



UNIVERSIDADE DA CORUÑA

FACULTADE DE INFORMÁTICA

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA INFORMÁTICA

Mención en Computación

**Generación de Escenarios de un Videojuego  
2D mediante Programación Lógica**

**Autor:** Rafael Alcalde Azpiazu

**Director:** José Pedro Cabalar Fernández

*A Coruña, 29 de agosto de 2018*



# Especificación

*Título del proyecto:* Generación de Escenarios de un Videojuego  
2D mediante Programación Lógica

*Clase:* Proyecto clásico de Ingeniería

*Alumno:* Rafael Alcalde Azpiazu

*Director:* José Pedro Cabalar Fernández

*Miembros del tribunal:*

*Fecha de lectura:*

*Calificación:*



DR. JOSÉ PEDRO CABALAR FERNÁNDEZ

Titular de universidad

Departamento de Computación

Universidade da Coruña

CERTIFICA

Que la memoria titulada **Generación de Escenarios de un Videojuego 2D mediante Programación Lógica** ha sido realizada por RAFAEL ALCALDE AZPIAZU, con DNI 47401974-D, bajo su dirección y constituye la documentación de su trabajo de Fin de Grado para optar a la titulación de Graduado en Ingeniería Informática por la Universidade da Coruña.

*A Coruña, 29 de agosto de 2018*



*Dedicatoria...*





# Agradecimientos

*Agradezco..*

Rafael Alcalde Azpiazu

*A Coruña, 29 de agosto de 2018*



# Resumen

Abstract



# Palabras clave

Keyword one, keyword two, etc.



# Índice general

<b>1. Introducción</b>	<b>1</b>
<b>2. Contexto</b>	<b>3</b>
2.1. Juegos de estrategia por turnos . . . . .	3
2.1.1. Sid Meier's: Civilization . . . . .	4
2.1.2. Proyecto Freeciv . . . . .	6
2.2. Tecnologías . . . . .	7
2.2.1. Answer Set Programming . . . . .	7
2.2.2. Lua . . . . .	9
2.2.3. JSON . . . . .	10
2.3. Herramientas . . . . .	11
2.3.1. Atom . . . . .	11
2.3.2. Git . . . . .	11
2.3.3. LÖVE . . . . .	13
<b>3. Trabajo desarrollado</b>	<b>15</b>
3.1. Propuesta . . . . .	15
3.1.1. Formato del escenario de Freeciv . . . . .	15
3.2. Proceso de ingeniería . . . . .	20
3.2.1. Metodología . . . . .	20
3.2.2. Gestión del proyecto . . . . .	20
3.3. Análisis del software . . . . .	20
3.3.1. Requisitos funcionales . . . . .	20
3.3.2. Requisitos no funcionales . . . . .	21

3.4. Diseño del sistema . . . . .	21
3.4.1. Arquitectura software . . . . .	21
3.4.2. Casos de uso . . . . .	23
3.4.3. Pantallas del sistema . . . . .	23
3.4.4. Implementación . . . . .	27
<b>4. Evaluación</b>	<b>35</b>
<b>5. Conclusiones</b>	<b>37</b>
<b>Apéndices</b>	<b>43</b>



# Capítulo 1

## Introducción



# Capítulo 2

## Contexto

### 2.1. Juegos de estrategia por turnos

También conocidos como *turn-based strategy* (TBS), son juegos donde el tiempo de juego no transcurre de forma continua, si no que se divide en partes bien definidas llamadas turnos. En un turno, un jugador dispone de un periodo de análisis antes de realizar una acción, lo cual realiza un salto de turno al siguiente jugador. Cuando todos los jugadores han terminado su turno, se comienza una nueva ronda.

Existen varios géneros dentro de la estrategia por turnos:

- **Juegos clásicos de tablero:** Son los primeros juegos de este tipo, padres del resto de géneros. La idea es controlar la acción del propio juego mediante turnos. En este género tenemos como ejemplos el ajedrez, el Go o el Revesi.
- **TBS and RTT** (Estrategia por turnos y táctica en tiempo real): Añade un componente en tiempo real al permitir que en cada turno se generen batallas en tiempo real contra los oponentes. Un ejemplo de este género es la saga de videojuegos Total War.
- **Man-to-man:** Se basa en el uso de unidades muy pequeñas, en donde controlamos cada uno de los individuos de nuestra partida. En esta

categoría está la saga de videojuegos XCOM.

- **TRPG** (RPG táctico): También conocidos en Japón como Simulation RPG (SRPG), se basan en incorporar elementos de estrategia por turnos al combate clásico de los RPG. En la parte de tablero, un ejemplo clásico es Dragones y Mazmorras (*Dungeons & Dragons*), mientras que en el caso de los videojuegos, los más conocidos son Darkest Dungeon, algunas entregas de Megami Tensei como Shin Megami Tensei y Persona, la saga Disgaea o Fire Emblem.
- **4X**: Llamado así por la idea principal del género (*eXplore, eXpand, eX-ploit and eXterminate*, en castellano “Explora, expande, explota y extermina”). Se basan en el control de un imperio, profundizando en el desarrollo económico, tecnológico y militar, con la idea final de que crear a largo tiempo un imperio sostenible. Existe una profunda complejidad debido a la cantidad de elementos que hay que tener en cuenta. Un ejemplo en tablero de este género es Risk, mientras que en videojuegos destaca la saga Master of Orion y Sid Meier’s: Civilization [ver sección 2.1.1].

### 2.1.1. Sid Meier’s: Civilization

Como ya comenté, Civilization es un videojuego de estrategia por turnos que cae dentro del género X4, lanzado por primera vez en 1991 por el programador y diseñador Sid Meier. El objetivo del juego es construir una civilización y avanzarla desde la prehistoria y hasta un futuro cercano. En cada turno, el jugador puede mover unidades por el mapa, construir o mejorar ciudades y unidades, y realizar negociaciones con el resto de jugadores. Una vez se realiza una ronda, el juego avanza unos años en la historia. Actualmente existen seis títulos principales (el último, Civilization VI, lanzado en 2016) y varios spin-off que se centran en potenciar distintos aspectos de los títulos principales.

Al empezar la partida, el jugador se encuentra en un mapa basado en casillas generado proceduralmente, el cual contiene distintos biomas (praderas,

desierto, montañas, ríos, lagos, etc...) con una serie de recursos (vino, caballos, trigo, fuel, carbón, etc...). La posición inicial normalmente es aleatoria, y el jugador puede controlar con una serie de colonos. En el resto del mapa cercano a los colonos, existe una niebla de guerra para evitar que el jugador pueda ver que hay cerca de él.

Si el jugador decide fundar una ciudad con los colonos, esta empieza a generar recursos que pueden ser los propios del mapa que hay en las casillas cercanas a la ciudad o simplemente mano de obra, cultura, ciencia o dinero. La cantidad de recursos que se puede generar en una ciudad se basa en como de grande es esta. Con estos recursos, se puede producir nuevas unidades como colonos, guerreros, distintos elementos de guerra como barcos, aviones y maquinaria pesada, o unidades con las que mantener comercio con otras civilizaciones.

Con la ciencia y la cultura se puede aumentar el progreso tecnológico, el cual se expresa en un árbol de tecnologías a desarrollar. El jugador puede eligiendo que tecnología se desarrolla en todo momento. Con respecto al dinero se puede usar para mejorar a las ciudades y unidades, crear carreteras, aumentar el ritmo de producción o crear regalos para otras civilizaciones.

En las ciudades se pueden mejorar al crear nuevos edificios, que ayudan a mejorar la producción y aumentar la población, como graneros, universidades, distritos comerciales o librerías. También pueden servir de fortaleza, como murallas, castillos de guardia o escuelas militares. Por último, existen edificios únicos que son las maravillas del mundo, que ayudan a aumentar el ritmo general de una civilización y que dan bonus al primero que las complete, de ahí que solo se pueda crear una vez en toda la partida.

Existen múltiples victorias por las que puede ganar un jugador: la victoria por conquista ocurre cuando el jugador toma el control o añade al imperio a

todas las ciudades capitales del resto de civilizaciones; la victoria diplomática se obtiene cuando el jugador construye las Naciones Unidas al entablar amistades con todas las civilizaciones; la victoria tecnológica se alcanza cuando el jugador construye una nave para llegar a Alfa Centauri; por último, la victoria cultural se obtiene al acumular la suficiente cultura con respecto a las otras civilizaciones y construir los edificios necesarios para guiar al resto a conseguir un estado utópico con alta cultura.

Debido al éxito de las entregas de Civilization y las disputas legales sobre el derecho de explotación entre Activision y Avalon Hills, se han creado distintos proyectos y secuelas que descienden del oficial, como la saga Call to Power o los proyectos Openciv y Freeciv [ver sección 2.1.2].

### 2.1.2. Proyecto Freeciv

Freeciv<sup>1 2</sup> es un videojuego de código abierto basado en la serie de Sid Meier's Civilization, sobre todo en Civilization II. Fue creado por tres estudiantes del departamento de computación de la universidad de Aarhus debido al bajo rendimiento de Openciv. Basándose en la arquitectura de X11, escribieron un sencillo cliente en donde se ejecutaría una prueba de concepto, la cual tuvo una enorme acogida y rápidamente se creó una comunidad, la cual pasó a gestionar el proyecto debido a que sus autores desearon la idea de continuar. Debido a esto expandiendo el juego al incluir elementos online, mejoras en la interfaz gráfica, mejor rendimiento, traducción para 30 idiomas distintos (entre ellos el castellano) y nuevos elementos al juego.

El juego está escrito en C y tiene compatibilidad con el estandar POSIX para ser lo más portable posible. Contiene un intérprete de Lua [ver sección 2.2.2] que permite cargar scripts con el que dar un componente dinámico a las partidas (lo usa sobre todo para tutoriales) y desde el 2006 IANA asignó el

---

<sup>1</sup><http://www.freeciv.org>

<sup>2</sup><https://github.com/freeciv>

puerto TCP/UDP 5556 a Freeciv [1].

Como el proyecto es de código abierto (tiene una licencia GPL [2]), cualquiera puede descargar el código fuente y modificarlo. Es por eso que se ha conseguido que el juego tenga diferentes tecnologías de interfaz gráfica, pasando por bibliotecas específicas como GTK<sup>3</sup> y Qt<sup>4</sup>, bibliotecas gráficas generales como SDL<sup>5</sup> o OpenGL<sup>6</sup>, e incluso bibliotecas del estándar web como WebGL<sup>7</sup>.

## 2.2. Tecnologías

Debido a las exigencias a la hora de desarrollar el proyecto, se ha optado por elegir un lenguaje de programación lógico sobre el que realizar la base declarativa del proyecto, ya que nos permitirá expresar las reglas de generación del mapa de forma matemática. También se ha escogido un segundo lenguaje multipropósito que nos servirá como soporte para crear un entorno gráfico con el que poder editar, guardar y cargar los mapas creados.

### 2.2.1. Answer Set Programming

*Answer Set Programming* (ASP) es un paradigma enfocado a la resolución declarativa de problemas difíciles, combinando un lenguaje simple con el que modelar los problemas lógicos y herramientas de alto rendimiento para la resolución de estos. ASP está basado en modelos estables [3], que usa para definir la semántica declarativa en los programas lógicos, y lógica no monótono [4], que añade razonamiento por defecto. Con esto, ASP permite resolver problemas *NP-hard* de forma uniforme.

Mediante estos mecanismos, los problemas lógicos se reducen al cómputo

---

<sup>3</sup><https://www.gtk.org>

<sup>4</sup><https://www.qt.io>

<sup>5</sup><https://www.libsdl.org>

<sup>6</sup><https://www.opengl.org>

<sup>7</sup><https://www.khronos.org/webgl>

de los modelos estables de un programa lógico dado, lo cual se hace mediante herramientas llamadas *solvers*, que se encargan de utilizar diferentes técnicas para la expansión de lo que se llama Forma Normal Conjuntiva (FNC), que es una conjunción de cláusulas.

Para ello, se necesita que los programas lógicos estén expresados con variables libres. Como la forma de expresar un programa lógico en ASP es mediante un lenguaje de alto nivel con ciudadanos de primer orden, muchas de las variables están ligadas. Es por eso que, antes de resolver el programa, se usa unas herramientas llamadas *grounders*, que permiten transformar el programa a su equivalente con variables libres.

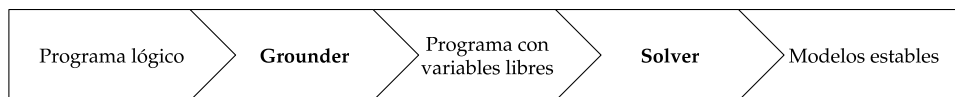


Figura 2.1: Ejemplo de ejecución de ASP

Unas de las herramientas más importantes de ASP son la de Potassco<sup>8</sup>, la cual son un conjunto de herramientas para ASP desarrolladas en la Universidad de Postdam. Contienen las herramientas fundamentales como un grounder llamado *gringo*; un solver que es *clasp*; y una herramienta que aglutina todo el sistema ASP, *clingo*. Así mismo, añade más funcionalidades al lenguaje ASP, como puede ser al resolución iterativa, debido a que permite embeber otros lenguajes como Lua [ver sección 2.2.2] o Python, que pueden interactuar con el programa escrito en ASP mediante la interfaz o *API* de *clingo*.

Estas herramientas serán claves a la hora de realizar este proyecto, ya que, como veremos más adelante, debido a la naturaleza de ASP, serán la base sobre la que se construirá todo el proyecto.

---

<sup>8</sup><http://potassco.org>



### 2.2.2. Lua

Lua<sup>9</sup> [5] (del portugués Luna), es un lenguaje de programación de propósito general, de scripting y multiparadigma (procedural, orientado a objetos basado en prototipos, funcional y *data-driven*). Fue pensado para ser embebido en cualquier aplicación independientemente de la plataforma sobre la que se ejecuta. Su intérprete esté escrito en ANSI C y contiene una API en C muy simple.

Uno de los puntos fuertes de Lua es que permite construir nuevos tipos en base a arrays asociativos, que también permiten extender la semántica del lenguaje al aplicar metadatos a estos. A parte de esto, también facilita la tarea al programador al tener un recolector incremental de basura que se encarga de gestionar automáticamente la memoria. Además, Lua es de tipado dinámico, y se ejecuta sobre un intérprete bytecode en una máquina virtual basada en registros, ejecutándose más rápido que otros lenguajes ejecutados sobre una máquina virtual basada en *stack*, como puede ser Java.

A parte de esto, existen otras implementaciones y dialectos basados en Lua, como LuaJIT<sup>10</sup>, que es una implementación de la máquina virtual de Lua que compilar en tiempo de ejecución y no antes de ejecutarse; y MoonScript<sup>11</sup>, un lenguaje de scripting basado en Python que extiende el lenguaje de Lua al añadir orientación a objetos basados en clases, una forma más eficiente de lenguaje funcional y nuevas estructuras y elementos que no contenía Lua.

Existen otras implementaciones que lo único en lo que se diferencian de la implementación estándar es que expanden las funciones básicas o implementan bibliotecas por defecto. A estas implementaciones se llaman *Lua Players* o *Lua Engines*, que principalmente son destinadas para la creación de videojuegos, ya que añaden la implementación de las bibliotecas gráficas para una plata-

---

<sup>9</sup><http://www.lua.org>

<sup>10</sup><http://luajit.org/luajit.html>

<sup>11</sup><https://moonscript.org>

forma concreta, como pueden ser SDL o OpenGL. Los más destacados son los intérpretes no oficiales para consolas de Sony y Nintendo, y los intérpretes para ordenador y móviles como Corona, Cocos-2D, Moai y LÖVE [ver sección 2.3.3].

Como ya expliqué en el apartado anterior [sección 2.2.1], las herramientas ASP que se usarán permiten usar este como un lenguaje embebido, lo que nos permitirá controlar la ejecución de la base. Así mismo, tomaremos el *engine* LÖVE para crear una interfaz gráfica para que los usuarios puedan trabajar mejor con la edición de los mapas.

### 2.2.3. JSON

JSON<sup>12</sup> (acrónimo de JavaScript Object Notation) es un formato abierto [6] ligero de intercambio de información. Está pensado para la transmisión de objetos de datos usando texto en un formato legible por humanos, pudiendo ser leído y modificado fácilmente.

A demás, puedes ser fácilmente analizado y generado por máquinas, de ahí que a pesar de que en sus inicios fue pensado como un subconjunto del lenguaje de programación JavaScript/ECMAScript, actualmente es un lenguaje independiente debido a que muchos otros lenguajes incluyen actualmente herramientas para poder leerlo y modificarlo de forma nativa.

JSON se basa en dos tipos de estructuras:

- Un mapa asociativo que guarda pares (nombre, valor). Esta estructura se reconoce en el estándar como *object*.
- Una lista ordenada que guarda valores. Esta estructura se reconoce en el estándar como *array*.

Estas estructuras se pueden mezclar independientemente, pudiendo generar listas de objetos u objetos que contienen listas de elementos.

---

<sup>12</sup><https://www.json.org>

Debido estas características, y a que Lua soporta este estándar mediante bibliotecas externas, se usará para el intercambio de datos entre el programa principal y la parte lógica, así como será utilizado para almacenar temporalmente los escenarios antes de ser exportados al software final.

## 2.3. Herramientas

Aparte de las tecnologías usadas para la construcción del proyecto, se ha usado distintas herramientas que nos han servido a la hora de elaborar y desarrollar el sistema planteado.

### 2.3.1. Atom

Atom<sup>13</sup> es un editor de texto gratuito y de código abierto basado en Electron, que es un *Framework* para crear aplicaciones nativas con tecnología Web. Está desarrollado por GitHub Inc. e integra dentro de su interfaz la herramienta Git para el control de versiones. Se puede extender las funcionalidades de Atom mediante *plugins* escritos con Node.js, los cuales permiten desde resaltar la sintaxis de un determinado lenguaje de programación, cambiar el tema del editor, hasta añadir terminales, autocompletado de texto o nuevas herramientas para desarrollar/depurar código fuente. Entre estos plugins cabe destacar que para este proyecto podemos encontrar algunos que nos permiten resaltar la sintaxis de Lua y de ASP, así como iniciar LÖVE y ASP desde el propio editor. Debido a estas características, se usará este editor como una de las herramientas principales.

### 2.3.2. Git

Git<sup>14</sup> es un sistema de control de versiones distribuido y de código abierto creado para la gestión de código fuente y desarrollo de software. Fue creado

---

<sup>13</sup><https://atom.io>

<sup>14</sup><https://git-scm.com>

por Linus Trovalds como una alternativa libre y gratuita a BitKeeper para el desarrollo del kernel Linux por la comunidad. Está enfocado en la rapidez y eficiencia al procesar proyectos grandes, la integridad de datos, la seguridad mediante autenticación criptográfica y el fuerte soporte de un flujo de trabajo no lineal y distribuido.

Debido a estas características, Git es uno de los sistemas de control de versiones más importantes hoy en día, por lo que ha permitido que existan servicios de alojamiento en línea que incluyen esta herramienta:

- GitLab<sup>15</sup>, creado por dos programadores ucranianos y que hoy en día es gestionado por GitLab Inc. Es usado por organizaciones como Sony, IBM, NASA, CERN o GNOME Foundation. Permite realizar gratuitamente repositorios privados, gestionar grupos y realizar rastreo de *issues* y funcionalidades de CI/CD.
- Phabricator<sup>16</sup>, creado por Facebook como herramienta interna que integra también Mercurial y Subversion. Actualmente es de código abierto y lo usan empresas como Blender, Cisco Systems, Dropbox o KDE.
- Bitbucket<sup>17</sup>, creado por Atlassian. Este sistema integra Mercurial desde sus inicios y Git desde 2011, y está pensado para ser integrado con el resto de productos como Atlassian como Jira, Confluence y Bamboo.
- GitHub<sup>18</sup>, creado por tres estudiantes estadounidenses, hoy en día es gestionado por GitHub Inc. empresa que fue adquirida por Microsoft en 2018. Permite realizar rastreo de *bugs*, wikis, gestión de tareas y petición de funcionalidades. Es usado por empresas como Microsoft, Google, Travis CI, Bitnami, DigitalOcean y Unreal Engine. Con la popularidad de GitHub nacieron varios servicios, como el Education Program, que

---

<sup>15</sup><https://gitlab.com>

<sup>16</sup><https://www.phacility.com>

<sup>17</sup><https://bitbucket.org>

<sup>18</sup><https://www.github.com>

permite a estudiantes el acceso gratuito a herramientas de GitHub y de partners; Gist, que permite usar GitHub como un hosting de *snippets* y GitHub Marketplace, que permite comprar servicios con los que aumentar las funcionalidades en los proyectos y que muchos de ellos son gestionados por partners.

### 2.3.3. LÖVE

LÖVE<sup>19</sup> (o también conocido como Love2D) es un motor de código libre multiplataforma para la creación de juegos en 2D mediante el lenguaje Lua. Está hecho en C++ y añade una API al lenguaje Lua para que el programador pueda usar las bibliotecas OpenGL ES y SDL de una forma muy simplificada mediante funciones y tipos creados en Lua. Con el paso del tiempo también fue incluyendo un motor de físicas 2D o bibliotecas para el manejo de fuentes FreeType, manejo de strings en UTF-8, funciones para usar sockets y una biblioteca especializada en conexión a videojuegos llamada ENet. Así mismo, debido al crecimiento de la comunidad, esta fue creando bibliotecas no oficiales que expanden el funcionamiento del motor, como bibliotecas para añadir interfaces inmediatas, soporte para técnicas usadas en videojuegos, orientación a objetos basada en clases o incluso portar el motor a otras plataformas o lenguajes.

Debido a la sencillez para poder dibujar en pantalla, así como poder integrar las bibliotecas gráficas creadas para LÖVE, este motor va a ser una de las herramientas principales que se usarán para la construcción de la interfaz de usuario en este proyecto.

---

<sup>19</sup><https://love2d.org>



# Capítulo 3

## Trabajo desarrollado

En este capítulo se detalla cada uno de los puntos llevados a cabo para la realización de este proyecto, empezando por definir la propuesta realizada, luego explicar el proceso ingenieril llevado a cabo y terminar desglosando el trazado de la ejecución de este proyecto.

### 3.1. Propuesta

El trabajo propuesto tiene como objetivo la creación de un elemento software funcional que, usando el paradigma lógico explicado en la Sección 2.2.1, permita la generación de un escenario jugable que pueda ser ejecutado en el juego Freeciv, como se indica en la Sección 2.1.2. Así mismo incluirá una interfaz gráfica interactiva que permitan al usuario marcar que zonas del terreno deben generarse y cuales no, indicando su contenido antes de lanzar el proceso de generación.

#### 3.1.1. Formato del escenario de Freeciv

Como uno de los puntos fuerte de este trabajo tiene que ver con la generación de escenarios, este sistema tiene que tener como salida un mapa válido que sea leído correctamente por el juego Freeciv. Es por eso que explicaré en detalle el formato usado.

Para empezar, el formato se basa en archivo de texto plano que contiene varios campos, haciendo que sea lo más simple posible y que en la teoría se pueda modificar a mano.

Listado 3.1: Ejemplo de formato de mapa

```
[scenario]
is_scenario=TRUE
name=_("My map")
description=_("This map is a example.")
players=TRUE

[savefile]
options=" +version2"
version=20
reason="Scenario"
rulesetdir="classic"
[...]

[settings]
set={"name", "value", "gamestart"
    "generator", "SCENARIO", "RANDOM"
    "mapsize", "FULLSIZE", "FULLSIZE"
    "maxplayers", "::PLAYERS::", "::PLAYERS::"
    "topology", "", "WRAPX|ISO"
    "xsize", 50, 12
    "ysize", 50, 12
    [...]
}
set_count=9

[map]
have_huts=FALSE
t0000="h+hf  aaaa  ff"
t0001="dgg  aat  ff"
t0002="hhh  pp"
t0003="hmp  ggghh"
t0004="hmh  s dghf"
```



```

t0005=" dg      phffpm "
t0006="          "
t0007="aa      sdg      aa"
t0008="aa      h      aa"
t0009="          jff p      "
t0010="mh      s pdp pm"
t0011="gf      pdhh      "
t0012="          gpsg      p"
t0013="          h hh      "
t0014="hff          gg s"
t0015="m+h      aaaa      m f"
startpos_count=5
startpos={"x","y","exclude","nations"
          0,2,FALSE," Russian"
          9,11,FALSE," Spanish"
          14,0,FALSE," "
          2,5,FALSE," "
          12,3,FALSE," "
}
b00_0000="0000000000000000"
[...]
r00_0000="0000000000000000"
[...]
```

Como se puede comprobar en el Listado 3.1, el archivo que contiene un mapa se divide en varias secciones de datos, los cuales se marcan con un título entre corchetes. De los definidos por Freeciv, se puede destacar algunos.

- **scenario**: Configura los ajustes básicos del mapa creado, como puede ser el nombre del mismo (con el campo **name**) o una breve descripción (con el campo **description**).
- **savefile**: Establece los valores por defecto de las opciones de juego, como puede ser la versión de Freeciv mínima (con el campo **options**), el conjunto de reglas de juego (con el campo **rulesetdir**) o las tecnologías disponibles (se establece en la lista guardada por **technology\_vector** y se indica el número de elementos en **technology\_size**).

- **settings**: Se puede definir una lista de valores del mapa y la topología del mismo (que se explica en detalle en la Sección 3.1.1) en el campo **set**. El número de valores definidos se guarda en el campo **set\_count**.
- **map**: En esta sección se indica cada uno de los valores de terreno de la rejilla del mapa (en los campos **t00\_XXXX**), así como la lista de puntos de inicio de jugadores (definidos en la lista **startpos** e indicado el tamaño de la lista en **startpos\_count**), las capas con recursos o incluso las capas de ríos.

#### Topología del mapa

El mapa es siempre una rejilla de dos dimensiones en el que cada celda es una baldosa o *tile*. Esta rejilla puede estar configurada de varias maneras con la variable **topology** en la sección **settings**.

- **warpx**: La topología de escenario es como un mapa terrestre, es decir el eje Este-Oeste se junta.
- **warpy**: La topología del escenario junta el eje Norte-Sur.
- **warpx warpy**: La topología del escenario es un toroide, es decir, tiene forma de donuts.
- **iso**: La rejilla del escenario es isométrico.
- **hex**: La rejilla del escenario es hexagonal.
- **iso hex**: La rejilla del escenario es en forma de panel de abeja.

#### Terrenos disponibles por defecto

En Freeciv, cada celda de la rejilla del mapa contiene una baldosa de terreno único, que viene definido por un identificador único, el cual puede ser uno de los siguientes:

Descripción	ID	Imagen
<i>Pradera</i> : Es uno de los terrenos más comunes. Las unidades se pueden mover fácilmente.	g	
<i>Llanura</i> : Es otro terreno muy común. Se puede usar para crear carreteras.	p	
<i>Colinas</i> : Las unidades se mueven lentamente. Es uno de los terrenos con mayor bonus defensivo (200 %).	h	
<i>Bosque</i> : Produce una unidad de producción (madera) con la que construir edificaciones. Tiene un bonus defensivo de 150 %	f	
<i>Jungla</i> : Puede llegar a producir 4 unidades de producción si se encuentran con recursos de gemas o fruta.	j	
<i>Montañas</i> : Es el terreno con mayor bonus defensivo (300 %). Solo las unidades aéreas (aviones, cazas, etc) pueden atravesarla.	m	
<i>Desierto</i> : Normalmente solo se puede usar para crear carreteras, pero si hay un modificador de oasis puede generar hasta 3 unidades de producción.	d	
<i>Pantano</i> : Se puede irrigar rápidamente. Puede producir de 5 a 9 unidades de producción si se encuentra con recursos como tundra o con especias.	s	
<i>Tundra</i> : Solo se pueden crear carreteras.	t	
<i>Glacier</i> : Ninguna de las unidades puede cruzarlo.	a	

Tabla 3.1: Tipos de terreno

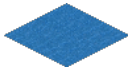
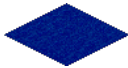
Descripción	ID	Imagen
<i>Mar</i> : Todas las unidades acuáticas pueden cruzarlo. Puede producir 2 unidades de producción si se encuentra con un banco de peces		
<i>Océano</i> : Solo las grandes embarcaciones y submarinos pueden pasar por encima.	:	

Tabla 3.2: Tipos de terreno

## 3.2. Proceso de ingeniería

### 3.2.1. Metodología

### 3.2.2. Gestión del proyecto

## 3.3. Análisis del software

Una vez definido el sistema y planificada su construcción, se ha realizado un análisis en donde se identifican los requisitos que debe cumplir el software una vez terminado el proyecto.

### 3.3.1. Requisitos funcionales

Los requisitos funcionales son aquellas condiciones indispensables que estipulan las funcionalidades que debe proporcionar el sistema. Para este proyecto se ha recogido los diferentes requisitos:

- Generación de un mapa que sea legible por el videojuego Freeciv.
- Permitir añadir restricciones sobre ciertas zonas del mapa.
- Poder guardar y recuperar el mapa en un formato sencillo.

### 3.3.2. Requisitos no funcionales

Los requisitos no funcionales, por su contra, son aquellas condiciones indispensables que debe cumplir el sistema a la hora de diseñar e implementar. Para este proyecto se han tenido en cuenta estos requisitos:

- Eficiencia y eficacia: El generador debe responder en el menor tiempo posible arrojando una respuesta óptima.
- Escalabilidad: El generador debe trabajar con mapas de diferentes tamaños, por lo que el sistema debe poder soportar cualquier tamaño de entrada.
- Usabilidad: La interfaz gráfica debe ser lo más sencilla posible, evitando que el usuario tenga que realizar tareas tediosas a la hora de construir mapas.

## 3.4. Diseño del sistema

Una vez definidos los requisitos se ha procedido a realizar el diseño software del sistema en cuestión, empezando a concretar la arquitectura propuesta y luego desarrollando los casos de uso y diagramas de clases.

### 3.4.1. Arquitectura software

Debido a que el sistema cuenta con una interfaz gráfica y un módulo que se encargará de generar el mapa, el sistema estará dividido en dos partes concretas tal y como se puede ver en la Figura 3.1:

- Una parte que será la aplicación gráfica, que actuará en todo momento como *Front-end* de cara al usuario. Esta parte sigue la estructura Modelo-Vista-Controlador (MVC), en donde el controlador se encargará de hacer de puente entre la interfaz gráfica, que es lo que manipulará el usuario, y los datos guardados en memoria. Así mismo proporcionará

las funcionalidades básicas del sistema como guardar o cargar el mapa y crear un mapa en blanco.

- La segunda parte actuará como *Back-end*, que se encargará de generar el mapa mediante un programa lógico escrito en ASP que se ejecutará Clingo. Contiene un controlador que se encargará de hacer la llamada a Clingo y de obtener su resultado, y luego otro controlador que se encargará de llamar primeramente al programa lógico que genere las regiones y luego, dada una región, rellene las casillas de la región. Para ello primeramente define que zonas son tierra y que zonas son agua, y con las zonas con tierra se rellenan con cordilleras y con áreas bióticas (las cuales son zonas grandes que contienen un solo tipo de terreno). Una vez realizado terminará generando los mares y los puntos de inicio para los jugadores.

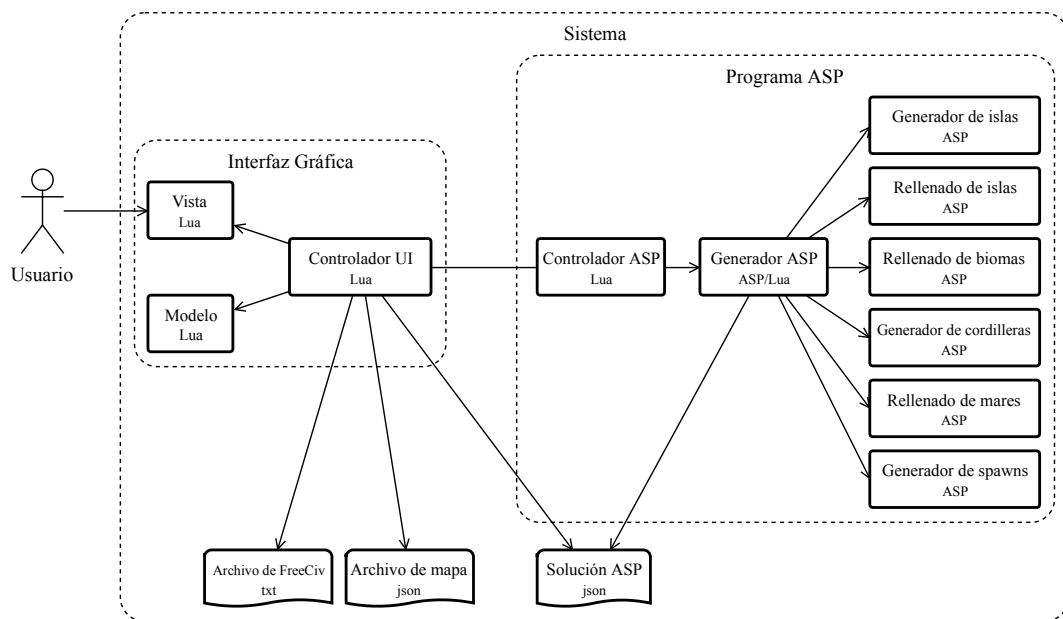


Figura 3.1: Arquitectura del sistema

Tanto el controlador de la interfaz de usuario como el controlador del programa ASP se ejecutarán en paralelo, pudiendo enviarse información de un controlador a otro para saber cuando hay que empezar una generación o si esta terminó. A pesar de esto, los resultados de la generación de ASP se guardarán en un archivo intermedio para evitar enviar gran cantidad de datos entre

los elementos.

### 3.4.2. Casos de uso

El sistema tiene en cuenta que se usará en todo momento por un único usuario, el cual llevará a cabo todas las funcionalidades propuestas en la Sección 3.3.1 a través de una interfaz gráfica que se explica en la Sección 3.4.3. Estos requisitos funcionales se transforman, por tanto, en los casos de uso del sistema que se proponen en la Figura 3.2.

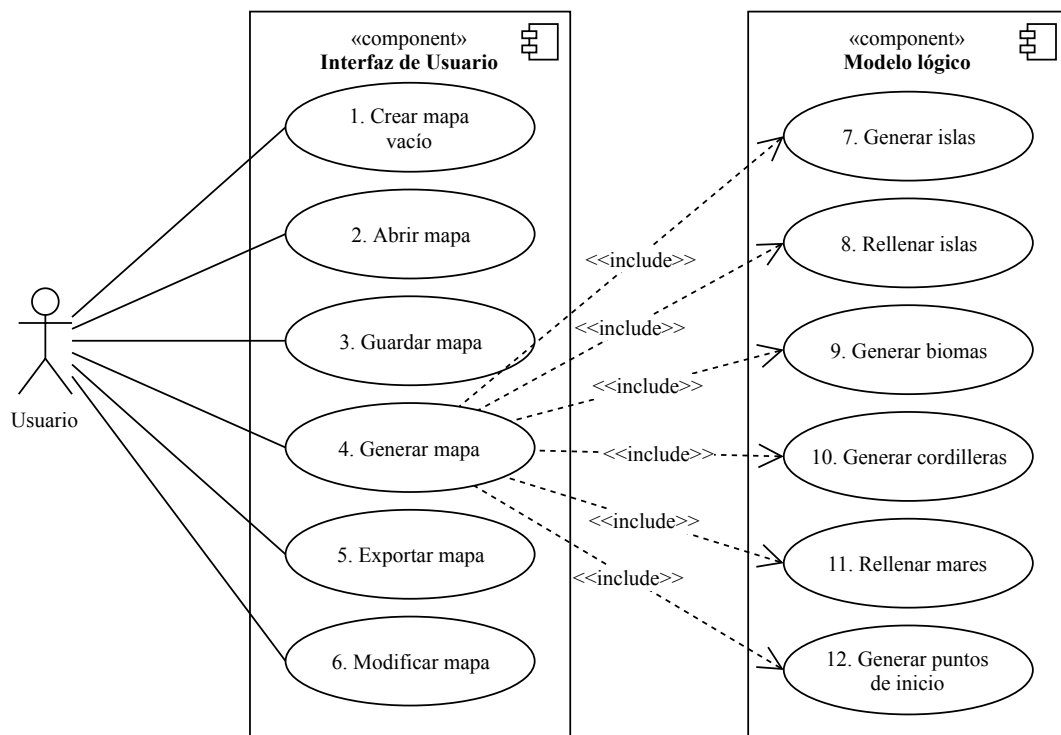


Figura 3.2: Casos de uso del sistema propuesto

### 3.4.3. Pantallas del sistema

El proyecto propuesto está pensado para ser usado a través de una interfaz gráfica de usuario, la cual es modelada mediante un patrón MVC como se indica en la Sección 3.4.1. Esta interfaz está dividida en varias vistas, las cuales corresponden con los casos de uso propuestos contra los que el usuario interacciona directamente en la Sección 3.4.2.

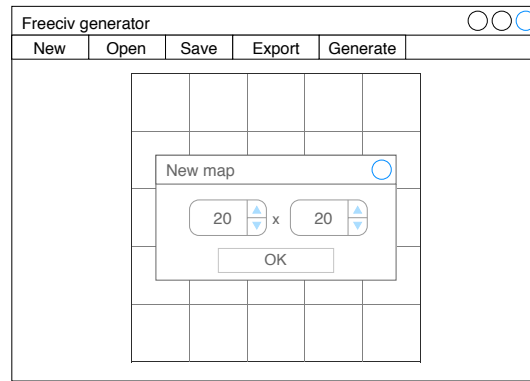


Figura 3.3: Pantalla de nuevo mapa

- 1. Crear mapa vacío: Se acciona cuando el usuario presiona en el botón de “Crear mapa” en la barra de herramientas, lo que despliega una ventana emergente parecida a la Figura 3.3. Aquí el usuario puede indicar el tamaño del mapa a crear y pulsar el botón de “Aceptar”. Con esto el sistema muestra una rejilla vacía.

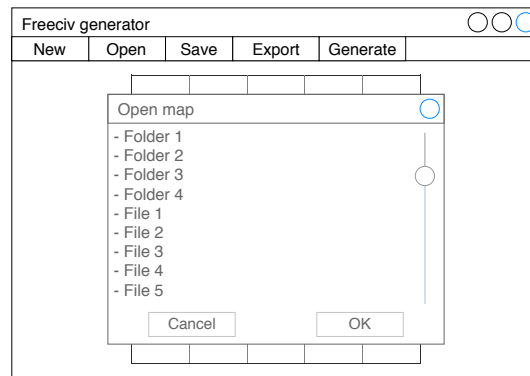


Figura 3.4: Pantalla de abrir archivo de mapa

- 2. Abrir mapa guardado en disco: Se inicia cuando el usuario presiona en el botón de “Abrir mapa” en la barra de herramientas, lo que despliega una ventana emergente parecida a la Figura 3.4. Aquí el usuario navega a través de una lista que representa los archivos guardados en disco y selecciona un archivo anteriormente guardado por la aplicación que representa un mapa. El sistema procede a cargarlo y mostrar los datos del mapa en la rejilla principal.



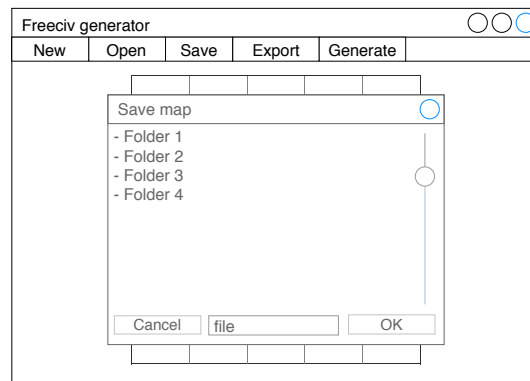


Figura 3.5: Pantalla de guardar mapa

- 3. Guardar mapa en disco: Se inicia cuando el usuario presiona en el botón de “Guardar mapa” en la barra de herramientas, mostrando una ventana emergente como la de la Figura 3.5. Aquí el usuario navega por las carpetas guardadas en disco a través de una lista y selecciona una donde se guardará. Además, escribe el nombre del nuevo archivo en un campo de texto debajo de esta lista y acepta. El sistema procede a transformar los datos del mapa en un fichero que se guarda en disco donde el usuario indicó.

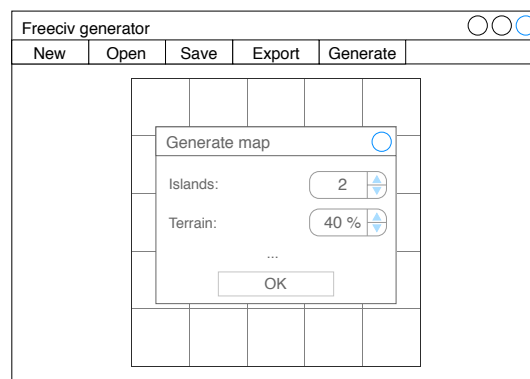


Figura 3.6: Pantalla de generar mapa

- 4. Generar mapa: Se inicia cuando el usuario acciona el botón de “Generar mapa” en la barra de herramientas, haciendo que el sistema muestre una ventana emergente parecida a la Figura 3.6. Aquí el usuario indica los parámetros que prefiera en la generación y acepta. En sistema procede a llamar a la parte del programa lógico creado en ASP, la cual va

ejecutando cada uno de los casos de uso. Entre medias, el sistema lanza una vista como la de la Figura 3.7 para proporcionar retro-alimentación al usuario. Una vez acabada la generación, el sistema muestra en la rejilla principal con el mapa que se ha generado finalmente.

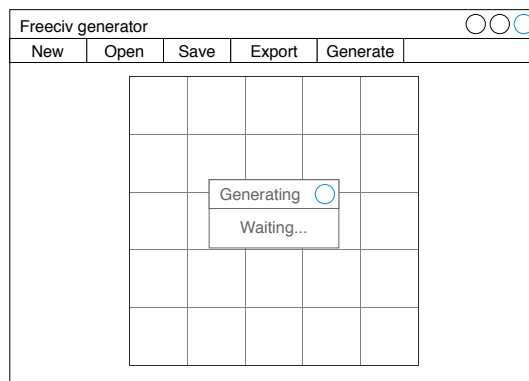


Figura 3.7: Pantalla con el mensaje de espera

- 5. Exportar mapa al formato de Freeciv: Se inicia cuando el usuario presiona en el botón de “Exportar mapa” en la barra de herramientas, mostrando una ventana emergente como la de la Figura 3.8. Aquí el usuario navega por las carpetas guardadas en disco a través de una lista y selecciona una donde se guardará. Además, escribe el nombre del nuevo archivo en un campo de texto debajo de esta lista y acepta. El sistema procede a transformar los datos del mapa en un fichero reconocible por Freeciv y se guarda en disco donde el usuario indicó.

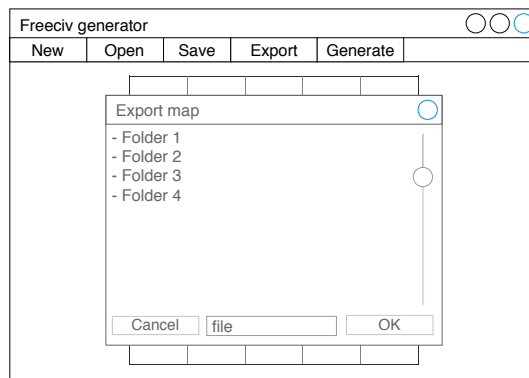


Figura 3.8: Pantalla de exportar mapa

- 6. Modificar mapa: Se inicia cuando el usuario presiona sobre una de las celdas de la rejilla como en la Figura 3.9. El sistema pasará a cambiar el terreno de la celda por otro, realizando un ciclo en el momento en el que no quede ninguna opción nueva.

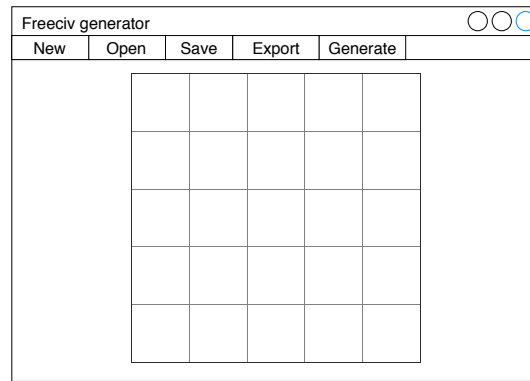


Figura 3.9: Pantalla principal de la aplicación

#### 3.4.4. Implementación

Una vez analizado el proyecto en cuestión, pasaré a detallar el diseño de cada uno de los diferentes componentes del sistema en su construcción, indicando mediante diagramas como se integran los diferentes elementos.

Como ya se indicó en la Sección 2.2.2, el lenguaje de programación Lua no tiene un paradigma de programación orientado a objetos basados en clases, más se ha preferido que en la realización de este sistema se use el módulo *class.lua* de la biblioteca *HUMP*<sup>1</sup> para una organización lo más estructurada posible. Así mismo, también hay que destacar que el lenguaje tampoco soporta la manipulación de archivos en formato JSON, por lo que se ha usado el módulo *json.lua*<sup>2</sup>, el cual permite transformar un texto en formato JSON a tipos compatibles en Lua.

Por último, indicar que para la realización de las vistas de la interfaz de

---

<sup>1</sup><https://github.com/vrld/hump>

<sup>2</sup><https://github.com/rxi/json.lua>

usuario se ha usado la biblioteca *SUIT*<sup>3</sup>, que permite generar una interfaz gráfica en modo inmediato de forma sencilla sobre LÖVE, pudiendo realizar un prototipo de los elementos más rápidamente y con más flexibilidad que con otro tipo de bibliotecas gráficas, como puede ser GTK+<sup>4</sup>. Por otra parte, habrá elementos gráficos más complejos (como pueden ser diálogos de selección de ficheros o ventanas emergentes) que se tendrán que generar a mano, tal y como se expone en la Sección 3.4.4.

#### Clase principal

Como se puede ver en la Figura 3.10 se ha diseñado la clase principal **Main**, la cual es una clase estática que sirve de controlador para las vistas y el modelo. Implementa los casos de uso de la interfaz descritos en la Sección 3.4.2 en los métodos `_newMap`, `_openMap`, `_saveMap`, `_exportMap` y `_generateMap`, que son eventos que se llaman desde los diálogos emergentes tal y como se explica desde la Sección 3.4.4. Para el caso de uso de modificar el mapa, la tarea recae en la clase **Editor**, que se explica en detalle en la Sección 3.4.4.

Con respecto al motor gráfico LÖVE, este usa varias funciones predefinidas cuando se activan distintos eventos, los cuales llaman a los métodos públicos de la clase principal:

- El método `init` es llamado en la función `love.load` la primera vez que se ejecuta el programa. Sirve para cargar los elementos gráficos e iniciar el resto de clases que serán usadas por el programa.
- Como LÖVE es un motor para programación de videojuegos, la función `love.update` es llamada en un bucle de eventos internos antes de actualizar la pantalla. Esta función llama al método homónimo de la clase principal, el cual prepara las vistas de la interfaz para su pintado en pantalla y llama a los métodos correspondientes según los eventos producidos en las vistas.

---

<sup>3</sup><https://github.com/vrld/SUIT>

<sup>4</sup><https://www.gtk.org/>

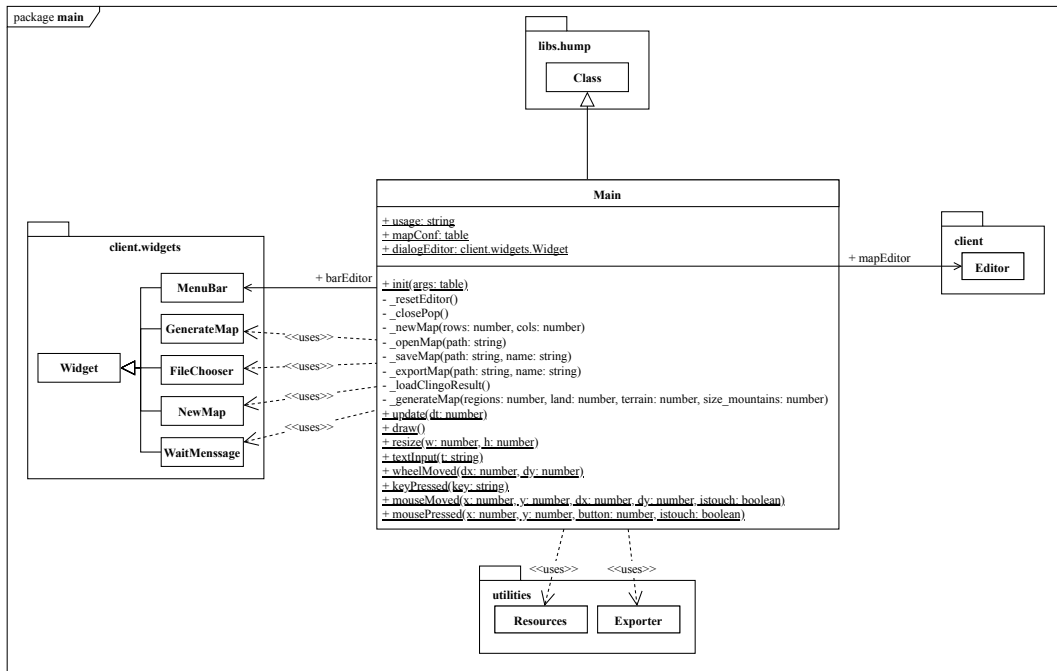


Figura 3.10: Diagrama de clases del paquete principal

- Una vez actualizada la lógica, se procede al pintado de pantalla mediante la función `love.draw`, la cual llama al método con el mismo nombre de la clase principal, que se encarga de pintar las vistas en pantalla.
- Existen otros eventos, los cuales LÖVE tiene contempladas varias funciones por defecto adicionales que se lanzan para contestar a estos. Es el caso de cuando se redimensiona la ventana (`love.resize`), se introduce un texto mediante un teclado virtual (`love.textinput`), se realiza un movimiento de la rueda del ratón (`love.wheelmoved`), se presiona una tecla (`love.keypressed`), se mueve el ratón (`love.mousemoved`) o se presiona un botón del ratón (`love.mousepressed`). Estas llaman a los métodos correspondientes de la clase principal.

## Editor gráfico

La clase **Editor** sirve como un controlador para responder a las modificaciones y cambios de representación del mapa, ya sea cuando se actualiza este mediante uno de los casos de uso que recoge la clase principal [ver Sección

3.4.4] o cuando el usuario procede a la modificación del mapa, tal y como se explica en la Sección 3.4.2.

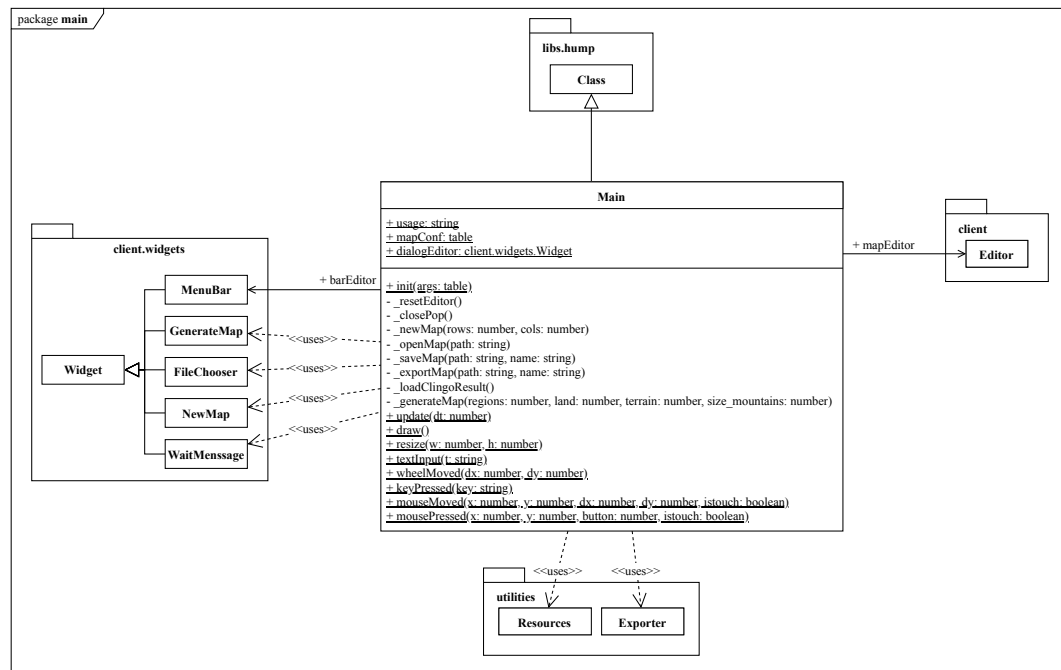


Figura 3.11: Diagrama de clases del paquete Client

Para ello, siguiendo el patrón decorador, esta se apoya en la clase **MapDecorator**, la cual se encarga de actualizar la vista del mapa, que es representada mediante una rejilla con los distintos terrenos mediante el objeto `love.SpriteBatch`, el cual es un mapa de *tiles* bidimensional. Debido a que hay terrenos que contienen diferentes imágenes para las posiciones de un *tile*, **MapDecorator** contiene varios métodos que permiten discretizarlas conociendo los vecinos de una celda. Finalmente esta clase llama al modelo en si, representado por la clase **Map**, la cual tiene una representación del mapa en forma de tabla.

Finalmente todas estas clases usan constantes que están definidas el módulo **Constants**.

### Elemento gráficos complejos

Como ya expliqué en la entrada de la Sección 3.4.4, debido a las limitaciones de la biblioteca *SUIT*, hay varios elementos gráficos que no se incluyen en ella

debido a que son complejos y no es del ámbito de esta biblioteca, es por eso que se han realizado a mano. Estos elementos tienen su propio controlador para poder usarlos más fácilmente dentro del programa.

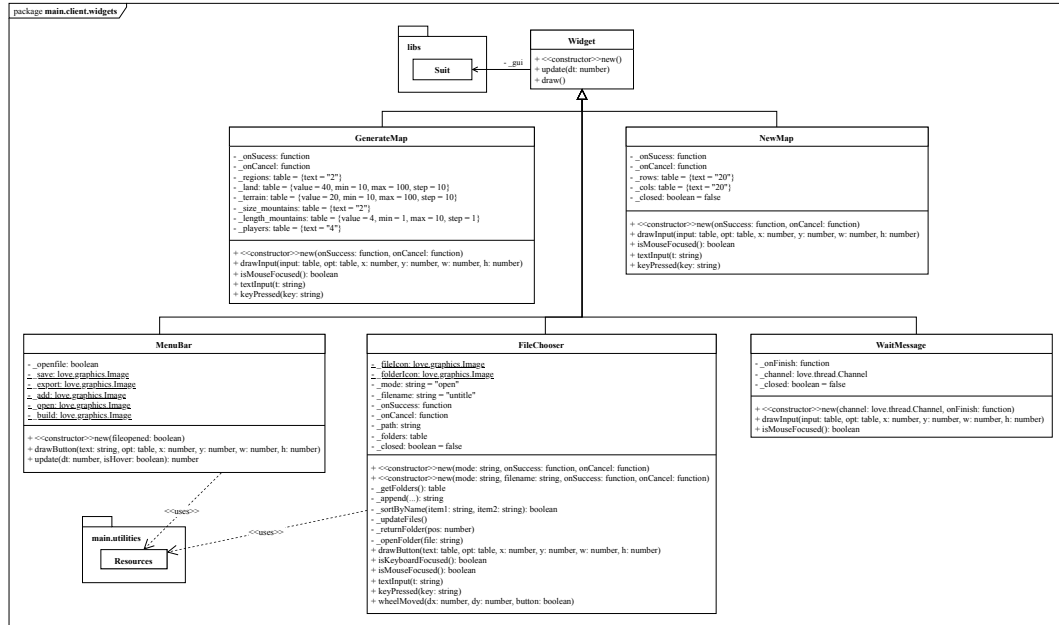


Figura 3.12: Diagrama de clases del paquete Widget

- Para la barra de herramientas se ha realizado la clase **MenuBar**, la cual controla y muestra la lista de botones que accionan los principales casos de uso. Al constructor se le pasa las funciones para los eventos de aceptar y cancelar diálogo. Tiene un método **update** que devuelve un número que se corresponde con el botón pulsado en la barra.
- La clase **NewMap** controla y muestra la ventana emergente que se puede ver en la Figura 3.3. Al constructor se le pasa las funciones para los eventos de aceptar y cancelar diálogo. Contiene el método **isMouseFocused** que indica si el ratón está encima de la ventana, y los métodos **textInput** y **keyPressed** para introducir texto en las entradas de la ventana.
- La clase **GenerateMap** controla y muestra la ventana emergente con el diálogo de generar mapa, tal y como se puede ver en la Figura 3.6. Contiene los mismos métodos que la clase **NewMap**.

- La clase `WaitMessage` controla y muestra la ventana emergente que se puede ver en la Figura 3.7. Al constructor se le pasa el canal del *thread* que se usa en la generación del mapa, tal y como se expone en la Sección 3.4.4, y la función que se lanza cuando la generación termina. Contiene el método `isMouseFocused` que indica si el ratón está encima de la ventana.
- La clase `FileChooser` controla y muestra la ventana emergente correspondiente a un diálogo de selección de archivo, tal y como se puede ver en las Figuras 3.4, 3.5 y 3.8. Contiene varios métodos privados para la ordenación y la apertura de carpetas en disco, así como el método `isMouseFocused` que indica si el ratón está encima de la ventana, los métodos `textInput` y `keyPressed` para introducir texto en las entradas de la ventana y el método `wheelMoved` que permite subir y bajar la lista de carpetas y ficheros mediante la rueda del ratón.

### Generador de mapas



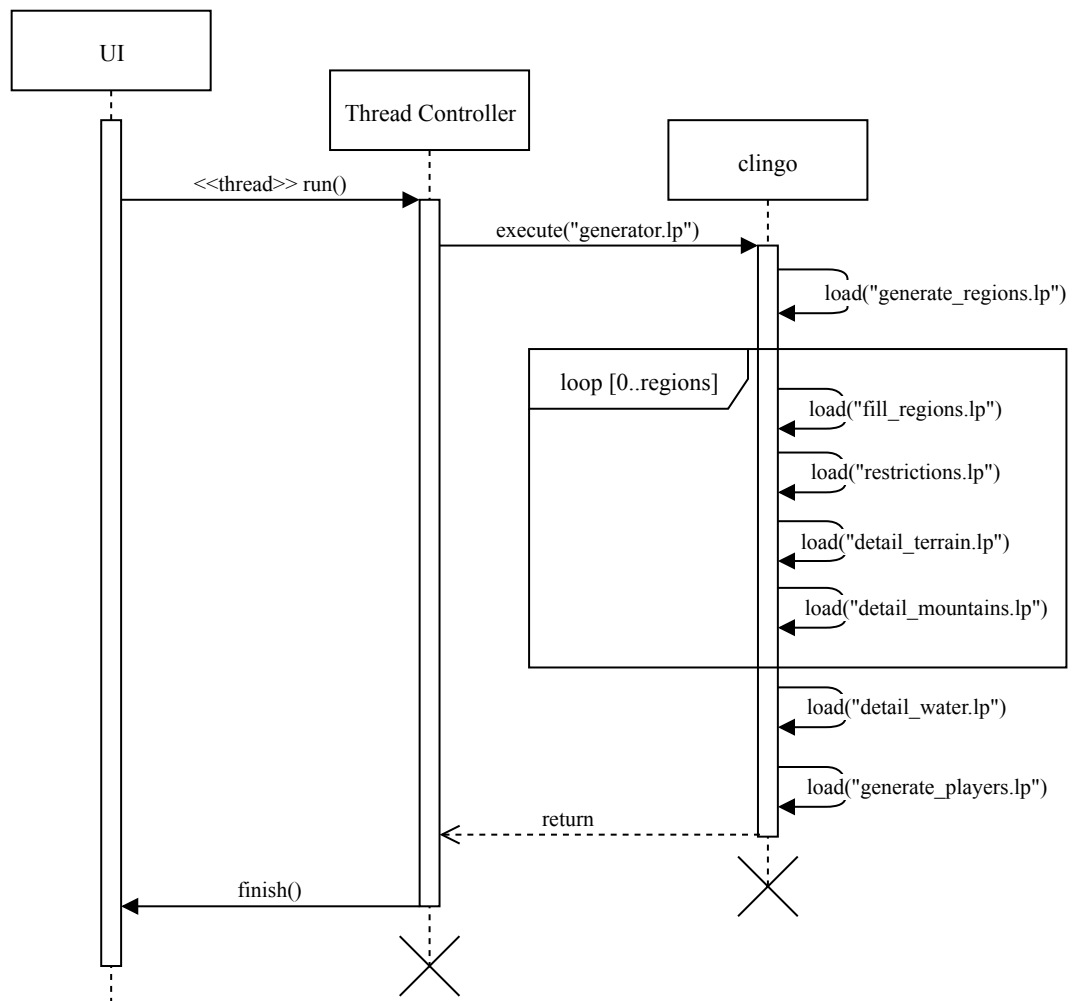


Figura 3.13: Diagrama de secuencia de la ejecución del generador



## Capítulo 4

### Evaluación



## Capítulo 5

## Conclusiones



# Índice de tablas

3.1. Tipos de terreno . . . . .	19
3.2. Tipos de terreno . . . . .	20





# Índice de figuras

2.1. Ejemplo de ejecución de ASP . . . . .	8
3.1. Arquitectura del sistema . . . . .	22
3.2. Casos de uso del sistema propuesto . . . . .	23
3.3. Pantalla de nuevo mapa . . . . .	24
3.4. Pantalla de abrir archivo de mapa . . . . .	24
3.5. Pantalla de guardar mapa . . . . .	25
3.6. Pantalla de generar mapa . . . . .	25
3.7. Pantalla con el mensaje de espera . . . . .	26
3.8. Pantalla de exportar mapa . . . . .	26
3.9. Pantalla principal de la aplicación . . . . .	27
3.10. Diagrama de clases del paquete principal . . . . .	29
3.11. Diagrama de clases del paquete <b>Client</b> . . . . .	30
3.12. Diagrama de clases del paquete <b>Widget</b> . . . . .	31
3.13. Diagrama de secuencia de la ejecución del generador . . . . .	33



# Bibliografía

- [1] “Service name and transport protocol port number registry,” IANA, Jan 2006. [Online]. Disponible en: <https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml?search=5556>
- [2] “Gnu general public license,” Free Software Foundation. [Online]. Disponible en: <http://www.gnu.org/licenses/gpl.html>
- [3] M. Gelfond y V. Lifschitz, “The stable model semantics for logic programming.” en *Proceedings of International Logic Programming Conference and Symposium*, vol. 88, 1988, pp. 1070–1080.
- [4] S. Hanks y D. McDermott, “Nonmonotonic logic and temporal projection,” *Artificial Intelligence*, vol. 33, n.º 3, pp. 379 – 412, 1987. [Online]. Disponible en: <http://www.sciencedirect.com/science/article/pii/0004370287900439>
- [5] R. Ierusalimschy, *Programming in Lua, Fourth Edition*. Lua.Org, 2016.
- [6] “The JSON Data Interchange Format,” ECMA, Tech. Rep. Standard ECMA-404 1st Edition / October 2013, Oct 2013. [Online]. Disponible en: <http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf>