



**ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA
INFORMÁTICA**

GRADO EN INGENIERÍA DEL SOFTWARE

Curso académico 2018/2019

Trabajo Fin de Grado

**Arquitectura, diseño y configuración de un simulador de bicicletas
compartidas**

**Autor: Carlos Ruiz Ballesteros
Tutor: Holger Billhardt**

*Dedico este trabajo a mis padres por su apoyo y dedicación,
a Eva por su paciencia y cariño,
a Oscar, Tao, Sandra por tan buenos momentos de trabajo,
a mis tutores de proyecto Holger y Alberto,
y a todas las personas que sin darme cuenta
han estado ahí.*

Índice

Resumen	9
1. Introducción	11
1.1. Motivación	11
1.2. Contexto	12
1.3. Objetivos	13
1.4. Metodología	14
2. Descripción informática	16
2.1. Especificación de requisitos	16
2.1.1. Perspectiva general del producto	16
2.1.2. Requisitos funcionales	17
2.1.3. Requisitos no funcionales	19
2.2. Análisis	19
2.2.1. Acciones de los usuarios	20
2.2.2. Interacción de los usuarios con el sistema de recomendaciones	21
2.2.3. Definición de un estado inicial (configuración)	22
2.3. Diseño	30
2.3.1. Arquitectura general	31
2.3.2. Simulador basado en eventos	31
2.3.3. Arquitectura detallada	33
2.4. Implementación	37
2.4.1. Tecnologías	38
2.4.2. Lógica del simulador (Backend)	39
2.4.3. Inicialización y ejecución del simulador	47
2.4.4. Aplicando reflexión para una mejor extensibilidad	47
2.4.5. Carga de mapas y cálculo de rutas	49
2.4.6. Interfaz de usuario del simulador y formularios dinámicos(Frontend) . . .	52
2.4.7. Formularios dinámicos en las configuraciones	55
3. Evaluación	58
3.1. Prueba simulador	61
4. Conclusiones	65
4.1. Lecciones aprendidas	65
4.2. Líneas futuras	66
5. Referencias	67
6. Anexos	68
Anexo 1 - Archivos de configuración	68
Configuración global	68
Estaciones	69
Puntos de entrada (Entry points)	69
Usuarios generados	70
Anexo 2 - Manual de configuración del entorno de desarrollo	70
Prerrequisitos	70
Setup	71

Ejecutar el simulador en el entorno de desarrollo	71
Comandos básicos para desarrollo	71
Anexo 3 - Como añadir un entry point	72
1. Creamos una clase para el Entry Point.	72
2. Añadir el entry point a los schemas	73
3. Compilar backend y schemas	73
4. Probar el entry point para crear una configuración.	74
Anexo 4 - Manual para crear implementaciones de usuario	75
1. Creación de la clase	75
2. Implementación de métodos	76
3. Añadir el usuario a los schemas	80
4. Compilar backend y schemas	81

Resumen

Imaginemos por un momento que necesitamos ir de un sitio a otro en nuestra ciudad y disponemos de un sistema de alquiler de bicicletas por estaciones. Como sabemos los sistemas de bicicletas públicas, ofrecen a los usuarios bicis en estaciones situadas en puntos concretos de la ciudad para poder utilizarlas como medio de transporte de un punto a otro. Estas estaciones están situadas en lugares concretos de la ciudad y podemos desplazarnos de un sitio a otro, cogiendo las bicis en una estación y dejándolas en otra cerca de nuestro destino.

Con la llegada de los smartphones, la utilización de estos sistemas es más sencilla, ya que los usuarios pueden disponer de mucha información del sistema a través de aplicaciones software. Estas aplicaciones pueden ofrecerle al usuario estaciones donde coger y dejar una bici en función del destino al que quieran ir. En sistemas como BiciMad es posible reservar bicicletas, ver el estado actual de todas las estaciones, etc.

En ciertas ocasiones, este tipo de sistemas pueden saturarse en estaciones concretas, haciendo que la experiencia de los usuarios que quieran utilizarlo no sea tan satisfactoria, por lo que es necesario aplicar soluciones. Hay que plantear posibles estrategias de balanceo para prever esas situaciones problemáticas. El objetivo es asegurar la disponibilidad tanto de bicis como de huecos en todas las estaciones, para que los usuarios puedan coger y devolver una bici en todas ellas.

Muchas soluciones y estrategias de balanceo son muy difíciles de probar directamente en el sistema real, ya que habría que implementar muchas cosas en un sistema en funcionamiento, y quizás nuestra estrategia de balanceo no sea la más adecuada, ocasionando gastos innecesarios y empeorando la experiencia de los usuarios. Por tanto para probar la viabilidad de las estrategias que se planteen es mejor usar un **simulador** capaz de probar estos sistemas y algoritmos de balanceo propuestos.

En eso se centra este trabajo, en la implementación de un simulador capaz de recrear de la manera más realista posible, un sistema de bicis compartidas. Este simulador deberá permitir la posibilidad de tener diferentes tipos de usuario, implementar un sistema de recomendaciones, implementar un sistema de incentivos y poder probar cualquier sistema de bicis públicas en cualquier lugar del mundo. En definitiva, crear un simulador que nos permita crear, simular y analizar situaciones reales e implementar sistemas de balanceo para comprobar su viabilidad. Se crearán archivos de configuración para crear situaciones con estaciones y ciudades reales. Posteriormente tras las simulaciones se podrán analizar los datos obtenidos y visualizarlos para comprobar su eficacia.

1. Introducción

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

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 Vèlib² 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 los 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, es decir, asegurar la disponibilidad de bicis y huecos en todas las 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 alquiler de bicis y probarlo en la realidad, puede ser muy tedioso y costoso.

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 probar y evaluar diferentes estrategias y acciones de control cuya finalidad es mejorar el balanceo.

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 probar soluciones en algunas ciudades y trasladarlas a otras.

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

¹BiciMad: <https://bicimad.com/>.

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

necesario que se pueda añadir en el código usuarios para que cualquier persona con conocimientos de programación pueda implementar sus propios algoritmos de comportamiento 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 dentro del ámbito de los sistemas de bicis compartidas.

1.2. Contexto

Para definir las diferentes partes de el simulador, es necesario tener una visión general del sistema de bicis en la realidad, que se presenta en la Figura 1

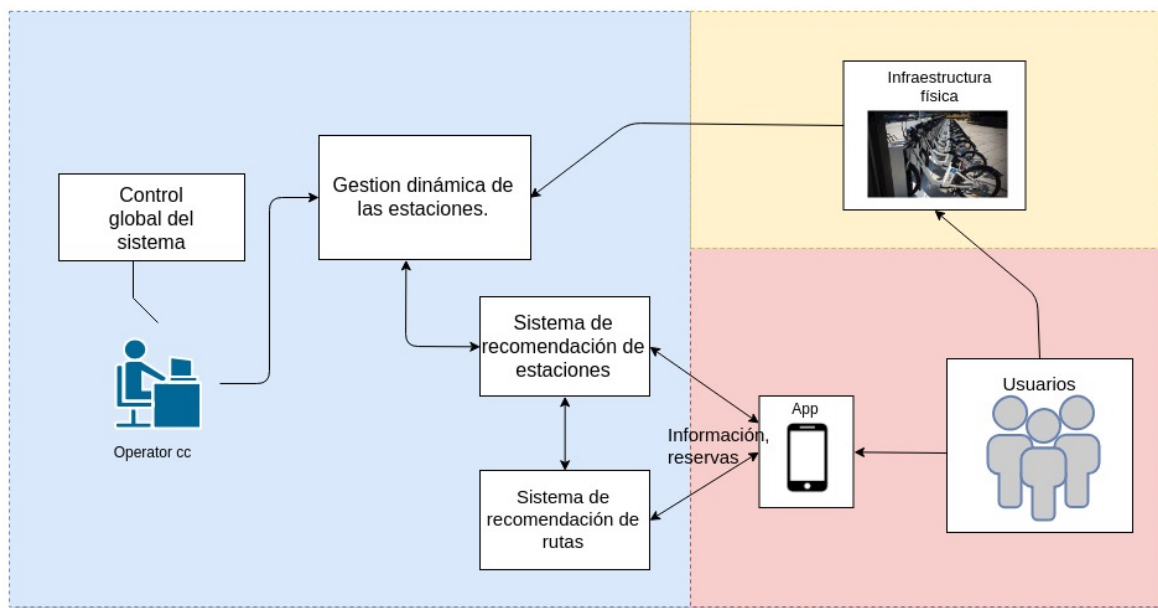


Figura 1: Sistema de bicis compartidas

Podemos distinguir claramente en esta figura tres partes principales dentro del sistema de bicis compartidas:

- Infraestructura física con estaciones y bicis (Amarillo).
- Usuarios con Smartphone y App del sistema (Rojo).
- Sistema de recomendaciones y gestión del sistema (Azul).

Estas tres partes diferenciadas constituyen tres partes importantes dentro del desarrollo del simulador. El **núcleo**, que incluirá la infraestructura y como deben los usuarios interactuar con el sistema, los **usuarios** que pueden tener comportamientos diferentes, y el **sistema de recomendaciones y de información** que podrá influir en el comportamiento de los usuarios.

Los usuarios hacen uso de la infraestructura cuando cogen o dejan una bici y también pueden hacer uso del sistema de recomendaciones a través de la App, para reservar una bici o un

hueco o ver el estado de una estación. Vemos que hay una interacción continua entre usuarios e infraestructura. El sistema de recomendaciones puede influenciar en las decisiones finales del usuario.

En el desarrollo de este simulador hemos participado varias personas hasta la fecha de publicación de esta memoria, las cuales han realizado diferentes partes, aunque parte de este desarrollo ha sido realizado de forma conjunta debido a la necesidad de tener una base común, que serían el núcleo y algunos estándares definidos.

Con esta visión global del sistema se especifica una serie de objetivos que se detallan a continuación.

1.3. Objetivos

Antes de plantear que objetivos perseguir, hay que analizar que necesidades va a tener el simulador en el futuro y con que fin se va a utilizar en términos globales. 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 disponibilidad de bicis y huecos en todas las estaciones.
- Plantear nuevas distribuciones de estaciones sobre una ciudad.

Basado en estos objetivos globales se derivan los siguientes objetivos más concretos para el desarrollo:

- Recrear infraestructuras reales en ciudades reales.
- Generar usuarios en cualquier punto de la ciudad.
- Generación de usuarios versátil y que puedan seguir distribuciones (Poisson).
- Poder definir qué tipos de usuarios queremos en el sistema y parametrizarlos para que puedan tener comportamientos variados.
- Facilidad para crear configuraciones (GUI).
- Creación de distintos tipos de recomendadores.
- Simulación realista.
- Análisis de los datos para evaluar los diferentes algoritmos y estrategias de recomendación.
- Visualizar las simulaciones.

En general el simulador tiene que ser capaz de recrear situaciones reales basadas en entornos reales, con infraestructuras existentes o que puedan existir en el mundo real. Es decir, uno de los objetivos básicos es dotar al simulador de mecanismos ágiles para poder crear configuraciones de diferentes situaciones.

Además estas configuraciones tienen que poder ser generadas con cierta independencia del simulador, añadiendo la posibilidad de que nuevos desarrolladores puedan crear sus propias herramientas que generen casos para la simulación y ofreciendo una interfaz ágil para la realización de simulaciones.

En el contexto global del desarrollo, la parte que ha correspondido al trabajo presentado en este

documento y que se presenta en mayor detalle es:

- Configuración.
- Interfaz de usuario.
- Desarrollo ágil de nuevos usuarios, sistemas de recomendación, generación de usuarios...
- Extensibilidad.

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 los tutores de proyecto) y hay casi una comunicación total día a día con ellos, sin roles intermediarios. Sin embargo, sí que se tienen reuniones cada semana en el equipo para ver cómo avanza el proyecto, retrospectivas, prototipos, integración, pruebas, etc. 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.

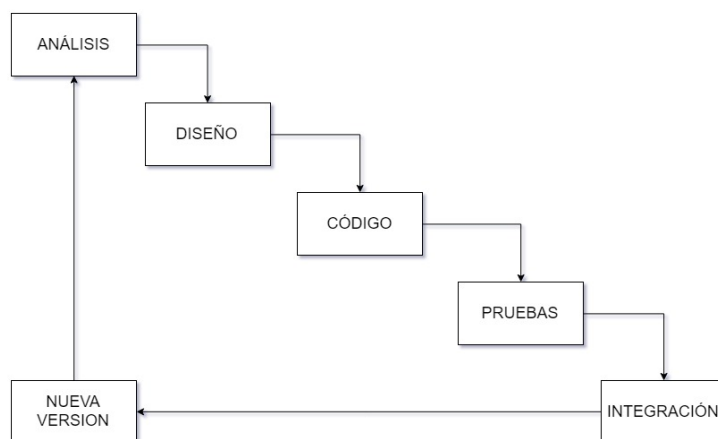


Figura 2: Ciclo iterativo e incremental

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 del sistema.

Cada 2 semanas se ha realizado una iteración donde se realizaron todos los pasos comentados anteriormente, donde todo el equipo de desarrollo decidió qué requisitos son más críticos e importantes y que bugs se detectarían. En base a estas decisiones y utilizando herramientas online como Trello³, se gestionó que tareas debe realizar cada uno.

³Es 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 que se está creando.

Un desarrollo iterativo e incremental ofrece varias ventajas con respecto a otras metodologías como puede ser el desarrollo en cascada. Una de las ventajas que ofrece es la entrega de software que se puede usar a mitad de desarrollo, mientras que en el modelo en cascada cada fase del proceso debe ser finalizada (firmada) para pasar a la siguiente fase. El desarrollo de software no es lineal y esto crea dificultades si se utiliza una metodología en cascada[2].

Cada cierto tiempo se han realizado releases. 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, corrección de bugs) y Z, que son micro versiones (parches, pequeños bugs críticos...).

2. Descripción informática

El presente capítulo aborda toda la fase de desarrollo del proyecto, desde la especificación de requisitos, el diseño y la implementación del mismo, contextualizando el trabajo realizado por todos los integrantes del equipo de desarrollo y explicando con más detalle la parte correspondiente de nuestra memoria.

2.1. Especificación de requisitos

Antes de comenzar las iteraciones y primeros prototipos del proyecto, es necesario crear 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 modificables con frecuencia.

2.1.1. Perspectiva general del producto

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

Necesitamos un software que sea capaz de, a partir de un estado inicial:

- a. Simular situaciones reales.
- b. Visualizar estas simulaciones
- c. Presentar una serie de datos correspondientes de la simulación.

El simulador tiene que mostrar estos datos a través de un histórico que posteriormente se utilizará para analizar los resultados del sistema de recomendaciones y de 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 comprobar el comportamiento de los distintos algoritmos de balanceo implementados.

Este TFG se centrará en dos partes:

- Configuración: Será diseñado con el objetivo de inicializar el simulador utilizando la ubicación de una ciudad real, poder generar usuarios en zonas específicas, parametrizar valores globales de la simulación, ubicar estaciones en cualquier punto y parametrizar que tipos de usuarios se van a utilizar, en que zonas y con qué distribución aparecerán.
- Participación en el desarrollo global, que incluye la separación del simulador en módulos, la aplicación de ciertos patrones de diseño y la implementación de la parte gráfica de configuración y simulación.

En las siguientes subsecciones se detallan los requisitos funcionales y no funcionales que corresponden a la parte del desarrollo realizado en el ámbito de este TFG. En la descripción se emplean las siguientes abreviaturas:

- 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.
- Usuario simulado (US): Agente simulado que interactúa dentro del sistema de bicis compartidas y que pueden hacer uso del sistema de recomendaciones y el sistema de bicis compartidas.
- Alquiler: 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).

2.1.2. Requisitos funcionales

Los requisitos funcionales consisten en una serie de comportamientos o módulos que deben ser integrados en el sistema de simulación y son los siguientes:

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 pueden cambiar, quitarse o añadirse según el progreso y uso del simulador. El fichero de configuración deberá especificar:

- Un parámetro (semilla) para la generación de sucesos aleatorios. Esta semilla permite 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.

Requisito funcional 2

Fichero de configuración de estaciones. Se podrá mediante un fichero de configuración, definir 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

Fichero de configuración para los US: La configuración deberá proporcionar un mecanismo o módulo con el cual se puedan generar usuarios en distintos puntos geográficos. Estos lugares donde los usuarios son generados se denominarán puntos de entrada (entry points). Este módulo deberá ser capaz de generar usuarios de dos formas:

- Con una distribución exponencial siguiendo un proceso de Poisson. Dada una cantidad de usuarios por una unidad de tiempo y un punto geográfico, generarlos como un proceso de Poisson en dicho punto. En este proceso de poisson λ será la cantidad de usuarios por segundo. (Véase 2.2.3.4).
- Usuarios individuales: Dado un punto geográfico, generar un único usuario.

Este fichero de configuración no tendrá límite de puntos de entrada, y podrán definirse más métodos de generación de usuarios.

Requisito funcional 4

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.
- Ficheros de configuración con usuarios generados a partir del anterior fichero.

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 simular.

Requisito funcional 5

Generador de usuarios: Del requisito anterior podemos deducir que debería 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 6

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

Requisito funcional 7

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 8

Herramientas GUI: La GUI deberá proporcionar las siguientes herramientas:

- Crear y cargar configuraciones

- Visualizar históricos
- Analizar y exportar datos.

Además la GUI debe facilitar la creación de los distintos elementos a través de un mapa. Los elementos de la configuración que se vayan añadiendo, deberán verse en un mapa y en una vista en forma de árbol para que sean accesibles.

2.1.3. 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 ser utilizado y desarrollado en las plataformas 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 de los ficheros 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 con un sistema para la verificación de datos.

2.2. Análisis

Previamente a la implementación es necesario hacer un análisis del sistema en la realidad y abstraer dichos conceptos para poder crear el simulador. A continuación se describen las tres partes esenciales en las que se ha trabajado en este TFG:

- Acciones de los usuarios: Debemos analizar que pueden hacer los usuarios en el sistema y que decisiones pueden tomar.
- Interacción de los usuarios con el sistema de recomendaciones: Los usuarios podrán tomar decisiones basándose en información externa proporcionada por aplicaciones móviles y de su entorno.

- Definición de un estado inicial (configuración): Para poder simular en su totalidad estas acciones y decisiones de los usuarios en el sistema, debemos partir de un estado inicial, que debe ser definido a partir de una configuración. Se hace especial incapié en esta última parte, ya que representa una gran parte del trabajo de este TFG.

En los siguientes subapartados se mencionan las distintas partes del simulador que se corresponden con cada punto descrito anteriormente.

2.2.1. Acciones de los usuarios

En el núcleo definiremos como se ejecutarán las acciones de las entidades en el simulador. Para definir como se van a ejecutar estas acciones vamos a plantear un diagrama de flujo de las posibles acciones del usuario dentro del sistema.

Estas 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 implementada como un **simulador basado en eventos discretos**.

Si analizamos el comportamiento de un usuario en el sistema de bicis real, se pueden identificar la siguiente secuencia de acciones y decisiones:

- Cuando un usuario aparece en el sistema elegirá primero a que estación quiere ir y posteriormente si reservar o no una bicicleta. Si hay bicicletas disponibles en dicha estación, podrá reservar una de ellas, aunque puede suceder que la reserva expire (ya que las reservas podrán contar con un tiempo límite de expiración). Si esto ocurre podrá decidir entre:
 - Abandonar el sistema.
 - Continuar hacia la estación sin reserva para coger una bici si está disponible.
 - Repetir el proceso y decidir a que otra estación ir y si reservar o no.
- Si el usuario llega a la estación sin reserva y no hay bicis disponibles, puede tomar la decisión de abandonar el sistema o decidir si ir a otra estación y repetir el mismo proceso.
- Una vez el usuario ha conseguido una bicicleta, deberá decidir si ir directamente a la estación para devolverla, o dar una vuelta por la ciudad para posteriormente, devolverla en una estación.
- Si el usuario reserva un anclaje y hay disponibles, podrá devolver la bici sin problemas y abandonar el sistema. Las reservas de anclajes también pueden expirar por lo que si la reserva expira, tendrá que decidir entre:
 - Continuar hacia la estación y devolverla si hay anclajes disponibles.
 - Repetir el proceso y decir a que otra estación ir para devolver la bici y si reservar o no.
- Si el usuario llega a la estación sin reserva de anclaje, puede suceder que no haya anclajes

libres. En tal caso tendrá que volver a decidir a que estación ir para devolverla y si reservar o no. El proceso se repite hasta que el usuario consigue devolver la bicicleta.

En la figura 3 se puede ver el flujo de decisiones del usuario con más detalle.

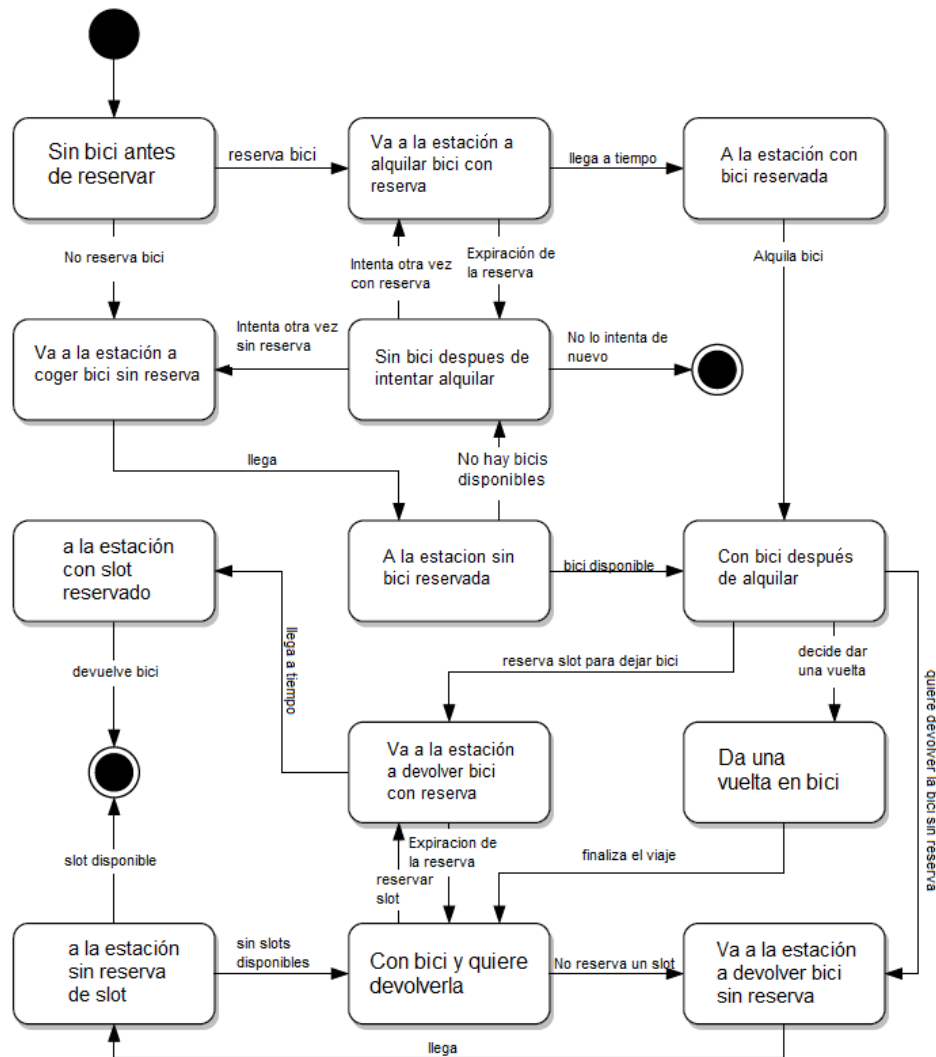


Figura 3: Flujo de decisiones de un Usuario Simulado

2.2.2. Interacción de los usuarios con el sistema de recomendaciones

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

Si pensamos 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 proporcione información, puede tener un amplio conocimiento del SBC en ese momento, además de poder seguir consejos, recomendaciones, etc.

Los US podrán obtener información del estado actual de las estaciones, o recomendaciones a través de distintas interfaces que proporcionará el simulador a los usuarios. Por ejemplo, una interfaz para la información de la infraestructura (estaciones y bicis) y otra para las recomendaciones. Podríamos considerar estas interfaces como las “apps” que utilizan los usuarios en el sistema real, y cada usuario, según su implementación, podrá o no hacer uso de ellas.

En la figura 4 vemos un diagrama en el que se explica cómo los US podrían consultar al SBC.

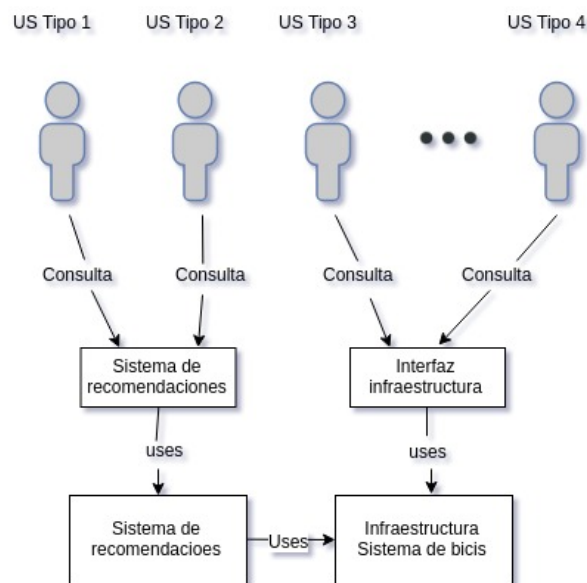


Figura 4: Usuarios simulados utilizando la información del sistema de recomendaciones y la infraestructura

El simulador podrá disponer en un futuro de más de un sistema de recomendaciones, pero de momento se ha incluido uno, el cual ha sido implementado por otra desarrolladora del simulador (Sandra Timón [5]).

La diferencia entre la interfaz de infraestructura y el sistema de recomendaciones, es que el primero da información del estado del sistema en ese momento, y el de recomendaciones, los criterios que aplica para realizar estas recomendaciones, responden a los intereses del sistema para balancearlo.

2.2.3. Definición de un estado inicial (configuración)

Para comenzar una simulación, será necesario establecer una serie de parámetros con los que poder inicializarlo. Estos parámetros serán archivos de configuración que deberán poder ser

legibles y modificables utilizando el formato JSON. Además proporcionaremos una interfaz de usuario para facilitar la creación de estos archivos. Estos archivos deben contener información acerca de la infraestructura (estaciones, bicis), de cómo y donde aparecerán los usuarios y de ciertos parámetros globales necesarios para controlar la simulación.

2.2.3.1. Configuración de las estaciones Para que los usuarios puedan ir a las estaciones y realizar acciones en ellas, debemos proporcionar al simulador información sobre dónde está cada estación y de cuantas bicis dispone en ese momento. Para ello debemos proporcionar un archivo de configuración que contenga esta información.

El contenido del archivo de configuración de las estaciones contiene:

- Capacidad: Número de anclajes disponibles.
- Cantidad de bicis
- Posición geográfica.

Con esta información aportada al simulador, se podrán realizar simulaciones en cualquier SBC del mundo, simplemente proporcionando la información necesaria al archivo de configuración.

2.2.3.2. Configuración global de la simulación. 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).
- 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 se pueda obtener la misma aleatoriedad de una simulación a otra, permitiendo así poder repetir pruebas y compartir simulaciones con otras personas que utilicen el simulador.

2.2.3.3. Configuración para la aparición de usuarios Para configurar los usuarios es necesario representar mediante unos datos iniciales cómo van a aparecer. Habrá dos posibles formas de realizar esto:

- Crear un fichero de configuración con el instante exacto en el que aparece un usuario.

- Crear un fichero con puntos de entrada de usuarios (Entry Points).

¿Qué es un Entry Point?

Para representar cómo van a aparecer los usuarios sin especificar el momento exacto de tiempo en el que aparecen necesitaremos un conjunto de datos que nos proporcione esta información. Estos datos son los Entry Points. Así pues 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 una distribución.

En este caso en particular nos interesa que los usuarios sean generados en puntos geográficos concretos siguiendo un proceso de Poisson y que estén distribuidos dado un radio de forma uniforme en el área abarcado por este. Un Entry Point de estas características podría tener las siguientes propiedades:

- Una posición geográfica y un radio, con los que establecer un área de aparición para los usuarios.
- Parámetro lambda: Un proceso de Poisson se caracteriza por un único parámetro lambda.
- Instante inicial y final: Rango de tiempo en el que se quiere que aparezcan los usuarios.
- Tipo de usuario: Tipo de usuario simulado que aparecerá. Recordemos que cada tipo de usuario tendrá un comportamiento diferente.

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

- Posición geográfica
- Instante de aparición: Momento exacto en el que el usuario aparece.
- Tipo de usuario y parámetros propios.

En definitiva, un **Entry Point** es un concepto genérico de entrada al simulador y pueden variar sus parámetros. Ahora la pregunta es, ¿cómo y de que forma generaremos los usuarios siguiendo un proceso de Poisson?

2.2.3.4. Generación de usuarios siguiendo un proceso Poisson. Con respecto a la generación de usuarios, la distribución que más se acerca a como los usuarios aparecen es la distribución exponencial que se aplica en procesos de Poisson. La distribución exponencial podemos considerarla como el modelo más adecuado para la probabilidad del tiempo de espera entre dos eventos que sigan un proceso de Poisson. Estos eventos que ocurren son las apariciones de usuario y para simularlo tendremos que resolver primero dos problemas:

1. Necesitamos computar de alguna manera los instantes de aparición de cada usuario siguiendo una distribución exponencial.
2. Situar las apariciones de forma aleatoria uniforme dado un punto geográfico y un radio

(dentro de una circunferencia).

Las soluciones a estos dos problemas se resuelven en los siguientes dos subapartados.

1. Calcular variables aleatorias de una distribución exponencial

El objetivo de este subapartado es definir un algoritmo que genere dado un valor λ , un tiempo total y un número máximo de usuarios, una secuencia de tiempos de aparición para los usuarios. Para ello primero debemos recordar como es la función de distribución de probabilidad exponencial:

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

Donde λ es la cantidad de usuarios que aparecen por cada unidad de tiempo.

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. Donald Knuth describe una forma de generar variables aleatorias siguiendo esta distribución utilizando la formula antes mencionada [6] [7]. Según explica Knuth, siempre que tengamos una distribución continua y su función de distribución de probabilidad 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 (2), como todos los valores de $f(x)$ están dentro del rango del cero al uno (3), 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 $u = U(0, 1)$ que se utilice como argumento de la inversa de la distribución de probabilidad exponencial(1).

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

Si hacemos esta inversa, 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.

Lo que estamos haciendo es a partir del valor de u (que como hemos mencionado anteriormente, es una variable aleatoria uniforme entre 0 y 1) aplicando la formula vista en (5) obtenemos X que sería el tiempo que falta para el siguiente suceso.

Por ejemplo, si el valor de u es 0.42 el valor que devolverá es de 2,64 segundos para un $\lambda = 1/5$

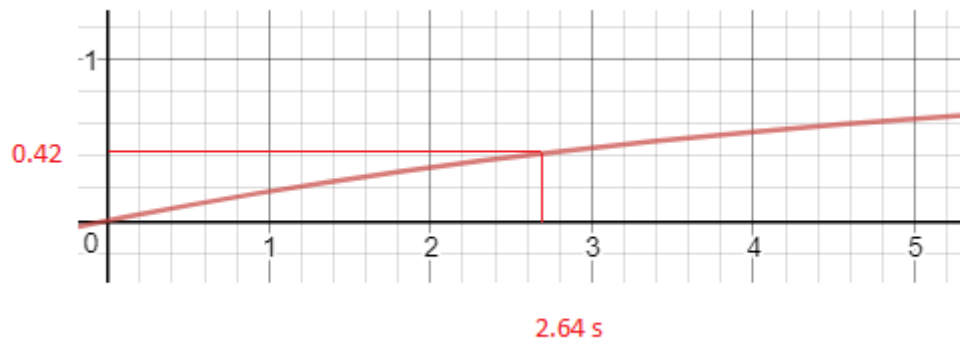


Figura 5: La función inversa (5) nos permite generar variables aleatorias que se correspondan a tiempos de aparición a partir de un valor aleatorio entre 0 y 1

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

Las variables de entrada de el 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

Algorithm 1: How to write algorithms

```

1 let L be a new List;
2  $ct \leftarrow 0$  // current time;
3  $uc \leftarrow 0$  // users counter;
4 while  $ct < e$  and  $uc < m$  do
5    $uc \leftarrow uc + 1$ ;
6   if  $r > 0$  then
7      $up \leftarrow \text{RandomPositionInCircle}(p, r)$  // User position;
8   else
9      $up \leftarrow p$ ;
10   $t \leftarrow -\ln(\text{Random}(0, 1))$  // Apparition time ;
11   $ct \leftarrow ct + t$ ;
12   $u \leftarrow \text{NewUser}(up, ut, ct)$ ;
13  L.add(u);
14 return L;
```

En la línea 10 aplicamos la fórmula deducida en (5). Por cada iteración del bucle 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.

En el siguiente apartado veremos como implementar la función *RandomPositionInCircle* de la línea 7

2. Generación de usuarios dentro de un punto geográfico y un radio

En primer lugar, debemos partir desde una base sencilla. Un primer acercamiento a la solución de este problema, sería la generación de puntos aleatorios en un círculo plano bidimensional.

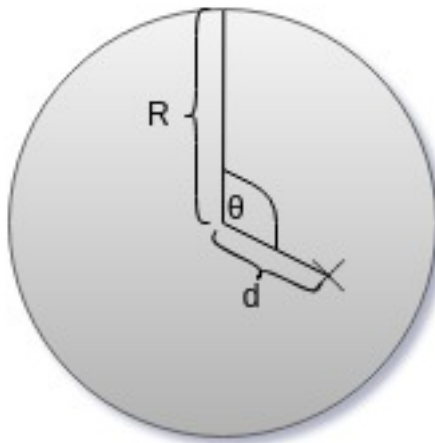


Figura 6: Generación de puntos en una circunferencia

En la figura 6, R es el radio del círculo, d es un valor aleatorio uniforme entre 0 y R , y θ es un ángulo cuyo valor es aleatorio uniforme entre 0 y 2π . Una posible solución sería generar los puntos en base a las siguientes formulas.

$$d = \text{random}(0, R); \quad \theta = \text{random}(0, 2\pi) \quad (1)$$

Sin embargo esto nos va a dar como resultado más cantidad de puntos en zonas cercanas al centro de la circunferencia. Esto se debe a que a medida que el radio aumenta, la superficie en la que se puede representar el punto es mucho mayor. Debemos tener en consideración este dato a la hora de generar d aleatoriamente.

Por lo tanto lo que haremos es generar d con la siguiente formula para el radio [8].

$$d = R * \sqrt{\text{random}(0, 1)} \quad (2)$$

Donde $\text{random}(0, 1)$ es un función que devuelve un valor uniforme entre 0 y 1, y R es el radio del círculo. En la figura 7 se puede ver como influye la generación de puntos utilizando la formula descrita anteriormente. El círculo en la derecha, corresponde a una generación correcta de puntos.

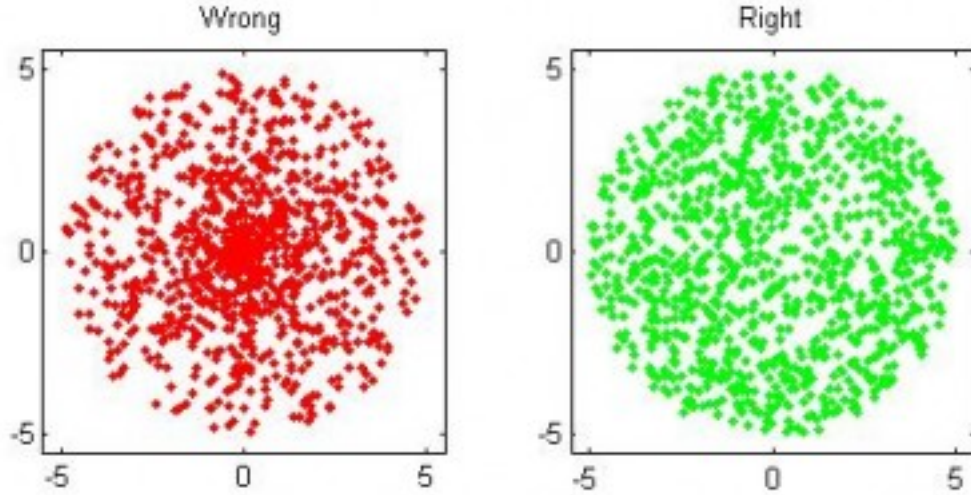


Figura 7: Comparativa generación puntos aleatorios

En segundo lugar tenemos que trasladar esta solución a la superficie de la tierra. Los puntos geográficos en la tierra, no se comportan como puntos cartesianos en un plano bidimensional. Supongamos los siguientes puntos geográficos, siendo x_1, x_2, x_3 y x_4 latitudes y y_1, y_2, y_3 y y_4 longitudes:

$$p1 = (x_1, y_1); \quad p2 = (x_2, y_2); \quad t1 = (x_3, y_3); \quad t2 = (x_4, y_4); \quad (3)$$

Supongamos ahora que los puntos $p1$ y $p2$ se encuentran cercanos al ecuador, y los puntos $t1$ y $t2$ se encuentran cercanos a los polos. Además supongamos también que las diferencias de las longitudes y latitudes de ambos puntos son las mismas, es decir:

$$x_2 - x_1 = x_4 - x_3; \quad y_2 - y_1 = y_4 - y_3 \quad (4)$$

Si consideramos los puntos $p1, p2, t1$ y $t2$ como puntos en un plano bidimensional, las distancia entre $p1$ y $p2$, y la distancia entre $t1$ y $t2$ sería la misma, pero como son coordenadas geográficas, al ser la tierra elipsoidal, son distancias diferentes.

Dado un punto inicial, un ángulo de dirección y una distancia, podemos calcular una nueva coordenada geográfica. Tenemos como punto inicial φ_1 (latitud) y λ_1 (longitud), un ángulo θ (en

sentido horario desde el norte) y una distancia d . Además necesitaremos también conocer la distancia angular, que sería $\delta = d/R$, donde R es el radio de la tierra.

Aplicando la siguiente fórmula[9] obtendríamos el punto (φ_2, λ_2) :

$$\varphi_2 = \text{asin}(\sin \varphi_1 * \cos \delta + \cos \varphi_1 * \sin \delta * \cos \theta) \quad (5)$$

$$\lambda_2 = \lambda_1 + \text{atan2}(\sin \theta * \sin \delta * \cos \varphi_1, \cos \delta - \sin \varphi_1 * \sin \varphi_2) \quad (6)$$

Si aplicamos la formula para generar de forma aleatoria uniforme el ángulo θ vista en (1) y la distancia d vista en (2), podemos calcular puntos aleatorios en cualquier círculo en la superficie terrestre. Es posible que esta solución carezca de sentido para muchos lectores por que el error puede parecer mínimo si se representan las coordenadas sobre un plano bidimensional, pero no es así. Esta generación de puntos la necesitamos para los Entry Point y estos pueden estar en cualquier ciudad del mundo. Si un usuario define un Entry Point en una ciudad de Suecia, por ejemplo, y no realizamos los cálculos sobre una superficie esférica, los usuarios en Suecia se generaran dentro de áreas que no serían circulares, sino elipses (en la figura 8 se puede ver la diferencia entre aplicar el calculo sobre 2 dimensiones y sobre la esfera). Esto se explica debido a que la distancia entre grados no es la misma según en la zona en la que estemos. Con esto estamos teniendo en cuenta ese factor, y los usuarios generados siempre se generaran en áreas circulares.

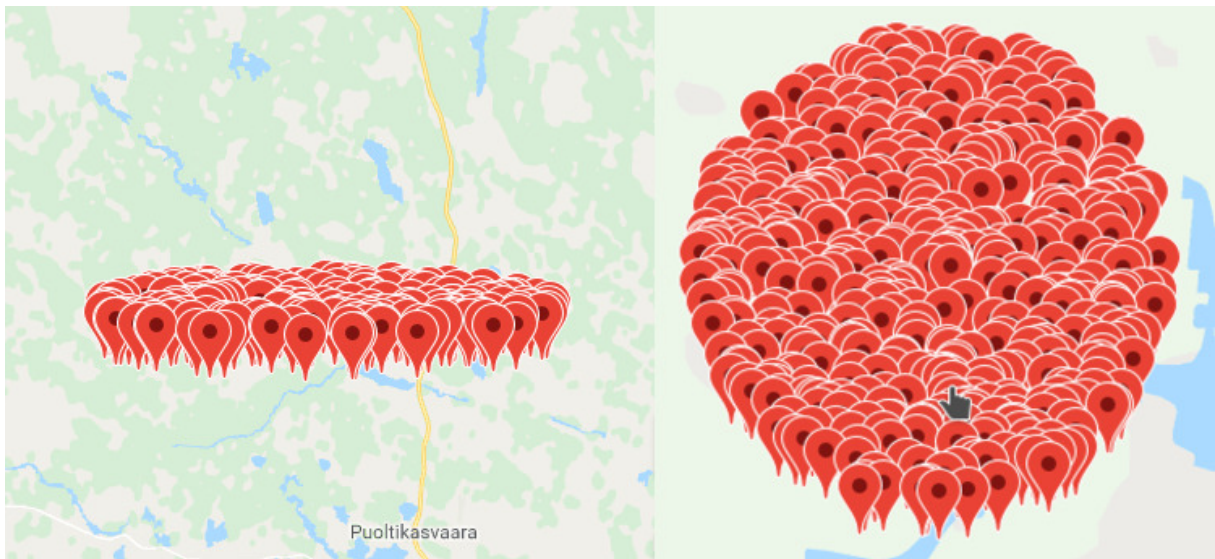


Figura 8: En la izquierda se pueden ver puntos aleatorios generados en base a un plano bidimensional y en la derecha puntos generados en base a la superficie de la tierra.

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 esta.

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:

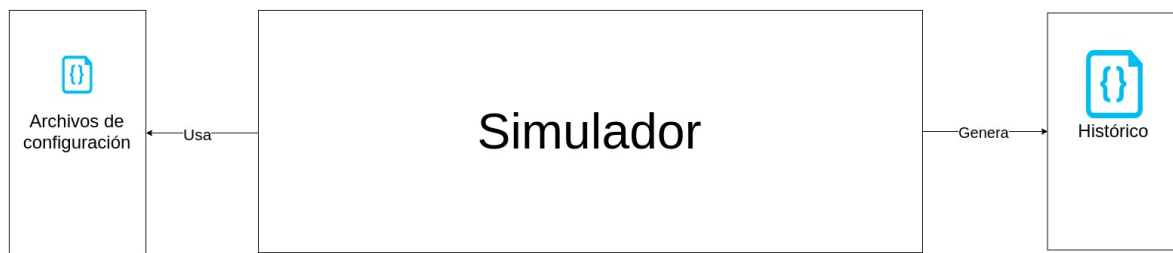


Figura 9: Vista básica del simulador

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

- Archivos de configuración, que como hemos comentado en el análisis serán:
 - Estaciones: Puntos geográficos de las estaciones, número de bicis, capacidad.
 - Entry Points: Puntos de entrada de los usuarios.
 - Parámetros globales: Tiempo total de la simulación, semilla. . .
- Simulador: De momento contendrá toda la lógica del simulador y sus interfaces de comunicación con el SDR (sistema de recomendaciones) y la infraestructura.
- 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.

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 este 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.

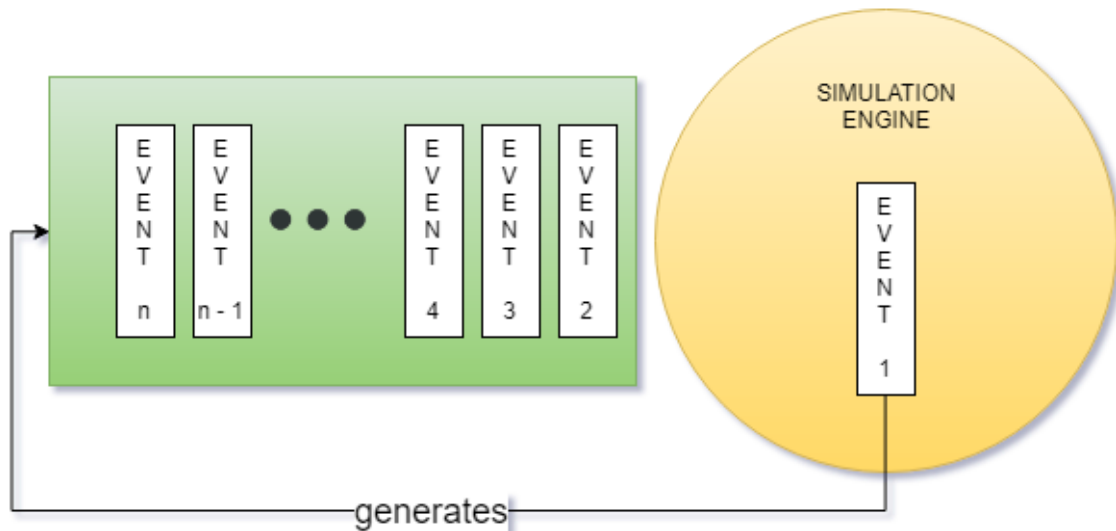


Figura 10: Simulador basado en eventos

En la figura 10 vemos una ejecución básica del motor de el 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 el 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
- Timeout⁴ 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[5].

⁴Momento en el que una reserva pierde su validez debido a un tiempo máximo alcanzado.

2.3.3. Arquitectura detallada

A partir de la figura 9, a continuación se desarrolla los detalles de la arquitectura del sistema incluyendo las decisiones que se han tomado que han llevado a un desarrollo evolutivo.

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 figura 11.

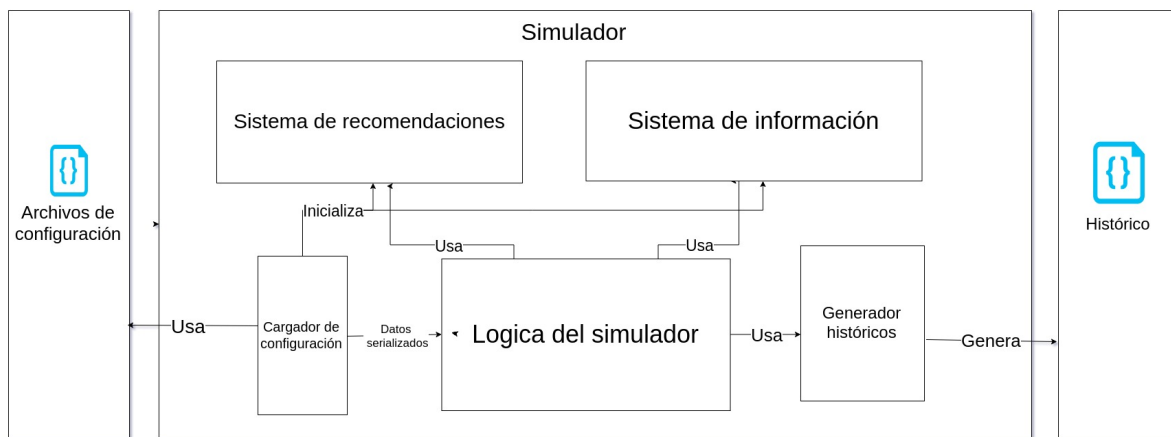


Figura 11: Vista básica del simulador

Esta figura muestra una parte importante de nuestra arquitectura y es el cargador de configuración. Este se encargará de convertir los datos expresados en los archivos a objetos serializados en el simulador para que puedan ser utilizados por este.

Se puede apreciar como la configuración, además de serializar los datos para la lógica del simulador, inicializa dos servicios que tendrán acceso a ciertos datos de la configuración y del estado de la infraestructura, que servirán como interfaz de comunicación para los usuarios simulados con el sistema de bicis. Cabe mencionar también el generador de históricos, que por cada Evento escribirá en un fichero de texto todos los cambios que se han producido por evento, dando lugar a un histórico final, con el cual, podremos visualizar la simulación y calcular datos sobre esta.

En la parte lógica del simulador estaría toda la parte relacionada con las interacciones de los usuarios y como estos actúan con la infraestructura y con el sistema de recomendaciones y de información. Esta lógica de simulador es la siguiente que vamos a desglosar en nuestra arquitectura.

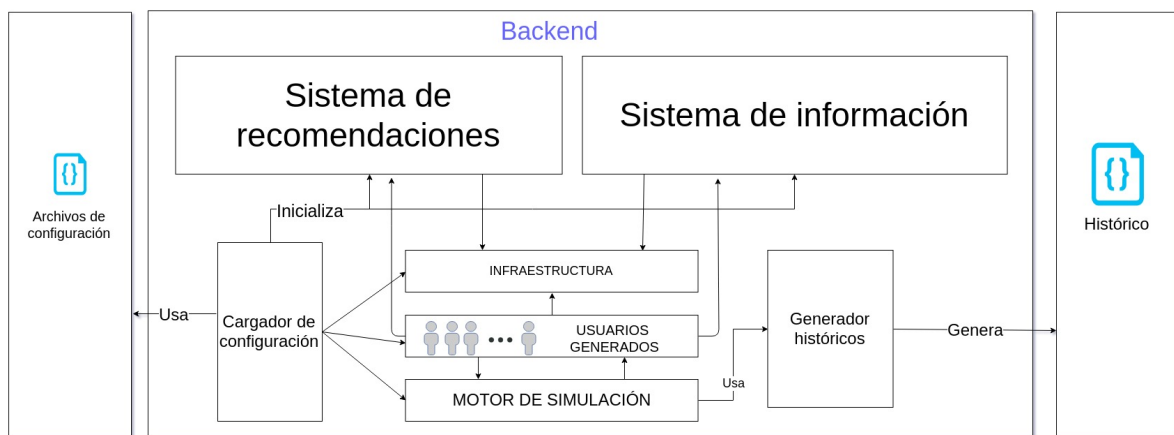


Figura 12: Simulador con arquitectura interna de la lógica desglosada

En la figura 12 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 el simulador basado en eventos. Este 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 12 iba a ser la arquitectura principal de el 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 eran los siguientes:

- Parámetros globales
- Estaciones
- Entry Points

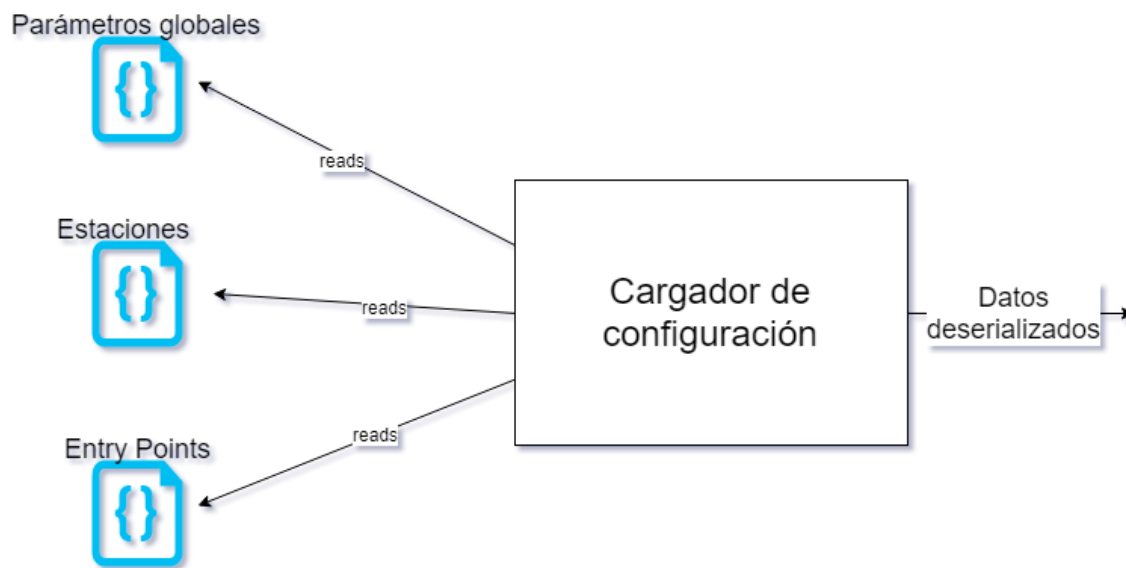


Figura 13: Primera idea para la configuración

Parece apropiado pensar que esos tres ficheros de configuración son suficientes, sin embargo, surgió la necesidad de independizar la generación de usuarios. Si solo utilizamos un archivo de configuración con los **entry points**, el 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. Es en este punto donde surge la necesidad de definir un nuevo módulo en nuestra arquitectura, un **generador de usuarios**. En la figura 14 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.

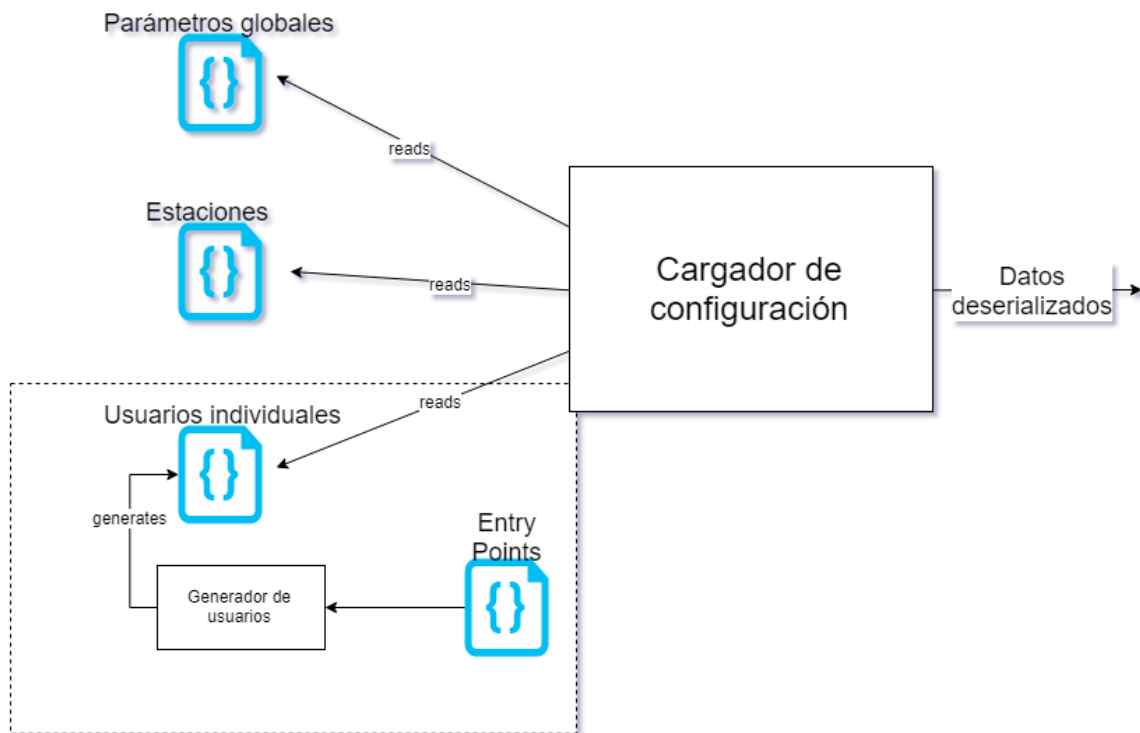


Figura 14: Cargador de configuración con generador de usuarios

Por otro lado, la generación de los históricos es también importante. El simulador 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[10].

En general este módulo se encargará de escuchar cada evento del simulador y escribir el resultado en uno o varios ficheros[10].

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 el simulador para:

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

En la figura 15 que se ve a continuación se añaden a la arquitectura los módulos encargados de esta tarea dentro de la interfaz de usuario.

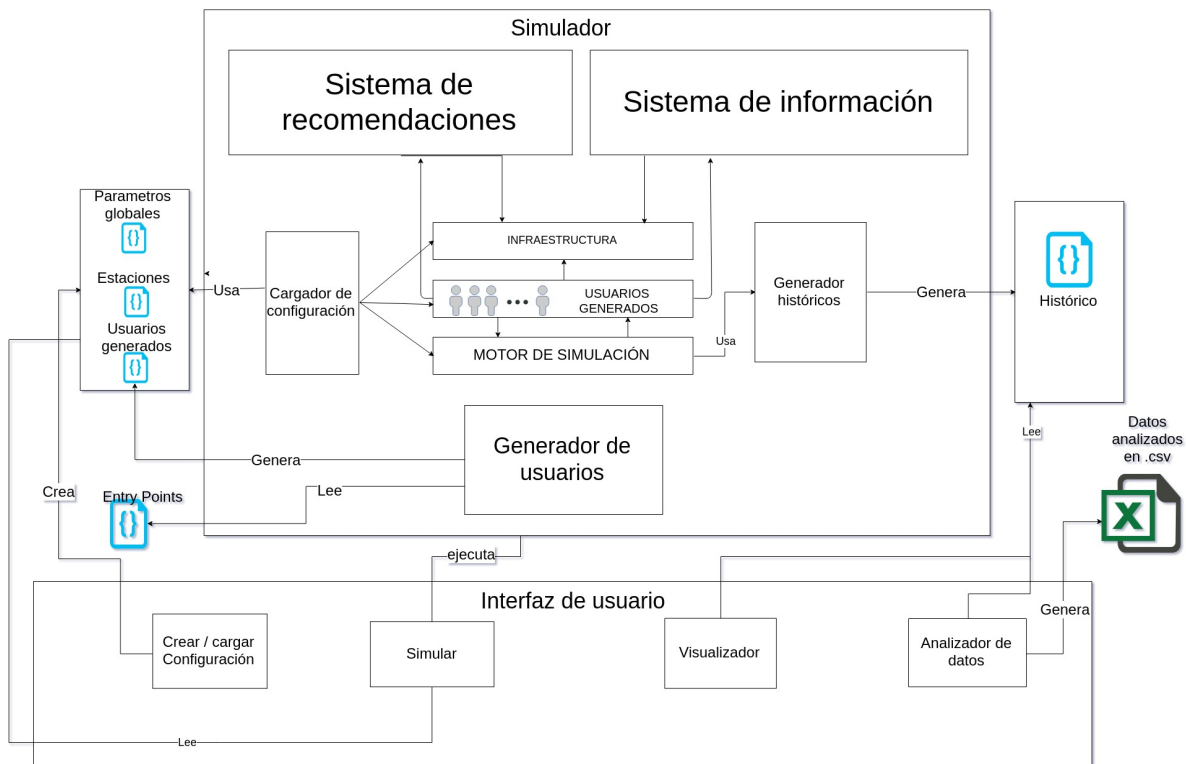


Figura 15: Arquitectura final con interfaz de usuario

2.4. Implementación

En esta sección, siguiendo las decisiones tomadas en el apartado de diseño, implementaremos todas las partes correspondientes a mi TFG, que se centran principalmente en la configuración, el uso de patrones de diseño y modularización que le de flexibilidad al código y la interfaz de usuario. Además utilizaremos un gestor de rutas para que los usuarios calculen en la simulación los caminos que deben tomar para coger y devolver las bicis. También explicaremos como hemos implementado un pequeño inyector de dependencias para que los usuarios puedan hacer uso de múltiples servicios fácilmente y como hemos implementado un sistema de logs para comprobar el comportamiento de los usuarios individualmente en el desarrollo de los mismos. Crearemos utilizando técnicas de reflexión factorías que faciliten la implementación de nuevos usuarios y puntos de entrada. Por otra parte también veremos como hemos implementado la interfaz gráfica para la configuración, la estructura de esta junto con la creación de formularios dinámicos para los usuarios.

2.4.1. Tecnologías

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 cumple las características deseadas, o la interfaz no se adapta a las necesidades del proyecto, esta se pueda reemplazar, adaptar, cambiar de tecnología, etc. . .

En la figura 16 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)

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 el 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 herramienta de visualizaciones que hay para visualizar datos, mapas, crear interfaces de usuario atractivas, pensamos que esta 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 incorporar a desarrolladores nuevos en el proyecto si tienen conocimiento 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 los 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.

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.

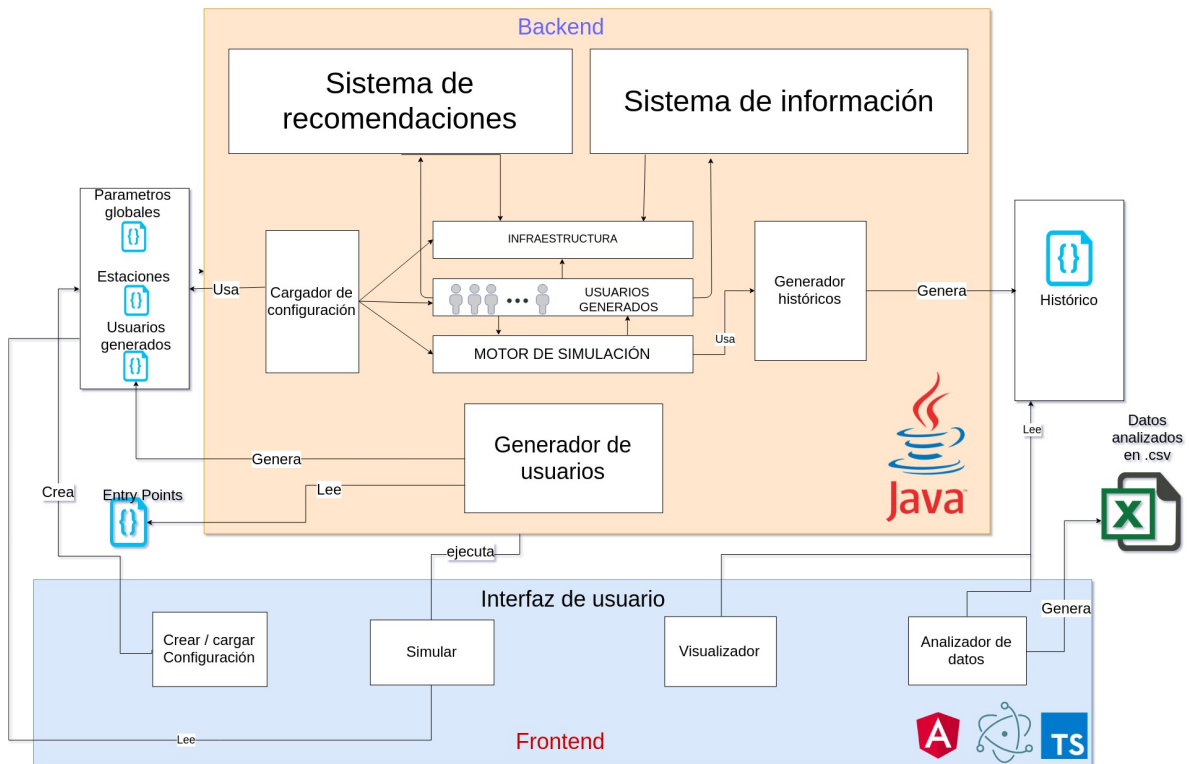


Figura 16: Arquitectura final con tecnologías

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.

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

En la figura 16 se pueden ver los diferentes módulos del backend.

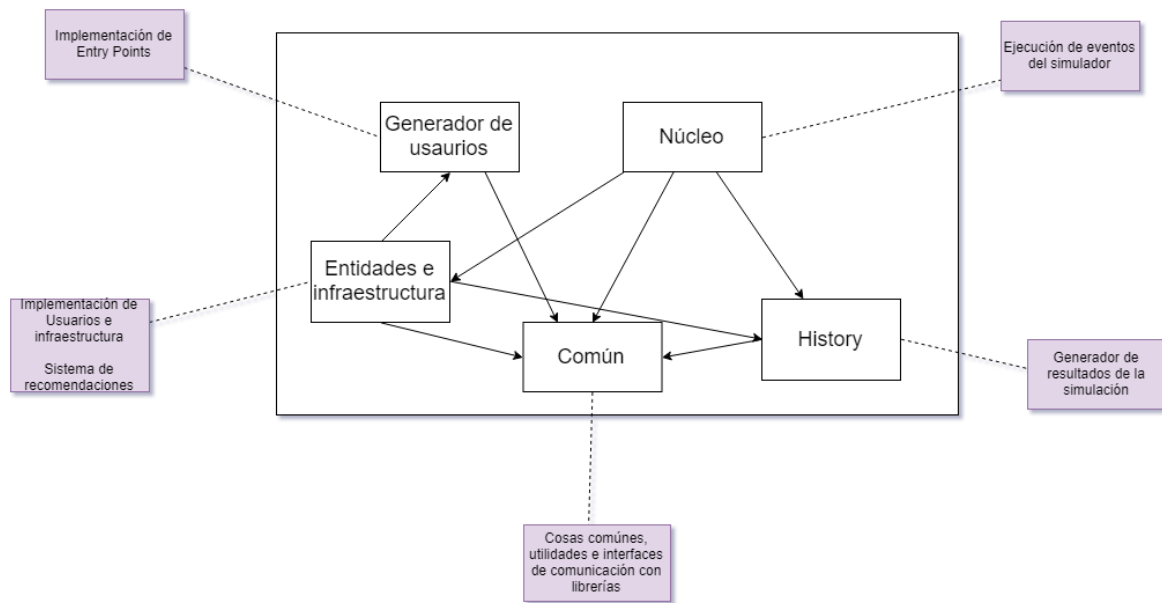


Figura 17: Módulos backend-simulator

A primera vista puede parecer muy diferente con lo definido en la arquitectura de la figura 16, pero eso se debe a que esta arquitectura solo define como se organizan y comunican las partes de el 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. En esta clase se implementa la creación de puntos aleatorios dentro de una circunferencia vista en el apartado 2.2.3.4.
 - **GraphManager, GraphHopperIntegration y GeoRoute:** Véase la sección 2.4.5 para su descripción
 - **DebugLogger:** Utilidad creada para depurar los usuarios implementados. Es muy útil para ver errores en las implementaciones de los usuarios. Una descripción en mayor detalle se puede ver en la sección 2.4.2.
- **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 la cola de eventos, y el cargador de los archivos de configuración. Los eventos siguen la jerarquía de clases de la figura 19. 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 e introducir los eventos en la cola (Véase figura 10). Una explicación más detallada del núcleo se encuentra en el trabajo final de Sandra Timón Mayo [5].

- **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 18.

En la clase `EntryPointFactory` se utiliza Gson⁵, que 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.

La clase abstracta `EntryPoint` define el método `generateUsers()`, la cual deben heredar todas las implementaciones de puntos de entrada en el sistema. Estos usuarios son de tipo `SingleUser` los cuales tienen como propiedad el instante de tiempo en el que aparecen y podrán ser insertados posteriormente en un evento de aparición en ese mismo instante de tiempo.

En el Anexo 3, se puede ver como implementar un Entry Point.

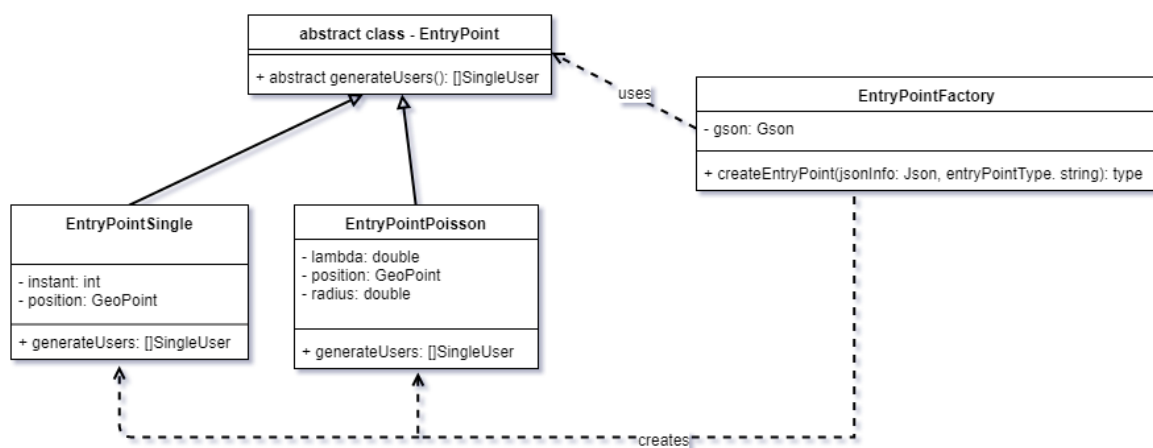


Figura 18: Entry Points Factory

⁵Gson - <https://github.com/google/gson>

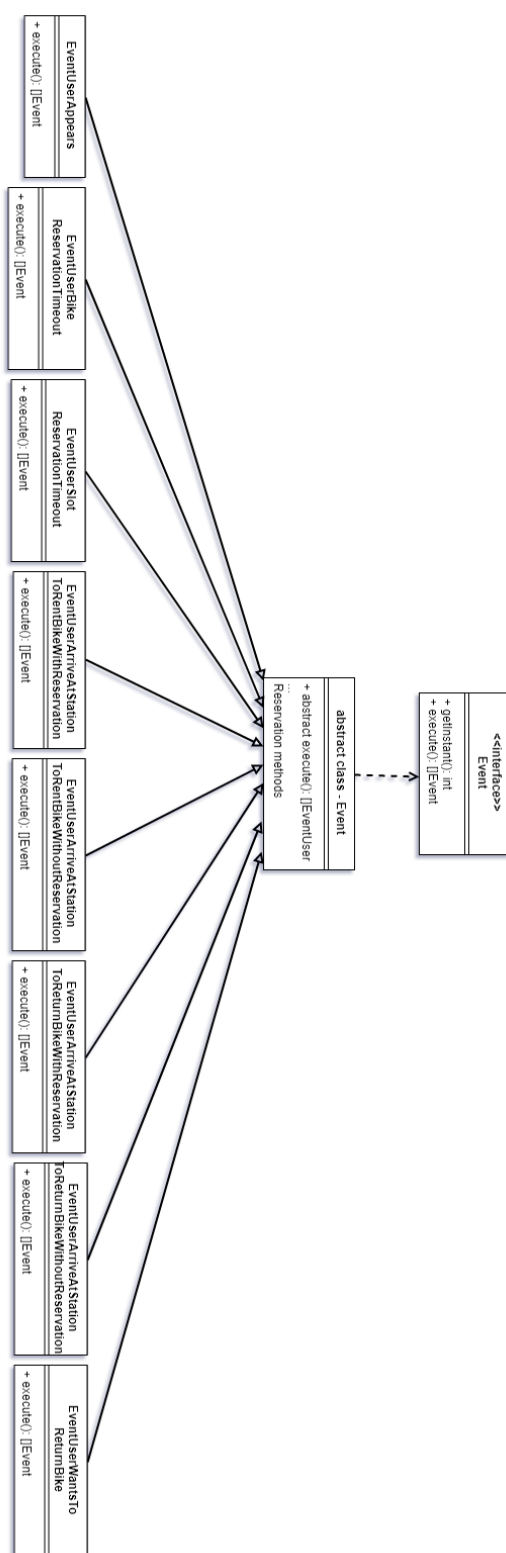


Figura 19: Jerarquía de clases de los eventos

- **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, 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 y la visualización en el trabajo final de Tao Cumplido[10].

- Entidades e infraestructura: En este módulo se definen todas las entidades de la simulación. Consideramos como entidad los objetos que cambian por cada evento que sucede. Entidades en el 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 20 se muestra el diagrama de clases de las entidades.

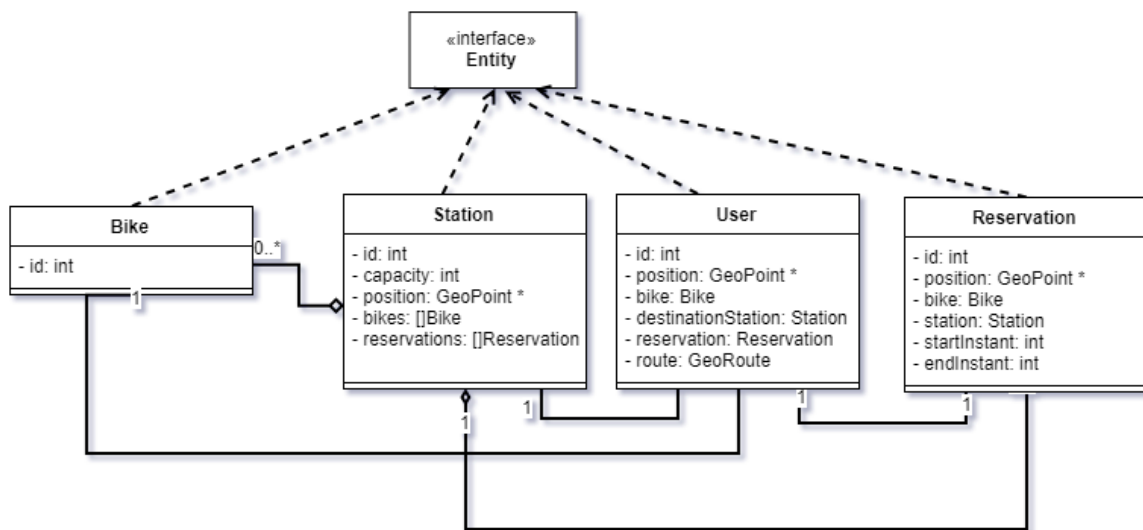


Figura 20: Diagrama UML de entidades e infraestructura

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 21. 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 influyan en sus decisiones. Todo está a disposición del implementador que desee programar un usuario en concreto.

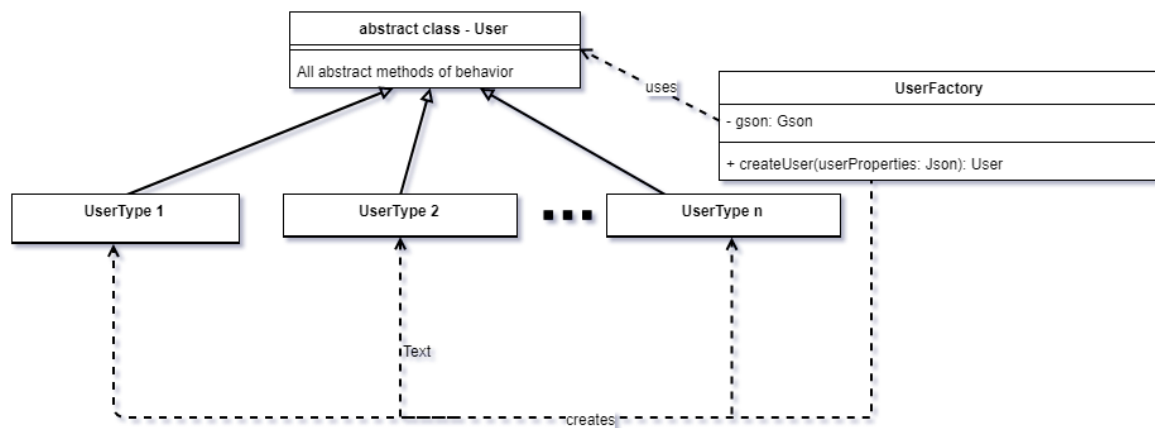


Figura 21: Factoría de usuarios

Los diferentes métodos que deben implementar los usuarios simulados son:

- `determineStationToRentBike(): Station` y `determineStationToReturnBike(): Station`

El usuario elige a que estación ir para alquilar o para devolver una bici. El implementador deberá hacer uso de los servicios a los que puede acceder el usuario para determinar a que estación ir. Este método devuelve una estación.

- `determineRoute(): GeoRoute`

En este método deberá hallarse una ruta hasta el destino del usuario. Cualquier gestor de rutas podrá ser usado, pero la ruta que devuelva debe ser un objeto de la clase `GeoRoute` para que pueda ser usada por el núcleo. Por defecto se puede usar el servicio `graph` que utilizará `GraphHopper` para calcular las rutas.

- `decidesNextPoint(): GeoPoint`

El usuario decide a que lugar de la ciudad quiere ir a dar una vuelta.

- `decidesToLeaveSystemAfterTimeout(): boolean`

El usuario decide si abandona el sistema tras una expiración de reserva.

- `decidesToLeaveSystemAfterFailedReservation(): boolean`

Tras un intento fallido de reserva de bici o de hueco, decide si abandonar el sistema o no. Es importante destacar que el usuario tiene acceso a un objeto llamado `memory` el cual contiene información acerca de las acciones que ha realizado el usuario simulado, para poder tomar decisiones e implementar algoritmos más avanzados que puedan tomar referencia de las acciones pasadas.

- `decidesToLeaveSystemWhenBikesUnavailable(): boolean`

El usuario decide si abandonar el sistema cuando no hay bicis disponibles en la estación.

- `decidesToReserveBikeAtSameStationAfterTimeout(): boolean`

El usuario decide si quiere volver a intentar reservar en la misma estación donde había hecho la reserva que acaba de expirar.

- `decidesToReserveBikeAtNewDecidedStation(): boolean`

El usuario decide si reservar en la estación que ha escogido como destino.

- `decidesToReserveSlotAtSameStationAfterTimeout(): boolean`

Tras haber reservado un hueco en una estación y haber expirado la reserva, el usuario decide si volver a reservar en esa estación.

- `decidesToReturnBike(): boolean`

El usuario decide si devolver la bicicleta. Si no la devuelve, se ejecutará el método `decidesNextPoint()`, el cual devolverá un punto al que el usuario irá a dar una vuelta con la bicicleta.

- `decidesToDetermineOtherStationAfterTimeout(): boolean`

Tras la expiración de una reserva, el usuario decide si elegir una nueva estación.

- `decidesToDetermineOtherStationAfterFailedReservation(): boolean`

Tras un intento de reserva fallido, el usuario decide si elegir una nueva estación.

Además en las implementaciones de estos métodos se podrá acceder a los siguientes atributos, que contienen información del usuario:

- `id: integer` - Identificador del usuario.
- `position: GeoPoint` - Posición actual del usuario.
- `walkingVelocity: double` - Velocidad caminando.
- `cyclingVelocity: double` - Velocidad en bici.
- `destinationStation: Station` - Estación a la que el usuario quiere ir para coger o devolver una bici.
- `destinationPoint: GeoPoint` - Punto al que el usuario quiere ir en bici para dar una vuelta.
- `route : GeoRoute` - Ruta actual del usuario.

- **bike:** **Bike** - Bici que tiene el usuario.
- **reservedBike:** **boolean** - **true** si tiene bici reservada, **false** en caso contrario.
- **reservedSlot:** **boolean** - **true** si tiene un hueco reservado, **false** en caso contrario.
- **reservation:** **Reservation** - Reserva actual del usuario.
- **memory:** **UserMemory** - Contiene atributos que registran los hechos sucedidos hasta el momento actual de la simulación

Si los usuarios simulados quieren consultar algún tipo de información o quieren consultar alguna de las recomendaciones del sistema, lo hará a través de ciertos servicios que implementarán una interfaz para realizar estas consultas. Estos servicios serán ofrecidos por la clase abstracta **User** a modo de atributos, por lo tanto cualquier clase que herede de **User**, tendrá acceso a estos servicios, que podrán ser usados en todos los métodos. Como se inicializan estos servicios se puede ver en la sección 2.4.2. Los servicios de los que disponen los usuarios son:

- **infraestructure:** **InfraestructureManager**: Información acerca del estado de las estaciones, su ubicación, etc.
- **recomendationSystem:** **RecomendationSystem**: Sistema de recomendaciones al que el usuario puede consultar.
- **graph:** **GraphManager**: Gestor de rutas por defecto. Hemos utilizado **GraphHopper** como veremos en el apartado 2.4.2

De momento se ha implementado un sistema de recomendaciones muy básico. La idea es crear implementaciones con funcionalidades más complejas para evaluar diferentes modelos de equilibrio del uso de las bicicletas. Estas nuevas implementaciones se podrían añadir como nuevos servicios a los que el resto de usuarios podrán acceder, pudiéndose usar uno o más sistemas de recomendaciones.

Por otro lado el simulador debe ofrecer la posibilidad de implementar usuarios fácilmente, pero además es importante poder depurarlos. Al haber una gran cantidad de usuarios decidí implementar un “logger” que en cada ejecución generase por cada usuario un fichero con las trazas de su ejecución. Para ello simplemente hemos creado una clase llamada **DebugLogger**, la cual ofrece los métodos necesarios para poder escribir automáticamente las acciones del usuario, escribiendo una simple instrucción cuando queramos crear una traza en un archivo log. Podemos escribir un log de dos formas diferentes:

- **debugLog()**: Escribe la información del evento, junto con información del Evento y parámetros del usuario.
- **debugLog("Message to log")**: Igual que **debugLog** solo que puede añadirse información más precisa con el mensaje introducido como argumento.

Cada usuario tendrá su propio fichero de log. Por ejemplo, el usuario con $id = 5$ tendrá

un fichero log llamado `User5.txt`. La implementación de esta clase `Logger` es una simple clase estática que es llamada en los eventos de la simulación.

2.4.3. Inicialización y ejecución del simulador

El simulador tiene dos modos de ejecución principales, uno para la generación de usuarios y otro para ejecutar una simulación. El simulador como tal, no necesita los entry points para empezar a simular, como comentamos en la sección 2.2.3.3, sino que se le pasa un archivo de configuración con todos los usuarios generados. Para crear este fichero de configuración ofrecemos un generador de usuarios que recibe el archivo de configuración de los entry points. Una vez generados los usuarios, se podrá ejecutar el simulador. Las fases por las que pasa el simulador para inicializarse son las siguientes:

1. Lectura de ficheros y deserialización.
2. Inicialización de las factorías e inyector de dependencias. En esta parte mediante reflexión se descubren y guardan las clases correspondientes a las implementaciones de usuario, gestores de rutas y sistemas de recomendación. Esto se explica con mayor detalle en la sección 2.4.4
3. Inicialización de Servicios. Estos servicios serán posteriormente utilizados por los usuarios.
4. Creación de eventos de aparición. Se introducen en la cola del simulador todos los eventos de simulación, incluyendo en estos los usuarios ya inicializados y preparados para ser simulados.

Con respecto al punto 3 utilizamos un patrón de inyección de dependencias que consiste en permitir que al crear un objeto, no sea necesario que ese mismo objeto se encargue de crear o inicializar los objetos de los que hace uso o que posee como atributos. En nuestro caso, por ejemplo, los usuarios pueden hacer uso de un gestor de rutas, de un recomendador, o pueden ver la información actual del sistema de bicis. Cada uno de estos servicios es inicializado al comienzo de la simulación y se pasa como parámetro al constructor de todos los tipos de `User`. La clase principal `User` tiene acceso entonces a este servicio, y los usuarios implementados que heredan de esta clase, pueden acceder a todos los servicios en sus métodos.

Una vez todo está inicializado, el motor de simulación (implementado en `SimulationEngine`, dentro del módulo `Core`), se encargará de ejecutar todos los eventos que se han introducido en la cola del simulador, que por evento irá generando los históricos hasta que la cola quede vacía.

2.4.4. Aplicando reflexión para una mejor extensibilidad

En el simulador se va a disponer de muchos tipos de usuarios a implementar, y de muchas formas de entrada de los usuarios al sistema (entry points), además de múltiples servicios que los usuarios podrán usar durante las simulaciones. Es por esto que necesitamos de un método que nos permita de la manera más eficiente posible implementar nuevas características, y para ello hemos utilizado la Java Reflection API. En la configuración de los entry points y los usuarios debemos indicar que tipo de usuario queremos generar. Cada tipo de usuario lo vamos a identificar con una cadena de texto. Así si quiero crear un usuario de la clase `UserRandom`, en la configuración ponemos el tipo “`USER_RANDOM`”. En el Anexo 1, en el ejemplo de configuración de usuarios y entry points, podemos ver como se especifica el tipo de usuario en la configuración. Como hemos

dicho con anterioridad, para crear los usuarios utilizamos una factoría. Para crear un usuario debemos escribir la siguiente instrucción en nuestro código Java:

```
1      UserFactory userFactory = new UserFactory();
2      User user = userFactory.createUser(userType, services);
```

En este código, `userType` es el tipo de usuario y `services` son todos los servicios inicializados del sistema. Ahora bien, ¿cómo sabe el simulador que clase se corresponde con el tipo de usuario `userType`? Especificando mediante anotaciones que clases son implementaciones de usuario y como se identifica en la configuración. Por ejemplo si quisiéramos crear un usuario, deberíamos hacerlo de la siguiente forma:

```
1      @UserType("USER_RANDOM")
2      public class UserRandom extends User {
3
4          @UserParameters
5          public class Parameters {
6              private int parameter1;
7              private double parameter2;
8              ...
9          }
10
11      ...
```

En la notación `@UserType` ponemos con que cadena de texto se identifica a este usuario y en `@UserParameters` definimos que parámetros concretos tiene este. Imaginemos que queremos instanciar un usuario de tipo `USER_RANDOM`. La factoría al inicializarse lo que hará es guardar en una lista todas las clases que tengan la interfaz `@UserType`, después se llamará al método `userFactory.createUser('USER_RANDOM', services)` que utilizando la API de reflexión de Java conseguirá identificar en la lista de clases que implementan `@UserType` cual se corresponde con el tipo `USER_RANDOM`. Posteriormente conseguimos el constructor de la clase que implementa `USER_RANDOM`, y instanciamos el usuario. La implementación de la factoría y la utilización de la API de reflexión se encuentra en el módulo `world-entities` en la clase `UserFactory`.

Con los entry points sucede exactamente lo mismo. Necesitamos diferenciar diferentes tipos de entry point en los archivos de configuración y además necesitamos añadirlos e identificarlos mediante anotaciones. La diferencia radica en que en vez de usar la anotación `@UserType`, utilizaremos la anotación `@EntryPointType` que ira junto con el tipo de entry point. De esta forma si queremos por ejemplo crear el tipo de entry point que siga un proceso de poisson tendríamos que hacerlo de esta forma:

```
1      @EntryPointType("POISSON")
2      public class EntryPointPoisson extends EntryPoint {
3          ...
4          @Override
5          public List<SingleUser> generateUsers() {
6              ...
7          }
```


8 }

Al igual que los usuarios, hemos implementado una factoría de entry points que sigue la misma lógica que la factoría de usuarios. La implementación concreta de este entry point se encuentra en el módulo `usersgenerator` en la clase `EntryPointPoisson` y la implementación de la factoría en `EntryPointFactory`.

Pero no solo en las factorías va a ser una muy buena herramienta la reflexión computacional. Como hemos mencionado con anterioridad en la parte de Diseño 2.3, los usuarios harán uso de servicios para consultar información y recomendaciones. Para no hacer dependientes las implementaciones de los usuarios hemos utilizado una inyección de dependencias, dejando responsable de la creación e inicialización de estos servicios a la clase `SimulationServices`. La clase `SimulationServices` se hará cargo de inicializar el gestor de rutas y el sistema de recomendaciones. Se pueden crear varios tipos de gestores de rutas y sistemas de recomendaciones por lo que hemos decidido emplear la misma lógica de reflexión utilizada para los entry points y los usuarios en estos servicios. En primer lugar se especifica en el archivo de configuración global que gestor de rutas y que sistema de recomendaciones queremos utilizar:

```
...
"recommendationSystemType": "AVAILABLE_RESOURCES_RATIO",
"graphManagerType": "GRAPH_HOPPER"
...
```

Las implementaciones de los gestores de rutas deberán tener la anotación `@GraphManagerType()` con el tipo correspondiente y `@GraphManagerParameters` para los parámetros necesarios de inicialización. Por otro lado las implementaciones de sistemas de recomendaciones deberán tener la anotación `@RecommendationSystemType()` con el tipo y `@RecommendationSystemParameters` para los parámetros. La clase `SimulationService` se encargará de, según la configuración, inicializar e inyectar dichas dependencias en los usuarios.

2.4.5. Carga de mapas y cálculo de rutas

Una cosa muy importante a implementar es la posibilidad a de crear simulaciones en ciudades reales. Implementar toda esta lógica y crear una estructura de datos eficiente para la carga y cálculo de las rutas nos podría llevar incluso más tiempo que el propio simulador. Es por eso que decidimos utilizar una librería externa, `GraphHopper`⁶. `GraphHopper` utiliza `Open Street Maps`, lo cual nos da cierta libertad al no depender de `Google Maps` que es una tecnología cerrada.

Para hacer que el simulador no dependa directamente de dicha librería, decidimos crear una interfaz (`GraphManager`) y un formato de rutas propio (`GeoRoute`), aunque el implementador puede crear cualquier gestor de rutas, simplemente deberá hacer que al calcularse las rutas en los métodos correspondientes del usuario, devuelva una ruta que sea una instancia de la clase `GeoRoute`. De esta forma si necesitamos nosotros crear nuestra propia utilidad o utilizar otra librería no sería difícil implementarla en el sistema.

A continuación explicaremos cada una de las clases referentes al gestor de rutas:

⁶<https://github.com/graphhopper/graphhopper>

- **GeoRoute**: Es la clase que representa una ruta en el simulador. Cualquier gestor de rutas implementado en el sistema debe devolver un objeto de esta clase para poder ser utilizado por el núcleo del simulador. Esta clase está compuesta de los siguientes atributos junto con los correspondientes métodos para obtenerlos:
 - **points**: `GeoPoint[]` - Contiene una lista de `GeoPoint` con los puntos de la ruta.
 - **totalDistance**: `double` - Distancia total de la ruta.
 - **intermediateDistance**: `Double[]` - Contiene una lista de las distancias entre cada punto dentro de la ruta.
- **GeoPoint**: Representa un punto geográfico. Dispone de los siguientes métodos:
 - **distanceTo(point: GeoPoint)** Distancia el punto geográfico del objeto y un punto dado, utilizando la formula de haversine⁷
 - **bearing(GeoPoint point): double** - Devuelve el ángulo formado por dos puntos dado el eje del norte.
 - **reachedPoint(distance: double, destination: GeoPoint): GeoPoint** - Devuelve el punto alcanzado tras partir del punto del objeto y dirigirse al punto **destination** tras haber recorrido una distancia.
- **GraphManager**: Es una interfaz creada con la intención de crear un estándar para que el usuario simulado en su implementación utilice siempre los mismos métodos, pero no es estrictamente obligatorio utilizarla. Los métodos de esta interfaz son:
 - **obtainAllRoutesBetween(originPoint: GeoPoint, destination: GeoPoint): GeoRoute[]** - Las clases que implementen este método deberán devolver una lista con todos los caminos posibles desde un punto de origen y un punto de destino.
 - **obtainShortestRouteBetween(originPoint: GeoPoint, destination: GeoPoint): GeoRoute** - Deberá devolver la ruta mas corta desde el punto origen al punto de destino.
 - **hasAlternativesRoutes(startPosition: GeoPoint, endPosition: GeoPoint): boolean** - Deberá devolver si desde un punto origen a un punto destino hay más de una ruta.
- **GraphHopperImplementation**: Esta clase implementa la interfaz **GraphManager**. **GraphHopper** es un motor de rutas rápido y eficiente en memoria. Aunque existían otras alternativas como **OSMR**, **GraphHopper**⁸ está implementado en Java por lo que nos facilita bastante su integración en el sistema, al contrario que **OSMR** que está implementado en C++.

GraphHopper es una tecnología que se puede utilizar de dos formas. A modo de servidor o integrando el motor de rutas en el código fuente utilizándolo a modo de librería. Decidí optar por integrar el motor en el simulador, para que las simulaciones se ejecutaran con mayor rapidez y consumiera menos recursos el simulador. Hay que tener en cuenta que un servidor de rutas ejecutado en paralelo con el simulador, podría consumir una gran cantidad de memoria.

Para calcular las rutas es necesario descargar el mapa de Open Street Maps y pasarlo como

⁷<http://www.movable-type.co.uk/scripts/gis-faq-5.1.html>

⁸<https://github.com/graphhopper/graphhopper>

argumento al instanciar el objeto que se encarga de utilizar GraphHopper. Posteriormente la librería crea una serie de ficheros con información que utilizará para calcular las rutas. El código del constructor del servicio de rutas es el siguiente:

```
1 public GraphHopperIntegration(GraphProperties properties) throws IOException {
2     //Check the last map loaded
3     boolean sameMap;
4     try {
5         CheckSum csum = new CheckSum();
6         sameMap = csum.md5Checksum(new File(properties.mapDir));
7     }
8     catch (Exception e) {
9         sameMap = false;
10    }
11
12    //If it is not the same map, we remove the latest temporary files
13    if(!sameMap) {
14        FileUtils.deleteDirectory(new File(GRAPHHOPPER_DIR));
15    }
16    this.hopper = new GraphHopperOSM().forServer();
17    hopper.setDataReaderFile(properties.mapDir);
18    hopper.setGraphHopperLocation(GRAPHHOPPER_DIR);
19    hopper.setEncodingManager(new EncodingManager("foot, bike"));
20    hopper.importOrLoad();
21 }
```

GraphHopper internamente genera unos ficheros a partir del mapa OSM, el cual utiliza para calcular las rutas, tardando un tiempo considerable en realizar esta tarea. Es por eso que, en el constructor de `GraphHopperIntegration`, comprobamos si el mapa que se está cargando ha sido cargado con anterioridad, evitando así la generación de nuevo de estos ficheros (líneas 3-15). En la línea 16 creamos el objeto que controla el gestor de rutas. Posteriormente le pasamos al objeto la información necesaria para cargar el mapa, como la ruta del mapa osm (línea 17), el directorio en el que generar los ficheros necesarios para el calculo de rutas (línea 18), y que tipos de ruta queremos sacar (línea 19). En este caso queremos sacar rutas a pie y en bici. Finalmente en la línea 19 cargamos el mapa. Los usuarios pues, hacen uso de esta implementación para poder calcular las rutas.

2.4.6. Interfaz de usuario del simulador y formularios dinámicos(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 cómo están implementados los usuarios, es tener un visualizador con el que observar toda la simulación. En esta sección explicaremos las diferentes partes de la parte gráfica de el simulador que se encargará de ofrecer una GUI capaz de realizar todo lo antes mencionado.

La interfaz de usuario está desacoplada completamente del simulador, por lo que otros desarrolladores podrían crear otras GUIs. Ésta es una de las ventajas que ofrece una arquitectura del tipo Cliente/Servidor. Nosotros hemos decido crear una interfaz de usuario utilizando tecnologías Web para un rápido desarrollo, pero se puede crear esta interfaz con otras tecnologías, si esto fuera necesario. Como se mencionaba en la sección 2.4.1, vamos a utilizar para la interfaz de usuario Electron, que al ejecutarse crea dos procesos, un proceso que llamaremos *Main* y otro

proceso que denominaremos *Renderer*:

- *Main*: Este proceso puede comunicarse con el SO y hacer operaciones de entrada salida. Está implementado 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
 - **Configuration**: Todos los archivos de configuración son validados a través de unos ficheros que determinan la estructura que debe tener la configuración. En este módulo se encuentra el parseador que convierte los esquemas de los datos en esquemas para generar formularios dinámicos. Esto es explicado en más detalle en el apartado 2.4.7
 - **Controllers**: Contiene todas las clases que definen la interfaz de comunicación entre el *Main* y el *Renderer*. Sigue una lógica tipo API REST en el que si el proceso *Renderer* pide algún dato o recurso, el *Main* recibirá este dato a modo de servidor y devolverá una respuesta a *Renderer*.
 - **DataAnalysis**: Contiene toda la lógica para analizar los históricos[5].
 - **Entities**: Clases utilizadas por **DataAnalysis**.
 - **Util**: Utilidades necesarias por otros módulos, que pueden ser de utilidad general.

Al ejecutar el frontend, éste debe ejecutar los módulos en Java del backend tanto para crear usuarios como para simular. Estos dos módulos son dos ejecutables java con extensión “*.jar*” disponibles en la ruta de la aplicación cuando es compilada. Simplemente hay que crear un proceso hijo que los ejecute. El proceso *Main* tendrá que poder ejecutar estos dos procesos, que se corresponden con los archivos *jar* compilados del backend:

- `java -jar bikesurbanfleets-config-usersgenerator-1.0.jar <parameters>`
- `java -jar bikesurbanfleets-core-1.0.jar <parameters>`

En estos dos comandos, *<parameters>* son las rutas a los archivos de configuración para comenzar el proceso de generar o de simular respectivamente.

- *Renderer*: Este proceso contiene toda la parte visual. Está programado en TypeScript y Angular. Está dividido en varios pequeños programas que forman parte de la misma interfaz de usuario. Angular está basado en componentes. Los componentes son los bloques de código más básicos de una interfaz de usuario en una aplicación Angular. Una aplicación Angular es un árbol de componentes y cada componente puede estar formado de otros componentes. Estos se podrán comunicar con el proceso *Main* si necesitan de algún recurso o tienen que hacer llamadas al simulador. Esta aproximación basada en componentes se utiliza para una mayor reutilización de código, ya que un mismo componente puede ser reutilizado en otros. Los distintos módulos son:
 - **App**: Contiene todos los componentes de la aplicación. Los principales son:
 - **configuration-component**: Componente encargado de la creación de configuraciones para el simulador.

- **simulate-component**: Permite cargar los ficheros de configuración para poder simular
- **visualization-component**: Permite visualizar como los usuarios interactúan en un mapa real tras realizar una simulación, cargando el histórico de ésta. Permite depurar los usuarios y además facilita la comprensión de los datos históricos arrojados por el simulador.
- **analyse-history-component**: A partir del histórico genera archivos csv con cálculos sobre la simulación.
- **Ajax**: Es el módulo encargado de comunicarse con el proceso *Main*, posteriormente hará las peticiones al simulador. El *renderer* hace las peticiones a modo de cliente, *Main* las recibe con los controladores definidos en el módulo **Controllers** de *Main*, y dependiendo de la petición, el proceso *Main* hará peticiones al simulador, peticiones de entrada/salida al SO, o otros tipo de peticiones como calculo de datos, generación de csv, etc.

En la figura 22 se muestra la arquitectura a grandes rasgos del frontend. Se puede observar que para iniciar una simulación la comunicación es de 3 niveles:

1. Desde el proceso *Renderer*, se renderiza la interfaz gráfica, con la cual el usuario que va a simular, introduce la ubicación de los ficheros de configuración y este pasa dicha información al proceso *Main*.
2. Los datos con las rutas de los ficheros de configuración son recibidos por el proceso *Main*, el cual ejecuta el simulador en java con los parámetros necesarios.
3. El simulador se ejecuta y envía por la salida estándar información que es leída por el proceso *Main* que a su vez envía esta información con formato al proceso *Renderer* que muestra el progreso de la simulación por pantalla, hasta que termina la simulación.

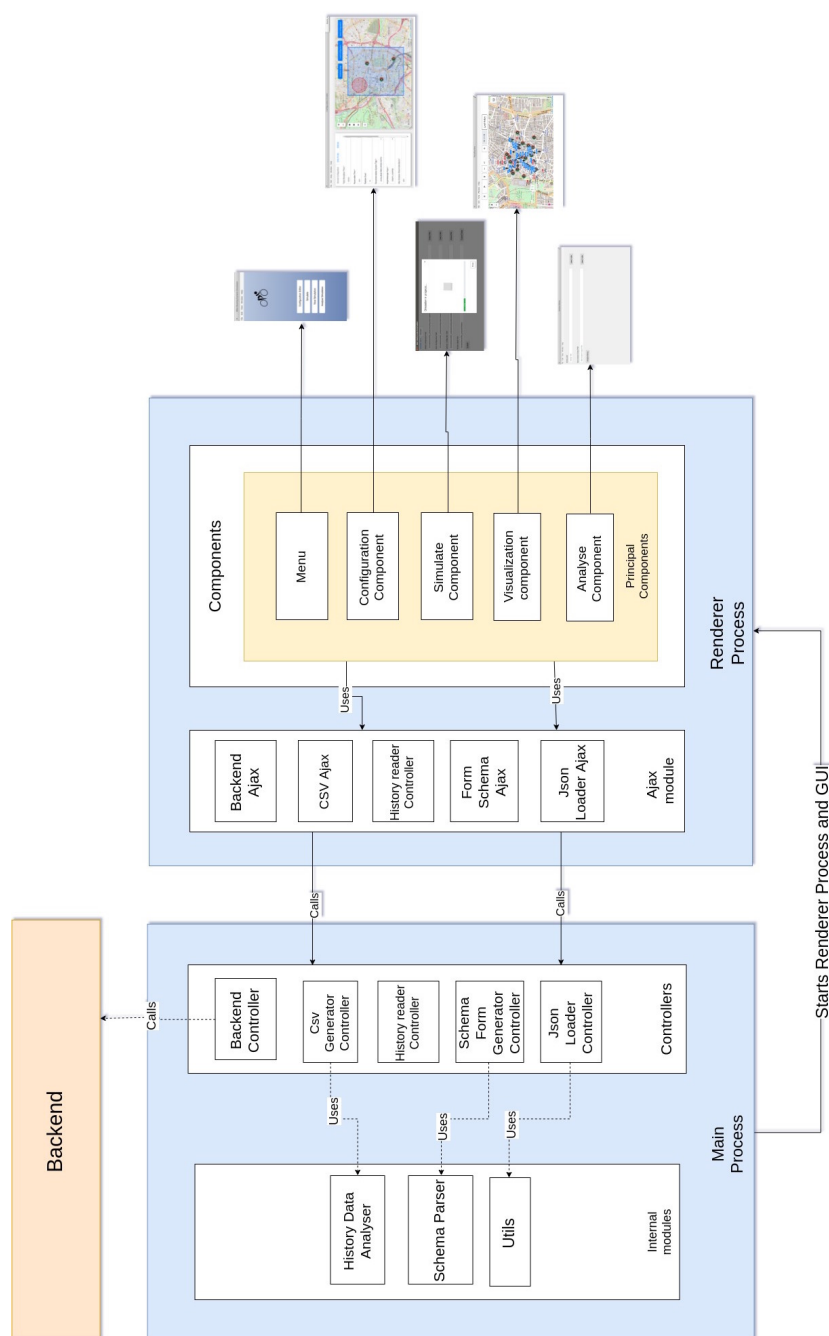


Figura 22: Arquitectura Frontend

2.4.7. Formularios dinámicos en las configuraciones

Al tener que introducir datos en la configuración y ser estos variables 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.

Como ya hemos mencionado, el simulador dispone de dos códigos base principales, el del Backend(Java) y el del Frontend(Typescript, Angular y Electron). Por cada usuario que creamos, tenemos que modificar ambos códigos. Por lo que se dificultaría en gran medida la implementación de nuevos usuarios, ya que habría que modificar el código del Frontend. Con un generador de formularios dinámicos podríamos modificar sólo el código del Backend y los esquemas para crear nuevos usuarios y mantener la interfaz de usuario actualizada con los nuevos usuarios que se pueden crear, facilitando la tarea de añadir o quitar nuevos parámetros en las configuraciones. Por lo tanto se obtienen bastantes ventajas de esta funcionalidad:

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

En la interfaz de usuario actual, se pueden añadir o quitar sin problemas parámetros a la configuración global de la simulación, y los parámetros de los usuarios.

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

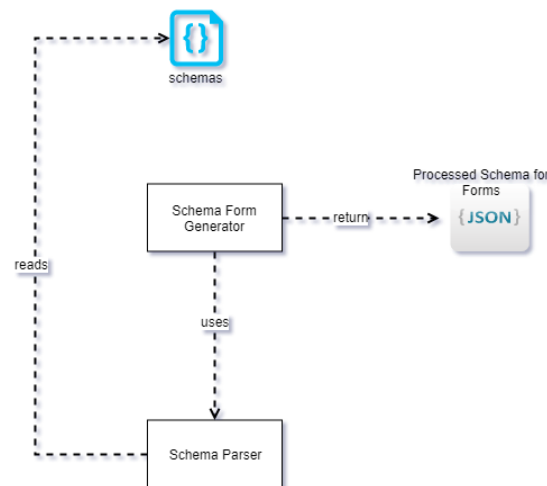


Figura 23: Generación dinámica de formularios

3. Evaluación

En este apartado se presentan las distintas partes de la interfaz de usuario además de los resultados obtenidos tras una serie de simulaciones realizadas para un artículo presentado a la PAAMS international conference del 2018⁹

A continuación se muestran capturas de las distintas partes del simulador:

- Menú: Las opciones disponibles son: crear configuración, simular, ver simulador y analizar datos.



Figura 24: Menú principal de la GUI

- Create configuration: Se pueden crear de forma interactiva simulaciones pudiendo crear entry points con radio de una forma intuitiva y delimitar la zona de simulación. En esta ventana se utilizan los formularios dinámicos.

⁹<https://www.paams.net/>

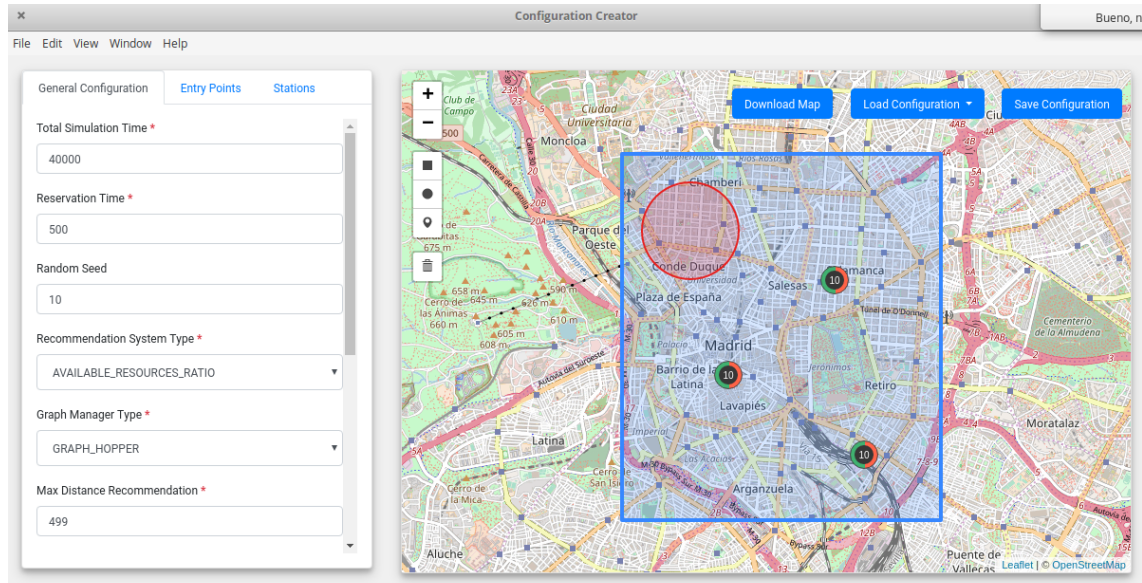


Figura 25: Simulación creada desde la GUI

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 26 se puede ver una lista de Entry Points en la configuración.

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

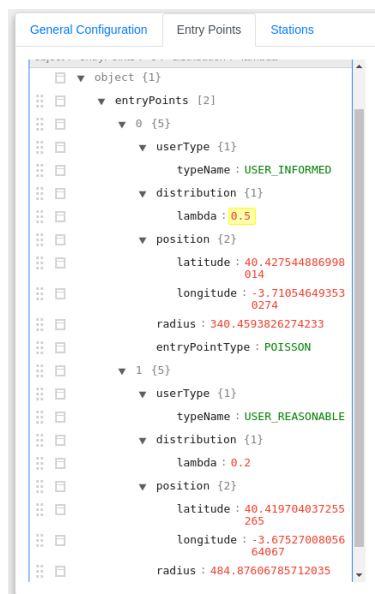


Figura 26: Lista de Entry Points en GUI

- Simulate configuration: 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 generados.

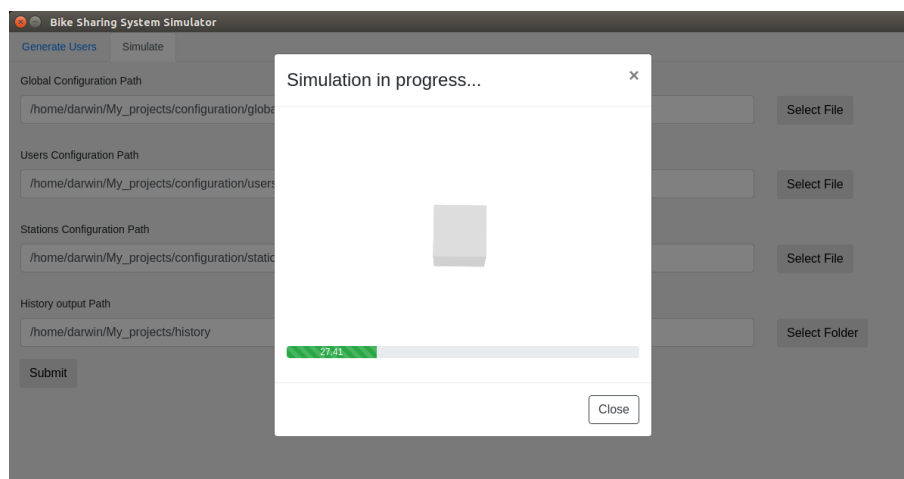


Figura 27: Simulación ejecutándose desde la GUI

- **View Simulation:** Desde este visualizador se puede cargar el histórico de la simulación que se desee y ver como los usuarios actúan dentro del sistema.

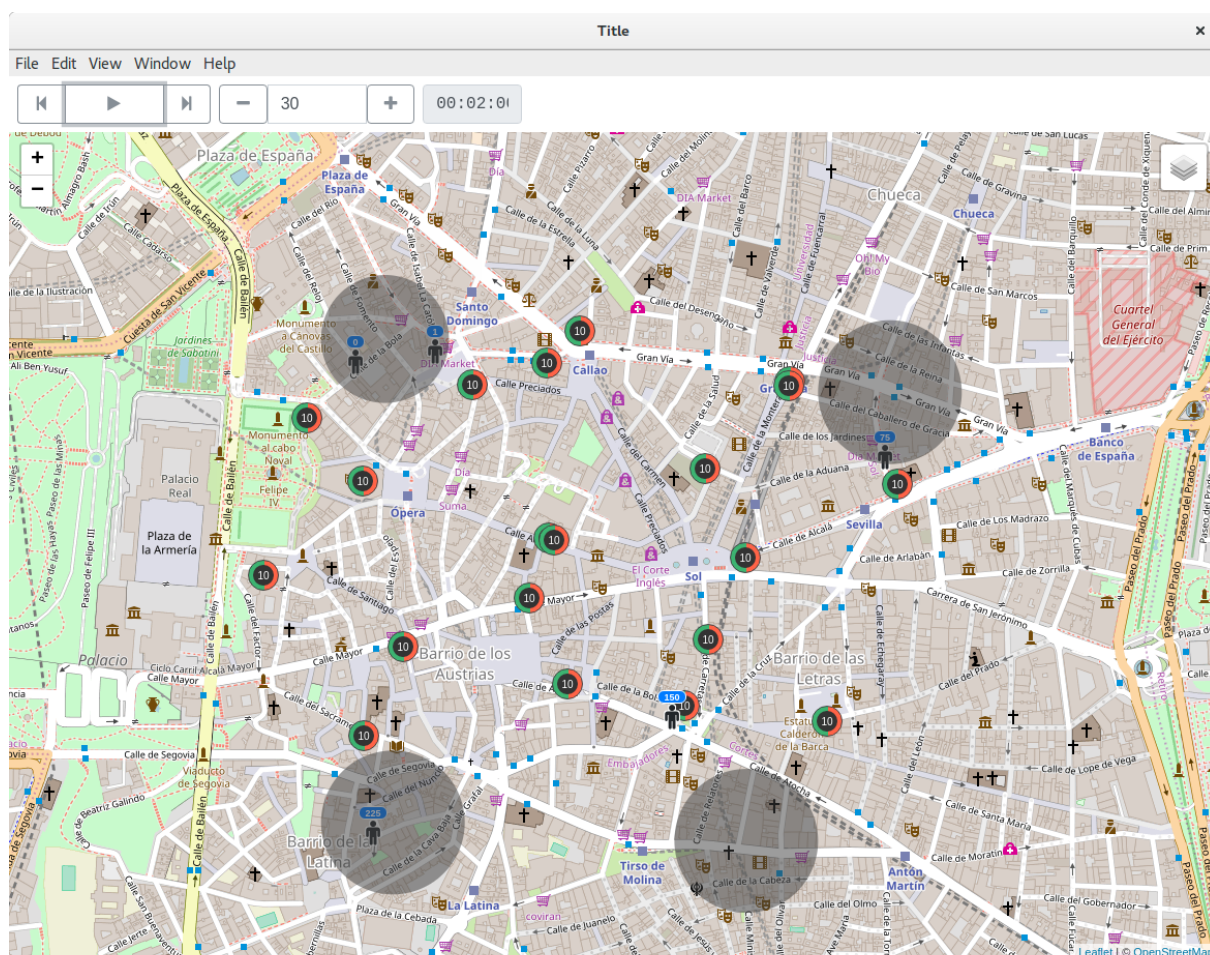


Figura 28: Visualización de histórico de una simulación

- **Analyse Simulation:** Al igual que en el visualizador, se carga el histórico de la simulación que se desee, generando un conjunto de archivos csv con información relativa a la simulación.

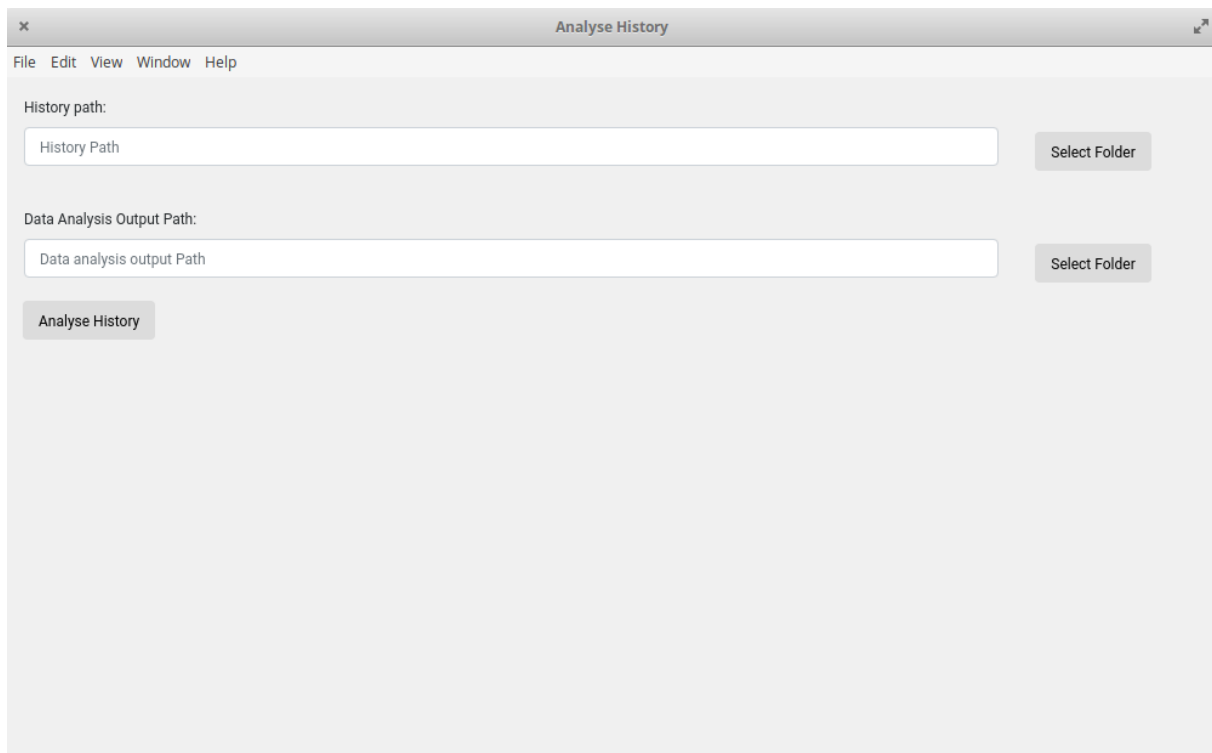


Figura 29: Analizador de históricos

3.1. Prueba 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 el punto de entrada. Este usuario no tiene información de la disponibilidad de bicis o de huecos en la estación.
- **Informed:** Este usuario usa información del sistema y sólo va a estaciones con bicis además de escoger la ruta más corta para ir a la estación.
- **Obedient:** Pide información al sistema de recomendaciones y siempre sigue sus sugerencias. Este recomendador devuelve estaciones en un rango de 600 metros a la posición del usuario, ordenadas por el ratio de bicis o huecos disponibles, dependiendo de si el usuario quiere coger o devolver una bici.
- **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 un experimento como el de la figura 28. Se tienen 20 estaciones

repartidas por el centro de Madrid y 4 entry points. Las medidas básicas que provee el simulador son:

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

A partir de estos valores se calculan los siguientes medidas de calidad:

- *DS* (Demamd 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.

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

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

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

Se realizaron experimentos en los que se aumenta progresivamente el número de usuarios en un mismo espacio de tiempo. Los resultados obtenidos se presentan en la figura 30 y en la figura 31.

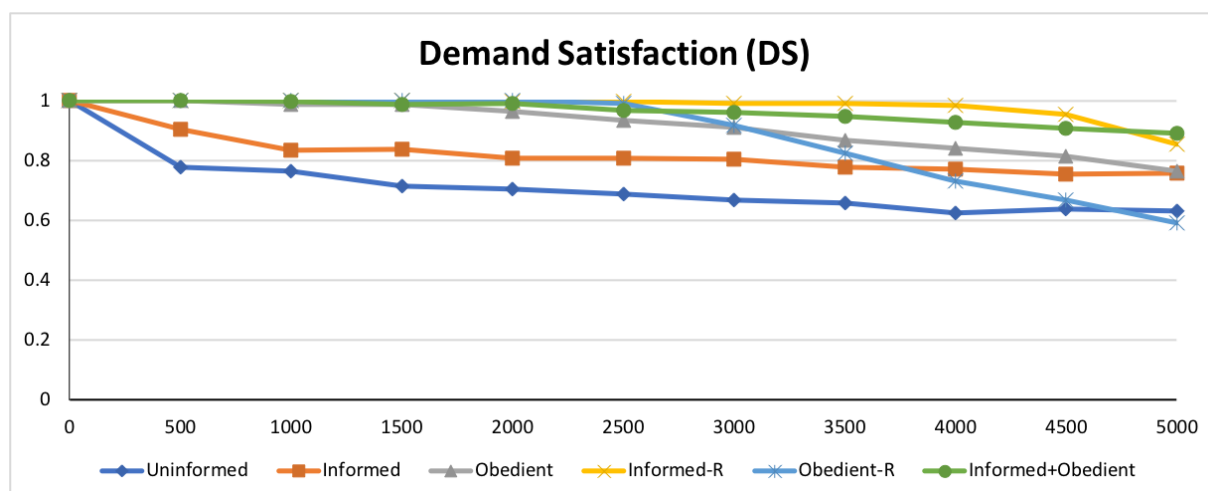


Figura 30: Satisfacción de demanda

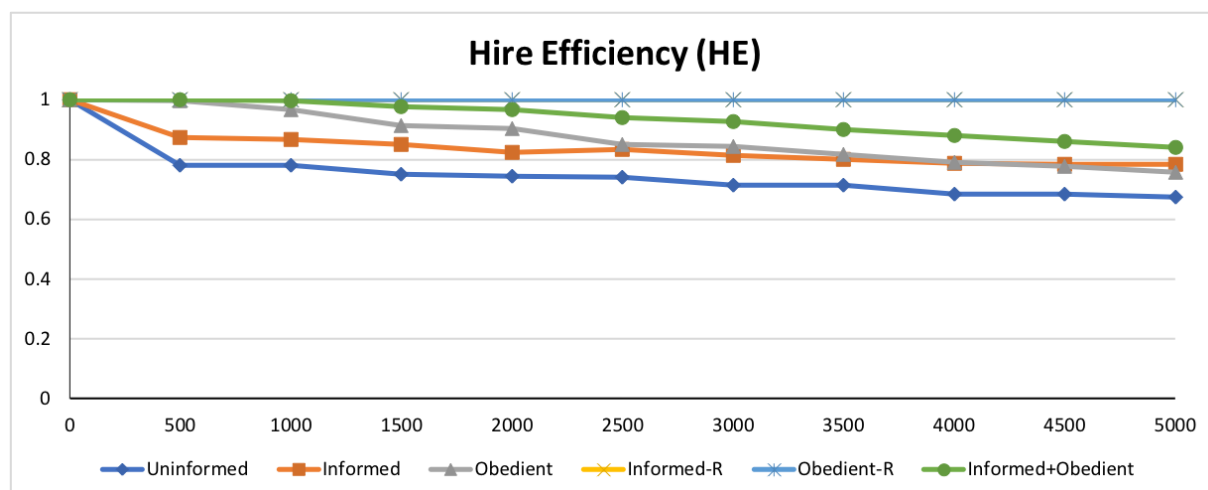


Figura 31: Eficiencia al alquilar

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 peor ya que tienen la bici reservada durante más tiempo, imposibilitando a otros usuarios el poder coger dichas bicis reservadas.

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 adaptable a cambios, extensible, fácil de modificar, con una interfaz de usuario que nos permite realizar simulaciones de una manera sencilla.

Cada una de las partes de este proyecto se ha hecho siempre pensando a futuro hacia nuevas implementaciones y cambios.

4.1. Lecciones aprendidas

No siempre las lecciones aprendidas son debido 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 schemas para los JSON 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, el 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 librería 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 cometer:

- 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. Aun así el simulador funciona correctamente.
- Angular y Electron son demasiado pesados. En un principio, Angular no se pensó como framework multiventana como nosotros lo hemos utilizado, y las ventanas tardaban mucho en cargar de forma individual. Además, Angular tiene mucho código por debajo ejecutando, y esto junto con Electron, suele conllevar a programas algo más lentos y de carga de memoria y disco bastante altos para una aplicación de escritorio. Una alternativa buena hubiera sido utilizar React o Vue.js que son frameworks más ligeros o utilizar una tecnología más sólida para la interfaz de usuario. Pero a pesar de esto, Angular es un framework bastante 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 usuarios a través de un 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 ejecute. 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. Aun así, los usuarios son bastante fáciles de implementar en el proyecto, por lo que no es algo realmente prioritario. Además contamos con la posibilidad de que los usuarios son parametrizables, y estos pueden comportarse de manera diferente a partir de sus parámetros.

- Extender y mejorar la actual versión y corregir posibles errores.
- Crear un sistema de recomendaciones externo, basado en datos reales, que pueda predecir la demanda de una estación y recomendar al usuario una estación que beneficie a ambos, al sistema y al usuario.
- Experimentos más completos para ver el funcionamiento en casos reales.

5. Referencias

- [1] J. S. Ken Schwaber, “La guía de scrum.” <https://www.scrumguides.org/docs/scrumguide/v2016/2016-Scrum-Guide-Spanish.pdf>, 2013.
- [2] I. Sommerville, *Ingeniería del software*. pp. 62–68.
- [3] “Semantic versioning 2.0.0.” <https://semver.org>.
- [4] R. M. Agut, *Especificación de requisitos software según el estándar de iee 830*.
- [5] S. T. Mayo, *Núcleo de un simulador y análisis de datos para sistemas de bicis compartidas*. 2018.
- [6] J. Preshing, “How to generate random timings for a poisson process.” <http://preshing.com/20111007/how-to-generate-random-timings-for-a-poisson-process/>.
- [7] D. Knuth, *The art of computer programming - volume 2*. 1983, p. 116.
- [8] “Uniform random points in a circle using polar coordinates.” <http://www.anderswallin.net/2009/05/uniform-random-points-in-a-circle-using-polar-coordinates/>.
- [9] “Calculate distance, bearing and more between latitude/longitude points.” <http://www.movable-type.co.uk/scripts/latlong.html>.
- [10] T. Cumplido, *An extensible visualization component for a bike sharing simulator*. 2018.

6. Anexos

Anexo 1 - Archivos de configuración

A continuación se presentan los diferentes archivos de configuración que son necesarios para crear simulaciones.

Configuración global

```
{
  "reservationTime": 500,
  "totalSimulationTime": 86400,
  "debugMode": true,
  "randomSeed": 201,
  "boundingBox": {
    "northWest": {
      "latitude": 40.4215886,
      "longitude": -3.7134038
    },
    "southEast": {
      "latitude": 40.4121931,
      "longitude": -3.69826
    }
  },
  "recommendationSystemType": "AVAILABLE_RESOURCES_RATIO",
  "graphManagerType": "GRAPH_HOPPER",
  "maxDistanceRecommendation": 1000
}
```

- Reservation Time: Tiempo máximo para las reservas (segundos).
- Total Simulation Time: Tiempo total de la simulación (segundos).
- Debug Mode: Modo depuración. Si es `true` se generarán unos archivos log con información individual de cada usuario.
- Random Seed: Semilla de aleatoriedad. Puede tener cualquier valor entero. Las simulaciones realizadas con el mismo valor de semilla, serán iguales.
- Bounding Box: Cuadro delimitador. Contiene dos puntos que delimitan el cuadro donde se realizará la simulación. `northWest` y `southEast` se corresponde con los puntos noroeste y sureste del cuadro delimitador.
- Recommendation System Type: Recomendador que se está utilizando en ese momento.
- Graph Manager Type: Gestor de rutas que se está utilizando en ese momento.

- Max Distance Recommendation: Máxima distancia a la que los recomendadores hacen sugerencias. (metros)

Estaciones

```
{
  "stations": [
    {
      "bikes": 10,
      "position": {
        "latitude": 40.4197429,
        "longitude": -3.7080733
      },
      "capacity": 20
    },
    ...
  ]
}
```

- Bikes: Numero de bicis en la estación.
- Position: Posición de la estación.
- Capacity: Capacidad total de la estación.

Puntos de entrada (Entry points)

```
{
  "entryPoints": [
    {
      "entryPointType": "POISSON",
      "userType": {
        "typeName": "USER_INFORMED",
        "parameters": {
          "minRentalAttempts": 2
        }
      },
      "position": {
        "latitude": 40.419969,
        "longitude": -3.709819
      },
      "radius": 100,
      "distribution": {
        "lambda": 1
      }
    },
    ...
  ]
}
```

```
]
}
```

- Entry Point Type: Tipo de entry point, en este caso Poisson.
- User Type: Tipo de usuario y parámetros:
 - Type Name: Tipo del usuario que se quiere generar.
 - Parameters: Parametros de los usuarios. En este caso está especificado que solo puede intentar alquilar una bicicleta 2 veces.
- Position: Posición origen del punto de entrada.
- Radius: Radio de aparición de los usuarios. (metros)
- Distribution: Parámetros de la distribución. En este caso lambda, que vale 1 (en minutos). Significa que se creará de media 1 usuario por minuto con un proceso de poisson.

Usuarios generados

```
{
  "initialUsers": [
    {
      "position": {
        "latitude": 40.420634503997825,
        "longitude": -3.7103315803956933
      },
      "userType": {
        "typeName": "USER_INFORMED",
        "parameters": {
          "minRentalAttempts": 2
        }
      },
      "timeInstant": 19
    },
    ...
  ]
}
```

- Position: Posición en la que aparece el usuario.
- User Type: Es el mismo userType que el utilizado en los puntos de entrada.
- Time Instant: Instante de tiempo en el que aparece el usuario. (segundos)

Anexo 2 - Manual de configuración del entorno de desarrollo

Prerrequisitos

Es necesario tener instaladas las siguientes herramientas:

1. JDK 1.8

2. Maven ≥ 3.5
3. Node.js ≥ 8.9
4. Git

Asegúrese de que todos los binarios estén registrados en la variable de entorno PATH.

El gestor de paquetes NPM es también necesario pero normalmente es instalado automáticamente con Node.js.

Setup

Ejecute las siguientes instrucciones en el directorio donde desee trabajar:

```
git clone https://github.com/stimonm/Bike3S.git
cd Bike3S
npm install
node fuse configure:dev
```

Si no aparece ningún error, ya está todo preparado para desarrollar, empaquetar y probar el simulador.

Ejecutar el simulador en el entorno de desarrollo

Para ejecutar el simulador en modo gráfico en el entorno de desarrollo, solo es necesario ejecutar el siguiente comando en el directorio raíz del proyecto:

```
node fuse build:frontend
```

Comandos básicos para desarrollo

Compilar Backend:

```
node fuse build:backend
```

Compilar Frontend:

```
node fuse build:frontend
```

Compilar todo:

```
node fuse build:dist
```

Crear ejecutable e instalador

```
npm run distribute
```

Anexo 3 - Como añadir un entry point

Para poder simular formas de entrada dentro del sistema, tenemos que crear un entry point. Vamos a crear un entry point de ejemplo a modo de guía. Este entry point creará usuarios de forma secuencial dado unos segundos como parámetro:

1. Creamos una clase para el Entry Point.

Se puede crear en el módulo `backend- bikesurbanfleets-config-usersgenerator` en el paquete `package es.urjc. ia.bikesurbanfleets.usersgenerator.entrypoint.implementations`

La clase debe tener el siguiente aspecto:

```
1 @EntryPointType("SEQUENTIAL")
2 public class EntryPointPoisson extends EntryPoint {
3
4     private GeoPoint position;
5
6     private UserProperties userType;
7 }
```

La anotación `EntryPointType("SEQUENTIAL")`, determina el nombre del Entry Point y lo hace visible a el simulador para poder crearlo. Los atributos `GeoPoint position` y `UserProperties userType` son obligatorios, determinan la posición y el tipo de usuario que se va a implementar junto con sus parámetros.

Necesitaríamos 3 parámetros para este entry point.

- Tiempo entre apariciones.
- Instante de inicio de generación de usuarios.
- Instante final de generación de usuarios.

La clase final quedaría de la siguiente forma:

```
1 @EntryPointType("SEQUENTIAL")
2 public class EntryPointSequential extends EntryPoint {
3
4     private GeoPoint position;
5
6     private UserProperties userType;
7
8     private int secondsBetweenUsers;
9
10    private int startTime;
11
12    private int endTime;
13
14    @Override
```



```
15     public List<SingleUser> generateUsers() {
16         List<SingleUser> users = new ArrayList<>();
17         int actualInstant = this.startTime;
18         while (actualInstant <= this.endTime) {
19             SingleUser user = new SingleUser(this.position,
20                 userType, actualInstant);
21             users.add(user);
22             actualInstant += this.secondsBetweenUsers;
23         }
24         return users;
25     }
26
27 }
```

Como se puede ver hemos creado un método denominado `generateUsers()`. Este es un método abstracto heredado de la clase de la que extendemos `EntryPoint` y debe devolver una lista con los usuarios ya generados. Aquí aplicaremos la lógica necesaria con los atributos que le hayamos atribuido al entry point para generar los usuarios. En este caso es un bucle `while`, el cual en cada iteración creará un usuario cada cierto número de segundos en un rango dado.

2. Añadir el entry point a los schemas

En el directorio raíz del proyecto, en la carpeta `schemas/schemas-and-form-definitions`, en el fichero `entrypoints-config.ts` añadimos lo siguiente:

```
sObject({
  entryPointType: sConst('SEQUENTIAL'),
  position: GeoPoint,
  userType: UserProperties,
  secondsBetweenUsers: UInt,
  startTime: UInt,
  endTime: UInt
}),
```

Este fragmento de código irá a continuación de la línea:

```
export const EntryPoint = sAnyOf(
```

Aquí se especifican todos los parámetros que debe tener un entry point. Los parámetros `entryPointType`, `position` y `userType` no cambian entre entry points.

De este modo generaremos el schema correspondiente del Entry Point y éste será configurable desde la interfaz de usuario sin añadir ninguna línea de código al frontend.

3. Compilar backend y schemas

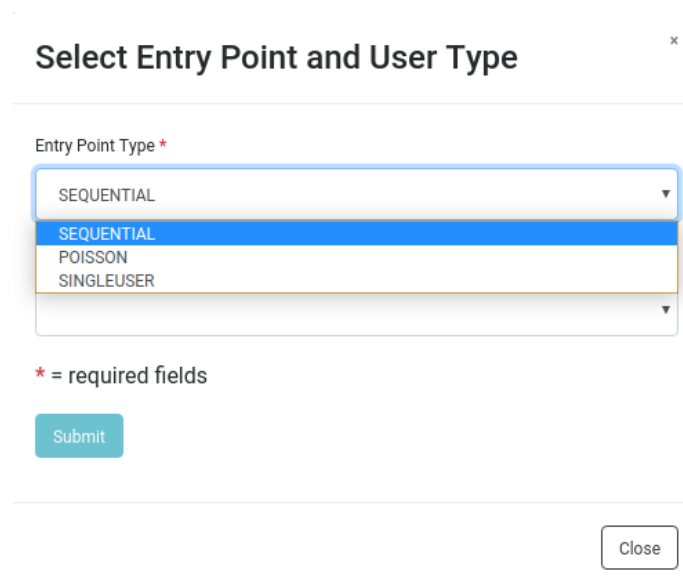
Para probar el nuevo Entry Point debemos ejecutar los siguientes comandos.

```
node fuse build:backend
node fuse build:schema
node fuse build:frontend
```

El primer comando compila el backend, el segundo genera los schemas y el tercero compila y ejecuta el frontend.

4. Probar el entry point para crear una configuración.

Si creamos un entry point en la parte de configuración veremos lo siguiente:



Select Entry Point and User Type

Entry Point Type *

SEQUENTIAL
SEQUENTIAL
POISSON
SINGLEUSER

* = required fields

Submit

Close

Figura 32: Entry Point - Menú de selección de tipo

El frontend nos detecta automáticamente que hay un nuevo tipo de Entry Point. Lo seleccionamos, seleccionamos también el tipo de usuario que queremos y la siguiente ventana que nos aparecerá será la siguiente:

Entry Point parameters

Position

Latitude *

40,42492819330793

Longitude *

-3,7033367156982426

Seconds Between Users

Start Time

End Time

Figura 33: Entry Point - Parámetros usuarios

También aparecen automáticamente los formularios para introducir los parámetros que especificamos en el schema. El entry Point ha sido implementado con éxito. Ya sólo es necesario para terminar la prueba crear una configuración con estos entry points, generar los usuarios y simular.

Anexo 4 - Manual para crear implementaciones de usuario

Para crear un nuevo tipo de usuario se recomienda hacerlo en el módulo `backend-bikesurbanfleets-world-entities` en el paquete `es.urjc.ia.bikesurbanfleets.users.types`.

Vamos a crear un usuario de ejemplo que elija una estación aleatoriamente y que haga reservas dado un porcentaje parametrizable desde el archivo de configuración.

1. Creación de la clase

La clase en un principio debe presentar el siguiente aspecto, antes de comenzar a programarlo:

```

1 @UserType("USER_RANDOM_STATION")
2 public class UserRandomExtension extends User {

```

```
3
4     @UserParameters
5     public class Parameters {
6
7     }
8
9     private Parameters parameters;
10
11     public UserInformed(Parameters parameters, SimulationServices services) {
12         super(services);
13         this.parameters = parameters;
14     }
15 }
```

Es necesario que todas las clases de usuarios tengan como anotación `@UserType` para que el simulador sea capaz de detectar esta nueva clase de usuario y utilizarla. También es obligatorio que la clase de usuario implemente una clase interna que tenga como anotación `@UserParameters`, donde incluiremos todos los parámetros configurables del usuario.

En este caso, queremos que el usuario reserve un porcentaje dado de veces, y que abandone el sistema tras haber intentado coger bici según el parámetro que se le especifique. Para ello añadiremos atributos para representar esta característica a la clase con la anotación `@UserParameters`

```
1     @UserParameters
2     public class Parameters {
3
4         private int minReservationPercentage;
5
6         private int minRentalAttempts;
7
8     }
```

El atributo `minReservationPercentage` representa el porcentaje de veces que realizará una reserva y `minRentalAttempts` el mínimo de veces que queremos que intente coger una bici.

2. Implementación de métodos

Como este usuario no va a realizar reservas, pondremos todos sus métodos de decisión a `false`:

- `boolean decidesToLeaveSystemAfterTimeout()`
- `boolean decidesToLeaveSystemAfterFailedReservation()`
- `boolean decidesToReserveBikeAtSameStationAfterTimeout()`
- `boolean decidesToReserveBikeAtNewDecidedStation()`
- `boolean decidesToReserveSlotAtSameStationAfterTimeout()`
- `boolean decidesToReserveSlotAtNewDecidedStation()`
- `boolean decidesToDetermineOtherStationAfterTimeout()`
- `boolean decidesToDetermineOtherStationAfterFailedReservation()`

Podríamos hacer que el usuario tome decisiones utilizando la memoria del usuario para controlar dichas decisiones. El usuario tiene acceso a una memoria que contiene información de todo lo que ha ido realizando a lo largo de la simulación:

- `bikeReservationAttemptsCounter` - Contador de intentos de reserva de bici
- `bikeReservationTimeoutsCounter` - Contador de timeouts de reserva de bici
- `slotReservationAttemptsCounter` - Contador de intentos de reserva de hueco
- `slotReservationTimeoutsCounter` - Contador de timeouts de reserva de hueco
- `rentalAttemptsCounter` - Intentos de coger bici
- `returnAttemptsCounter` - Intentos de devolver bici

Podemos utilizar esta información para hacer que el usuario decida que hacer en ciertos casos. Por ejemplo en este caso queremos que abandone tras un mínimo de intentos, el método `decidesToLeaveSystemWhenBikesUnavailable()` quedará de la siguiente forma:

```
1      @Override
2      public boolean decidesToLeaveSystemWhenBikesUnavailable() {
3          return this.getMemory().getRentalAttemptsCounter() > 2;
4      }
```

Este método devolverá `true` si se ha superado el numero mínimo de intentos y abandonará el sistema.

También el usuario tiene la posibilidad de dar una vuelta en bici, en vez de ir directamente a la estación. Si no queremos que de una vuelta y vaya directamente a la estación simplemente el método `boolean decidesToReturnBike()` debe devolver `true`.

Procederemos ahora a implementar que ruta debe escoger el usuario para ir a coger o devolver una bici según el gestor de rutas.

```
1      @Override
2      public GeoRoute determineRoute() throws Exception{
3          List<GeoRoute> routes = null;
4          routes = calculateRoutes(getDestinationPoint());
5          if(routes != null) {
6              int index = infrastructure.getRandom().nextInt(0, routes.size());
7              return routes.get(index);
8          }
9          else {
10             return null;
11         }
12     }
```

Con este método se escogerá la primera ruta generada por el gestor de rutas, que en el caso de `GraphHopper` será la más corta.

Ahora implementaremos como queremos que determine la estación a la que queremos que coja una bici y que la devuelva:

```
1      @Override
```

```
2     public Station determineStationToRentBike() {
3         List<Station> allStations = this.services.getInfrastructureManager()
4             .consultStations();
5         int stationRandIndex = infrastructure.getRandom()
6             .nextInt(0, allStations.size());
7         return allStations.get(stationRandIndex);
8     }
9
10    @Override
11    public Station determineStationToReturnBike() {
12        List<Station> allStations = this.services.getInfrastructureManager()
13            .consultStations();
14        int stationRandIndex = infrastructure.getRandom()
15            .nextInt(0, allStations.size());
16        return allStations.get(stationRandIndex);
17    }
```

La clase al completo quedara de la siguiente manera:

```
1  @UserType("USER_STATION_RANDOM")
2  public class UserStationRandom extends User {
3
4      @UserParameters
5      public class Parameters {
6
7          private int minReservationPercentage = 50;
8
9          private int minRentalAttempts = 2;
10
11      }
12
13      private Parameters parameters;
14
15      public UserStationRandom(Parameters parameters, SimulationServices services) {
16          super(services);
17          this.parameters = parameters;
18      }
19
20      @Override
21      public boolean decidesToLeaveSystemAfterTimeout() {
22          return false;
23      }
24
25      @Override
26      public boolean decidesToLeaveSystemAffterFailedReservation() {
27          return false;
28      }
29
30      @Override
```

```
31     public boolean decidesToLeaveSystemWhenBikesUnavailable() {
32         return this.getMemory()
33             .getRentalAttemptsCounter() > parameters.minRentalAttempts;
34     }
35
36     @Override
37     public boolean decidesToReserveBikeAtSameStationAfterTimeout() {
38         return false;
39     }
40
41     @Override
42     public boolean decidesToReserveBikeAtNewDecidedStation() {
43         return false;
44     }
45
46     @Override
47     public boolean decidesToReserveSlotAtSameStationAfterTimeout() {
48         return false;
49     }
50
51     @Override
52     public boolean decidesToReserveSlotAtNewDecidedStation() {
53         return false;
54     }
55
56     @Override
57     public boolean decidesToReturnBike() {
58         return true;
59     }
60
61     @Override
62     public boolean decidesToDetermineOtherStationAfterTimeout() {
63         return false;
64     }
65
66     @Override
67     public boolean decidesToDetermineOtherStationAfterFailedReservation() {
68         return false;
69     }
70
71     @Override
72     public Station determineStationToRentBike() {
73         List<Station> allStations = this.services.getInfrastructureManager()
74             .consultStations();
75         int stationRandIndex = infrastructure.getRandom()
76             .nextInt(0, allStations.size());
77         return allStations.get(stationRandIndex);
78     }
79
```

```
80     @Override
81     public Station determineStationToReturnBike() {
82         List<Station> allStations = this.services.getInfrastructureManager()
83             .consultStations();
84         int stationRandIndex = infrastructure.getRandom()
85             .nextInt(0, allStations.size());
86         return allStations.get(stationRandIndex);
87     }
88
89     @Override
90     public GeoRoute determineRoute() throws Exception{
91         List<GeoRoute> routes = null;
92         routes = calculateRoutes(getDestinationPoint());
93         if(routes != null) {
94             int index = infrastructure.getRandom()
95                 .nextInt(0, routes.size());
96             return routes != null ? routes.get(index) : null;
97         }
98         else {
99             return null;
100         }
101     }
102
103     @Override
104     public GeoPoint decidesNextPoint() {
105         return null;
106     }
107
108 }
```

3. Añadir el usuario a los schemas

Para poder configurarlo desde la interfaz de usuario en la configuración y validarlo debemos añadirlo a los schemas. Para ello nos vamos al directorio `schemas/schemas-and-form-definitions` y en el archivo `users-config.ts` debemos añadir lo siguiente:

```
1     USER_STATION_RANDOM: {
2         minReservationPercentage: Percentage,
3         minRentalAttempts: UInt
4     },
```

Este fragmento de código debe ir después de la línea:

```
export const typeParameters =
```


4. Compilar backend y schemas

Para probar este nuevo usuario debemos ejecutar los siguientes comandos.

```
node fuse build:backend  
node fuse build:schema  
node fuse build:frontend
```

Así tendremos acceso desde la interfaz gráfica a la creación de configuraciones con dichos usuarios.