Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Un simulador de propagación de enfermedades infecciosas basado en el juego de la vida de Conway

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: José Ángel Perea García

Tutor: Carlos Herrero Cucó

2020-2021

Resumen

A pesar de que el ámbito tecnológico y el de la salud son dos ámbitos completamente distintos, siempre ha habido proyectos en los que han podido colaborar ambos, con el objetivo de conseguir un mejor resultado que si hubiera sido por separado.

En pleno 2021 nos encontramos en medio de una pandemia mundial, y muchas personas no son todavía del todo conscientes de lo peligroso que puede resultar un virus para el resto de las personas si no se tiene cuidado.

En este trabajo se plantea desarrollar una aplicación en formato de página web que servirá como un simulador de propagación de enfermedades basado en el juego de la vida de Conway donde, tal y como ocurre en este, para cada entidad de la simulación se realizarán distintos cálculos para alcanzar un nuevo estado en el siguiente ciclo.

El objetivo del proyecto es realizar una simulación completa con ciertos parámetros que definirá el usuario o definidos al azar por el sistema para así posteriormente poder obtener una conclusión con los datos que se han obtenido. Con esto se pretende también que la aplicación sirva para concienciar a las personas del riesgo al que se pueden exponer debido a una enfermedad.

El proyecto se realizará en Javascript, en concreto con la librería llamada React, además de las distintas dependencias que ofrece Node.js.

Palabras clave: simulador, React, aplicación, web, enfermedades, Conway

Abstract

Although technology and healthcare are two completely different fields, there have always been projects in which both have been able to collaborate, with the aim of achieving a better result than if they had been done separately.

In 2021 we are in the middle of a global pandemic, and many people are still not fully aware of how dangerous a virus can be for everyone else if they are not careful.

In this work, it is proposed to develop an application in web page format that will serve as a disease spread simulator based on Conway's game of life, where, as in this game, different calculations will be execute for each entity in the simulation to reach a new state in the next cycle.

The objective of the project is to execute a complete simulation with certain parameters that will be user defined or randomly defined by the system to subsequently be able to obtain a conclusion with the data that have been obtained. With this it is also intended that the application serves to make people aware of how dangerous a disease can be.

The project will be carried out in Javascript, specifically with the library called React, in addition to the different dependencies offered by Node.js.

Keywords: simulator, React, application, web, diseases, Conway

Resum

A pesar de que l'àmbit tecnològic i el de la salut són dos àmbits completament distints, sempre hi ha hagut projectes en què han pogut col·laborar ambdós, amb l'objectiu d'aconseguir un millor resultat que si haguera sigut per separat.

En ple 2021 ens trobem enmig d'una pandèmia mundial, i moltes persones no són encara del tot conscients el perillosos que pot resultar un virus per a la resta de les persones si no es va amb compte.

En aquest treball es planteja desenvolupar una aplicació en format de pàgina web que servirà com un simulador de propagació de malalties basat en el joc de la vida de Conway, on tal i com passa en aquest, per a cada entitat de la simulació es realitzaran diferents càlculs per assolir un nou estat en el següent cicle.

L'objectiu d'el projecte és realitzar una simulació completa amb certs paràmetres que definirà l'usuari o definits a l'atzar pel sistema per així posteriorment poder obtenir una conclusió amb les dades que s'han obtingut. Amb això es pretén també que l'aplicació serveix per conscienciar les persones de lo perillosa que pot arribar a ser una malaltia.

El projecte es realitzarà en Javascript, en concret amb la llibreria anomenada React, a més de les diferents dependències que ofereix Node.js.

Paraules clau: simulador, React, aplicació, web, enfermetats, Conway

Agradecimientos

Son muchas las personas que he conocido a lo largo de estos últimos 4 años, y me gustaría expresar todo lo que han significado, ya que sin ellas toda esta experiencia sería completamente distinta. Sin embargo, hay 2 personas en concreto que merecen ser las primeras en agradecerles todo lo que han hecho por mí.

Gracias, papá, por todo el sacrificio que has hecho a lo largo de estos últimos años, con el fin de darme la oportunidad para poder estudiar y formarme en la carrera que desde pequeño siempre he encontrado fabulosa.

Gracias, mamá, por apoyarme en los peores momentos, donde siempre estabas ahí para darme el empujón que me era necesario para seguir adelante y no tirar la toalla.

No me olvido de mi hermano, Daniel, que más que ser familia, es mi amigo y compañero. ¡Te quiero mucho!

También tengo que agradecer al grupo de amigos que he formado desde el día 1 que entré en la facultad, y el cual, ha hecho que Valencia se sienta como mi segundo hogar. Gracias Miguel, Álvaro, Jorge, Pablo, Jaime. Os elegiría otra vez sin dudarlo, ya que sin vosotros estoy seguro de que, hoy en día, sería una persona completamente distinta.

Debo dar las gracias a Óscar, epidemiólogo que decidió dedicar un poco de su tiempo a ofrecer una entrevista para resolverme algunas dudas y obtener algunos datos que resultaban de interés a la hora de desarrollar la aplicación.

Por último, agradezco a mi tutor, Carlos, el cual se ha encargado de orientar y supervisar tanto el desarrollo de la aplicación como la redacción de la memoria de esta.

Índice de contenidos

Índice de ilustraciones

Índice de tablas

1. Introducción

Estas últimas décadas, el conocimiento científico ha ido en aumento, debido en gran parte a la introducción de nuevos métodos de investigación que han servido para intentar conocer la mejor manera con la que hacer frente a los distintos problemas que la humanidad ha sufrido en estos últimos años, entre los que se incluyen, calentamiento global, biodiversidad en peligro o pandemias entre otros.

Muchos de estos métodos de investigación se basan en el uso de herramientas procedentes del sector tecnológico, donde la mayoría de estos se ven relacionados en mayor o menor medida con la informática. Un ejemplo bastante claro puede ser la bioinformática, en la cual, dos ciencias completamente distintas se unen para lograr responder a problemas biológicas que, por separadas no serían capaces de resolver. Una práctica de esta disciplina puede ser el análisis y almacenaje masivo de datos biológicos a través del uso de algoritmos cuya función es extraer el máximo conocimiento de los datos.

Los simuladores son herramientas que resultan de gran interés, ya que estas logran reproducir el comportamiento de un sistema en determinadas condiciones, sirviendo así, como aprendizaje y un medio del que extraer datos. Normalmente suelen separar lo fundamental de lo irrelevante, consiguiéndolo mediante el uso de algoritmos que suelen imitar los comportamientos y reglas de dichos sistemas. También permite realizar simulaciones con unas características en concreto, que, de otra manera, reproducir en un laboratorio sería inviable, ya sea por un tiempo, coste o riesgo demasiado alto.

En la presente memoria se realiza y detalla el desarrollo de un simulador de propagación de enfermedades programado en **JavaScript+React**, mediante el cual, podremos realizar ejecuciones en las que observar cómo se expande una enfermedad, las consecuencias de esta, y los datos que pudieran ser de interés de esta. El simulador está basado en el clásico juego de la vida de Conway, un autómata celular que simula un sistema dinámico siguiendo ciertas reglas establecidas.

Los valores por defecto que se han establecido han sido los de la reciente enfermedad que ha asolado todo lo que ha comprendido 2020 y 2021, Covid-19. Sin embargo, no solo se ha elegido al Covid-19 por el impacto que ha tenido recientemente, sino, también debido a que Conway, el creador del juego de la vida, murió el 11 de abril de 2020 a causa de esta enfermedad. Se espera que este trabajo sirva como “humilde homenaje” al profesor Conway.

Los valores mencionados se han obtenido en base a la investigación de los distintos documentos pertenecientes a PubMed, una base de datos especializada en ciencias de la salud y con millones de referencias bibliográficas. Además, también se ha realizado una entrevista a **Óscar Zurriaga**, profesor titular del Departamento de Medicina Preventiva y Salud Pública de la Universidad de València, experto en epidemiología, con el fin de poder resolver algunas dudas y saber desde su punto de vista que sería interesante ver en el desarrollo de la aplicación.

1.1 Motivación

Una de las principales razones que ha llevado a realizar el desarrollo del simulador es que debido a que actualmente la sociedad se encuentra en medio de una pandemia, y muchas personas no son conscientes del riesgo que supone esto, por lo tanto, se espera que sirva a no solo para ayudar a que las personas tomen conciencia de lo peligrosa que es la situación, sino a que también sirva como medio por el cual poder extraer datos o incluso sobre el que aprender cómo funciona la expansión de cierta enfermedad.

También pareció interesante la idea de desarrollar un proyecto basado en el concepto del juego de la vida de Conway, ya que los programas derivados de este suelen mostrar lo que podría llamar una ejecución orgánica o, dicho de otra forma, un programa que da la sensación de estar vivo, y no hay mejor manera para representar este concepto que en un simulador.

1.2 Objetivos

El objetivo principal, como se mencionó anteriormente, consiste en el desarrollo de un simulador de propagación de enfermedades basado en el juego de la vida de Conway, con el fin de que no solo sirva de aprendizaje, si no a también poder obtener ciertos datos de interés que se producirían en determinadas situaciones muy complicadas de reproducir en la vida real. Por lo tanto, a partir de este objetivo principal, podemos extraer un conjunto de objetivos secundarios que servirán para conseguir el desarrollo del simulador:

- Visualización del estado de cada entidad del sistema.
- Un editor que permita al usuario definir un escenario desde el cual realizar la simulación.
- Parametrizar todas las variables del programa para que el usuario pueda definir el valor de sus variables.
- Integrar en el simulador las distintas reglas que se ejecutarán en cada ciclo de nuestro programa, tal y como puede ser infectar, aislar o morir.
- Disponer de distintas gráficas desde las que poder observar la evolución de distintos datos a lo largo de la ejecución de las generaciones del programa.

Gracias a los objetivos mencionados se pueden especificar las distintas características y tareas pertenecientes realizar al proyecto a realizar.

Además, también es de interés conseguir otras clases de objetivos, en este caso de aquellos que sirvan como un beneficio a la sociedad:

- Conseguir que el simulador sirva como aprendizaje para aquellas personas que se están formando en el campo de las ciencias de la salud.
- Hacer crecer la consciencia pública de que ciertas enfermedades infeccionas pueden ser mucho más peligrosas de lo que creen.
- Poder recrear mediante el simulador situaciones que no serían viables en un laboratorio o cualquier otro entorno.

1.3 Contexto

John Horton Conway fue un matemático británico (1937-2020), el cual, creó trabajos que lograron atribuirle un gran prestigio. Algunos de estos trabajos son la teoría de grupos, teoría de nudos o teoría de probabilidades, sin embargo, unas de sus propuestas más conocidas es el juego de la vida.

El juego de la vida de Conway consiste en un modelo matemático con un sistema dinámico que avanza a un estado futuro dependiendo de las condiciones del estado anterior y al cálculo de ciertas reglas, provocando así en cada instante de tiempo (también llamado generación), una evolución del sistema.

El juego se representa en una especie de tablero o cuadrícula, donde los cuadrados de esta son lo que se denomina células, las cuales pueden estar en 2 estados, vivas o muertas, viéndose representadas blancas o negras respectivamente. Conforme van avanzando las generaciones podemos ver la interacción de cada célula del sistema, provocando así, un sistema dinámico que evoluciona por sí solo, como si de algo orgánico o vivo se tratase.

Es curioso, ya que, a pesar de llamarse juego de la vida, es un juego que no necesita a ningún jugador para ser jugado, ya que la única entrada que requiere es la del estado inicial, denominada también población inicial o generación cero. Todas las células se actualizan simultáneamente en cada instante de tiempo, donde siguiendo las siguientes reglas establecen su nuevo estado:

- Una célula **sobrevive** si en las 8 casillas alrededor de la misma tiene 2 o 3 vecinos vivos.
- Una célula vive que tenga en sus 8 casillas alrededor menos de 2 vecinos vivos, **fallece** por aislamiento.
- Una célula viva que tenga más de 3 vecinos vivos **fallece** por superpoblación.
- Una célula muerta que tenga a su alrededor un total de 3 células vivas, en la siguiente generación pasará a ser una célula **viva**.

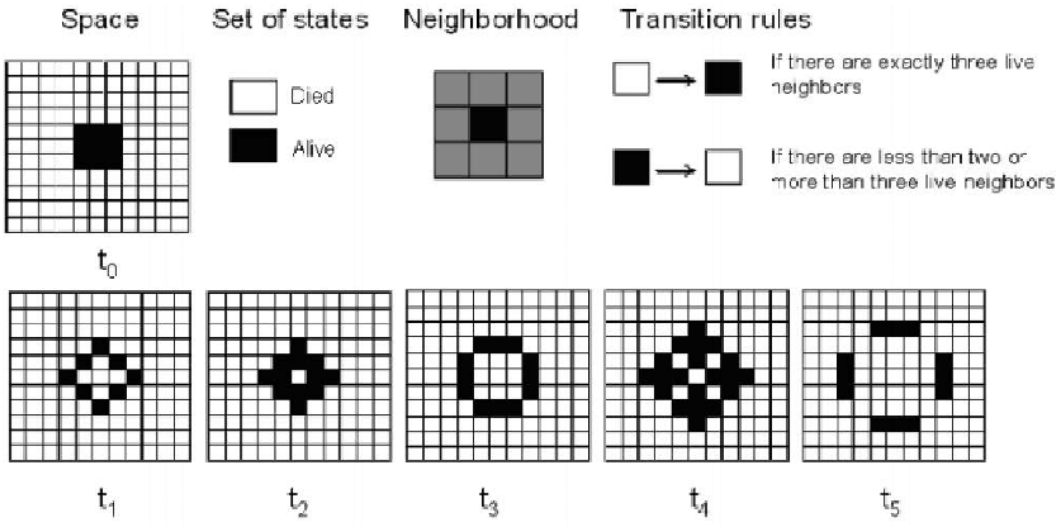


Ilustración 1. Evolución de una población a lo largo de las generaciones

Gracias a las reglas mencionadas anteriormente, una población, definida por un grupo de células, en su siguiente generación puede acabar en uno de los siguientes 3 estados: **extinguida,** donde no queda ninguna célula viva, **estabilizada,** en un estado o más que va oscilando entre uno y otros, o **crecimiento,** donde se ve un aumento de las células vivas.

El juego de la vida de Conway llamó rápidamente la atención de todo el mundo debido a que, con sus pocas reglas, conseguía resolver cualquier problema que fuera computable algorítmicamente, siendo equivalente a una maquina universal de Turing, y por lo tanto, capaz de crear relojes o sistemas lógicos a partir de la evolución de los patrones generados por las reglas. Además, hay que mencionar la cantidad de patrones que se pueden llegan a formar durante la ejecución del sistema, ya que llega a tal punto, que hasta se han llegado a clasificar a los mismos:

- Osciladores: Patrones que después de un número finito de generaciones vuelven de nuevo a su estado inicial, y, por lo tanto, formando un bucle que se repite infinitamente.
- Inmortales: Patrones que no varían de una generación a la otra, por lo que se mantienen estables, pero siempre y cuando no se vean influenciados por la interacción con otro patrón o población independiente de la misma.
- Naves espaciales: Patrones que se desplazan por el tablero en una misma dirección, donde después de un número finito de generaciones vuelven a su estado inicial, pero en una ubicación distinta.
- Matusalenes: Patrones que después de un número determinado de generaciones se estabiliza o muere.
- Reflectores: Patrones inmortales inicialmente, pero que tras recibir una colisión de un patrón nave espacial, genera otra igual en una dirección distinta a la inicial.

Ilustración 2. Tipos de patrones del juego de la vida de Conway

Las reglas mencionadas anteriormente, también llamadas leyes genéticas de Conway, son consideradas las que inicialmente definió Conway. Sin embargo, es posible añadir o modificar toda regla o estado que se quiera, estableciendo así un sistema de acuerdo con nuestras necesidades.

Un ejemplo de los muchos autómatas celulares que surgieron a raíz de variaciones del juego de la vida fue Wireworld, creado por Brian Silverman, siendo capaz de simular dispositivos electrónicos, gracias a que cada célula perteneciente al autómata utiliza cuatro posibles estados de celda, cada una representada por un color: vacío (negro), cabeza de electrones (blanco), cola de electrones (azul) y conductor (naranja). Además, para decidir el estado de cada célula en la siguiente generación se deben seguir siguientes reglas:

- Una celda vacía siempre estará vacía, por lo que no puede cambiar de estado.
- Una cabeza de electrón en su siguiente estado será una cola de electrón.
- Una cola de electrón pasará a ser un conductor.
- Un conductor seguirá siendo un conductor a no ser que si una o dos células vecinas son cabezas de electrón

Al igual que en el juego de la vida, siguiendo las reglas que se acaban de mencionar podemos crear estructuras como diodos o puertas lógicas, tales como AND o XOR. En la siguiente imagen, podemos observar cómo se ha creado un banco de registros capaz de almacenar en cada uno de los mismos un número de 16 bits, asimismo, es capaz de leer o escribir ciclos.

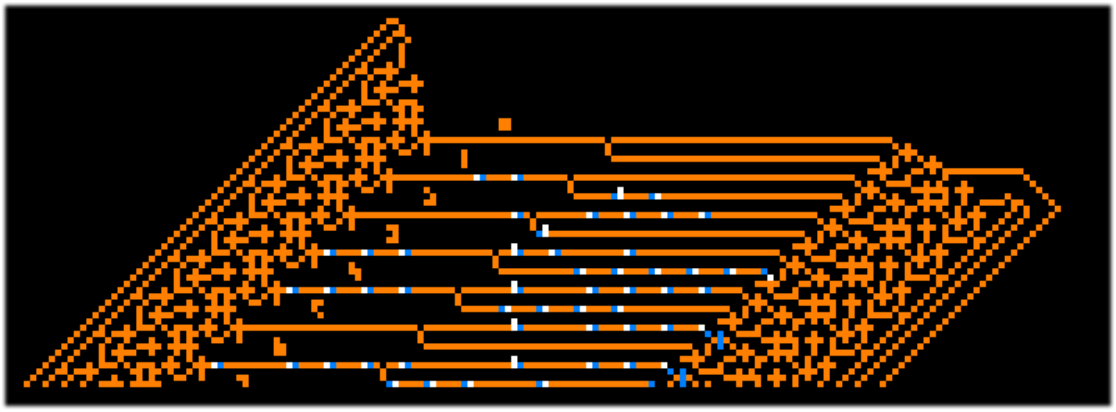


Ilustración 3. Banco de registros creado con Wireworld

Una vez mostrado el potencial que se puede alcanzar modificando los estados o las reglas pertenecientes al juego de la vida, queda claro que el único límite es la imaginación. Por lo tanto, se plantea la siguiente pregunta: ¿sería posible crear un simulador de propagación de enfermedades?

La respuesta, tal y como se puede deducir, es si, de hecho, se ha tratado de encontrar alguna aplicación parecida a la que se plantea desarrollar, sin embargo, aunque existen una gran cantidad de simuladores de infección, no se ha logrado encontrar ninguno que se base en el juego de la vida de Conway.

1.4 Metodología

Para el desarrollo de la aplicación se ha decidido aplicar la metodología en cascada, un proceso de desarrollo que consiste en un conjunto de etapas que se van realizando de forma secuencial en el orden que están definidas, pero solamente es posible avanzar a la siguiente etapa una vez se han terminado completamente las etapas previas. Esto no significa que no haya vuelta atrás, es posible, pero puede llevar a altos costes.

Debido a que no se va a estar en contacto con un cliente para que supervise el producto en los llamados Sprints o entregas a lo largo del desarrollo, se ha decidido que lo más viable era elegir este método de desarrollo, ya que se definen los requisitos al principio del desarrollo, y, por lo tanto, estos estarán cerrados y no se podrán ver alterados puesto que no hay ningún cliente.

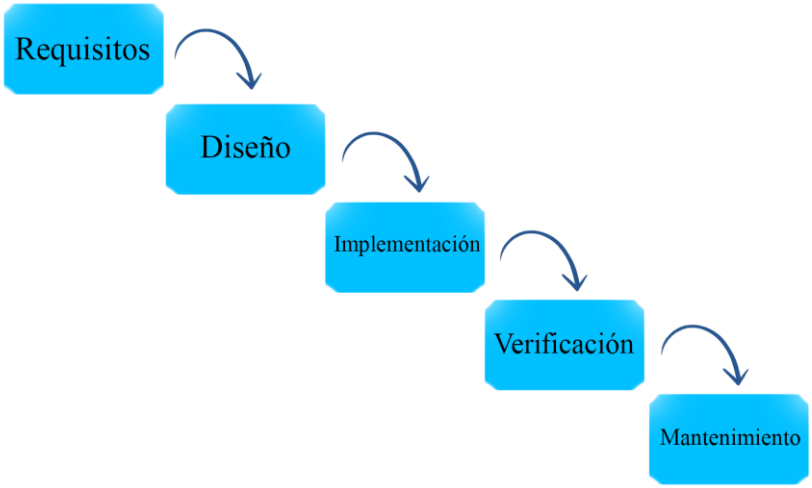


Ilustración 4. Etapas del modelo en cascada

1. **Requisitos**: Se realiza un análisis en profundidad de lo que se espera que conforme el conjunto de funcionalidades de la aplicación, por lo que se deben definir conjuntos como requisitos funcionales y no funcionales o diagramas de clase.

2. **Diseño**: En base a los resultados obtenidos de la fase anterior se elige la mejor solución posible, eligiendo lo que serán tanto la tecnología como las herramientas utilizadas.

3. **Implementación**: Fase en la que se comienza a programar la aplicación con la intención de que se cumpla todo lo especificado en las etapas anteriores.

4. **Verificación**: Una vez se ha completado el desarrollo de la aplicación se crean un conjunto de pruebas para validar el correcto funcionamiento de esta.

5. **Mantenimiento**: Una vez finalizadas las etapas de verificación y implementación, la etapa de mantenimiento se encarga de detectar y corregir posibles fallos que no se hayan encontrado en las etapas anteriores o bien incluir funcionalidades que se deberían haber implementado, pero han pasado desapercibidas en la fase de requisitos.

1.5 Estructura

En este último apartado de la Introducción, se mencionan lo puntos principales que se van a encontrar a lo largo de la memoria, además de adjuntar a cada una de las mismas una explicación sobre lo que van a tratar:

1. **Introducción:** encargado de dejar claro al lector el contenido de la memoria, además de explicar el contexto, motivos y objetivos que han llevado a la elección del proyecto a realizar.

2. **Análisis del problema:** aquí se define el conjunto de requisitos de la aplicación a realizar, los cuales se identifican a través de los casos de uso o entrevistas realizadas.

3. **Diseño de la solución:** después de establecer los requisitos del problema, aquí se habla del diseño elegido para aplicación, así como la tecnología utilizada.

4. **Desarrollo de la solución:** apartado en el cual se habla de todo el desarrollo de la aplicación, explicando cómo se han realizado ciertas partes del código tanto los problemas que se han tenido que afrontar y como se han resuelto.

5. **Pruebas:** se muestran las distintas pruebas que se han implementado para demostrar que las funcionalidades y requisitos planteados en el apartado de análisis del problema se cumplen correctamente.

6. **Conclusiones:** se mencionan las conclusiones obtenidas sobre la realización del trabajo, además de también realizar una valoración personal sobre lo que ha supuesto la realización del proyecto.

7. **Trabajos a futuro:** posibles funcionalidades que se podrían implementar en un futuro con tal de que la aplicación siga evolucionando hacía una versión más completa.

8. **Referencias:** conjunto de información que se ha empleado a la hora de elaborar tanto la aplicación como la memoria.

2. Análisis del problema

Tal y como se presentó en la introducción, el objetivo es desarrollar una aplicación capaz de simular la propagación de enfermedades infecciosas con tal de que sirva de utilidad para los distintos puntos que se mencionaron en el apartado de motivaciones y objetivos. Sin embargo, antes de comenzar el desarrollo de cualquier proyecto, es vital realizar previamente un análisis sobre el mismo.

En este apartado se trata de elegir el mejor modo para abordar la realización del trabajo, y para ello, se usan distintas técnicas elicitación con tal de descubrir y extraer los requisitos, además de también especificar y analizar a los mismos a través de distintos modelos, como por ejemplo, los casos de uso.

Antes que nada, debemos tener claro las distintas fuentes de requisitos de las que extraer los mismos, entre las que destacan los objetivos que definimos en la introducción, y los *stakeholders*, considerados todos los que tengan alguna relación con el sistema, siendo por ejemplo una persona que ha padecido de alguna enfermedad infecciosa.

Ya que en el apartado de objetivos se vieron las distintas funcionalidades que resultan interesantes para implementar a la aplicación, a continuación, se habla de otra técnica que es útil para identificar requisitos, las entrevistas.

2.1 Entrevistas

Las entrevistas son consideradas una técnica bastante frecuente tradicional de elicitación de requisitos, las cuales suelen resultar bastantes útiles ya que se interactúa con un stakeholder de interés con el fin aclarar aspectos o conceptos muy específicos que pueden resultar complicados de entender. Aunque pueden llegar a resultar una técnica un poco complicada, puesto que requiere de que el entrevistador tenga experiencia y habilidades de comunicación.

Para la realización de este proyecto, puesto que se trata de una temática relacionada con la propagación de enfermedades, se decidió entrevistar a **Óscar Zurriaga**, epidemiólogo y profesor titular del Departamento de Medicina Preventiva y Salud Pública de la Universitat de València.

A pesar de que antes de la entrevista se prepararon una serie de preguntas con la intención de ser lanzadas al entrevistado para encontrar respuestas (como pueden ser probabilidades sobre ciertos casos que resultan de interés), también se hablaron sobre temas que fueron surgiendo y que no estaban planteados en un principio. Por ello se puede determinar que el tipo de entrevista es mixta, debido a que resultó ser una mezcla de una estructurada y otra abierta.

La primera pregunta que se suele realizar en las entrevistas es una puesta en contexto, en este caso se le preguntó cómo hoy en día se ha acabado en la situación que el mundo se encuentra, es decir, una pandemia mundial.

Después se presentó una idea general sobre la aplicación que se quería desarrollar en este trabajo, el simulador, donde se presentaron las distintas funcionalidades y conceptos que se querían añadir en un principio, además de explicar un poco en qué consistía el juego de la vida de Conway, concepto en qué se iba a basar el simulador. Al entrevistado le pareció bastante interesante todo lo planteado, por lo que no le pareció mal ninguna de las funcionalidades.

Más tarde se preguntaron algunas dudas que se tenían, que si bien muchas de ellas pudieron ser respondidas previamente si se investigaba un poco, se ha querido obtener respuestas de una fuente de primera mano, no sólo para confirmar los datos, sino también para contrastarlos con los que ya se tenían. A pesar de que muchas de las dudas eran sobre ciertas probabilidades o casos concretos, hubo una pregunta en concreto que fue muy útil, donde esta era la duración de los virus en las superficies. El entrevistado dijo que no suele haber un tiempo fijo, si no que depende de la aireación del recinto, ya que no es lo mismo un virus en la superficie de un banco de un parque que en la puerta del baño de un bar.

Por último, se pidió que, si se le ocurrió alguna sugerencia que pudiera ser curiosa como funcionalidad, a lo que él respondió que podría ser interesante añadir una especie de barra de progreso que represente la creación de la vacuna, la cual, una vez haya sido completada de comienzo la vacunación de la población que se encuentra dentro del simulador.

Todo lo dicho en la entrevista quedó apuntado, con la intención de tener todo lo hablado guardado, con sus preguntas y respuestas, para así más tarde poder consultarlo y hacer una reflexión sobre todo lo obtenido. Por lo tanto, la realización de la entrevista ha servido para lo siguiente:

- Obtener probabilidades aproximadas como pueden ser sobre la transmisión del virus o la letalidad de este en distintos casos, como por ejemplo personas con mascarilla o personas que sean de riesgo.

- Incorporar una funcionalidad que permita en base a un atributo llamado aireación, infectar a los recintos y las personas de este. Además de que la infección de la superficie tarde más rápido o lento dependiendo de la aireación.

- Crear una barra de progreso que represente el porcentaje de completitud de la vacuna. También, la vacuna debe de tener un porcentaje de efectividad, como también debe de seguir un orden sobre a quién aplicarla antes o después.

- Incorporar una cuadrícula para que cada cierto tiempo se aísle a los infectados hasta que ya no lo estén, y otra para los muertos.

2.2 Casos de uso

Dentro del análisis de requisitos se suele realizar una tarea que resulta bastante útil, la cual, no solo ayuda a visualizar el conjunto de requisitos funcionales que tiene la aplicación, viéndose representados mediante casos de uso, sino también a saber qué tipos de usuarios van a ser los que realizan dicha acción. Esta tarea se llama diagrama de casos de uso, y son representados usando el lenguaje de modelado unificado (UML).

Un caso de uso es una descripción de un conjunto de acciones para realizar una funcionalidad determinada que produzca un resultado de valor para los actores o *stakeholders* pertenecientes al dominio de la aplicación.

Normalmente se suele representar para cada aplicación su propio diagrama, donde este está unido a los distintos casos de usos a través de asociaciones. Sin embargo, en la aplicación que se ha desarrollado solamente se tiene en cuenta un solo tipo de actor, la persona que entra en la página web e inicia una simulación. Por tanto, no habrá funciones exclusivas para distintos tipos de actores, puestos que todos los que interactúen con la aplicación serán el mismo.

A continuación, se muestra una imagen en la que se ha representado para el usuario de nuestra aplicación el conjunto de funcionalidades que puede realizar dentro de la misma.

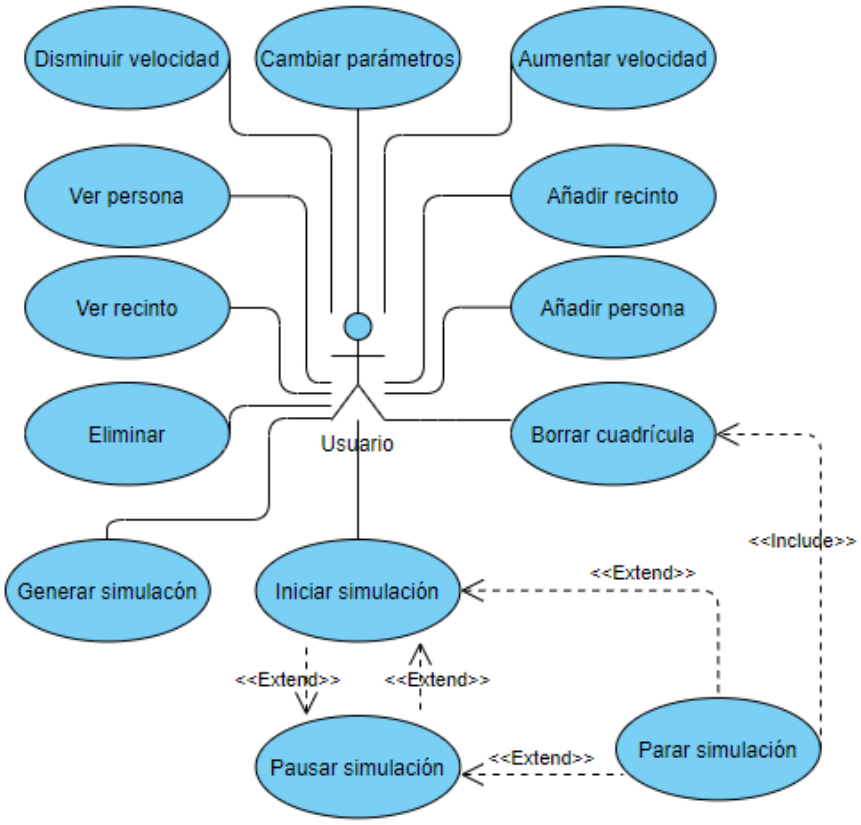


Ilustración 5. Diagrama de caso de uso para usuario

Se encuentran funcionalidades como pueden ser generar una simulación o aumentar la velocidad de esta, pero cabe destacar en concreto tres de ellas.

Iniciar Simulación es una de ellas, funcionalidad que el usuario inicia y tiene un ***extend***, cosa que implica que, dependiendo de un condicional, en este caso dependiente de si el usuario pulsa un botón o no, se realiza la siguiente funcionalidad a mencionar.

Pausar Simulación, al igual que la funcionalidad anterior, no solo posee también un ***extend***, sino que tiene dos, los cuales hacen que se pueda volver a **Iniciar Simulación** o la última funcionalidad.

Por último, **Parar Simulación**, funcionalidad que además de tener un ***extend*** a **Iniciar Simulación,** tiene un ***include***, el cual representa que la funcionalidad de la que parte no puede funcionar sin la que se une, ya que es parte esencial de esta para que su funcionamiento sea correcto.

Una vez representados los requisitos funcionales de nuestro sistema, es hora de especificarlos a través de tablas, en la que se detallan con campos como una descripción a fin de que sean más entendibles.

ID	CA-01
Nombre	Iniciar Simulación
Descripción	El usuario selecciona la opción de iniciar la simulación, donde seguidamente, el sistema entra en un bucle, representando cada iteración con el número de generaciones, en el que se aplica a cada entidad de la simulación las reglas que están establecidas para que alcancen su siguiente estado.
Actor	Usuario
Relaciones	Extend con CA-02 y CA-03
Precondición	

Tabla 1. Caso de uso Iniciar Simulación

ID	CA-02
Nombre	Pausar Simulación
Descripción	El usuario al seleccionar la opción de pausar la simulación, el sistema pausa el bucle en el que se encuentra, por tanto, el número de generaciones deja de aumentar, lo que implica que las entidades permanecen en el estado en que estaban en el momento de haber ejecutado la funcionalidad.
Actor	Usuario
Relaciones	Extend con CA-01 y CA-03
Precondición	El sistema debe de estar con una simulación iniciada y en marcha.

Tabla 2. Caso de uso Pausar Simulación

ID	CA-03
Nombre	Parar Simulación
Descripción	El usuario al seleccionar la opción de parar la simulación, el sistema detiene el bucle en el que se encuentra, para seguidamente reiniciar el número de generaciones y borrar todas las entidades que se encuentran en la cuadrícula de la simulación.
Actor	Usuario
Relaciones	Extend con CA-01, CA-02. Intend con CA-04
Precondición	El sistema debe de estar con una simulación iniciada y en marcha.

Tabla 3. Caso de uso Parar Simulación

ID	CA-04
Nombre	Borrar Cuadrícula
Descripción	El usuario al seleccionar esta opción, el sistema eliminará de la cuadrícula a todas las entidades, tanto recintos como personas.
Actor	Usuario
Relaciones	Intend con CA-03
Precondición	

Tabla 4. Caso de uso Borrar Cuadrícula

ID	CA-05
Nombre	Añadir Persona
Descripción	El usuario al seleccionar el botón que representa la funcionalidad añadir persona, además de quedar presionado, aparece una barra lateral desde la cual puede escribir los valores de las variables de la persona que se va a añadir. Finalmente, al clicar en un cuadrado de la cuadrícula del simulador se añadirá la persona que se ha definido.
Actor	Usuario
Relaciones	
Precondición	

Tabla 5. Caso de uso Añadir Persona

ID	CA-06
Nombre	Añadir Recinto
Descripción	El usuario al seleccionar el botón que representa la funcionalidad añadir recinto, además de quedar presionado, aparece una barra lateral desde la cual puede escribir el valor aireación, variable del recinto que se va a añadir. Finalmente, al clicar en un cuadrado de la cuadrícula del simulador se añade el recinto que se ha definido.
Actor	Usuario
Relaciones	
Precondición	

Tabla 6. Caso de uso Añadir Recinto

ID	CA-07
Nombre	Ver Persona
Descripción	El usuario al seleccionar el botón que representa la funcionalidad ver persona, aparte de quedar presionado, al clicar en una persona del simulador aparece una barra lateral en la cual se mostrará información sobre la entidad pulsada.
Actor	Usuario
Relaciones	
Precondición	El sistema debe de tener como mínimo una persona en la cuadrícula.

Tabla 7. Caso de uso Ver Persona

ID	CA-08
Nombre	Ver Recinto
Descripción	El usuario al seleccionar el botón que representa la funcionalidad ver recinto, aparte de quedar presionado, al clicar en un cuadrado perteneciente a la cuadrícula del simulador, aparece una barra lateral en la cual se mostrará información sobre la entidad pulsada.
Actor	Usuario
Relaciones	
Precondición	

Tabla 8. Caso de uso Ver Recinto

ID	CA-09
Nombre	Aumentar Velocidad
Descripción	Cada vez que el usuario pulsa el botón que realiza esta funcionalidad, disminuye el tiempo que dura cada generación de la simulación, por tanto, la simulación ira más rápida. Sin embargo, tiene un límite y no se puede aumentar infinitamente.
Actor	Usuario
Relaciones	
Precondición	

Tabla 9. Caso de uso Aumentar Velocidad

ID	CA-10
Nombre	Disminuir Velocidad
Descripción	Cada vez que el usuario pulsa el botón que realiza esta funcionalidad, aumenta el tiempo que dura cada generación de la simulación, por tanto, la simulación ira más lenta. Sin embargo, tiene un límite y no se puede disminuir infinitamente.
Actor	Usuario
Relaciones	
Precondición	

Tabla 10. Caso de uso Disminuir Velocidad

ID	CA-11
Nombre	Cambiar Parámetros
Descripción	El usuario al pulsar un botón abre un modal desde el cual puede definir los valores que va a tener la simulación. Una vez definidos, el usuario cancela los cambios o confirma los valores y el sistema los establece.
Actor	Usuario
Relaciones	
Precondición	

Tabla 11. Caso de uso Cambiar Parámetros

ID	CA-12
Nombre	Eliminar
Descripción	El usuario al pulsar el botón de eliminar, además de quedar este pulsado, al clicar en cualquier entidad situada en la cuadrícula del simulador, esta queda eliminada.
Actor	Usuario
Relaciones	
Precondición	Debe de haber entidades en la cuadrícula, ya sean recintos o personas.

Tabla 12. Caso de uso Eliminar

ID	CA-13
Nombre	Generar Simulación
Descripción	El usuario al clicar en el botón que representa la funcionalidad generar simulación, el sistema genera aleatoriamente, en base a los parámetros por defecto o los que ha definido el usuario, una cuadrícula llena de entidades, donde cada una tiene un estado definido.
Actor	Usuario
Relaciones	
Precondición	

Tabla 13. Caso de uso Generar Simulación

.

2.3 Plan de trabajo

En este punto, se representa mediante un Diagrama de Gantt todo el conjunto de tareas que se van a realizar dentro del plan y la metodología de trabajo que se ha decidido usar, la cual, abarca un total de cuatro meses, y que como se menciona anteriormente, es la cascada, comprendiendo las etapas pertenecientes a esta.

El primer gran bloque corresponde a la investigación y análisis sobre el trabajo que se realiza. Además de formar parte de este bloque algunas tareas como la realización de entrevistas a expertos o casos de uso, también se incluye un estudio sobre el contexto, importancia y funcionamiento del juego de la vida, concepto en el que se basa el proyecto.

En el siguiente bloque, diseño, se realizan tareas como la realización de mockups o la elección de las tecnologías que se utilizan en el proyecto, en base a la información que se ha recopilado procedente del bloque anterior.

El tercer bloque es el de implementación, el más grande debido a que tiene una duración total de 55 días. Es la parte del proyecto en la que se comienza tanto a instalar todo el entorno necesario, como a crear el proyecto y comenzar a desarrollar todo el código con el objetivo de conseguir que la aplicación incorpore todas las funcionalidades que se han visto en los bloques anteriores.

A lo largo de la elaboración del trabajo, se ha realizado en paralelo, junto al resto de las tareas anteriores, la redacción de la memoria, teniendo como fecha de inicio y final el primer y último día de la realización del proyecto.

Finalmente, se encuentra el bloque de pruebas, en el cual se programan todas las pruebas necesarias para poder validar correctamente los requisitos funcionales de la aplicación.

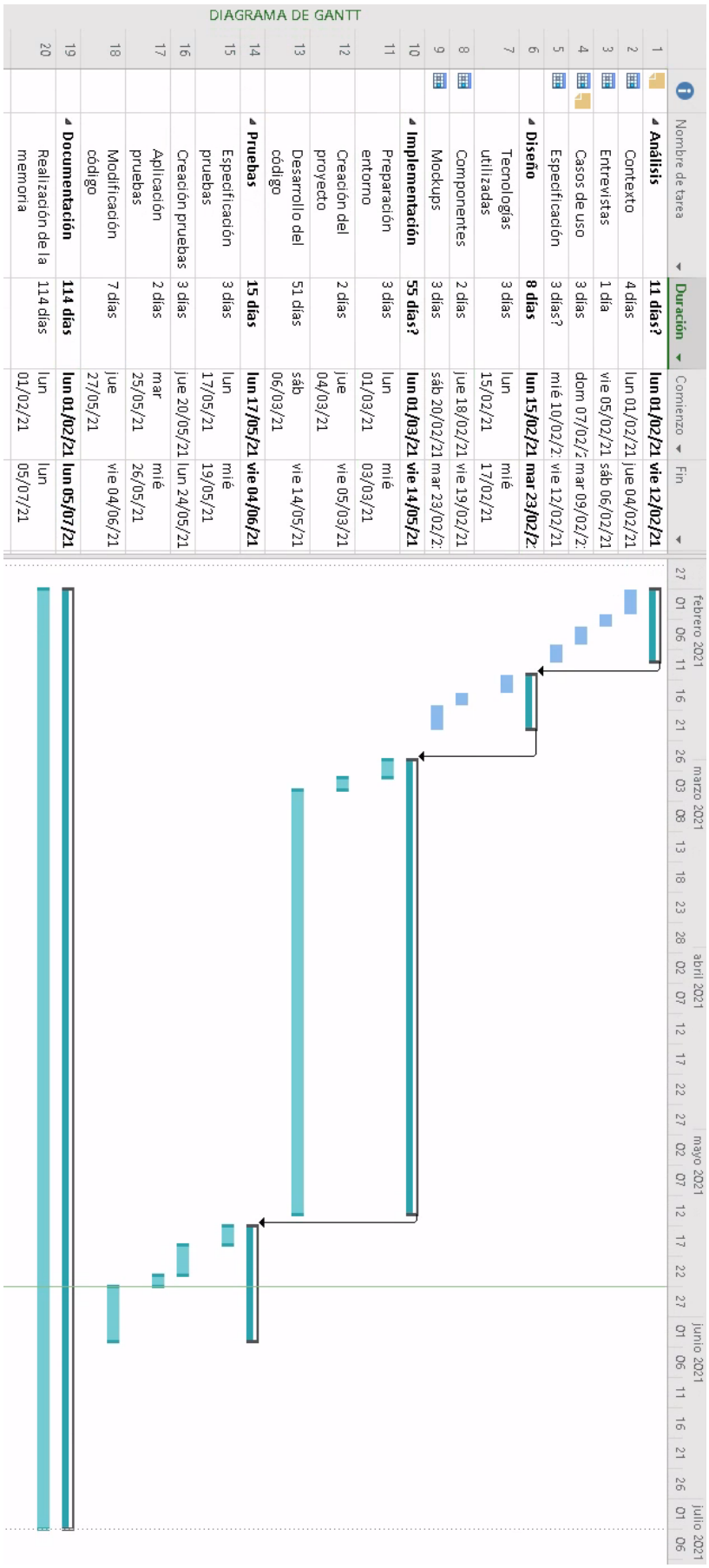


Ilustración 6. Diagrama de Gantt

3. Diseño de la solución

Una vez realizado previamente todo el análisis del problema en el apartado anterior, llega la hora del diseño de la solución. En este apartado se decide la solución que se va a dar, y a raíz de esta, se elige la estructura de la aplicación que se va a desarrollar, además de las tecnologías que se usan. También, se han realizado distintos mockups.

3.1 Solución propuesta

El trabajo plantea la realización de una aplicación que trate de simular la propagación de enfermedades, usando el concepto del juego de la vida de Conway. Por ello, la simulación debe de tener lugar dentro de una cuadrícula, además de contar con una secuencia de generaciones, mediante las cuales, las entidades del sistema alcanzan su siguiente estado.

Para poder cumplir con las necesidades anteriores, se ha decidido que la mejor solución para ello es desarrollar la aplicación como una página web, y por lo tanto usando JavaScript, con soporte de una librería llamada React.

Se ha elegido React por distintas razones, donde una de ellas es que gracias a esta podemos crear componentes reutilizables que son capaces de recibir datos de entrada mediante this.props. El hecho de que sea reutilizable sirve por ejemplo para crear todos los cuadrados de la cuadrícula de la simulación con apenas código.

Otra de las razones por la que se ha decidido usar React ha sido porque por cada componente que se crea implementa una función llamada **render()**, encargada de renderizar visualmente al componente. Lo curioso es que este método se ejecuta cada vez que se actualicen los valores pertenecientes al estado. Por lo tanto, se puede declarar un valor dentro del estado para que después de generación se haga un renderizado del nuevo estado de nuestro sistema.

3.2 Componentes

Un diagrama de clases sirve para representar de manera estática la estructura de una aplicación, mostrando desde sus atributos y funciones, hasta las relaciones y clases.

No obstante, como se ha mencionado en el anterior punto, en este caso, la aplicación está desarrollada en React, una librería de JavaScript, donde la mayor parte de las aplicaciones creadas con esta hacen uso de componentes. Es por eso por lo que si se llega a tratar de representar un diagrama de clases para una aplicación desarrollada en React puede llegar a resultar bastante compleja y caótica debido a la cantidad de estados o variables de entradas que cada componente posee.

Por ello, tal y como se observa en la siguiente imagen, se ha tratado de crear mediante lenguaje UML una representación del conjunto de componentes que conforman la aplicación con sus respectivas relaciones.

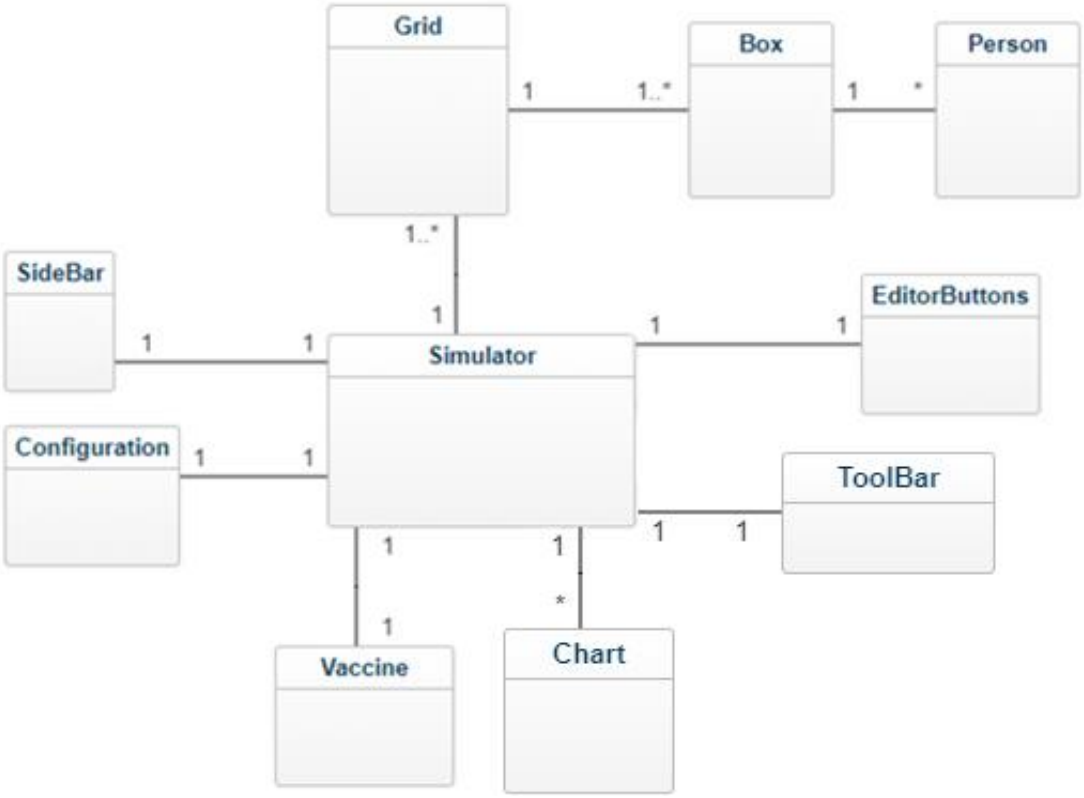


Ilustración 7. Conjunto de componentes que forman la aplicación

A continuación, se explican los componentes que forman el diagrama de la imagen anterior:

- **Vaccine**: componente formado por una barra de progreso que representa el porcentaje de completitud de la vacuna. Una vez se llegue al 100%, se comenzará a vacunar a las entidades.

- **Graphics**: componente gracias al cual se crean las gráficas a partir de dos variables de entrada, una para cada eje, para así poder representar visualmente los datos.

- **SideBar**: componente que consiste en una barra lateral web que sirve para poder visualizar los datos de un componente del **Grid** que haya sido seleccionado. También se pueden establecer los parámetros mediante los cuales se va a añadir una nueva componente de las dichas anteriormente.

- **Box**: componente representado como un cuadrado y que actúa como recinto para las distintas entidades. Tendrá asociado un atributo que represente la aireación, para así poder determinar la velocidad con la que se desinfecte el lugar en caso de que este infectado.

- **Grid**: componente que hace uso del componente Box para formar la cuadrícula del simulador a partir de dos valores de entrada, **rows** y **columns**, que representan el ancho y alto respectivamente.

- **Person**: entidades que recorren el **Grid** a lo largo de la ejecución del simulador y varían su estado a lo largo de las generaciones.

- **Configuration**: componente formado por un botón, el cual, al clicar en este se abre un modal desde el que definir los nuevos parámetros de entrada de las ejecuciones del simulador.

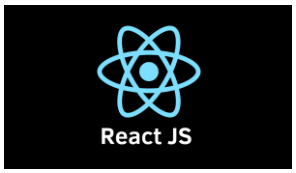
- **EditorButton**: componente que consiste en un conjunto de botones cuya función al clicarlos corresponde a manipular la ejecución del simulador, como puede ser pararlo, ejecutarlo, pausarlo, aumentar su velocidad…

- **ToolBar**: componente formado por un grupo de botones cuyo funcionamiento al clicar suele estar orientado a modificar la cuadrícula de nuestro simulador, siendo por ejemplo añadiendo o eliminando entidades.

- **Simulator**: componente principal conectado con el resto de los componentes, y que, además, es el encargado de ejecutar en cada generación las reglas definidas para decidir en nuevo estado de cada entidad.

3.3 Tecnologías empleadas

En el apartado actual se presentan las distintas tecnologías que se han utilizado a lo largo de la realización el proyecto. No solo se encuentran las utilizadas a la hora de programar la aplicación, sino también las necesarias para realizar la memoria.



ReactJS es una biblioteca de *front-end* de JavaScript orientada a aplicaciones Web, que permite facilitar la creación de interfaces de usuarios gracias a la creación de componentes que ofrece entre otras.

JavaScript es el lenguaje de programación que se ha decidido utilizar para el desarrollo de la aplicación, junto con **HTML** y **CSS**. Utilizando estos tres lenguajes junto con **ReactJS** permite programar y modificar todos los aspectos de una aplicación Web con un gran nivel de detalle y facilidad.

Se ha utilizado **GitHub**, para facilitar la gestión de los repositorios remotos, de tal forma que el proyecto tenga control de versiones, ofreciendo así la posibilidad de incorporar nuevas funcionalidades a la aplicación sin el riesgo de perder información.

El entorno de programación escogido es **Visual Studio Code**. El motivo principal por el que se ha elegido es debido a que se dispone de experiencia previa con este, además de que es gratuito y ofrece características como la instalación de *plugin* o la incorporación de terminales propios.

Photoshop ha sido utilizado para la creación y desarrollo de las versiones finales de las imágenes que la aplicación hace uso, tal y como pueden ser los distintos iconos de los botones. También se ha usado para retocar o realizar algún *mockup*.

NodeJS es una herramienta de *runtime* para JavaScript, que permite diseñar y construir aplicaciones escalables, conocido por ser *asíncron*. Estos atributos han permitido crear una aplicación con un tiempo de respuesta muy rápido y capaz de servir muchas peticiones a la vez.

Gracias a la herramienta **balsamiq**, se pueden realizar *mockups* de gran calidad para poder tener una mejor imagen visual de cómo van a ser las distintas interfaces que van a componer la aplicación que se va a desarrollar.

Word es la herramienta escogida para redactar la memoria del proyecto. El motivo de su elección es sencillo, la Universidad Politécnica de Valencia ofrece de manera gratuita la aplicación.

Gracias al acceso de las máquinas virtuales mediante Polilabs es posible usar gratuitamente **Microsoft Project 2019**, el cual ha servido para poder realizar los diagramas de Gantt.

3.4 Estructura de datos

La aplicación necesita una estructura de datos en la que poder ir guardando toda la información que ocurre a lo largo de la ejecución de una simulación, para luego, poder representarla. Información que puede ser interesante de almacenar puede ser algo tan simple como la posición de las entidades o la cantidad de infectados en una generación.

Se consideraron varias opciones, donde una de ellas era la creación de una base de datos. Sin embargo, esta idea se desechó casi inmediatamente, debido en parte a que no se quiere que estos datos persistan, es decir, que una vez se recargue la página web los datos previos a la recarga no deben mostrarse de nuevo, además de que ocurriría sobrecarga debido al gran número de peticiones que se tendrían que realizar

Es por eso, que la mejor opción resultó ser realizar una estructura que almacenara los datos en memoria mediante el uso de variables en un formato JSON. Gracias a esto, se puede definir una variable para cada estructura de datos que se actualice a lo largo de una ejecución, un ejemplo sería un array que contiene en cada posición un JSON que representa toda la información relacionada con cada recinto, representando información como su posición o si se encuentra infectado o no.

A continuación, se muestra una estructura que representa el estado de un recinto de la cuadrícula.

```
placesInfo={"solid": false, "infected": false, "timeInfected" : 0, 'air
eation' : 1, 'selected' : false}
```

3.5 Mockups

En base a las funcionalidades que se han visto anteriormente, y las condiciones que necesita la aplicación para poder estar basada en el juego de la vida de Conway, se ha debido encontrar la forma de distribuir los elementos de la mejor manera posible. Para ello, gracias a la herramienta **balsamiq** se ha realizado un conjunto de *mockups* que representan la visión final de las interfaces de usuario del simulador que se propone desarrollar.

El primer *mockup* que se muestra corresponde a la vista general de la aplicación, en concreto, un momento en el que hay una simulación en curso. Se ha optado por esta manera de organizar los elementos porque se ha determinado que era la óptima.

Como se puede observar se disponen de tres cuadrículas, siendo la más grande la principal, en la cual se encuentran las distintas entidades del simulador, es decir, recintos, siendo cuadrados grises, o las personas, representándose con círculos de colores. Cada cuadrícula situada debajo de la principal tiene como objetivo albergar un tipo de entidad determinada, siendo para una si la persona ha muerto o no (actuando por lo tanto como un “cementerio”), y la otra para aislar personas infectadas por la enfermedad hasta que estas se curen.

También se observan los elementos que se encuentran encima de la cuadrícula principal, siendo estas, aparte del número de generación actual, la barra de progreso que representa el porcentaje de completitud de la vacuna, o las distintas disposiciones de botones que realizan muchas de las funcionalidades especificadas en los apartados anteriores.

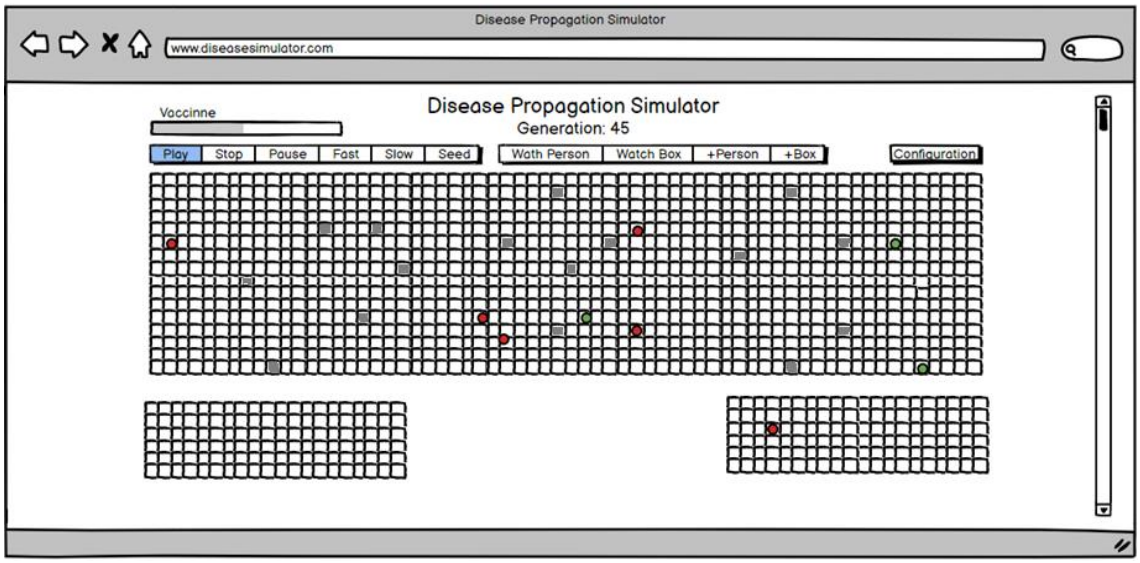


Ilustración 8. Mockup de la ventana general de la aplicación

Sin embargo, hay una funcionalidad que hay que definir el cómo mostrarla, y esta es la que corresponde con el caso de uso cuyo identificador es **CA-11, Cambiar Parámetros**. Necesita un *mockup* debido a que se debe especificar donde y como se van a modificar los parámetros que usa la aplicación durante la ejecución de la simulación. En este caso, se ha decidido que al clicar en un botón se abra un modal desde el cual el usuario sea capaz de ver los valores actuales del simulador, y, si así lo desea, cambiar cada variable según sus necesidades o para que se adecue a la futura simulación a realizar.

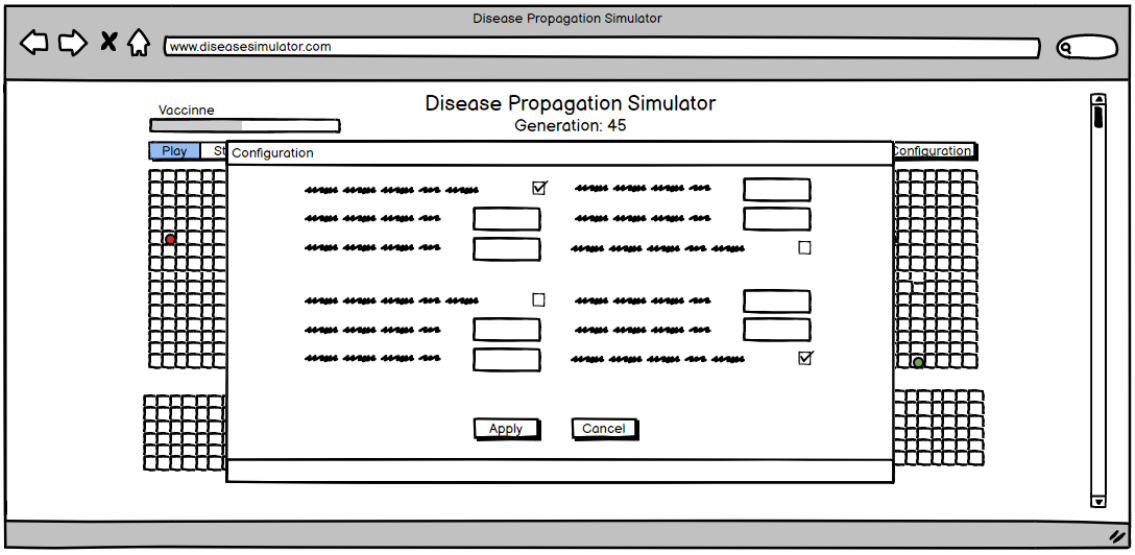


Ilustración 9. Mockup del modal de configuración

Si en la ventana principal, se desplaza la página hacia abajo con la rueda del ratón, se pueden ver como se generan las distintas gráficas a partir de los datos de los que dispone la simulación.

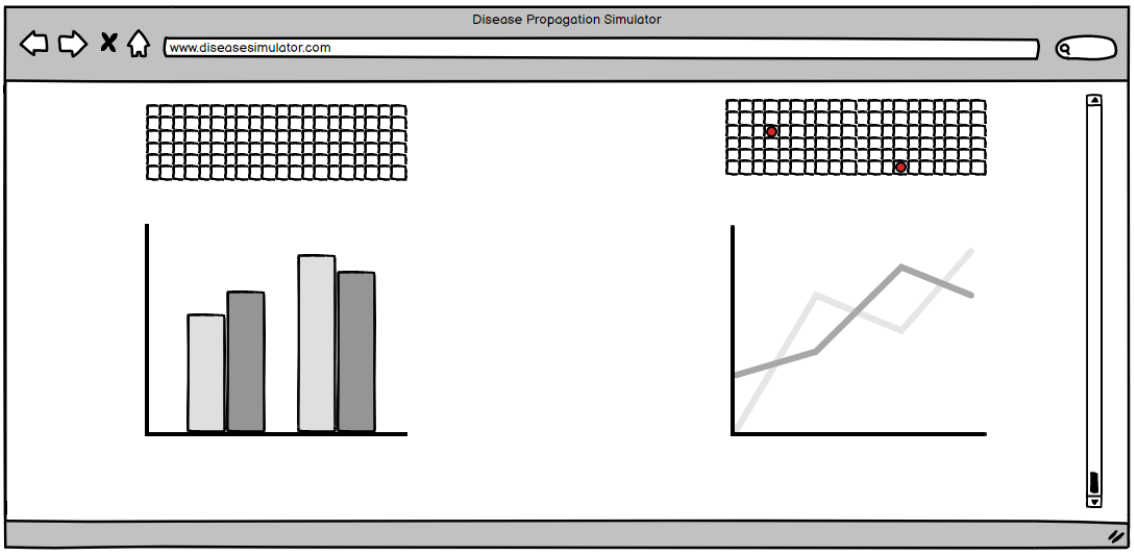


Ilustración 10. Mockup de las gráficas

Para finalizar, en la imagen se observa cómo al presionar un botón de la segunda agrupación de botones, se expande una barra lateral, la cual, dependiendo de la herramienta seleccionada muestra información correspondiente a la entidad que se haya seleccionado o, por otro lado, el usuario debe proporcionar los datos para añadir una nueva entidad a la cuadrícula principal.

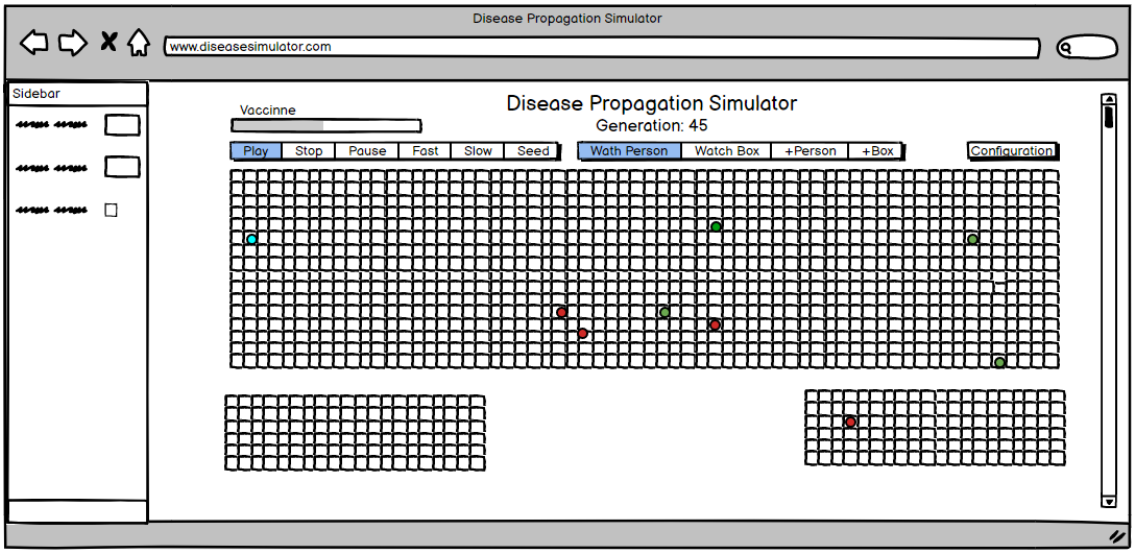


Ilustración 11. Mockup de la barra lateral

4. Desarrollo de la solución

Al finalizar la etapa de diseño, la siguiente es la de desarrollo, en la cual se comienza a desarrollar la aplicación que se ha planteado desde el comienzo del trabajo. Sin embargo, antes de comenzar a escribir código, y empezar a implementar las funcionalidades vistas en los puntos anteriores, es necesario preparar el entorno de trabajo.

Lo primero es instalar **Visual Studio Code**, el entorno de desarrollo elegido para desarrollar la solución, y una vez ya instalado, es posible comenzar a crear el proyecto. Para ello, como se ha visto, entre las tecnologías mencionadas que se van a utilizar, se encuentra **node.js**, la cual, aparte de permitir tener un entorno *runtime* para ejecutar nuestro código JavaScript en un servidor, también ofrece el administrador de paquetes llamado *npm*. El administrador de paquetes permite tanto publicar como instalar distintas librerías que distintos programadores hayan compartido a partir de Node.js, por lo tanto, esto permite una manipulación de estas muy cómoda, pudiendo actualizar, eliminar o añadir a las mismas con escasas líneas de comando.

A continuación, se dispone a crear el proyecto, donde para ello, tal y como se ve en la siguiente imagen, hay que abrir un terminal y ejecutar los comandos que se muestran. El primero de ellos, realiza la instalación de todo lo necesario para crear el proyecto. Al finalizar la instalación anterior, se ejecuta la siguiente línea de comandos de la imagen, la cual crea el proyecto, donde **disease_simulator** es el nombre que se ha decidido.

```
PS C:\Users\theza> npm install -g create-react-app
C:\Users\theza\AppData\Roaming\npm\create-react-app -> C:\Users\theza\AppData\Roaming\npm\node_modules\create-react-app\index.js
+ create-react-app@4.0.3
updated 1 package in 2.228s
PS C:\Users\theza> npx create-react-app disease_simulator
```

Ilustración 12. Instalación y creación del proyecto

La creación del proyecto puede tardar unos minutos, pero una vez haya finalizado, está todo listo para comenzar a desarrollar. Para lanzar una ejecución de la aplicación web, se necesita acceder al directorio en el que se encuentra el proyecto y ejecutar el comando **npm start**, el cual, levantará una instancia de la aplicación en http://localhost:3000/. Si todo ha salido bien se obtiene un resultado simular a la siguiente imagen.

```
Compiled successfully!

You can now view disease_simulator in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://192.168.0.162:3000

Note that the development build is not optimized.
To create a production build, use npm run build.
```

Ilustración 13. Compilación y ejecución exitosa

Como se mencionó anteriormente, durante el desarrollo de este trabajo se han hecho uso de distintas librerías que ofrece node, siendo una de estas, **React.js**. Con simplemente ejecutar en un terminal el comando **npm install <nombre del paquete>**, se instala la última versión disponible del paquete.

En el directorio donde se ha creado el proyecto, se puede observar un fichero llamado **package.json,** resultando ser este, el lugar donde se declaran todas las librerías con sus respectivas versiones que la aplicación hace uso. Cabe mencionar que se declaran, no instalan, haciendo posible que con realizar el comando **npm install**, se instalen todas las librerías que se encuentran en el **package.json** sin necesidad de tener que ir una por una.

La siguiente imagen corresponde al **package.json** del proyecto que se ha desarrollado, mostrando concretamente la parte de las dependencias, es decir, todas las librerías de las que se hace uso, y por lo tanto, necesita para su correcto funcionamiento.

Ilustración 14. Dependencias del package.json del proyecto

No se han usado muchas librerías, de hecho, se pueden resumir básicamente en dos bloques, siendo uno de ellos donde se encuentran todas las dependencias relacionadas con **React.js**, tal como **react-modal** o **react-bootstrap**, librerías que, si bien cada una presenta funcionalidades distintas (como es el caso de la primera, que ofrece la posibilidad de abrir un modal), todas ellas están creadas de tal forma que se usen en una aplicación que haga uso del *framework* de **React.js**. Por otra parte, se encuentran todas las librerías que no están creadas expresamente para que funcionen en conjunto con React.js, como es el caso de **victory**, una librería que ha hecho posible la generación de gráficas a partir de los datos de la simulación.

Lo siguiente que se muestra es una vista de la aplicación final, concretamente lo que el usuario encuentra nada más acceder a la web. Como se observa, se han respetado bastante los mockups mostrados anteriormente y solo se han realizado cambios mínimos con el objetivo de obtener un diseño mucho más eficiente e intuitivo, como es por ejemplo la adición de imágenes o el cambio en la disposición de los botones.

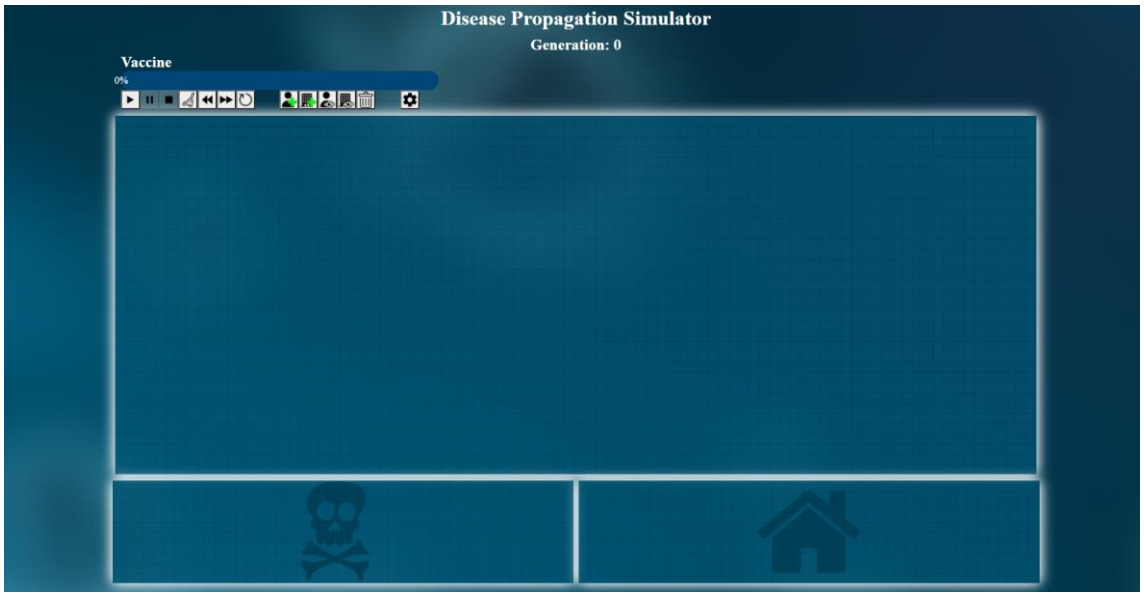


Ilustración 15. Vista principal de la aplicación web

Nada más se accede, no hay ninguna simulación en marcha, de hecho, no hay entidades. Sin embargo, no es obligatorio que haya entidades en la cuadrícula para iniciar una simulación, ya que esta puede comenzar inicialmente vacía, y conforme vayan pasando las generaciones ir añadiéndolas gracias a las herramientas que se han implementado.

También se observa cómo hay tres cuadrículas. Se refiere a ellas como cuadrícula general, de muerte y de aislamiento.

Si se clica cualquiera de los cuatro primeros botones pertenecientes al segundo conjunto agrupado de estos, se abre una barra lateral, desde la cual, dependiendo de la herramienta que se haya seleccionado se muestra de una manera u otra. Por ejemplo, si se selecciona una herramienta cuya función sea observar una entidad, al clicar en una, en la barra lateral mostrará los datos pertenecientes a esta. Si, por el contrario, se seleccionan herramientas con la funcionalidad de añadir entidades, se abre también la barra lateral, pero con la diferencia de que se dispone de *inputs* con el fin de configurar los atributos de una entidad antes de añadirla a la cuadrícula cuando se clique en esta.

Se observa en la siguiente ilustración, el conjunto de apariencias que puede tener la barra lateral, siendo un total de cuatro.

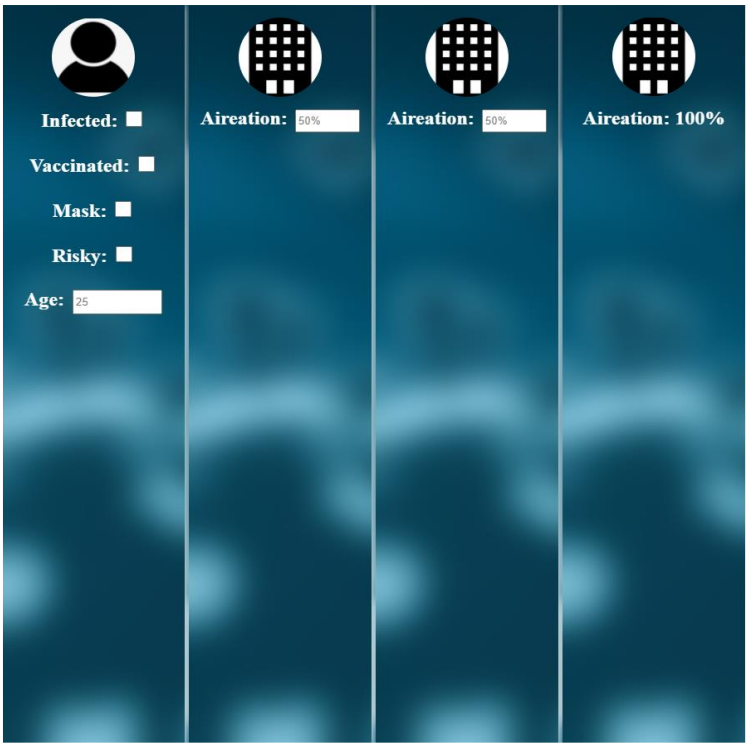


Ilustración 16. Todas las apariencias de la barra lateral

Se debe mencionar que, tal y como se definió en el análisis del problema, antes de iniciar la simulación se tiene la posibilidad de configurar los parámetros de la simulación que se va a ejecutar. Para acceder a esta ventana emergente, el usuario tiene que pulsar el botón que se observa en la vista principal, cuyo icono corresponde al de un engranaje. Una vez abierta, podremos configurar cada variable por separado y confirmar los cambios, o bien, descartando a los mismos pulsando fuera del modal o pulsando en el botón **"Confirmar"** sin haber rellenado ningún *input*.

Ilustración 17. Modal desde donde definir los parámetros de la simulación

Si se añaden entidades al simulador desde un inicio, ya sea bien por medio de las herramientas o generando un estado inicial al azar gracias al botón con forma de flecha, en la ejecución de la simulación que se ejecuta podemos observar cómo cada entidad tiene una evolución propia a lo largo del avance de las generaciones, gracias a las reglas y conjuntos de estados que se han declarado.

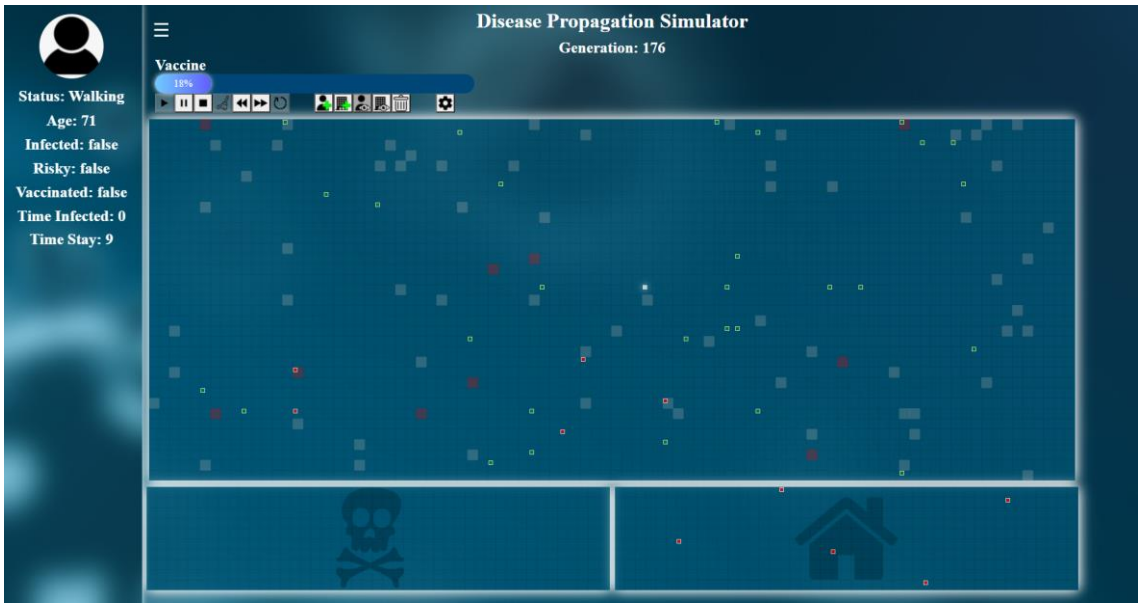


Ilustración 18. Simulación en marcha

Con tal de integrar el funcionamiento deseado en el simulador, se ha tenido que definir **this.state**, es decir, las variables del programa, las cuales, al cambiar su valor realizan una actualización del método **render()**, encargado de mostrar mediante **HTML** y **CSS** como se va a ver el programa. Además, dentro del **this.state** también se encuentran las variables que actúan como estructuras de datos, tales como **gridSolid** o **gridIsolated**, las cuales almacenan información del estado de las cuadrículas en un formato JSON. El **this.state** que se ha definido lo siguiente:

- **toolSelected**: nombre de la herramienta que está seleccionada actualmente.
- **generation**: número de la generación actual de la simulación.
- **gridPeople**: representa el conjunto de personas que hay actualmente en la cuadrícula general, donde cada una de ellas tiene sus atributos, siendo estos, datos de interés como puede ser su localización en la generación actual.
- **gridSolid**: representa el conjunto recintos que hay actualmente en el simulador, definiendo datos para cada uno de ellos y almacenándolos en un **JSON**.
- **gridIsolated:** representa el conjunto de personas que están aisladas en la generación actual, y, por lo tanto, todos los elementos de este **grid** pertenecen a la cuadrícula de aislamiento.

- **gridDeath:** representa el conjunto de personas que han muerto durante la simulación, y, por lo tanto, todos los elementos de este **grid** pertenecen a la cuadrícula de muerte.

- **gridInfAux:** representa lo mismo que **gridSolid**, solo que este se ha creado específicamente para que no contenga ningún recinto y no se realice ningún cambio sobre estos, para así, asignárselo a las cuadrículas de muerte y aislamiento.

```
this.state = {
    toolSelected:'none',
    generation: 0,
    gridPeople: Array(this.rows).fill().map(() => Array(this.cols).fill([])),
    gridDeath: Array(10).fill().map(() => Array(45).fill([])),
    gridIsolated: Array(10).fill().map(() => Array(45).fill([])),
    gridSolid: Array(this.rows).fill().map(() => Array(this.cols).fill({"solid": false,
        "infected" : false, "timeInfected" : 0, 'aireation' : 1, 'selected' : false})),
    gridInfAux: Array(10).fill().map(() => Array(45).fill({"solid": false, "infected" : false,
        "timeInfected" : 0, 'aireation' : 1, 'selected' : false})),
}
```

Ilustración 19. this.state del main

En la imagen se observa como la entidad cuadrado tiene atributos como **solid** (define si es un recinto o no), **selected** (si esta seleccionado por alguna herramienta), **infected** (si está infectado o no), **timeInfected** (número de generaciones que va a estar infectado), y **aireation** (porcentaje de aireación). Sin embargo, la entidad más importante del simulador, y la cual ha sido mencionada anteriormente, es la que representa a las personas. Se ha decidido definir a estas a través de una clase con los siguientes atributos, además de definir obviamente los **getters** y **setters** para estos:

- **selected**: informa si la persona está siendo seleccionada actualmente o no por alguna herramienta.

- **status**: representa el estado en el que se encuentra la persona actualmente, pudiendo ser muerta, aislada, caminando o hablando.

- **timeStatus, timeInfected y timeVaccinated**: para el primer caso representa cuántas generaciones lleva la persona en un mismo estado, mientras que para el segundo y tercero informa cuanto lleva la persona infectada o vacunada respectivamente.

- **risky**: informa si es una persona de riesgo.

- **age:** edad de la persona.

- **vaccinated**: informa si una persona esta vacunada o no.

- **mask**: informa si una persona lleva mascarilla o no.
- **timeStay**: número de generaciones que la persona tiene intención de estar en el lugar que tiene como destino.
- **walkingTime:** número de generaciones que lleva la persona caminando.
- **actualX y actualY:** representa el conjunto de coordenadas en el que se encuentra la persona en la generación actual, siendo el eje X e Y respectivamente.
- **destinyX y destinyY:** representa el conjunto de coordenadas hacía donde tiene como objetivo la persona llegar, siendo el eje X e Y respectivamente.

```
class Person {
    constructor(selected, status, timeStatus, timeInfected, timeVaccinated,
        risky, age, vaccinated, mask, infected, timeStay, walkingTime, actualX,
        actualY, destinyX, destinyY){

        this.timeStay = timeStay;
        this.status=status
        this.timeStatus=timeStatus
        this.timeVaccinated = timeVaccinated;
        this.timeInfected = timeInfected;
        this.selected = selected;
        this.actualX = actualX;
        this.actualY = actualY;
        this.destinyX = destinyX;
        this.destinyY = destinyY;
        this.infected = infected;
        this.mask = mask;
        this.vaccinated = vaccinated;
        this.age = age;
        this.risky = risky;
        this.walkingTime = walkingTime;
    }
```

Ilustración 20. Atributos de la clase Person

Como se ha visto anteriormente, el juego de la vida de Conway hace uso de reglas para que cada elemento del simulador avance hacia su siguiente estado. Es por eso, que al querer que el simulador se base en el juego de la vida, se necesita encontrar alguna forma de adaptar o implementar el mismo principio.

Se ha decidido que una vez se pulse el botón de **“Play”** se crea un intervalo en el que se va a ejecutar una función que se ha implementado, **nextGeneration().** La rapidez con la que se ejecuta cada intervalo viene definida por la variable **speed** de **this.state**. Una vez creado el intervalo, se puede pausar, eliminar o variar la velocidad de este gracias al resto de botones que se disponen.

```
nextgeneration = () => {
    let gridSolidCopy = arrayClone(this.state.gridSolid);
    let gridIsolatedCopy = arrayClone(Array(10).fill().map(() => Array(45).fill([])));
    let gridDeathCopy = arrayClone(Array(10).fill().map(() => Array(45).fill([])));
    let gridPersonCopy = arrayClone(Array(this.rows).fill().map(() => Array(this.cols).fill([])));
    this.movePerson(gridPersonCopy);
    this.movePersonOut(gridDeathCopy, gridIsolatedCopy);
    this.infectPlace(gridSolidCopy,gridPersonCopy);
    this.downTimeBox(gridSolidCopy);
    this.infectedByPlace(gridSolidCopy,gridPersonCopy);
    this.timeInfectedDown(gridPersonCopy, gridIsolatedCopy);
    this.timeVaccinatedDown(gridPersonCopy, gridIsolatedCopy);
    this.speakPeople(gridPersonCopy);
    this.isolatePeople(gridPersonCopy, gridIsolatedCopy);
    this.killPeople(gridPersonCopy, gridDeathCopy, gridIsolatedCopy);
    this.dataInfecteds.push({generation:this.state.generation, infecteds: this.numberInfecteds})
    this.dataDeaths.push({generation:this.state.generation, deaths: this.numberDeaths})
    this.setState({
        generation: this.state.generation + 1,
        gridPeople : gridPersonCopy,
        gridSolid : gridSolidCopy,
        gridDeath : gridDeathCopy,
        gridIsolated : gridIsolatedCopy
    });

}
```

Ilustración 21.Reglas que se ejecutan en cada generación

Se comenta a continuación todo lo que ocurre dentro de la función **nextGeneration()**, la cual, contiene el conjunto de funciones que actúan como reglas del simulador.

Lo primero que se realiza es una copia de cada cuadrícula que compone nuestra simulación, concretamente de la generación actual. Esto se hace para no manipular directamente los datos que están siendo utilizados en el momento por el simulador y así no causar inconsistencias. Una vez ya creadas las copias, se habla de las reglas en sí que se han definido.

En primer lugar se encuentran **movePerson() y movePersonOut(),** donde las dos realizan la misma funcionalidad, solo que la primera calcula en base a las entidades de la cuadrícula principal y la segunda en el resto. Básicamente lo que ocurre, es que, para cada persona de la simulación, siempre y cuando se encuentre en el estado **“Walking”**, se decide cual va a ser el siguiente cuadrado de la cuadrícula al cual se va a mover en la siguiente generación, con tal de que la distancia entre las nuevas coordenadas y el destino al que se desea llegar sea cada vez menor hasta que al fin llega al mismo.

Si se eligiera siempre la ruta más corta, las personas entre generación y generación se moverían en línea recta o en diagonal. Para evitar esto, se ha añadido la probabilidad de que en base a la mejor posición a la que moverse, se varíe en un valor de uno, el eje y, x o ambos. También evitan entrar en recintos siempre y cuando no sea necesario.

La siguiente función es **infectPlace()**. Cada recinto de nuestra cuadrícula, dependiendo del número y el estado de las personas que contengan, puede llegar a infectarse durante un tiempo determinado, provocando que las futuras personas que lleguen a este tengan peligro de infectarse.

Al poder infectar los recintos, hay que encontrar alguna manera de desinfectarlos, y aquí es donde entra en juego **downTimeBox()**. Como vimos, cada entidad cuadrado está definida por un atributo llamado **aireation** que representa como de bien ventilado está un lugar. Por lo tanto, en base a este atributo, un recinto se desinfecta más rápido para un valor alto, o más lento con un valor más bajo.

También se pueden infectar las personas, y una opción de conseguirlo es a través de **infectedByPlace()**. Lo que ocurre en esta, es que en cada generación se comprueba el interior de cada recinto, donde gracias a variables como si el recinto está infectado o el número y estado de personas que contiene, se calcula si cada persona del interior que no esté infectada lo estará en la siguiente generación.

Otra manera de infectar a la entidad persona es a través de **speakPeople()**. Mientras las personas del simulador se estén moviendo a lo largo de la cuadrícula, es probable que en un mismo cuadrado se encuentren más de una persona. Si ocurre esto, hay una posibilidad de que las personas establezcan una conversación, y por lo tanto, si alguna de estas está infectada, pueda llegar a infectar al resto de personas con las que habla.

Después, tenemos funciones como **timeInfectedDown()** o **timeVaccinatedDown(),** cuya función no es más que reducir en cada generación, el tiempo que una persona está infectada o vacunada respectivamente.

Para determinar cómo y porque acaban las personas en otras cuadrículas que no sean la general, también es necesario el uso de funciones.

Con la función **isolatePeople()**, en cada generación hay una probabilidad de que si una persona está infectada, esta acabe en la cuadrícula de aislamiento hasta que esté curada.

La función **killPeople()**, hace que en cada generación cada persona tenga una probabilidad de morir, y por lo tanto acabar en la cuadrícula de muerte. No todas las personas tienen la misma probabilidad de morir, ya que, los atributos que estas tienen afectan a este porcentaje, como puede ser la edad, si es una persona de riesgo, si está vacunada…

Las siguientes dos líneas de código tienen como función guardar los valores de las muertes e infectados de la generación actual para poder representar dichos datos a través de las gráficas.

Por último, se actualiza el estado de cada cuadrícula con el nuevo calculado en las funciones anteriores, y se suma en uno el número de generación, provocando que se ejecute el método render() y, por lo tanto, realice una actualización de vista de la aplicación.

Sin embargo, hay una función más que mencionar, y no se ha hecho antes, puesto que esta no comienza a ejecutarse hasta que se cumpla cierta condición. La función es **vaccinate()**, cuya función no sirve más que para vacunar a las personas del simulador y así volverlas inmunes a la enfermedad.

El orden en el que se aplica la vacuna no es al azar, sino que se sigue un orden específico. Primero se vacunan a las personas mayores de 80 años, luego a las que se encuentran entre 65 y 80 años, después a las personas de riesgo, y finalmente al resto de personas. Cabe mencionar que la vacuna no se va a aplicar al siguiente grupo hasta que las personas del grupo que precede a esta no estén todas vacunadas.

```
vaccinate = () => {
    let g2 = arrayClone(this.state.gridPeople);
    for (var i = 0; i < g2.length; i++) {
        for (var j = 0; j < g2[i].length; j++) {
            var personArr = this.state.gridPeople[i][j];
            if(personArr.length!==0){
                for(var z = 0; z < personArr.length; z++){
                    if(!this.isVaccinated(1) && personArr[z].getAge()>=80 && !personArr[z].getVaccinated()){
                        personArr[z].setVaccinated(Math.random()<this.successVaccine/100);
                        if(personArr[z].getVaccinated()){personArr[z].setTimeVaccinated(this.timeVaccinated)};
                    }

                    else if(this.isVaccinated(1) && !this.isVaccinated(2) && personArr[z].getAge()>=65
                    && personArr[z].getAge()<80 && !personArr[z].getVaccinated()){
                        if(!personArr[z].getVaccinated()){
                            personArr[z].setVaccinated(Math.random()<this.successVaccine/100);
                            if(personArr[z].getVaccinated()){personArr[z].setTimeVaccinated(this.timeVaccinated)}}
                    }

                    else if(this.isVaccinated(1) && this.isVaccinated(2) && !this.isVaccinated(3)
                    && personArr[z].getRisky() && !personArr[z].getVaccinated()){
                        if(!personArr[z].getVaccinated()){
                            personArr[z].setVaccinated(Math.random()<this.successVaccine/100);
                            if(personArr[z].getVaccinated()){personArr[z].setTimeVaccinated(this.timeVaccinated)}}
                    }

                    else if(this.isVaccinated(1) && this.isVaccinated(2) && this.isVaccinated(3)){
                        if(!personArr[z].getVaccinated()){
                            personArr[z].setVaccinated(Math.random()<this.successVaccine);
                            if(personArr[z].getVaccinated()){personArr[z].setTimeVaccinated(this.timeVaccinated)}}
                    }

                }
            }
        }
    }
}
```

Ilustración 22. Función vaccinate()

La razón por la que esta función no aparece en **nextGeneration()**, es debido a que hasta que el componente que representa la barra de progreso de la vacuna no llegue al 100%, no se comienza a aplicar la vacuna a las personas. Por tanto, al haber una condición en el componente **ProgressBar** comprobándose cada generación, hasta que en algún momento se cumpla, es lógico que uno de los parámetros (en este caso, **isComplete**) que se pasan al componente sea la función que realiza cuando cumple la condición.

```
class ProgressBar extends React.Component {

    constructor() {
        super();
    }

    componentDidUpdate(){
        if(this.props.porcentaje>=100 && this.props.porcentaje%10===0)
        this.props.isComplete();
    }

    render(){
    return (
        <div>
            <h2 style={{textAlign: 'left', margin:0 }}>{this.props.text}</h2>

        <div className="progress">
            <div className="progress-done" style={{width: this.props.porcentaje<100 ? this.props.porcentaje +"%": "100%"}}>
                {this.props.porcentaje<100
                ? Math.round(this.props.porcentaje) + '%'
                : "100%"}
            </div>
        </div>
        </div>
    )
    }
}
```

Ilustración 23. Componente ProgressBar

Finalmente, queda mencionar como, mientras se ejecuta una simulación, se generan simultáneamente dos gráficas que representan la cantidad de infectados y muertes con relación al número de generaciones. Todo esto se consigue pasando al componente que representa la gráfica un *array* para cada eje que representa el conjunto de datos.

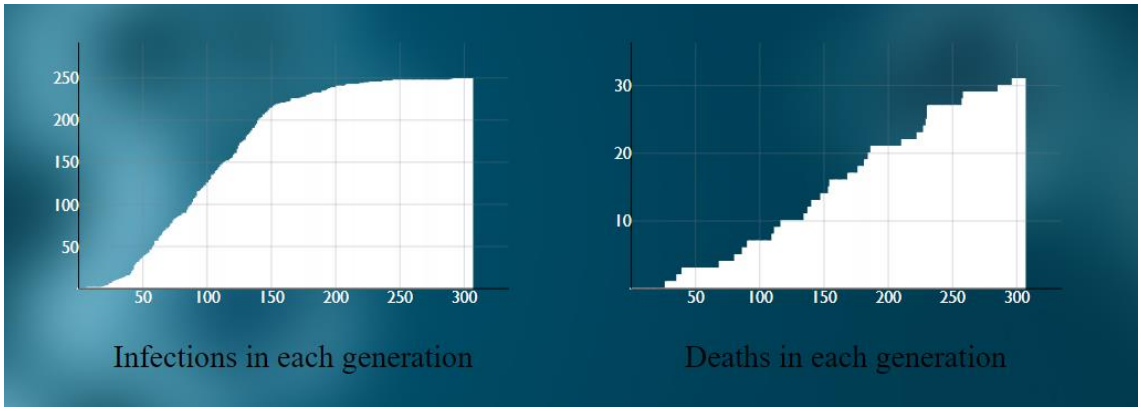


Ilustración 24. Gráficas generadas en una simulación en marcha

5. Pruebas

Para realizar las pruebas, se ha decidido usar **Jest**, un *framework* de *testing*, escrito en JavaScript y que hoy en día es mantenido por la comunidad. A pesar de que inicialmente nació enfocado para **React.js**, más tarde se expandió al resto de *frameworks* de JavaScript. Además del motivo principal por el que se ha decidido usar **Jest**, su facilidad, también se han tenido en cuenta otras características, como son las siguientes:

- **zero config:** en una primera instancia no necesita establecer configuración, al menos en la mayoría de los proyectos de JavaScript.

- **code coverage:** Jest puede recopilar información de cobertura de código de proyectos completos, incluidos archivos no probados.

- **isolated:** las pruebas se paralelizan ejecutándolas en sus propios procesos para maximizar el rendimiento.

- **great api:** tiene todo un conjunto de herramientas en un solo lugar, bien mantenido y documentado.

Gracias a todas las herramientas que ofrece **Jest**, es posible crear pruebas que verifiquen que el funcionamiento de la aplicación es el esperado. Para ello, lo interesante es enfocar la creación de estas pruebas de manera que cada una de ellas compruebe que se cumple una funcionalidad de los casos de usos que se mostraron en el apartado de análisis del problema.

En la siguiente imagen se muestra un ejemplo de una de las tantas pruebas que se han creado, en concreto se ha elegido el caso de uso **Iniciar Simulación**. A continuación, se trata de explicar el funcionamiento de esta con tal de comprender las posibilidades que ofrece **Jest**.

Lo primero, se monta el componente que se desea verificar, con tal de obtener una instancia, para así poder ejecutar las funciones asociadas a este y poder llegar a un estado que interese comprobar. Gracias a **expect**, función que ofrece Jest, podemos comparar el valor actual de un atributo con el esperado. En este caso, se realiza una comprobación de que las generaciones inicialmente son 0, pero al ejecutar la función **play()**, y por lo tanto, poner en marcha la simulación, después de esperar un segundo, se vuelve a comprobar para verificar que efectivamente el valor ha aumentado.

```
test('Iniciar Simulación', async () => {
    const wrapper = mount(<Main />);
    expect(wrapper.instance().state.generation).toBe(0);
    wrapper.instance().play();
    await new Promise((r) => setTimeout(r, 1000));
    expect(wrapper.instance().state.generation).toBe(1);
})
```

Ilustración 25. Ejemplo de método test de Borrar Cuadrícula.

Una vez realizadas las pruebas, si se quieren ejecutar simplemente tendremos que escribir en el terminal el comando **npm test**. Una vez finalizada la ejecución de las pruebas se muestra un resumen con el resultado de cada una de las pruebas, viendo así, si la ejecución de cada una de ellas se ha completado satisfactoriamente, o, por el contrario, ha ocurrido algún error.

```
PASS src/testing.test.js (19.946 s)
  √ Iniciar Simulación (2079 ms)
  √ Pausar Simulación (2870 ms)
  √ Parar Simulación (3093 ms)
  √ Borrar Cuadrícula (1272 ms)
  √ Añadir Persona (769 ms)
  √ Añadir Recinto (774 ms)
  √ Aumentar Velocidad (692 ms)
  √ Disminuir Velocidad (884 ms)
  √ Generar Simulación (709 ms)
  √ Eliminar (1299 ms)
  √ Ver Recinto (923 ms)
  √ Ver Persona (1974 ms)
  √ Cambiar Parámetros (798 ms)

Test Suites: 1 passed, 1 total
Tests:       13 passed, 13 total
Snapshots:   0 total
Time:        21.1 s, estimated 22 s
Ran all test suites related to changed files.
```

Ilustración 26. Ejecución de npm test

6. Conclusiones

Anteriormente, en los apartados previos se estableció cuáles eran los objetivos que el proyecto debe cumplir una vez se haya finalizado la realización de este. Ahora, ya en el apartado de conclusiones, la finalización del proyecto se ha completado.

Se considera que los objetivos que se plantearon al comienzo del proyecto, al menos los principales, se han cumplido correctamente. Los objetivos principales eran los siguientes:

- Visualización del estado de cada entidad del sistema.
- Un editor que permita al usuario definir un escenario desde el cual realizar la simulación.
- Parametrizar todas las variables del programa para que el usuario pueda definir el valor de sus variables.
- Integrar en el simulador las distintas reglas que se ejecutarán en cada ciclo de nuestro programa, tal y como puede ser infectar, aislar o morir.
- Disponer de distintas gráficas desde las que poder observar la evolución de distintos datos a lo largo de la ejecución de las generaciones del programa.

Sin embargo, cabe destacar que inicialmente también se definieron unos cuantos objetivos más, en este caso, los que dependían en parte de la sociedad, y, por lo tanto, de la publicación del proyecto con el fin de poder obtener conclusiones y verificar el cumplimiento de los objetivos. Es por esto por lo que no se puede afirmar directamente que se han cumplido este tipo de objetivos, pero, se espera que así sea.

- Conseguir que el simulador sirva como aprendizaje para aquellas personas que se están formando en el campo de las ciencias de la salud.
- Hacer crecer la consciencia pública de que ciertas enfermedades infeccionas pueden ser mucho más peligrosas de lo que creen.
- Poder recrear mediante el simulador situaciones que no serían viables en un laboratorio o cualquier otro entorno.

Durante la realización del proyecto se ha podido observar cómo se ha usado conocimiento que se ha visto a lo largo del grado. Algunas asignaturas que caben destacar son Análisis y Especificación de Requisitos, Proyecto de Ingeniería de Software y Diseño de Software.

Análisis y Especificación de Requisitos ha resultado ser fundamental no solo poder realizar diagramas UML, sino a saber identificar y especificar los requisitos de nuestra aplicación, ya que, si todas estas tareas, que se realizan en la fase inicial del proyecto, se hacen mal, puede llegar a tener consecuencias en el futuro.

Proyecto de Ingeniería de Software, además de aprender a conseguir un producto software de calidad, ha servido para poder realizar una mejor gestión y planificación a la hora de plantear un desarrollo. De esta manera se ha aprovechado mucho mejor el tiempo del que se disponía, y por consecuencia, a cumplir los plazos que se han autoimpuesto.

Diseño de Software es otra asignatura que ha sido de gran ayuda, ya que gracias a esta se ha podido desarrollar código siguiendo un orden sin que este termine en caos. También ha resultado ser útil para las pruebas que se han desarrollado para comprobar ciertas funcionalidades de la aplicación.

Para finalizar, como opinión, el autor considera que el desarrollo del proyecto ha sido una gran experiencia. No solo ha servido para poder aumentar el conocimiento sobre tecnologías de las que se desconocía, sino también a poder darse cuenta de que gracias a mucho del conocimiento que se ha obtenido a lo largo del grado, es posible utilizarlo para poder desarrollar cualquier producto software. Es cierto que siempre se tiene que buscar información, sobre todo de nuevas tecnologías que van surgiendo, sin embargo, el plan de estudios es adecuado, ya que, aun así, sirve como base para poder seguir formándose.

7. Trabajos a futuro

Gracias a la gran cantidad de funcionalidades que se han implementado en la aplicación, se ha logrado desarrollar el producto software que se esperaba, y, por lo tanto, uno de calidad.

Sin embargo, es posible, y, de hecho, sería interesante añadir más funcionalidades con tal de conseguir una aplicación mucho más completa. La razón por la que no se han añadido más, se debe al poco tiempo que se disponía para realizar el proyecto, cosa que ha acabado por obligar a elegir que funcionalidades se querían priorizar sobre otras, con tal de implementarlas en esta primera versión que ha resultado del producto.

A continuación, se presentan algunas tareas o funcionalidades que llegan a resultar como mínimo interesantes para plantearlas en un futuro y decidir si es viable realizarlas o no.

La primera de ellas ya se ha mencionado anteriormente infinidad de veces. Resulta ser la publicación de la aplicación, con tal de recibir *feedback* por parte de las personas que las usen y así poder mejorarla en base a los comentarios que se reciban de estas. Además, serviría para verificar si se cumplen los objetivos que dependen de una sociedad para ser verificados.

También se puede plantear realizar versiones del simulador para otros dispositivos, tales como dispositivos Android, iOS o inclusive una aplicación de escritorio. De esta forma se podría abarcar a más usuarios que puedan escoger el medio que mejor se ajuste a sus necesidades.

En cuanto a las funcionalidades, se puede implementar en un futuro una base de datos para disponer de un sistema de usuarios, donde cada persona se registrase y guardara un registro de cada simulación que ha realizado, consultando unas simulaciones con otras con tal de comparar los resultados de cada una de ellas y poder obtener conclusiones más precisas.

También sería interesante añadir una forma de poder trasladar los datos que se han obtenido en la simulación a un documento PDF generado por la aplicación que se permitiría al usuario descargarlo una vez haya terminado una simulación. Dentro de este documento se encontraría información tal como las gráficas generadas, datos de interés como muertes, infectados, configuración inicial, e incluso puede llegar a ser útil añadir una semilla, representada mediante una combinación de caracteres, mediante la cual, cualquier usuario que la introduzca podrá visualizar una especie de visualización que representa lo que ha ocurrido en cada generación referente a la simulación del documento generado.

Se podría añadir un sistema que definiera en qué fase se encuentra el simulador. Es decir, dependiendo de la fase seleccionada se añadirían ciertas reglas extras, como pueden ser por ejemplo mantener una distancia entre entidades o limitar el aforo por recinto.

Inicialmente, se planteó realizar una versión del simulador que, a partir de un plano, pudiendo ser este por ejemplo un edificio de la universidad, se mostrara el movimiento e interacción de las personas en su interior. Básicamente sería una versión para simular los interiores de los recintos.

Por último, como es obvio, se pueden añadir infinidad de reglas y estados que harían que el simulador fuera más complejo y fiel a una situación real. Como se ha mencionado, se puede añadir por ejemplo una regla para establecer una distancia de seguridad entre entidades, o añadir estados nuevos como pueden ser el género u oficio de la entidad.

8. Referencias

[1] ReactJS. Getting Started: https://reactjs.org/docs/getting-started.html. guía oficial de la librería, desde la cual, se ha podido tanto aprender como consultar cualquier duda que se haya tenido referente al *framework*. Fecha aproximada de última consulta: mayo de 2021.

[2] PubMed: https://pubmed.ncbi.nlm.nih.gov/. Base de datos de todo tipo de artículos científicos que ha servido para obtener información relacionada con el virus COVID-19, tales como probabilidades, síntomas o comportamiento de este. Fecha aproximada de última consulta: junio de 2021.

[3] Juego de la Vida de Conway: https://playgameoflife.com/. Página web desde la cual se puede jugar al clásico juego de la vida, tal y como lo definió el profesor Conway. Ha sido de gran utilidad ya que se ha consultado constantemente debido a estar la aplicación desarrollada basada en el juego de la vida. Fecha aproximada de última consulta: junio de 2021.

[4] JestJS: https://jestjs.io/docs/tutorial-react. Documentación oficial de JestJS, la librería que se ha decidido utilizar para desarrollar las pruebas que comprueban el correcto funcionamiento de los casos de uso. Ha servido para iniciarse y tener una idea del funcionamiento de la librería. Fecha aproximada de última consulta: junio de 2021.

[5] Victory: https://formidable.com/open-source/victory/. Documentación de la librería Victory, la cual, gracias a la guía que propone y ejemplos mostrados, ha servido para poder implementar las gráficas que se encuentran en la aplicación desarrollada y que representan los datos de una simulación. Fecha aproximada de última consulta: mayo de 2021.

[6] Li X, Wang W, Zhao X, et al. Transmission dynamics and evolutionary history of 2019-nCoV. J Med Virol. 2020 May;92(5):501–511.

[7] Guo Y-R, Cao Q-D, Hong Z-S, et al. The origin, transmission and clinical therapies on coronavirus disease 2019 (COVID-19) outbreak – an update on the status. Mil Med Res. 2020 March 13;7(1):11.

[8] Gardner, Martin (October 1970). "Mathematical Games - The Fantastic Combinations of John Conway's New Solitaire Game 'Life'".

[9] Paul Rendell (January 12, 2005). "A Turing Machine in Conway's Game of Life". Retrieved July 12, 2009.

[10] Mark Owen. "The Wireworld Computer".

[11] Gardner, Martin (October 1970). "Mathematical Games: The fantastic combinations of John Conway's new solitaire game "Life"". *Scientific American*. 223 (4): 120–123.

9. Anexos

A pesar de que en el apartado de desarrollo se mostraron algunas capturas de pantalla referentes al código de la aplicación, son pocas teniendo en cuenta el tamaño total del proyecto.

Por ello, se va a mostrar más código, pero no de cada clase, sino del contenido de cada función que actúa como una regla en el simulador que se ha desarrollado.

Código timeVaccinateDown()

```
timeVaccinatedDown = (gridperson, gridIsolated) => {
    let g2 = arrayClone(this.state.gridPeople);
    let g3 = arrayClone(this.state.gridIsolated);
    for (var i = 0; i < g2.length; i++) {
      for (var j = 0; j < g2[i].length; j++) {
        var personArr = gridperson[i][j];
        if(personArr.length!==0){
          for(var z = 0; z < personArr.length; z++){
            if(personArr[z].getTimeVaccinated()>0){
              personArr[z].setTimeVaccinated(personArr[z].getTimeVaccinated()-1)
            }
            else{personArr[z].setVaccinated(false);}
          }
        }
      }
    }
    for (var i = 0; i < g3.length; i++) {
      for (var j = 0; j < g3[i].length; j++) {
        var personArr = gridIsolated[i][j];
        if(personArr.length!==0){
          for(var z = 0; z < personArr.length; z++){
            if(personArr[z].getTimeVaccinated()>0){
              personArr[z].setTimeVaccinated(personArr[z].getTimeVaccinated()-1)}
            else{personArr[z].setVaccinated(false);}
          }
        }
      }
    }
  }
```

Código movePerson()

```
movePerson = (gridPerson) => {
    let g2 = arrayClone(this.state.gridPeople);
    for (var i = 0; i < g2.length; i++) {
      for (var j = 0; j < g2[i].length; j++) {
        var personArr = this.state.gridPeople[i][j];
        if(personArr.length!==0){
          for (var z = 0; z < personArr.length; z++) {
            var person = Person.convertObject(g2[i][j].shift());
            if(person.getStatus()==='Walking'){
              var posToGo = this.whereGo(person);
              if(person.getWalkingTime()>54){person.resetWalkingTime();}
              if(posToGo[0]===person.getDestinyX() && posToGo[1]===person.getDestinyY()){
                person.setTimeStay(person.getTimeStay()-1);
                if(person.getTimeStay()=== 0){
                  var destination = this.destinations[Math.floor(Math.random() * (this.destinations.length))];
                  person.setTimeStay(Math.floor(Math.random()*9)+1);
                  if(destination===undefined){
                    console.log(Math.floor(Math.random() * (this.rows-1)));
                    person.setDestinyX(Math.round(Math.random() * (this.rows-1)));
                    person.setDestinyY(Math.round(Math.random() * (this.cols-1)));
                  }
                  else{
                    person.setDestinyX(destination.x);
                    person.setDestinyY(destination.y);
                  }
                }
              }
              person.setX(posToGo[0]);
              person.setY(posToGo[1]);
              person.upWalkingTime();
              gridPerson[posToGo[0]][posToGo[1]].push(person);
            }
            else if(person.getStatus()==='Talking'){
              gridPerson[i][j].push(person);
            }
          }
        }
      }
    }
  }
```

Código movePersonOut()

```
movePersonOut = (gridDeath, gridIsolated) => {
      let g2 = arrayClone(this.state.gridDeath);
      let g3 = arrayClone(this.state.gridIsolated);
      for (var i = 0; i < g2.length; i++) {
         for (var j = 0; j < g2[i].length; j++) {
            var personArr = this.state.gridDeath[i][j];
            if(personArr.length!==0){
               for (var z = 0; z < personArr.length; z++) {
                  var person = Person.convertObject(g2[i][j].shift());
                  var posToGo = this.whereGoOut(person);
                  person.setX(posToGo[0]);
                  person.setY(posToGo[1]);
                  gridDeath[posToGo[0]][posToGo[1]].push(person);
               }
            }
         }
      }
      for (var i = 0; i < g3.length; i++) {
         for (var j = 0; j < g3[i].length; j++) {
            var personArr = this.state.gridIsolated[i][j];
            if(personArr.length!==0){
               for (var z = 0; z < personArr.length; z++) {
                  var person = Person.convertObject(g3[i][j].shift());
                  var posToGo = this.whereGoOut(person);
                  person.setX(posToGo[0]);
                  person.setY(posToGo[1]);
                  gridIsolated[posToGo[0]][posToGo[1]].push(person);
               }
            }
         }
      }
   }
```

Código downTimeBox()

```
downTimeBox = (gridSolid) => {for (var i = 0; i < gridSolid.length; i++) {
         for (var j = 0; j < gridSolid[i].length; j++) {
            if(gridSolid[i][j].timeInfected > 0){
               gridSolid[i][j].timeInfected = gridSolid[i][j].timeInfected-gridSolid[i][j].aireation*100
            }
            else{gridSolid[i][j].infected = false;}
         }
      }
   }
```


Código infectPlace()

```
infectPlace = (gridSolid,gridPerson) => {
    let gridPersonCopy = arrayClone(gridPerson);
    for (var i = 0; i < gridPerson.length; i++) {
      for (var j = 0; j < gridPerson[i].length; j++) {
        var personArr = gridPerson[i][j];
        if(personArr.length!==0 && gridSolid[i][j].solid){
          var maskPeople = 0; var infectedPeople = 0;
          for(var z = 0; z < personArr.length; z++){
            var person = Person.convertObject(gridPersonCopy[i][j].shift());
            if(person.getMask() && person.getInfected()){maskPeople+=1;}
            if(!person.getMask() && person.getInfected()){infectedPeople+=1;}
          }
          if(Math.random() < ((infectedPeople * (this.probInfectNoMask/100)) + (maskPeople * (this.probInfectMask/100)))){
            gridSolid[i][j].infected = true;  gridSolid[i][j].timeInfected = 2500;
          }
        }
      }
    }
}
```

Código infectedByPlace()

```
infectedByPlace = (gridSolid,gridPerson) => {
    for (var i = 0; i < gridPerson.length; i++) {
      for (var j = 0; j < gridPerson[i].length; j++) {
        var personArr = gridPerson[i][j];
        if(personArr.length!==0 && gridSolid[i][j].solid){
          for(var z = 0; z < personArr.length; z++){
            if(personArr.length>1){
              var array = [];
              for(var m = 0; m < personArr.length; m++){if(m!==z){array.push(personArr[m])}}
              if(Math.random() < this.probInfected(personArr[z], array)){personArr[z].infect(this.timeInfected);}
            }
            if(gridSolid[i][j].infected){
              if(Math.random() < this.probInfectedByPlace(personArr[z])){
                if(!personArr[z].getInfected()) this.numberInfecteds=this.numberInfecteds+1;
                personArr[z].infect(this.timeInfected);
              }
            }
          }
        }
      }
    }
}
```

Código timeInfectedDown()

```
timeInfectedDown = (gridperson, gridIsolated) => {
      let g2 = arrayClone(this.state.gridPeople);
      let g3 = arrayClone(this.state.gridIsolated);
      for (var i = 0; i < g2.length; i++) {
         for (var j = 0; j < g2[i].length; j++) {
            var personArr = gridperson[i][j];
            if(personArr.length!==0){
               for(var z = 0; z < personArr.length; z++){
                  if(personArr[z].getTimeInfected()>0){personArr[z].setTimeInfected(personArr[z].getTimeInfected()-1)}
                  else if(personArr[z].getInfected()){
                     personArr[z].setInfected(false);
                     if(!personArr[z].getInfected()) this.numberInfecteds=this.numberInfecteds-1;}
               }
            }
         }
      }
      for (var i = 0; i < g3.length; i++) {
         for (var j = 0; j < g3[i].length; j++) {
            var personArr = gridIsolated[i][j];
            if(personArr.length!==0){
               for(var z = 0; z < personArr.length; z++){
                  if(personArr[z].getTimeInfected()>0){personArr[z].setTimeInfected(personArr[z].getTimeInfected()-1)}
                  else if(personArr[z].getInfected()){personArr[z].setInfected(false);
                     if(!personArr[z].getInfected()) this.numberInfecteds=this.numberInfecteds-1;}
               }
            }
         }
      }
   }
```

Código speakPeople()

```
speakPeople = (gridPerson) => {
    for (var i = 0; i < gridPerson.length; i++) {
        for (var j = 0; j < gridPerson[i].length; j++) {
            var personArr = gridPerson[i][j];
            if(personArr.length>1){
                for(var z = 0; z < personArr.length; z++){
                    if(Math.random()<(this.probTalk/100) && (z+1 < personArr.length) && personArr[z].getStatus()!=='Talking')
                    {
                        var randomTime = Math.random()*10;
                        personArr[z].setStatus(1);
                        personArr[z].setTimeStatus(randomTime)
                        personArr[z+1].setStatus(1);
                        personArr[z+1].setTimeStatus(randomTime)
                    }

                    else if(personArr[z].getStatus()==='Talking'){
                        var array=[];
                        for(var m = 0; m < personArr.length; m++){if(m!==z){array.push(personArr[m])}}
                        if(Math.random() < this.probInfected(personArr[z], array)){
                            if(!personArr[z].getInfected()) this.numberInfecteds=this.numberInfecteds+1;
                            personArr[z].infect(this.timeInfected)
                        }
                        personArr[z].setTimeStatus(personArr[z].getTimeStatus()-1);
                        if(personArr[z].getTimeStatus()<0){personArr[z].setStatus(0); personArr[z].setTimeStatus(0)}
                    }
                }
            }

            else if(personArr.length===1 && personArr[0].getStatus()==='Talking'){personArr[0].setStatus(0); personArr[0].setTimeStatus(0)}
        }
    }
}
```

Código isolatePeople()

```
isolatePeople = (gridPerson, gridIsolated) => {
    for (var i = 0; i < gridPerson.length; i++) {
        for (var j = 0; j < gridPerson[i].length; j++) {
            var personArr = gridPerson[i][j];
            if(personArr.length!==0){
                for(var z = 0; z < personArr.length; z++){
                    if(personArr[z].getInfected() && Math.random()<((this.probIsolate/100)/72))
                    {
                        var person = Person.convertObject(gridPerson[i][j].shift());
                        var col = Math.round(Math.random()*44);
                        var row = Math.round(Math.random()*9);
                        person.setX(row);
                        person.setY(col);
                        person.setStatus(2);
                        gridIsolated[row][col].push(person);
                    }
                }
            }
        }
    }
    for (var i = 0; i < gridIsolated.length; i++) {
        for (var j = 0; j < gridIsolated[i].length; j++) {
            var personArr = gridIsolated[i][j];
            if(personArr.length!==0){
                for(var z = 0; z < personArr.length; z++){
                    if(!personArr[z].getInfected())
                    {
                        var person = Person.convertObject(gridIsolated[i][j].shift());
                        var col = Math.round(Math.random()*(this.cols-1));
                        var row = Math.round(Math.random()*(this.rows-1));
                        person.setX(row);
                        person.setY(col);
                        person.setStatus(0);
                        gridPerson[row][col].push(person);
                    }
                }
            }
        }
    }
}
```

Código killPeople()

```
killPeople = (gridPerson, gridDeath, gridIsolated) => {
    for (var i = 0; i < gridPerson.length; i++) {
        for (var j = 0; j < gridPerson[i].length; j++) {
            var personArr = gridPerson[i][j];
            if(personArr.length!==0){
                for(var z = 0; z < personArr.length; z++){
                    if(personArr[z].getInfected() && Math.random()<this.probDeath(personArr[z]))
                    {
                        var person = Person.convertObject(gridPerson[i][j].shift());
                        var col = Math.round(Math.random()*44);
                        var row = Math.round(Math.random()*9);
                        person.setX(row);
                        person.setY(col);
                        person.setStatus(3);
                        gridDeath[row][col].push(person);
                        this.numberDeaths=this.numberDeaths+1;
                    }
                }
            }
        }
    }
    for (var i = 0; i < gridIsolated.length; i++) {
        for (var j = 0; j < gridIsolated[i].length; j++) {
            var personArr = gridIsolated[i][j];
            if(personArr.length!==0){
                for(var z = 0; z < personArr.length; z++){
                    if(personArr[z].getInfected() && Math.random()<this.probDeath(personArr[z]))
                    {
                        var person = Person.convertObject(gridIsolated[i][j].shift());
                        var col = Math.round(Math.random()*44);
                        var row = Math.round(Math.random()*9);
                        person.setX(row);
                        person.setY(col);
                        person.setStatus(3);
                        gridDeath[row][col].push(person);
                    }
                }
            }
        }
    }
}
```

Código vaccinate()

```
    vaccinate = () => {
    let g2 = arrayClone(this.state.gridPeople);
    for (var i = 0; i < g2.length; i++) {
      for (var j = 0; j < g2[i].length; j++) {
        var personArr = this.state.gridPeople[i][j];
        if(personArr.length!==0){
          for(var z = 0; z < personArr.length; z++){
            if(!this.isVaccinated(1) && personArr[z].getAge()>=80 && !personArr[z].getVaccinated()){
              personArr[z].setVaccinated(Math.random()<this.successVaccine/100);
              if(personArr[z].getVaccinated()){personArr[z].setTimeVaccinated(this.timeVaccinated)};
            }
            else if(this.isVaccinated(1) && !this.isVaccinated(2) && personArr[z].getAge()>=65 && personArr[z].ge
tAge()<80 && !personArr[z].getVaccinated()){
              if(!personArr[z].getVaccinated()){
                personArr[z].setVaccinated(Math.random()<this.successVaccine/100);
                if(personArr[z].getVaccinated()){personArr[z].setTimeVaccinated(this.timeVaccinated)}}
            }
            else if(this.isVaccinated(1) && this.isVaccinated(2) && !this.isVaccinated(3) && personArr[z].getRisky(
) && !personArr[z].getVaccinated()){
              if(!personArr[z].getVaccinated()){
                personArr[z].setVaccinated(Math.random()<this.successVaccine/100);
                if(personArr[z].getVaccinated()){personArr[z].setTimeVaccinated(this.timeVaccinated)}}
            }
            else if(this.isVaccinated(1) && this.isVaccinated(2) && this.isVaccinated(3)){
              if(!personArr[z].getVaccinated()){
                personArr[z].setVaccinated(Math.random()<this.successVaccine);
                if(personArr[z].getVaccinated()){personArr[z].setTimeVaccinated(this.timeVaccinated)}}
            }

          }
        }
      }
    }
  }
```