactuar con el simulador. Es así como se definen los nuevos **US (Usuarios simulados)**. Además, estos pueden tener parámetros que influencien en sus decisiones, o memoria. Todo está a disposición del implementador que desee programar un usuario en concreto.

Toda interacción con el **Sistema de recomendaciones** o la **infraestructura** se realiza a través de una clase llamada **SystemManager**. Ésta puede a través de sus métodos darle rutas al usuario, recomendarle estaciones, etc.

Como no disponemos de un sistema de recomendaciones aún, hemos implementado un **sistema de recomendaciones** muy básico en este módulo. Cuando se disponga de uno, se conectará el **SystemManager** a dicho módulo sin problemas.

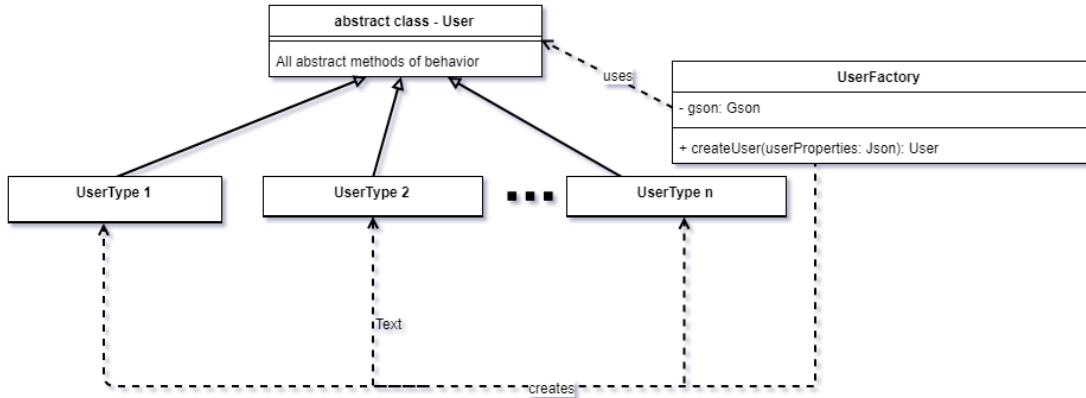


Figura 19 - Factoría de usuarios

En el Apéndice C se muestra el código de la clase abstracta de usuario. En esa clase están definidos todos los métodos que debe implementar cualquier tipo de usuario.

- **History:** Contiene toda la lógica necesaria para que evento tras evento, los resultados sean escritos en un histórico. Es un módulo que escribe los resultados que se han realizado expresando los cambios en cada evento, lo cual hace que los históricos no sean tan pesados. Los instantes son almacenados en diferentes archivos JSON de 100 en 100, haciendo que sean legibles por un ser humano y además manejables para módulos externos sin necesidad de usar streams. Se puede ver más información sobre la generación de históricos en el trabajo final de Tao Cumplido [8].

2.4.3 Carga de mapas y cálculo de rutas

Algo interesante a implementar es la posibilidad de crear simulaciones en ciudades reales. Implementar toda esta lógica y crear una estructura de datos eficiente para la carga y cálculo de rutas nos podría llevar incluso más tiempo que el propio simulador. Es por eso por lo que decidimos utilizar una librería externa, **GraphHopper**¹⁰. GraphHopper internamente utiliza Open Street Maps, lo cual nos da cierta libertad al no depender de Google Maps que es una tecnología cerrada.

Para hacer que nuestro simulador no dependa directamente de dicha librería, decidimos crear una interfaz (Graph Manager) y un formato de rutas propio (GeoRoute) con la que poder posteriormente implementar el uso de esta librería. De esta

¹⁰ <https://github.com/graphhopper/graphhopper>

forma si necesitamos nosotros crear nuestra propia utilidad o utilizar otra librería no sería un gran problema.

Las implementaciones de GraphManager, GeoRoute y GraphHopperImplementation se encuentran en el Apéndice D.

2.4.4 Interfaz de usuario del simulador (Frontend)

Crear las configuraciones para cada simulación puede ser algo tedioso, debido a la gran cantidad de datos que hay que introducir y los puntos geográficos de las entidades o los entry points a veces son difíciles de ubicar. Además, una buena forma de ver de primera mano, como están implementados nuestros usuarios, es tener un visualizador con el que observar toda la simulación.

Como hemos comentado en la sección 2.4.1, vamos a utilizar para la interfaz de usuario Electron.

Electron al ejecutarse ejecuta dos procesos:

- **Main:** Este proceso puede comunicarse con el SO y hacer operaciones de entrada/salida. Está implementado enteramente en TypeScript. En esta parte hemos definido toda la lógica que no tiene que ver con la interfaz de usuario. Los módulos que hay implementados son los siguientes:
 - **DataAnalysis:** Contiene toda la lógica para analizar los históricos [7].
 - **Entities:** Entidades utilizadas por **DataAnalysis**.
 - **Util:** Contiene clases básicas para la comunicación entre el proceso Main y Renderer, además del procesador de históricos History-Reader [8].
 - **Configuration:** Contiene lo referente a la creación de formularios dinámicos. Véase sección 2.4.4
- **Renderer:** Este proceso contiene toda la parte visual. Está programado en Angular y TypeScript.
Angular está basado en componentes. Cada parte visual es un componente, y se puede comunicar con el **Main** si necesita algún recurso o tiene que hacer alguna llamada al simulador. Los distintos módulos son:
 - **App:** Contiene todos los componentes.

- **Ajax**: Contiene la lógica implementada para comunicar datos al proceso **Main**.

2.4.4 Formularios dinámicos configuración(Frontend)

Al tener que introducir tantos datos en la configuración y ser estos tan variables, nos surgió la necesidad de que la interfaz gráfica fuera capaz de detectar los campos de todos los datos a introducir y generar los formularios de forma dinámica.

¿Por qué? Imaginemos que tenemos un usuario, y que necesitamos los siguientes datos para crear un entry point de tipo poisson con ese tipo de usuario.

- Valor lambda
- Posición
- Radio de aparición
- Probabilidad de reservar bici
- Probabilidad de reservar después de un TimeOut

Para crear un formulario con estos datos, necesitaríamos crear validadores para cada dato, tener un formulario único por cada tipo de usuario y, además, tendríamos que modificar dicho formulario cada vez que un parámetro cambie. Y no sólo eso, sino que cada vez que añadimos un usuario tenemos que modificar la GUI. Sin embargo, los esquemas ya definen como deben ser los archivos de configuración, por lo que hemos creado un generador de esquemas adaptados a una librería¹¹ que a partir de este esquema nos crea un formulario de forma dinámica.

Las ventajas de tener formularios dinámicos para los usuarios son las siguientes:

- No es necesario implementar formularios por cada usuario, se generan en tiempo de ejecución.
- No hay que tocar el código de la GUI para añadir, modificar o quitar una implementación de usuario, solo tocar los esquemas.

¹¹ <https://github.com/dschnelldavis/angular2-json-schema-form>

- En el siguiente diagrama se muestra el funcionamiento básico de los formularios dinámicos.

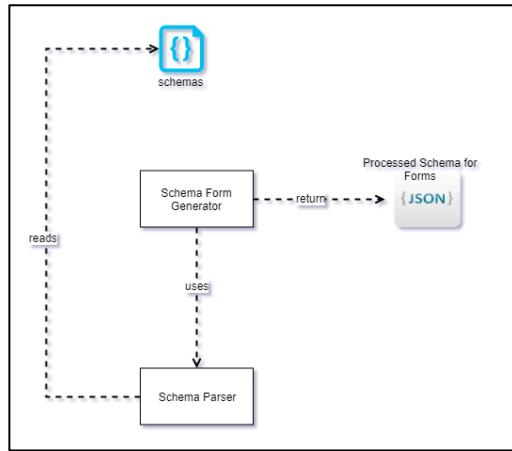


Figura 20 - Generador de schemas para los formularios

En el Apéndice E se puede ver una demostración de un esquema y su correspondiente formulario generado.

3 Evaluación

En este apartado vamos a ver las distintas partes que se han llegado a implementar de la interfaz de usuario además de los resultados obtenidos tras una serie de simulaciones realizadas para la PAAMS international conference del 2018¹².

3.1 GUI

A continuación, se mostrarán capturas de las distintas partes del simulador que se pueden utilizar.

- **Menú.** Las opciones disponibles en la finalización de este proyecto son: crear configuración, simular configuración y ver simulación.

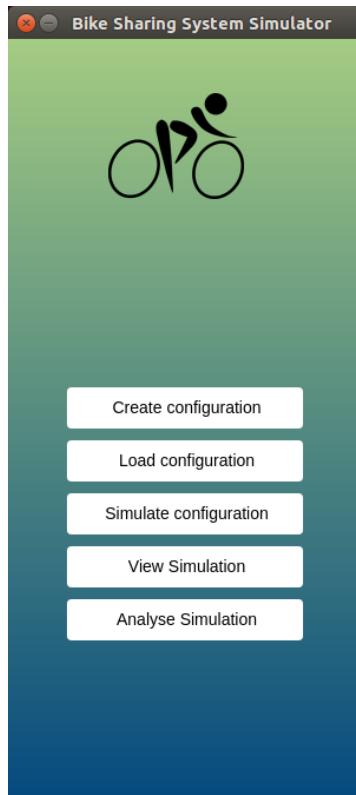


Figura 21 - Menú principal de la GUI

¹² <https://www.paams.net/>

- Crear configuración: Se pueden crear de forma interactiva simulaciones, pudiendo crear entry points con radio de una forma intuitiva y delimitar la zona de simulación. Los formularios son todos generados por el módulo de la sección 2.4.4.

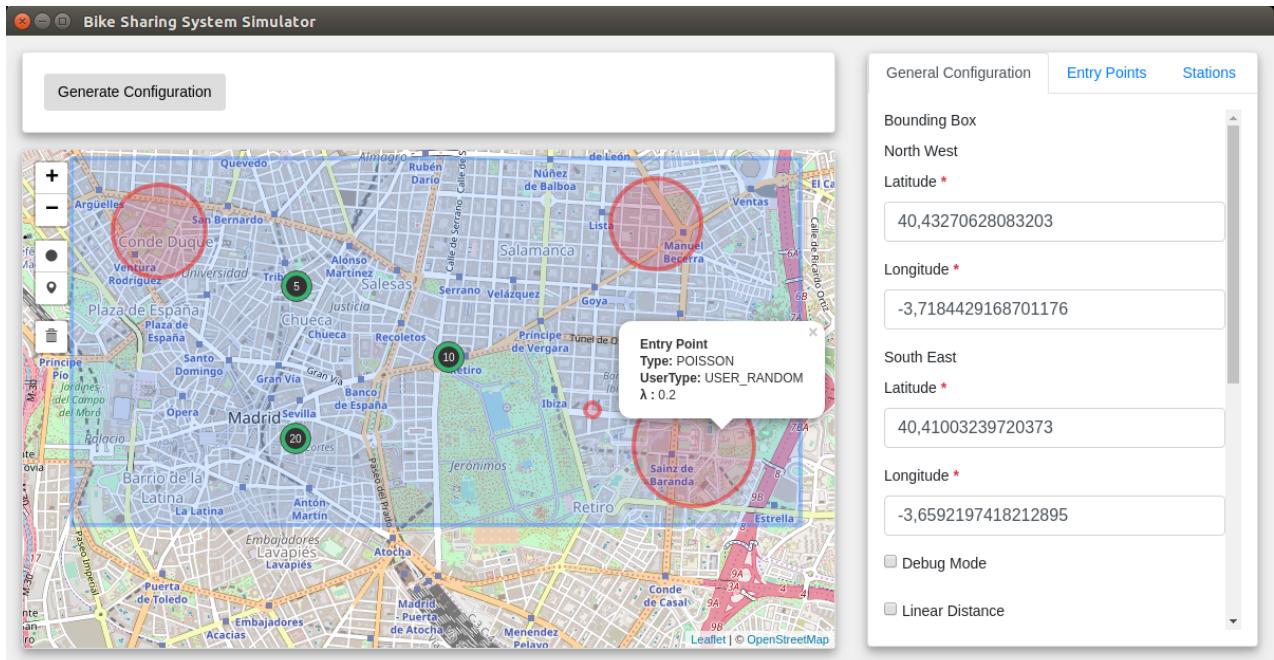


Figura 22 - Simulación creada desde la GUI

Al añadir entidades en el mapa se puede ver en el momento los datos de la configuración que estamos realizando, como la posición de los Entry Points, el radio... Pudiendo modificarlos desde un editor incorporado. En la Figura 23 se puede ver una lista de Entry Points en la configuración.

Al pulsar el botón **Generate Configuration** si los datos están correctamente se generarán los archivos de configuración necesarios en el directorio que se le indique.



Figura 23 - Lista de Entry Points en GUI

- Simular configuración. En esta ventana de la GUI se pueden generar usuarios individuales (cargando los entry points) y llamar al backend para simular con los archivos de configuración y los usuarios individuales ya generados.

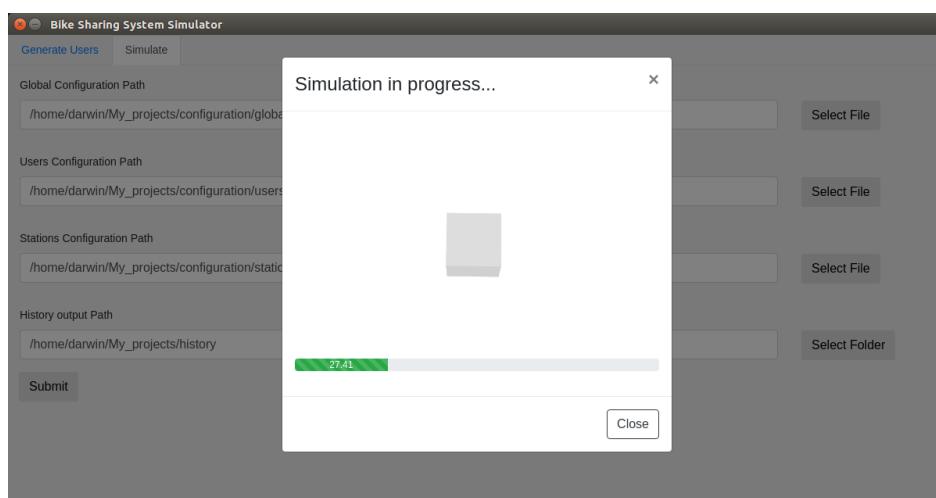


Figura 24 - Simulación ejecutándose desde la GUI

- Visualizador: Desde el visualizador se puede cargar el histórico de la simulación que se desee y ver como los usuarios actúan dentro del sistema [8].

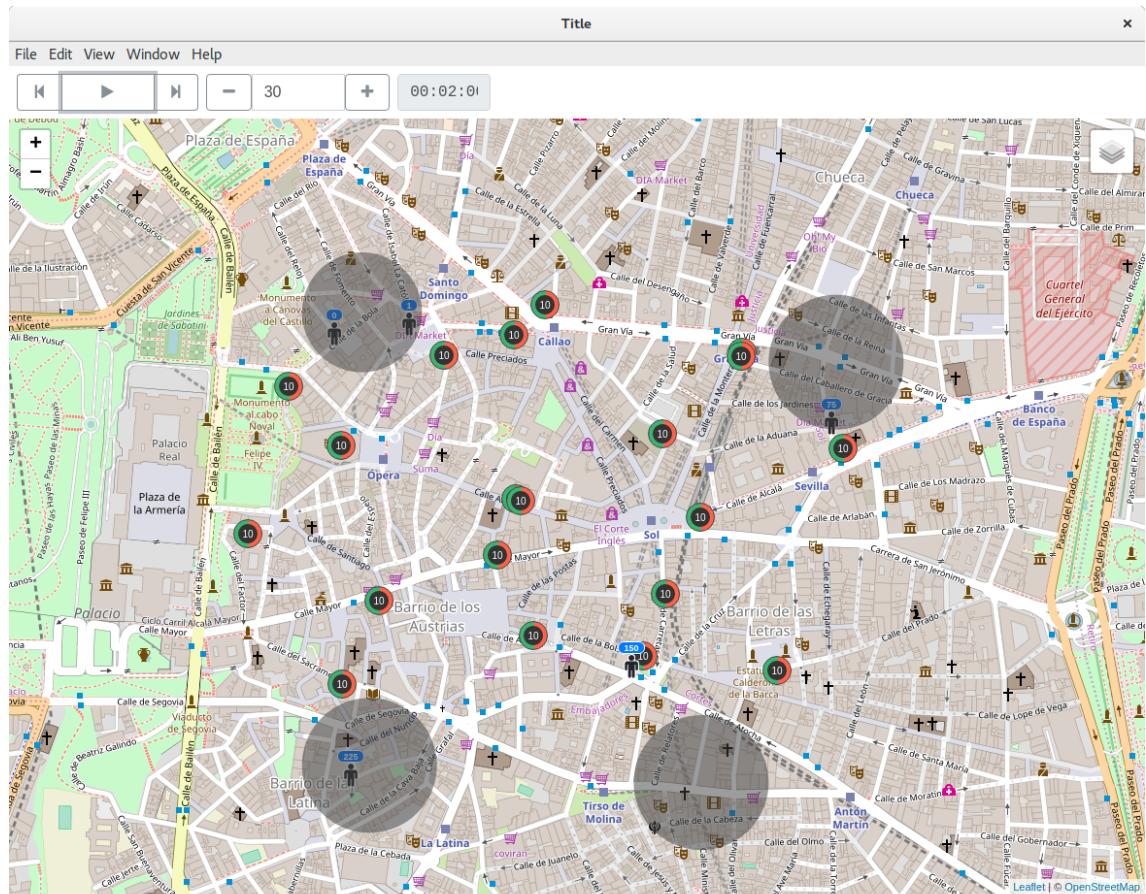


Figura 25 - Visualización histórico

3.2 Prueba simulador

En mitad del desarrollo del simulador, se realizaron una serie de pruebas con los siguientes tipos de usuarios implementados.

- Uninformed: Este usuario trata de coger o devolver la bici de la estación más cercana que tenga desde punto de entrada. Este usuario no tiene información de la disponibilidad de bicis o de huecos en la estación.
- Informed: Este usuario sabe información del sistema y sólo va a estaciones con bicis. Este usuario escoge la ruta más corta para ir a la estación.
- Obedient: Pide información al sistema de recomendaciones y siempre sigue sus sugerencias.

- Informed-R: Es el mismo tipo de usuario que el informado, solo que siempre hace reservas
- Obedient-R: Es el mismo tipo de usuario que el obediente, solo que siempre reserva.

Con estos usuarios se decide hacer el siguiente un experimento como el de la Figura 25. Se tienen 20 estaciones repartidas por el centro de Madrid y 4 entry points. El sistema de recomendaciones de las medidas básicas que provee ahora mismo el simulador son

- SH (Successful hires) - Número de bicis alquiladas con éxito.
- FH (Failed hires) - Número de intentos de alquilar que no se han realizado con éxito.
- SR (Successful returns) - Devoluciones exitosas.
- FR (Failed returns) - Devoluciones fallidas.
- N - Número total de usuarios

Y a partir de estos valores se quieren calcular los siguientes.

- DS (Demand satisfaction) - Satisfacción de demanda: Proporción de usuarios que son capaces de alquilar una bici con éxito.

$$DS = SH / N$$

- RS (Return satisfaction) - Satisfacción de devolución: Proporción de usuarios que son capaces de devolver con éxito por primera vez o en las siguientes.

$$RS = SR / SH$$

- HE (Hire efficiency) - Eficiencia al alquilar: Proporción de alquileres exitosos entre el número total de intentos de alquiler.

$$HE = SH / (SH+FH)$$

- RE (Return efficiency) - Eficiencia de devolución: Proporción de devoluciones exitosas y el número total de intentos de devolución.

$$RE = SR / (SH+FR)$$

A lo largo de la siguiente simulación lo que hacemos es aumentar progresivamente el número de usuarios en un mismo espacio de tiempo.

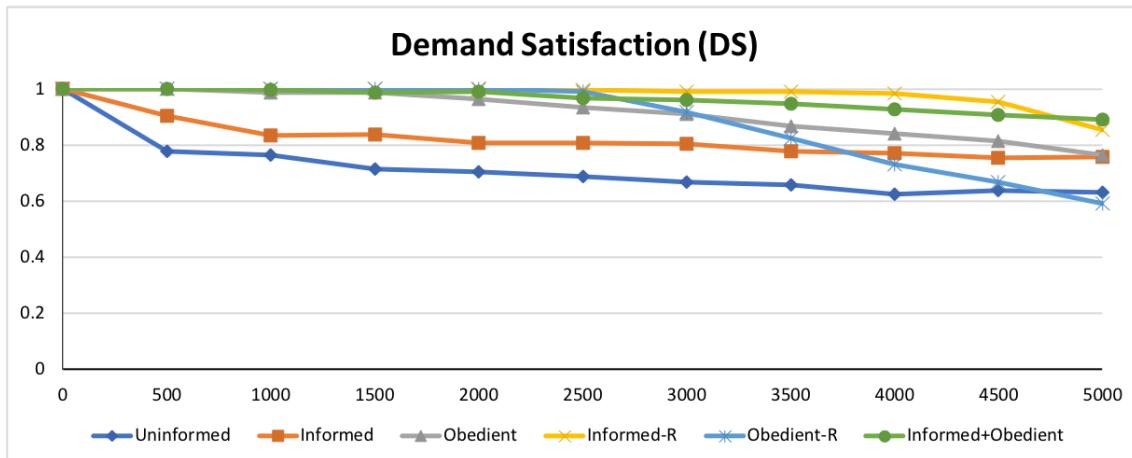


Figura 26 - Satisfacción de demanda

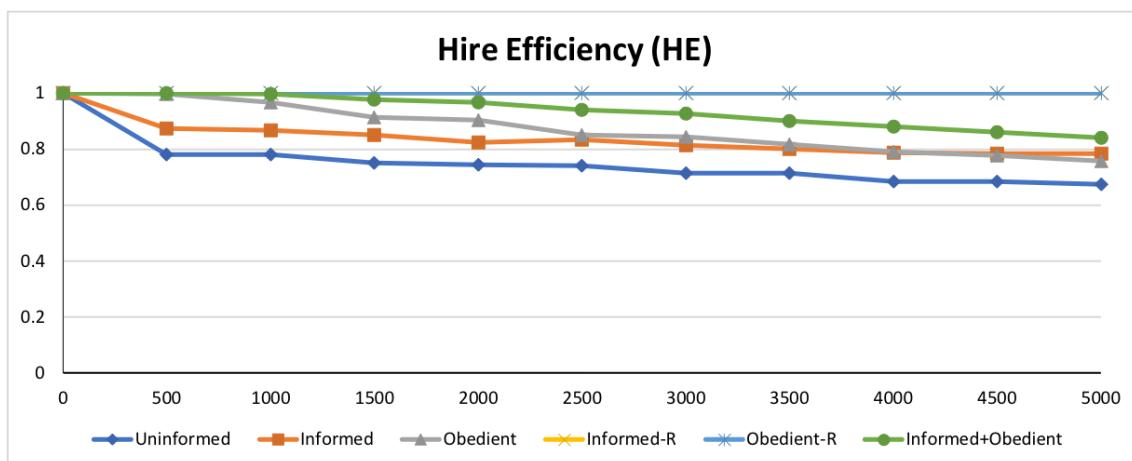


Figura 27 - Eficacia al alquilar

Como se puede ver el usuario desinformado es el peor de los casos tanto en la satisfacción de demanda como en el alquiler de bicis. Se dan ciertas incongruencias con el usuario Obediente que debería ser el mejor de todos, debido a que al reservar siempre, a medida que se aumenta el número de usuarios, el resultado es pero ya que tienen la bici alquilada durante más tiempo.

4 Conclusiones

A lo largo de este trabajo hemos aprendido a realizar un simulador, a plantear los archivos necesarios de configuración para llevarla a cabo, planteando su arquitectura e implementación, concluyendo con el alcance de nuestro objetivo. Se ha conseguido implementar un simulador con una configuración aceptable a cambios, extensible, fácil de modificar, con una interfaz de usuario que nos permite realizar las simulaciones de una manera mucho más sencilla.

Cada una de las partes de este proyecto se ha hecho siempre pensando a futuro hacia nuevas implementaciones y cambios, pero obviamente nada de esto se podría haber llevado sin un buen trabajo en equipo.

4.1 Lecciones aprendidas.

No siempre las lecciones aprendidas son en base a malas experiencias. En este caso ha habido más lecciones de buenas experiencias que de errores.

- Un equipo bien organizado, comunicativo y motivado da como resultado un buen software.
- Los patrones de diseño, bien aplicados, son muy importantes. En nuestro caso hemos aplicado dos factorías, una para los **US** (Usuarios simulados) y otro para los **entry points**.
- Es importante verificar las entradas y salidas si son grandes, con herramientas como los esquemas. En nuestro caso la utilización de los Json Schema no sólo han aportado una forma para validar los datos de los archivos de configuración e históricos, si no también un medio para generar formularios en la GUI de forma dinámica.
- Toda librería debe ser usada de la forma más independiente posible. Por ejemplo, nuestro simulador utiliza los mapas a través de una interfaz, lo cual lo hace independiente a como estén implementados dichos mapas y como se calculan las rutas. Podríamos sustituirlo por otra librera o incluso crear una nosotros mismos.

Pero no siempre las cosas salen bien, es por eso por lo que tampoco debemos olvidar los pequeños errores de diseño que hayamos podido tener.

- Un evento dentro del simulador nos hubiera ahorrado muchísimos quebraderos de cabeza y cuando nos dimos cuenta ya era demasiado tarde. Este

evento es del usuario haciendo reserva de bici o de hueco. Aún así el simulador funciona correctamente salvo que la información de cambio de estado de las reservas en los históricos está presente en muchos Eventos, cuando podría estarlo sólo en uno.

- Angular es demasiado pesado para Electron. En un principio, Angular no se pensó como framework multiventana como nosotros hemos hecho, y las ventanas tardaban mucho en cargar de forma individual. Además, angular tiene mucho código por debajo ejecutando, y suele ser algo más lento y ocupar más espacio que otros frameworks. Una alternativa buena hubiera sido React o Vue.js que son frameworks muy ligeros. Pero a pesar de esto, Angular es un framework muy elegante, ordenado con inyección de dependencias (para utilizar servicios) y muy potente.

4.2 Líneas futuras.

- Una de las ideas que más me llama la atención es la posibilidad de convertir el simulador en un framework, que nos permita, por ejemplo, añadir usuarios y entry points, sin necesidad de modificar el propio código del simulador.

Una de estas posibilidades es la de poder crear los usuarios a través de que lenguaje de scripting, como Python o Lua. Imaginemos que el simulador, antes de arrancar, hace un barrido de todos los usuarios implementados en python que haya en una carpeta, y los ejecuta. Con esto podríamos hacer que la propia interfaz de usuario incluya un pequeño editor de texto con el que se puedan añadir nuevos usuarios sin la necesidad de configurar ningún entorno y de compilar todo el proyecto. Estaríamos facilitando la labor de investigación de una manera muy eficiente.

Esto es perfectamente posible ya que Python.org deja a nuestra disposición librerías con los que invocar a métodos y clases definidos en Python.

- Llegar a una primera versión estable, tras corregir todos los posibles bugs encontrados de la interfaz.

- Crear un sistema de recomendaciones externo, basado en datos reales, que pueda predecir la demanda de una estación y recomendar en base a esto al usuario.

Apéndice A

En los siguientes ejemplos se muestra el código correspondiente al esquema global de configuración. Esta forma de escribir los esquemas no es más que la forma en lenguaje TypeScript que necesita el generador de esquemas.

CONFIGURACIÓN GLOBAL:

Ejemplo de esquema configuración global:

```
export default new JsonSchema(options, sObject({
    totalSimulationTime: sInteger().min(0).max(rData('1/totalSimulationTime')),
    debugMode: sBoolean(),
    reservationTime: sInteger(),
    randomSeed: sInteger(),
    linearDistance: sBoolean(),
    boundingBox: sObject({
        northWest: GeoPoint,
        southEast: GeoPoint,
    }).require.all().restrict(),
    map: sString().pattern(/.+\.osm/)
}).require.all().restrict());
```

Ejemplo de configuración global:

```
{
    "reservationTime": 500,
    "totalSimulationTime": 86400,
    "debugMode": true,
    "randomSeed": 201,
    "linearDistance": true,
    "boundingBox": {
        "northWest": {
            "latitude": 40.4215886,
            "longitude": -3.7134038
        },
        "southEast": {
            "latitude": 40.4121931,
            "longitude": -3.69826
        }
    },
    "map": ".../backend-configuration-files/maps/madrid.osm"
}
```

CONFIGURACIÓN ESTACIONES

Ejemplo de esquema estaciones:

```
export position default sObject{  
    capacity: GeoPoint,  
    bikes: UInt,  
    sAnyOf(  
        sInteger().min(0).max(rData('1/capacity')),  
        sArray(sAnyOf(Bike, sNull())).min(rData('1/capacity')).max(rData('1/capacity'))  
    ),  
}).require.all().restrict();
```

Ejemplo de configuración estaciones:

```
{  
    "stations": [  
        {  
            "bikes": 10,  
            "position": {  
                "latitude": 40.4197429,  
                "longitude": -3.7080733  
            },  
            "capacity": 20  
        },  
        {  
            "bikes": 10,  
            "position": {  
                "latitude": 40.4192095,  
                "longitude": -3.711504  
            },  
            "capacity": 20  
        },  
        {  
            "bikes": 10,  
            "position": {  
                "latitude": 40.4182146,  
                "longitude": -3.7103538  
            },  
            "capacity": 20  
        },  
        {  
            "bikes": 10,  
            "position": {  
                "latitude": 40.4200783,  
                "longitude": -3.7065376  
            },  
            "capacity": 20  
        },  
        {  
            "bikes": 10,  
            "position": {  
                "latitude": 40.4170009,  
                "longitude": -3.7024207  
            },  
            "capacity": 20  
        }  
    ]  
}
```

CONFIGURACION DE ENTRY POINTS

Ejemplo de esquema de entry point:

```
export default sAnyOf([
  sObject({
    entryPointType: sConst('POISSON'),
    distribution: Distributions.poisson,
    userType: UserProperties,
    position: GeoPoint,
    sObject({
      end: UInt,
      start: UInt
    }).require.all().restrict(),
    radius: sNumber().xMin(0),
    totalUsers: sInteger().xMin(0)
  }).require('entryPointType', 'userType', 'distribution', 'position'),
  ...
]);
```

Ejemplo de configuración entry points

```
{
    "entryPoints": [
        {
            "entryPointType": "POISSON",
            "userType": "USER_INFORMED",
            "parameters": {
                "minRentalAttempts": 2
            },
            "position": {
                "latitude": 40.419969,
                "longitude": -3.709819
            },
            "radius": 100,
            "distribution": {
                "lambda": 0.0166666666666667
            }
        },
        {
            "entryPointType": "POISSON",
            "userType": "USER_INFORMED",
            "parameters": {
                "minRentalAttempts": 2
            },
            "position": {
                "latitude": 40.418969,
                "longitude": -3.700819
            },
            "radius": 100,
            "distribution": {
                "lambda": 0.0166666666666667
            }
        },
        {
            "entryPointType": "POISSON",
            "userType": "USER_INFORMED",
            "parameters": {
                "minRentalAttempts": 2
            },
            "position": {
                "latitude": 40.4131931,
                "longitude": -3.70456
            },
            "radius": 100,
            "distribution": {
                "lambda": 0.0166666666666667
            }
        }
    ]
}
```

```

        "radius": 100,
        "distribution": {
          "lambda": 0.01666666666667
        }
      },
      {
        "entryPointType": "POISSON",
        "userType": {
          "typeName": "USER_INFORMED",
          "parameters": 2
        },
        "position": {
          "latitude": 40.4131931,
          "longitude": -3.7104038
        },
        "radius": 100,
        "distribution": {
          "lambda": 0.01666666666667
        }
      }
    ]
  }
}

```

CONFIGURACIÓN USUARIOS INDIVIDUALES

Ejemplo de esquema de usuarios individuales:

```

export default new JsonSchema(options, sObject({
  initialUsers: sArray(SingleUser),
}).require.all().restrict());

const SingleUser = sObject({
  userType: UserProperties,
  position: GeoPoint,
  timeInstant: UInt
});

const Percentage = sNumber().min(0).max(100);

export const UserType = sEnum(...Object.keys(typeParameters));

export const UserProperties = sAnyOf(...Object.entries(typeParameters).map((user) => {
  const type = user[0];
  const parameters: any = user[1];
  return sObject({
    typeName: sConst(type),
    parameters: parameters && sObject(parameters)
  })
}));

const typeParameters = {
  USER_RANDOM: {},
  USER_UNINFORMED: {},
  USER_INFORMED: {
    willReserve: sBoolean(),
    minReservationAttempts: UInt,
    minReservationTimeouts: UInt,
    minRentalAttempts: UInt,
    bikeReservationPercentage: Percentage,
    slotReservationPercentage: Percentage,
    reservationTimeoutPercentage: Percentage,
    failedReservationPercentage: Percentage
  },
  ...
};

```

Ejemplo de configuración de usuarios individuales

```
{ "initialUsers": [ { "position": { "latitude": 40.420634503997825, "longitude": -3.7103315803956933 }, "userType": { "typeName": "USER_INFORMED", "parameters": { "minRentalAttempts": 2 } }, "timeInstant": 19 }, { "position": { "latitude": 40.420026403464625, "longitude": -3.7089663814304945 }, "userType": { "typeName": "USER_INFORMED", "parameters": { "minRentalAttempts": 2 } }, "timeInstant": 42 }, { "position": { "latitude": 40.41931477317255, "longitude": -3.7095898771891456 }, "userType": { "typeName": "USER_INFORMED", "parameters": { "minRentalAttempts": 2 } }, "timeInstant": 51 }, { "position": { "latitude": 40.41961799493365, "longitude": -3.7099779343040784 }, "userType": { "typeName": "USER_INFORMED", "parameters": { "minRentalAttempts": 2 } }, "timeInstant": 58 }, ... ]}
```


Apéndice B

EntryPoint.java

```
package es.urjc.ia.bikesurbanfleets.usersgenerator.entrypoint;

import es.urjc.ia.bikesurbanfleets.usersgenerator.SingleUser;

import java.util.List;

/**
 * It is an event generator for user appearances.
 * It represents an entry point at system geographic map where a unique user or several
users
 * appear and start interacting with the system.
 * @author IAgroup
 *
 */
public abstract class EntryPoint {

    public enum EntryPointType {
        POISSON, RANDOM, SINGLEUSER
    }

    protected EntryPointType entryPointType;

    public static int TOTAL_SIMULATION_TIME;
    /**
     * It generate single users for the configuration file,
     * which are the main events that starts the simulation execution.
     * @return a list of single users
     */

    public EntryPointType getEntryPointType() {
        return entryPointType;
    }

    public abstract List<SingleUser> generateUsers();
}
```

EntryPointPoisson.java

```
package es.urjc.ia.bikesurbanfleets.usersgenerator.entrypoint.implementations;

import es.urjc.ia.bikesurbanfleets.common.graphs.GeoPoint;
import es.urjc.ia.bikesurbanfleets.common.util.BoundingCircle;
import es.urjc.ia.bikesurbanfleets.common.util.SimulationRandom;
import es.urjc.ia.bikesurbanfleets.common.util.TimeRange;
import es.urjc.ia.bikesurbanfleets.usersgenerator.SingleUser;
import es.urjc.ia.bikesurbanfleets.usersgenerator.UserProperties;
import es.urjc.ia.bikesurbanfleets.usersgenerator.entrypoint.EntryPoint;
import es.urjc.ia.bikesurbanfleets.usersgenerator.entrypoint.distributions.DistributionPoisson;

import java.util.ArrayList;
import java.util.List;

/**
 * This class represents several users that appears at system with a Poisson
 * distribution, taking as a reference to generate them the same entry point.
 * It provides a method which generates a variable set of users.
 * Number of users that are generated depends on the value of parameter of followed
 * distribution.
 * @author IAgroup
 */
public class EntryPointPoisson extends EntryPoint {

    /**
     * If a radius is given, position is the center of circle
     * In other case, position is the specific point where user appears
     */
    private GeoPoint position;

    /**
     * It is the radius of circle is going to be used to delimit area where users
     * appears
     */
    private double radius;

    /**
     * Type of distribution that users generation will follow
     */
    private DistributionPoisson distribution;

    /**
     * Type of users that will be generated
     */
    private UserProperties userType;

    /**
     * It is the range of time within which users can appears, i. e.,
     */
    private TimeRange timeRange;

    /**
     * It is the number of users that will be generated.
     */
    private int totalUsers;

    @Override
    public List<SingleUser> generateUsers() {
        List<SingleUser> users = new ArrayList<>();
        int currentTime, endTime;
        int usersCounter = 0;
        int maximumUsers;

        if (timeRange == null) {
            currentTime = 0;
            endTime = TOTAL_SIMULATION_TIME;
        }
        else {
            currentTime = timeRange.getStart();
            endTime = timeRange.getEnd();
        }

        while (usersCounter < maximumUsers) {
            SingleUser user = new SingleUser();
            user.setPosition(position);
            user.setRadius(radius);
            user.setDistribution(distribution);
            user.setUserType(userType);
            user.setTimeRange(timeRange);
            users.add(user);
            usersCounter++;
        }
        return users;
    }
}
```

```

    }

    maximumUsers = totalUsers == 0 ? Integer.MAX_VALUE : totalUsers;

    while (currentTime < endTime && usersCounter < maximumUsers) {
        usersCounter++;
        GeoPoint userPosition;

        //If not radius is specified, user just appears in the position submitted.
        if (radius > 0) {
            BoundingCircle boundingCircle = new BoundingCircle(position, radius);
            userPosition =
        boundingCircle.randomPointInCircle(SimulationRandom.getUserCreationInstance());
        } else {
            userPosition = position;
        }
        int timeEvent = distribution.randomInterarrivalDelay();
        currentTime += timeEvent;
        SingleUser user = new SingleUser(userPosition, userType, currentTime);
        users.add(user);
    }
    return users;
}

@Override
public String toString() {
    String result = position.toString();
    result += "| EntryPointType" + this.getEntryPointType();
    result += "| distributionParameter " + distribution.getLambda() + "\n";
    result += "user Type: " + userType;
    return result;
}
}

```

DistributionPoisson.java

```

package es.urjc.ia.bikesurbanfleets.usersgenerator.entrypoint.distributions;

import es.urjc.ia.bikesurbanfleets.common.util.SimulationRandom;
/**
 * This class represents a Poisson math distribution.
 * It contains the typical parameter that characterizes a Poisson math distribution.
 * @author IAgroup
 */
public class DistributionPoisson {
    /**
     * This is the Poisson distribution characteristic parameter.
     * It represents the number of users that appear per second.
     */
    private double lambda;
    public DistributionPoisson(double lambda) {
        this.lambda = lambda;
    }
    public double getLambda() {
        return lambda;
    }
    /**
     * It calculates, according to the distribution formula, the time between a user
     * appearance and the next user appearance.
     * It calculates an exponential instant given a lambda parameter.
     * @return a realistic exponential value given a lambda parameter.
     * @see <a href="https://en.wikipedia.org/wiki/Exponential_distribution#Generating_exponential_variates">Generating exponential variates</a>
     */
    public int randomInterarrivalDelay() {
        SimulationRandom random = SimulationRandom.getUserCreationInstance();
        double randomValue = Math.log(1.0 - random.nextDouble(Double.MIN_VALUE, 1));
        Double result = -randomValue /lambda;
        Long longResult = Math.round(result);
        return longResult.intValue();
    }
}

```


Apéndice C

User.java

```
package es.urjc.ia.bikesurbanfleets.entities;

import es.urjc.ia.bikesurbanfleets.common.graphs.GeoPoint;
import es.urjc.ia.bikesurbanfleets.common.graphs.GeoRoute;
import es.urjc.ia.bikesurbanfleets.common.graphs.exceptions.GeoRouteCreationException;
import es.urjc.ia.bikesurbanfleets.common.graphs.exceptions.GeoRouteException;
import
es.urjc.ia.bikesurbanfleets.common.graphs.exceptions.GraphHopperIntegrationException;
import es.urjc.ia.bikesurbanfleets.common.interfaces.Entity;
import es.urjc.ia.bikesurbanfleets.common.util.IdGenerator;
import es.urjc.ia.bikesurbanfleets.common.util.SimulationRandom;
import es.urjc.ia.bikesurbanfleets.history.entities.HistoricUser;
import es.urjc.ia.bikesurbanfleets.history.History;
import es.urjc.ia.bikesurbanfleets.history.HistoryReference;
import es.urjc.ia.bikesurbanfleets.systemmanager.SystemManager;
import es.urjc.ia.bikesurbanfleets.users.UserMemory;

import java.util.ArrayList;
import java.util.List;

/**
 * This is the main entity of the system.
 * It represents the basic behaviour of all users type that can appear at the system.
 * It provides an implementation for basic methods which manage common information for
all kind of users.
 * It provides a behaviour pattern (make decisions) which depends on specific user type
properties.
 * @author IAgroup
 */
@HistoryReference(HistoricUser.class)
public abstract class User implements Entity {

    private static IdGenerator idGenerator = new IdGenerator();

    private int id;

    /**
     * Current user position.
     */
    private GeoPoint position;

    /**
     * Before user removes a bike or after returns it, this attribute is null.
     * While user is cycling, this attribute contains the bike the user has rented.
     */
    private Bike bike;

    /**
     * It is the station to which user has decided to go at this moment.
     */
    private Station destinationStation;

    /**
     * Speed in meters per second at which user walks.
     */
    private double walkingVelocity;

    /**
     * Speed in meters per second at which user cycles.
     */
    private double cyclingVelocity;

    /**
     * It indicates if user has a reserved bike currently.
     */
    private boolean reservedBike;

    /**
     * It indicates if user has a reserved slot currently.
     */

}
```

```

    */
private boolean reservedSlot;

    /**
     * It is the user current (bike or slot) reservation, i. e., the last reservation
     user has made.
     * If user hasn't made a reservation, this attribute is null.
     */
private Reservation reservation;

    /**
     * It is the route that the user is currently traveling through.
     */
private GeoRoute route;

    /**
     * It saves the unsuccessful facts that have happened to the user during the entire
     simulation.
     */
private UserMemory memory;

protected SystemManager systemManager;

.....
Los métodos que vienen a continuación son los que se tienen que implementar por cada tipo de usuario. El resto de
funcionalidad intermedia se ha omitido
.....


```

 /**
 * User decides if he'll leave the system when bike reservation timeout happens.
 * @param instant: itt is the time instant when h'll make this decision.
 * @return true if he decides to leave the system and false in other case (he
decides to continue at system).
 */
public abstract boolean decidesToLeaveSystemAfterTimeout(int instant);

 /**
 * User decides if he'll leave the system after not being able to make a bike
reservation.
 * @param instant: itt is the time instant when h'll make this decision.
 * @return true if he decides to leave the system and false in other case (he
decides to continue at system).
 */
public abstract boolean decidesToLeaveSystemAffterFailedReservation(int instant);

 /**
 * User decides if he'll leave the system when there're no available bikes at
station.
 * @param instant: itt is the time instant when h'll make this decision.
 * @return true if he decides to leave the system and false in other case (he
decides to continue at system).
 */
public abstract boolean decidesToLeaveSystemWhenBikesUnavailable(int instant);

 /**
 * User decides to which station he wants to go to rent a bike.
 * @param instant: it is the time instant when he needs to make this decision.
 * @return station where user has decided to go.
 */
public abstract Station determineStationToRentBike(int instant);

 /**
 * User decides to which station he wants to go to return his bike.
 * @param instant is the time instant when he needs to make this decision.
 * @return station where user has decided to go.
 */
public abstract Station determineStationToReturnBike(int instant);

 /**
 * User decides if he'll try to make again a bike reservation at the previosly
 * chosen station after timeout happens.
 * @return true if user decides to reserve a bike at the initially chosen station.
 */
public abstract boolean decidesToReserveBikeAtSameStationAfterTimeout();

```


```

```

/**
 * User decides if he'll try to make a bike reservation at a new chosen station.
 * @return true if user decides to reserve a bike at that new station and false in
other case.
 */
public abstract boolean decidesToReserveBikeAtNewDecidedStation();

/**
 * User decides if he'll try to make again a slot reservation at the previously
 * chosen station after timeout happens.
 * @return true if user decides to reserve a slot at the initially chosen station.
 */
public abstract boolean decidesToReserveSlotAtSameStationAfterTimeout();

/**
 * User decides if he'll try to make a slot reservation at a new chosen station.
 * @return true if user decides to reserve a slot at that new station and false in
other case.
 */
public abstract boolean decidesToReserveSlotAtNewDecidedStation();

/**
 * User decides the point (it is not a station) to which he wants to ride the rented
bike
 * after removing it from station.
 * @return the point where he wants to go after making his decision.
 */
public abstract GeoPoint decidesNextPoint();

/**
 * Just after removing the bike, user decides if he'll ride it directly to a
station,
 * in order to return it.
 * @return true if user decides to cycle directly to a station in order to return
 * his bike and false in other case (he decides to ride it to another point before
returning it).
 */
public abstract boolean decidesToReturnBike();

/**
 * When timeout happens, he decides to continue going to that chosen station or to
go to another one.
 * @return true if user chooses a new station to go and false if he continues to the
previously chosen one.
 */
public abstract boolean decidesToDetermineOtherStationAfterTimeout();

/**
 * The user chooses the route which he'll travel to arrive at selected destination.
 * @param routes It's a list of possible routes to the chosen destination.
 * @return the route which the user will follow.
 */
public abstract GeoRoute determineRoute(List<GeoRoute> routes) throws
GeoRouteException;

/**
 * When user hasn't been able to make a reservation at the destination station,
 * he decides if he wants to choose another station to which go.
 * @return true if he decides to determine another destination station and false in
 * other case (he keeps his previously decision).
 */
public abstract boolean decidesToDetermineOtherStationAfterFailedReservation();

```


Apéndice D

GraphManager.java

```
package es.urjc.ia.bikesurbanfleets.common.graphs;

import es.urjc.ia.bikesurbanfleets.common.graphs.exceptions.GeoRouteCreationException;
import es.urjc.ia.bikesurbanfleets.common.graphs.exceptions.GraphHopperIntegrationException;
import java.util.List;

/**
 * This interface provides methods to manage the routes of a geographic map.
 * @author IAgroup
 *
 */
public interface GraphManager {

    List<GeoRoute> obtainAllRoutesBetween(GeoPoint originPoint, GeoPoint
destinationPoint) throws GeoRouteCreationException, GraphHopperIntegrationException;

    /**
     * It calculates which is the shortest route.
     * @return the shortest route of all possible routes between 2 points.
     * @throws GeoRouteCreationException
     * @throws GraphHopperIntegrationException
     */
    GeoRoute obtainShortestRouteBetween(GeoPoint originPoint, GeoPoint destinationPoint)
throws GraphHopperIntegrationException, GraphHopperIntegrationException,
GeoRouteCreationException;

    /**
     * It indicates if there are more than one possible route between two points.
     * @return true if there're several possible routes between 2 points or false in
other case.
     * @throws GraphHopperIntegrationException
     * @throws GeoRouteCreationException
     */
    boolean hasAlternativesRoutes(GeoPoint startPosition, GeoPoint endPosition) throws
GraphHopperIntegrationException, GeoRouteCreationException;
}
```

GraphManager.java

```
package es.urjc.ia.bikesurbanfleets.common.graphs;

import com.graphhopper.GHRequest;
import com.graphhopper.GHResponse;
import com.graphhopper.GraphHopper;
import com.graphhopper.PathWrapper;
import com.graphhopper.reader.osm.GraphHopperOSM;
import com.graphhopper.routing.util.EncodingManager;
import com.graphhopper.util.PointList;
import com.graphhopper.util.shapes.GHPoint3D;
import es.urjc.ia.bikesurbanfleets.common.graphs.exceptions.GeoRouteCreationException;
import es.urjc.ia.bikesurbanfleets.common.graphs.exceptions.GraphHopperIntegrationException;
import org.apache.commons.io.FileUtils;

import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

public class GraphHopperIntegration implements GraphManager {

    private final String GRAPHHOPPER_DIR = "graphhopper_files";

    private GraphHopper hopper;
    private GHResponse rsp;

    private GeoPoint startPosition;
    private GeoPoint endPosition;

    public GraphHopperIntegration(String mapDir) throws IOException {
        FileUtils.deleteDirectory(new File(GRAPHHOPPER_DIR));
        this.hopper = new GraphHopperOSM().forServer();
        hopper.setDataReaderFile(mapDir);
        hopper.setGraphHopperLocation(GRAPHHOPPER_DIR);
        hopper.setEncodingManager(new EncodingManager("foot"));
        hopper.importOrLoad();
    }

    private GeoRoute responseGHToRoute(PathWrapper path) throws
GeoRouteCreationException{
    List<GeoPoint> geoPointList = new ArrayList<>();
    PointList ghPointList = path.getPoints();
    geoPointList.add(startPosition);
    GHPoint3D prev = null;
    for(GHPoint3D p: ghPointList) {
        if(prev != null && p.equals(prev)){
            continue;
        }
        geoPointList.add(new GeoPoint(p.getLatitude(), p.getLongitude()));
        prev = p;
    }
    geoPointList.add(endPosition);
    GeoRoute route = new GeoRoute(geoPointList);
    return route;
}

    private void calculateRoutes(GeoPoint startPosition, GeoPoint endPosition) throws
GraphHopperIntegrationException {
        if(!startPosition.equals(this.startPosition) ||
!endPosition.equals(this.endPosition)) {
            GHRequest req = new GHRequest(
                startPosition.getLatitude(), startPosition.getLongitude(),
                endPosition.getLatitude(), endPosition.getLongitude())
                .setWeighting("fastest")
                .setVehicle("foot");
            GHResponse rsp = hopper.route(req);

            if(rsp.hasErrors()) {
                for(Throwable exception: rsp.getErrors()) {

```

```

        throw new GraphHopperIntegrationException(exception.getMessage());
    }
}
this.rsp = rsp;
this.startPosition = startPosition;
this.endPosition = endPosition;
}
}

@Override
public GeoRoute obtainShortestRouteBetween(GeoPoint startPosition, GeoPoint
endPosition) throws GraphHopperIntegrationException, GeoRouteCreationException {
    calculateRoutes(startPosition, endPosition);
    PathWrapper path = rsp.getBest();
    return responseGHToRoute(path);
}

@Override
public List<GeoRoute> obtainAllRoutesBetween(GeoPoint startPosition, GeoPoint
endPosition) throws GraphHopperIntegrationException, GeoRouteCreationException {
    calculateRoutes(startPosition, endPosition);
    List<GeoRoute> routes = new ArrayList<>();
    for(PathWrapper p: rsp.getAll()) {
        routes.add(responseGHToRoute(p));
    }
    return routes;
}

@Override
public boolean hasAlternativesRoutes(GeoPoint startPosition, GeoPoint endPosition)
throws GraphHopperIntegrationException {
    calculateRoutes(startPosition, endPosition);
    return rsp.hasAlternatives();
}
}

```

GeoRoute.java

```
package es.urjc.ia.bikesurbanfleets.common.graphs;

import com.google.gson.annotations.Expose;
import es.urjc.ia.bikesurbanfleets.common.graphs.exceptions.GeoRouteCreationException;
import es.urjc.ia.bikesurbanfleets.common.graphs.exceptions.GeoRouteException;

import java.util.ArrayList;
import java.util.List;

/**
 * This class represents a geographic route.
 * @author IAgroup
 */
public class GeoRoute {

    /**
     * These are the points which forms the route.
     */
    @Expose
    private List<GeoPoint> points;

    /**
     * This is the distance of the entire route.
     */
    @Expose
    private double totalDistance;

    /**
     * These are the distances between the different pairs of geographic points of the
     * route.
     * The number of subroutes is the number of points of the route minus one.
     */
    private List<Double> intermediateDistances;

    /**
     * It creates a route which has at least 2 points.
     * @param geoPointList It is the list of points which form the route.
     */
    public GeoRoute(List<GeoPoint> geoPointList) throws GeoRouteCreationException {
        if(geoPointList.size() < 2) {
            throw new GeoRouteCreationException("Routes should have more than two
points");
        }
        else if(geoPointList.get(0).equals(geoPointList.get(geoPointList.size()-1))) {
            this.points = new ArrayList<>();
            this.totalDistance = 0;
        }
        else {
            this.points = geoPointList;
            this.intermediateDistances = new ArrayList<>();
            calculateDistances();
        }
    }

    public List<GeoPoint> getPoints() {
        return points;
    }

    public double getTotalDistance() {
        return totalDistance;
    }

    /**
     * It calculates the distances of the different sections of the route and its total
     * distance.
     */
    private void calculateDistances() {
        Double totalDistance = 0.0;
        for(int i = 0; i < points.size()-1; i++) {
            GeoPoint currentPoint = points.get(i);
            GeoPoint nextPoint = points.get(i+1);
            Double currentDistance = currentPoint.distanceTo(nextPoint);
            intermediateDistances.add(currentDistance);
        }
    }
}
```

```

        totalDistance += currentDistance;
    }
    this.totalDistance = totalDistance;
}

public GeoPoint calculatePositionByTimeAndVelocity(double finalTime, double
velocity) throws GeoRouteException, GeoRouteCreationException {
    double totalDistance = 0.0;
    double currentTime = 0.0;
    double currentDistance = 0.0;
    GeoPoint currentPoint = null;
    GeoPoint nextPoint = null;
    int i = 0;
    while(i < points.size()-1 && currentTime < finalTime) {
        currentPoint = points.get(i);
        nextPoint = points.get(i+1);
        currentDistance = intermediateDistances.get(i);
        totalDistance += currentDistance;
        currentTime += currentDistance/velocity;
        i++;
    }
    if(currentTime < finalTime) {
        throw new GeoRouteException("Can't create intermediate position");
    }
    double x = totalDistance - finalTime*velocity;
    double intermedDistance = currentDistance - x;
    GeoPoint newPoint = currentPoint.reachedPoint(intermedDistance, nextPoint);
    return newPoint;
}

@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    GeoRoute other = (GeoRoute) obj;
    if (Double.doubleToLongBits(totalDistance) !=
Double.doubleToLongBits(other.totalDistance))
        return false;
    if (points == null) {
        if (other.points != null)
            return false;
    } else if (!points.equals(other.points))
        return false;
    return true;
}

@Override
public String toString() {
    String result = "Points: \n";
    for(GeoPoint p: points) {
        result += p.getLatitude() + "," + p.getLongitude() + "\n";
    }
    result += "Distance: " + totalDistance + " meters \n";
    result += "Distances between points: ";
    for(Double d: intermediateDistances) {
        result += d + "\n";
    }
    return result;
}
}

```

Apéndice E

Ejemplo de esquema de usuario para formulario dinámico

```
const Percentage = sNumber().min(0).max(100);  
  
const typeParameters = {  
    USER_RANDOM: {},  
    USER_UNINFORMED: {},  
    USER_INFORMED: {  
        willReserve: sBoolean(),  
        minReservationAttempts: UInt,  
        minReservationTimeouts: UInt,  
        minRentalAttempts: UInt,  
        bikeReservationPercentage: Percentage,  
        slotReservationPercentage: Percentage,  
        reservationTimeoutPercentage: Percentage,  
        failedReservationPercentage: Percentage  
    },  
};  
...  
}  
;
```

Ejemplo de formulario dinámico a partir del esquema anterior

The screenshot shows a user interface for generating a configuration file. On the left is a map of Madrid, Spain, with various neighborhoods and landmarks labeled. A red circle highlights a specific area in the center of the city. To the right of the map are two panels: 'General Configuration' and 'Entry Points'. The 'General Configuration' panel contains fields for 'Bounding Box' (North West Latitude: 40,43974941261535, South East Latitude: 40,38979182120838), 'Longitude' (-3,743247985839844, -3,6282348632812504), 'Debug Mode' (checkbox), 'Linear Distance' (checkbox), 'Map' (checkbox), 'Random Seed' (text input), 'Reservation Time' (text input), and 'Total Simulation Time' (text input). The 'Entry Points' panel is currently active, showing a list of parameters: 'End', 'Start', 'Total Users', 'User Type', 'Parameters', 'Bike Reservation Percentage', 'Failed Reservation Percentage', 'Min Rental Attempts', 'Min Reservation Attempts', 'Min ReservationTimeouts', 'Reservation Timeout Percentage', 'Slot Reservation Percentage', and a checkbox for 'Will Reserve'. Below these fields is a note stating '* = required fields' and a 'Submit' button. At the bottom of the interface are 'Close' and 'Generate Configuration' buttons.

Por el momento se ordenan por orden alfabético. También aparecen los campos del entry point que se quiere crear.

5 Referencias

- [1] J. Sutherland y J. Sutherland, «La guía de Scrum,» 2013. [En línea]. Available: <http://www.scrumguides.org/docs/scrumguide/v1/scrum-guide-es.pdf>.
- [2] I. Sommerville, de *Ingeniería del software*, 2005, pp. 62-68.
- [3] «Semantic Versioning 2.0.0,» [En línea]. Available: <https://semver.org/>.
- [4] R. M. Agut, «Especificación de Requisitos Software según IEEE 830,» 2000-2001. [En línea]. Available: http://zeus.inf.ucv.cl/~bcrawford/AULA_ICI_3242/ERS_IEEE830.pdf.
- [5] J. Preshing, «Preshing on Programming,» [En línea]. Available: <http://preshing.com/20111007/how-to-generate-random-timings-for-a-poisson-process/>.
- [6] D. Knuth, «The Art of Computer Programming - Volume 2,» 1983, p. 116.
- [7] S. T. Mayo, Bike Sharing System Simulator - Core And Data Analysis.
- [8] T. Cumplido, An extensible visualization component for a bike sharing simulator.
- [9] D. Chemla, F. Meunier, T. Pradeau y R. W. Calvo, «Self-service bike sharing systems: simulation,» 2013. [En línea]. Available: <https://hal.archives-ouvertes.fr/hal-00824078/document>.