Trabajo de Fin de Grado

Back-end de *Progrezz*

Progrezz's back-end

Daniel Herzog Cruz

La Laguna, 9 de junio de 2015

D. Carina Soledad González González, con N.I.F. 54.064.251-Z profesora Titular de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutora

D. Luis Gonzalo Aller Arias, con N.I.F. 10.081.577-X director de proyectos de la empresa Improve Change S.L., como cotutor

C E R T I F I C A (N)

Que la presente memoria titulada:

"Back-end de Progrezz."

ha sido realizada bajo su dirección por D. Daniel Herzog Cruz, con N.I.F. 42.224.763-Y.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 9 de junio de 2015.

Agradecimientos

En primer lugar, agradecer a Luis Gonzalo Aller Arias por brindarme la gran oportunidad de poder trabajar de manera altruista. Todo sea por una buena causa social.

Asimismo, agradezco a mi compañero y amigo Cristo por haberme ofrecido un espacio de trabajo tan cómodo (desayunos incluidos), en un ambiente agradable e ideal para trabajar.

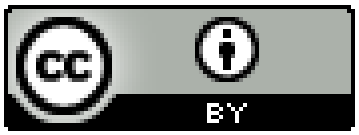
Agradecer también a Jorge García del Arco su afán y buena visión del proyecto, brindándonos acceso a eventos y meetings para hablar o desarrollar partes del mismo, y a mi directora de proyecto Carina Soledad González González por su apoyo y emoción por los resultados obtenidos en el proyecto.

Además, gracias a Alby Ojeda por dedicar su tiempo como guionista del juego, además de ayudarnos a tomar decisiones sobre jugabilidad y usabilidad.

Gracias también a mi familia y amigos por su gran apoyo incondicional a lo largo del desarrollo de este trabajo, en esos momentos difíciles de desgana y de aversión al trabajo.

A todos aquellos que han hecho posible el desarrollo de este proyecto, muchas gracias.

Licencia



Resumen

Los videojuegos serios son un subconjunto que define a aquellos cuyo propósito va más allá que el de entretener; educación, mejora de la sociedad,... Progrezz es un videojuego serio geo-localizado que busca hacer de este mundo un sitio mejor por medio de búsqueda de voluntariado y la realización y premiación de acciones positivas.

El objetivo del desarrollo de este proyecto fue diseñar, analizar e implementar los aspectos que componen el núcleo del servidor o back-end, como una aplicación web, partiendo de un prototipo inicial carente de usabilidad.

El trabajo se ha estructurado en una serie de apartados que se han desarrollado en orden, incluyendo diseño, análisis y desarrollo, basándose fuertemente en el modelo de integración continua: preparación del prototipo existente, desarrollo del sistema de gestión de usuarios, desarrollo del sistema de gestión de objetos y desarrollo del sistema de gestión de eventos sociales.

Se ha logrado completar todos los objetivos especificados en el Proyecto del Trabajo de Fin de Grado en tiempo y forma, cumpliendo con la estimación inicial de tiempo.

En conclusión, el haber trabajado en este proyecto ha conseguido ampliar mis horizontes, permitiendo conocer y trabajar junto a profesionales de gran talento relacionados con proyectos de esta índole. Además, se ha recabado información sobre los juegos serios y los proyectos de gran tamaño; éstos necesitan de mucho esfuerzo y apoyo por parte de la comunidad para que sean mínimamente viables.

Palabras clave: *Progrezz, juego serio, back-end, servidor, aplicación web, trabajo de fin de grado.*

Abstract

Serious games or applied games are a group of games defining those whose purpose goes further than entertainment; educational, society improvement ... Progrezz is a serious and geo-located game whose goal is making this world a better place through voluntary help and the realization or rewarding of positive actions.

The main objective of the development phase of this project was to design, analyze and implement the modules that establish the core of the back-end service as a web application, starting from an initial prototype lacking of usability.

The project has been split into sections that have been developed in order, including design, analysis and implementation, strongly based on the model of continuous integration: preparation of the existing prototype, development of the user management system, development of the items management system, development of the social events management system.

All the objectives and goals specified in the grade dissertation's draft have been achieved on time, based on the initial time estimation.

In conclusion, working on this project has managed to broaden my horizon, allowing me to meet and work with talented professionals related to similar projects. It has also been helpful to acquire knowledge about serious games and large projects; a lot of effort and support if needed from the community for the project to make it minimally viable.

Keywords: *Progrezz, serious game, applied game, back-end, server, web-app, web application, grade dissertation.*

Índice General

Índice de figuras

Índice de tablas

Capítulo 1. Introducción

Resumen del proyecto

Los juegos serios se caracterizan por ir más allá del juego en sí, ya que buscan potenciar elementos tan básicos como la educación, la publicidad, la sanidad, la sociabilidad,... Intentando llegar a un grupo de usuarios por medio de un producto atractivo y usable: un videojuego.

Progrezz es un videojuego serio, social y libre (bajo la licencia MIT) para apoyar una causa social: tratar de mejorar la interacción entre las personas, promoviendo proyectos sociales para desarrollar una comunidad solidaria y sostenible, mediante mini juegos y recompensas, haciendo uso de geo-localización y realidad aumentada.

En este mismo ámbito, Progrezz se define como un videojuego social pensado para smartphones diseñado como una aplicación web que busca hacer un mundo mejor, mediante acciones sociales como prestación de servicios, ayudas económicas, búsqueda de voluntariado, cualquier otro aspecto que pueda ser de utilidad. Todo esto puede ser logrado mediante una estética e historia bastante simple para el juego, en la que la energía negativa denominada entropía representa la problemática social actual, que puede ser combatida y evitada por medio de acciones sociales solidarias, potenciando el bienestar social, siendo el usuario o jugador un voluntario que busca combatir por dicha causa.

Al ser un videojuego social, requiere de una parte centralizada o back-end[1] para poder operar correctamente, además de permitir una interacción completa entre los distintos usuarios. Como cualquier otro videojuego, el servidor se encargará de procesar todas las peticiones de los usuarios respetando las mecánicas del juego, además de realizar las interacciones oportunas entre los

[1] Servicio o parte centralizada de un producto software disponible para los clientes.

usuarios y los elementos virtuales (minerales geo-localizados, otros usuarios, mini juegos,...).

Por tanto, el objetivo del trabajo de fin de grado es diseñar, analizar y codificar el contenido necesario para poder operar correcta y eficientemente en el lado del servidor, haciendo uso de las tecnologías necesarias, como la geo-localización o la programación modular, creando un prototipo inicial del juego más avanzado respecto al estado actual del proyecto. En resumen, se trata de desarrollar un segundo prototipo mejorable que permita realizar más acciones que las que se han implementado hasta la fecha.

1.2 Estructura del documento

Para una mayor comodidad, se estructurará la documentación del desarrollo en tantos apartados como módulos de tareas se deben desarrollar:

1. Introducción al documento.
2. Desarrollo; Preparación del contenido existente: adaptar el prototipo actual de manera debida para poder integrar nuevas funcionalidades.
3. Desarrollo; Gestión de usuarios: todo lo relativo al desarrollo de la parte de gestión de usuarios o jugadores.
4. Desarrollo; Gestión de objetos: todo lo relativo al desarrollo de la parte de gestión de objetos, inventario y depósito.
5. Desarrollo; Gestión de eventos sociales: desarrollo del sistema de balizas para crear puntos de energía.
6. Conclusiones del proyecto: conclusiones extraídas de la ejecución del trabajo de fin de grado.
7. Apéndices del documento.
8. Bibliografía del documento.

Metodología de trabajo

Para una mayor agilidad y simplicidad en las actividades a realizar, se utilizará una metodología ágil tipo Scrum[2], pero de una manera simplificada, usando sprints[3] de 1 o 2 semanas.

1.3 Los pasos a seguir para llevar a cabo la metodología de planificación son los siguientes:

1. Reunión de planificación del sprint: Decidir cuál será la próxima tarea a realizar bajo supervisión del tutor o cotutor.
2. Diseño, análisis y desarrollo: Llevar a cabo la tarea marcada en la fase de planificación.
3. Solución de problemas: Arreglar problemas y fallos encontrados en la fase desarrollada.
4. Reunión de planificación del sprint: Suele ser en la misma sesión que la reunión de planificación. Se comprobará que el trabajo realizado es válido.

Finalmente, para trabajar sobre los repositorios git, se usará un modelo *git feature branches* o (ramas por características o adiciones), tal como se muestra en la Figura 1.1.

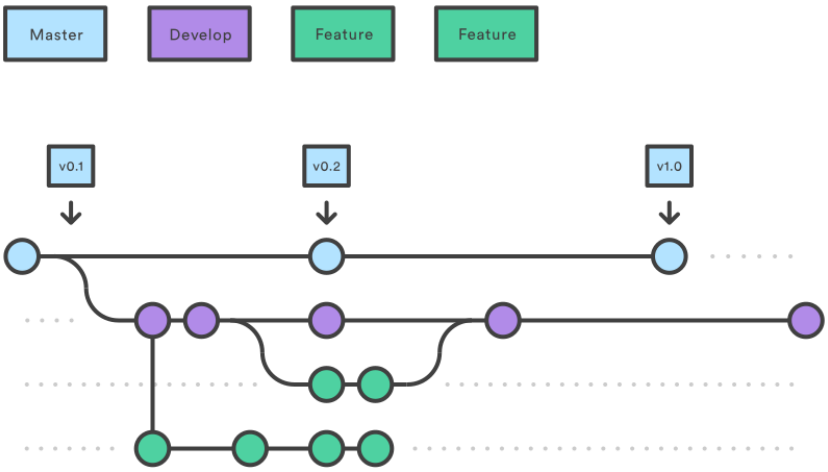


Figura 1.1. Modelo git del desarrollo del proyecto.

[2] Metodología ágil basada en desarrollo y en el solapamiento de las diferentes fases de desarrollo.

[3] Ciclos de corta duración para planificar y revisar.

Capítulo 2. Preparación

2.1 Antecedentes del proyecto

El proceso para llevar a cabo un proyecto de semejante envergadura para que sea usable consiste en producir software de manera evolutiva, de tal forma que haya transiciones entre prototipos o versiones del producto, y pueda mejorarse poco a poco, y con ayuda de toda la comunidad implicada en el proyecto.

Así mismo, el proyecto está en una fase muy temprana, en un primer prototipo inicial, realizado en un *hackathon*[4] el 13 y 14 de diciembre de 2014. El servidor de dicho prototipo fue enfocado para interaccionar con mensajes geo-localizados de manera sencilla, pudiendo tener un cliente y un servidor funcional. En este punto la aplicación es capaz de realizar las siguientes acciones:

- Almacenar mensajes geo-localizados encontrados por cada usuario.
- Enviar y recibir mensajes personalizados de usuarios.
- Obtener de manera simple mensajes próximos a un punto dado (geo-localización) para que puedan ser recolectados por los jugadores.
- Generar mensajes fragmentados, de tal manera que el usuario pueda acceder al contenido del mensaje siempre y cuando haya obtenido todos los fragmentos.

Para cumplir con todas esas funciones se hace uso de las siguientes tecnologías (abierto a cualquier nueva posibilidad):

[4] Evento de pequeño o gran tamaño de corta duración (uno o dos días) para intentar crear un producto usable lo antes posible.

- Sinatra[5] bajo el intérprete de Ruby[6] 2.2 como plataforma de ejecución del servidor.
- ORM[7] DataMapper[8] para gestionar las bases de datos como programación orientada a objetos u OOP.
- PostgreSQL[9] como gestor de bases de datos para almacenar todo el contenido de los usuarios, mensajes, etc.
- API[10] REST[11] desarrollada por el equipo Progrezz para enviar y recibir la información necesaria por el cliente, referente al juego y al estado del mismo.

Se ha desarrollado con una estructura modular[12], de tal manera que incorporar nuevos módulos sea sencillo y abierto para toda la comunidad del software libre, usando conectores que permitan multitud de lenguajes de programación.

El siguiente objetivo del proyecto es realizar un prototipo que pueda ser jugable, proporcionando a los usuarios una serie de mecánicas de juego que hagan atractivo al mismo, y modificando todo aquello que sea necesario.

2.2 Problemas encontrados

El prototipo inicial no es perfecto, ya que se han encontrado una serie de problemas graves que podrían confligir con las nuevas implementaciones del proyecto.

---

[5] Librería para definir servidores HTTP de manera sencilla.

[6] Lenguaje de programación interpretado.

[7] Conector para relacionar objetos de un lenguaje de programación con una base de datos relacional (tablas).

[8] ORM para Ruby y PostgreSQL.

[9] Motor de bases de datos basado en un modelo relacional.

[10] Conjunto de funciones que ofrece una biblioteca software.

[11] Arquitectura software basada en el protocolo HTTP que permite la comunicación entre un cliente y un servidor.

[12] Estructura que define que cada componente debe ser un módulo encapsulado e independiente.

Las más destacadas son las siguientes:

1. Modelo de base de datos complejo: demasiadas tablas para una funcionalidad tan reducida. Además, las relaciones entre las distintas tablas no están correctamente codificadas, por lo que no funcionan.
2. Motor de bases de datos obsoleto: se ha considerado que PostgreSQL bajo DataMapper está desactualizado, dando problemas notorios de eficiencia.
3. Estructura poco modular para futuras implementaciones: es necesario estructurar de manera más concisa las entidades creadas; esto es, crear más ficheros de código fuente, más espacios de nombres[13], más clases[14], etcétera.
4. Módulo de geo-localización ineficiente: para buscar elementos en un área circular determinada, el sistema iteraba por todos los elementos geo-localizados, buscando aquellos que estuviesen los suficientemente cerca. Es un diseño inadecuado, ya que, cuanto mayor sea la base de datos, más tardará en completar este tipo de consultas. Por ello, se usará un modelo completamente distinto.

2.3 Nuevas tecnologías

Se utilizarán las siguientes tecnologías para varios aspectos de la aplicación:

- Neo4j[15] como nuevo motor de bases de datos. Para conectarlo con Ruby, se utilizará el OGM[16] neo4jrb[17], permitiendo definir clases o modelos de objetos de la base de datos, además de ofrecer tiempos de ejecución de consultas moderadamente bajos.

[13] Entidad de un lenguaje de programación que permite agrupar elementos bajo un mismo identificador.

[14] Entidad software usada para crear objetos bajo un mismo modelo o plantilla.

[15] Motor de bases de datos basada en grafos.

[16] Conector para relacionar objetos de un lenguaje de programación con una base de datos basada en grafos.

[17] Conector OGM entre Neo4j y Ruby.

- progrezz-geolocation como módulo de geo-localización. Utiliza una extensión Ruby-C para una mayor eficiencia.
- Ejecución de scripts[18] del sistema. Para mayor modularidad, los desarrolladores podrán programar en cualquier lenguaje, y posteriormente podrán llamar al script o ejecutable desde el back-end.
- Documentación vía *yardoc*: Generador de documentación de código automático.

2.4 Refactorización

Se ha reestructurado todo el código fuente para tener un árbol de directorios más coherente y organizado.

La nueva estructura se muestra de manera resumida en la Figura 2.1.

```
progrezz-server         # Directorio raíz
|- data                 # Datos a cargar (por ejemplo: ficheros JSON)
|- public               # Ficheros estáticos web
|- rb                   # Código fuente Ruby
|  |- db                # Ficheros de la base de datos
|  |  |- objects        # Modelos de objetos de la base de datos
|  |  |- relations      # Modelos de relaciones de la base de datos
|  |- game              # Entidades referentes al juego
|  |  |- api            # APIs de acceso del cliente
|  |  |- mechanics      # Gestores de mecánicas de juego
|  |- managers          # Gestores del back-end
|- scripts              # Scripts externos
|  |- python            # Scripts de python
|- test                 # Directorio de pruebas (unitarias)
|- tmp                  # Carpeta temporal (logs, avisos, ...)
|- views                # Directorio de vistas o plantillas web
```

Figura 2.1. Árbol de directorios.

La mayor parte de los ficheros han sido modificados en gran medida para aportar una mayor modularidad.

Principalmente, se ha creado la clase `Sinatra::ProgrezzServer` (en el documento *main.rb*) como una aplicación Sinatra modular, permitiendo así

[18] Ficheros de código ejecutables (interpretables).

dividir en una gran cantidad de ficheros los distintos módulos de peticiones web al servidor.

Por comodidad, se han creado el fichero rakefile[19] para ejecutar tareas rutinarias, tales como lanzar el servidor, generar documentación de código, etcétera.

Además, se ha añadido *pry*[20] en modo desarrollo, que permite ejecutar comandos e interactuar con el servidor sin necesidad de usar un explorador.

Finalmente, las peticiones REST al servidor se han trasladado a un módulo aparte, formando, junto con los WebSockets, la API de comunicación entre cliente y servidor.

Ajustes del sistema gestor de la base de datos

2.5 La mecánica principal del prototipo era clara y sencilla: que los usuarios pudiesen compartir mensajes geo-localizados, y que estos pudiesen estar fragmentados, teniendo que recolectar todas las partes del mensaje para poder leer el contenido del mismo.

En el modelo original de la base de datos, hacían falta demasiadas tablas para que fuese medianamente viable.

Para evitar tal complejidad ahora y en futuras actualizaciones, se ha optado por usar otro tipo de bases de datos más efectivo para este proyecto: bases de datos basadas en grafos.

Neo4j se caracteriza por ser un motor de bases de datos innovador que incluso permite representar la información de manera visual. Los registros se corresponden con nodos en el grafo. Las relaciones, en cambio, conectan dichos nodos mediante etiquetas: por ejemplo, *A* (nodo) *es amigo de* (relación) *B* (nodo).

Así pues, usando el conector neo4jrb, se pueden redefinir las clases de la base de datos para compatibilizarlo con Neo4j.

---

[19] Fichero Ruby para ejecutar tareas bien definidas.

[20] Consola interactiva de Ruby en tiempo de ejecución.

Como Neo4j está fuertemente basado en ActiveRecord[21] y DataMapper, sólo ha hecho falta cambiar aspectos menores. Las diferencias son minúsculas, ya que son conectores y funcionan de manera idéntica. Basta con cambiar algunos identificadores (nombres de métodos estáticos o módulos incluidos) para hacerlo funcional.

Además, se han eliminado atributos duplicados en la base de datos (ya que las relaciones no funcionaban) y se han sustituido debidamente por relaciones.

Existen dos tipos de relaciones para los diversos nodos:

- Relaciones predefinidas: Se crean clases o modelos para definir debidamente la estructura de una relación. En el proyecto, se encuentran en la carpeta */rb/db/relations*. Permiten declarar de manera cómoda propiedades como atributos, características del nodo de origen y de destino, la etiqueta del enlace y métodos.
- Relaciones declaradas: Para aquellas relaciones sencillas donde sólo basta con definir su etiqueta, neo4jrb permite su creación en la misma línea en la que se define la relación. Por defecto no tienen atributos ni estructura.

En ambos casos se debe definir la relación en las clases objetivo con los métodos `has_many` y `has_one`.

La estructura de base de datos del prototipo cambia radicalmente y pasa a ser la de la Figura 2.2, teniendo al fin relaciones entre los objetos predefinidos de la base de datos.

[21] Conector ORM similar a DataMapper.

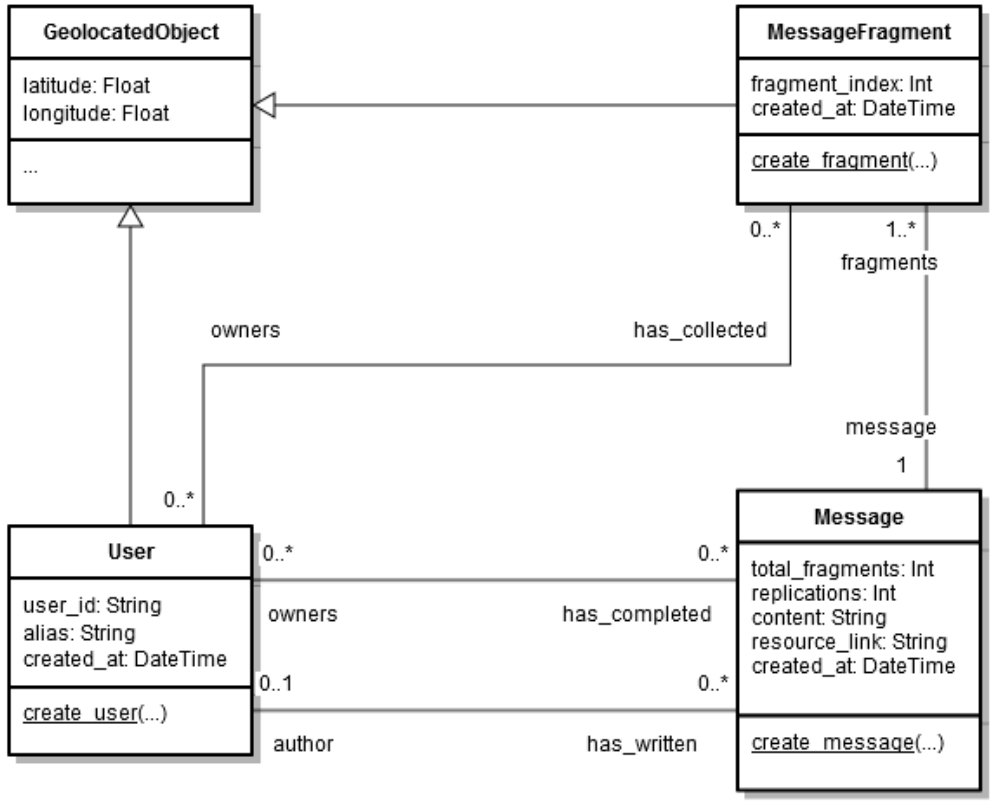


Figura 2.2. Modelo inicial de objetos y relaciones.

Esto permite unas consultas extremadamente rápidas, ya que no es necesario computar aquellos registros relacionados (como en el modelo relacional tradicional), sino que se guardan las referencias de las relaciones entre nodos.

Este mecanismo es muy conveniente para el proyecto, ya que las consultas que más suelen interesar son aquellas de nodos relacionados entre sí (mensajes de un usuario, objetos que ha recolectado, nivel actual, etcétera).

Finalmente, mencionar varios aspectos de considerable importancia para el nuevo prototipo:

- El identificador de usuario, que será su correo electrónico o un id numérico, debe ser único.
- Los mensajes están divididos en fragmentos. Si el mensaje tiene un único fragmento, se considera completo. En caso contrario, el usuario deberá recolectar todos los fragmentos para poder desbloquear el mensaje.
- Cuando un usuario recolecta todos los fragmentos de un mensaje, se borran todas las relaciones con los fragmentos de dicho mensaje y se crea una nueva relación con el mensaje padre.

- Los fragmentos deben ser creados por un mensaje, y no crearse por su cuenta, ya que siempre deben estar relacionados con el mensaje padre. Dicha relación es inmutable.
- La clase `GeolocatedObject` no debe ser instanciada; sólo deberán instanciarse aquellas clases que sean hijas de la anterior. No pasa nada por crear objetos, pero no tiene sentido, ya que sólo posee coordenadas.
- Existen dependencias de destrucción de objetos en la base de datos. Esto significa que, por ejemplo, si el nodo de un usuario es borrado, serán borrados los mensajes que ha escrito. Dicho efecto se consigue añadiendo la opción `dependant: destroy` a la relación.
- Por defecto, neo4jrb trata cada modificación de la base de datos como una operación atómica, escribiendo en disco en cada actualización. Para mejorar la eficiencia, se ha creado la clase gestora `DatabaseManager` en el fichero */rb/managers/_db.rb*, concretamente, el método `run_nested_transaction`, que permite ejecutar transacciones[22] con operaciones en conjunto, actualizando las modificaciones de una sola vez, cuando finalice el bloque de código. En caso de que se genere una excepción no controlada, se desharán los cambios de la transacción, dejando la base de datos intacta.

[22] Operación atómica, consistente, aislada y perdurable (propiedades ACID) de la base de datos en la que se pueden ejecutar múltiples consultas como si fueran una única una sola.

Capítulo 3.
Gestión de usuarios

La gestión de usuarios implica todo proceso de manipulación de datos referentes a los usuarios o jugadores que se hayan registrado en el servicio de Progrezz.

A continuación se listan los distintos apartados según el orden en el que se desarrollaron los mismos.

3.1 Autenticación

Una de las partes más importantes de este módulo es la seguridad. Para que el usuario *A* no pueda acceder, crear, actualizar o borrar datos que sean del usuario *B* (esto es, por ejemplo, escribir mensajes en su nombre) mediante las distintas APIs, es necesario implementar un sistema de autenticación que restrinja las distintas acciones permitidas por un usuario. También debe existir un sistema para bloquear el acceso a usuarios no deseados.

Antes de explicar el funcionamiento del gestor de este apartado hay que aclarar el funcionamiento de los gestores o *managers* de la aplicación.

Un *manager* se ubica en la subcarpeta */rb/managers*. Al iniciar la aplicación, se cargarán todos los ficheros *rb*[23] ubicados en el directorio. Nótese que el orden de carga es puramente alfabético.

La clase gestora definida debe, por tanto, encargarse de inicializarse así misma (por ejemplo, con un método *setup*).

Para el sistema de autenticación, se ha codificado la clase `AuthManager` en el fichero */rb/managers/auth.rb*, que, utilizando la gema OmniAuth[24]

[23] Formato de ficheros ejecutables del intérprete de Ruby.

[24] Librería de Ruby que permite usar de manera cómoda servicios basados en el estándar OAuth.

funcionando bajo OAuth[25]. El funcionamiento interno de OmniAuth está descrito en la Figura 3.1.

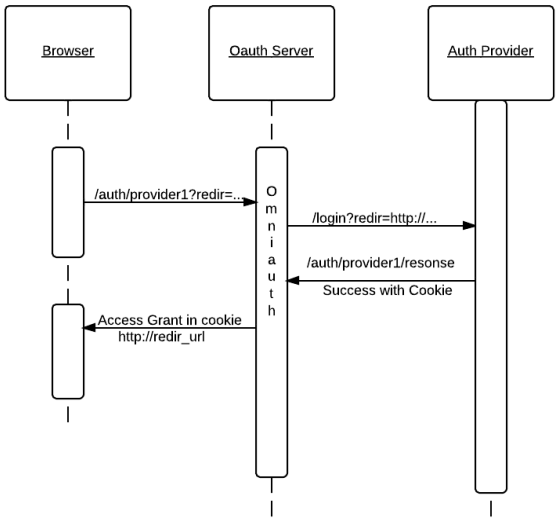


Figura 3.1. Autenticación mediante OmniAuth.

Asimismo, este gestor se encarga de inicializar la librería con una serie de proveedores (Facebook, Twitter, Google+, Steam y GitHub) usando desde la aplicación *Sinatra* la clase `OmniAuth::Builder`.

Posteriormente, se han definido las rutas HTTP[26] Sinatra para la los callbacks[27] de autenticación con el formato */auth/<proveedor>/callback*, al cual serán redirigidos los usuarios cuando se hayan autenticado con el servicio que elijan. Por el contrario, el callback de autenticación fallida se encuentra en la ruta */auth/failure*.

A costa de esto se guardarán los datos del usuario autenticado en las cookies[28] del dominio usando el objeto `session`, el cual funciona de manera similar a un Hash[29], permitiendo comprobar en todo momento qué usuario está autenticado

[25] Estándar de autenticación proporcionado por ciertas redes (Facebook, Twitter, Google+, etcétera).

[26] Protocolo usado en cada transacción o petición de la World Wide Web.

[27] Referencia a función que será llamada por otra subrutina.

[28] Paquete de información de un cierto dominio web guardado en el navegador de un usuario.

[29] Estructura de datos que permite crear pares tipo clave-valor, como un diccionario.

en el sistema, y si se corresponde con el que realiza las peticiones con la API de comunicación.

Hay que tener en cuenta que si un usuario que no está registrado en la base de datos se autentica, pasará a ser un usuario registrado.

Se puede deshabilitar la autenticación cambiando el valor de la variable de entorno *users_auth_disabled* a *"true"*.

3.2 Interacción entre usuarios mediante mensajería

En primer lugar, mencionar que se ha denegado la idea de crear un servicio de mensajería directa similar a un chat, pero dado que uno de los pilares base del juego es el poder compartir información con todos de manera anónima, fue necesario plantear una alternativa.

Por tanto, se ha optado por la opción de que los usuarios puedan escribir mensajes en lugares geo-localizados.

La implementación se encuentra ubicada en la clase `User`, en el fichero */rb/db/objects/user_actions.rb* (en el que se definen todas las "acciones" que tiene permitido realizar un usuario a nivel de jugabilidad) en la función con la siguiente cabecera:

```
def write_message(content, extra_params = {})
```

Nótese que es obligatorio especificar el contenido del mensaje, mientras que se pueden pasar una serie de parámetros opcionales (actualmente, un hipervínculo a un recurso como una imagen) como un Hash. Por ejemplo:

```
user1.write_message("Esto es un mensaje!!", :resource_link => "http...")
```

El tamaño del contenido del mensaje y del recurso se validará en otras funciones relativas al nivel del usuario, descritas en el apartado 3.4 Gestión de estadísticas del jugador.

Los mensajes escritos por usuarios serán divididos en un único fragmento, por lo que no hará falta desbloquearlos ni codificarlos; esto es, el mensaje se encuentra en su totalidad en la zona en la que se escribió. No obstante, cabe la posibilidad de, en un futuro, permitir a usuarios de alto nivel fragmentar sus mensajes en varios bloques.

Con respecto a lo anterior, un usuario puede buscar los fragmentos cercanos con la acción `search_nearby_fragments`. Si está lo suficientemente cerca, podrá recolectarlo con `collect_fragment`.

Cabe mencionar también que un mensaje que no tenga autor será considerado un mensaje de sistema, generado por los administradores, que por lo general podrá ser replicado de manera automática. Este proceso será explicado con más detalle en el punto 3.3 Fragmentación automática de mensajes.

3.3 Fragmentación automática de mensajes

Uno de los pilares las importantes del juego es la generación de contenido para todos los jugadores, sin importar desde dónde se han conectado.

Se ha pensado para la parte de mensajes, que puedan generarse fragmentos de manera aleatoria cuando un usuario se conecta (o si realiza una petición de buscar fragmentos cercanos) en una ubicación en la que no hay suficientes para recolectar.

En ese caso, si el usuario que se ha conectado tiene radio de búsqueda n, se generarán $n \cdot N\_FRAGMENTS\_PER\_KM2$ fragmentos a su alrededor en un círculo de manera totalmente aleatoria.

Por desgracia, cabe la posibilidad de que un fragmento generado aleatoriamente caiga en un punto inaccesible por un jugador, tal como se muestra en la Figura 3.2.

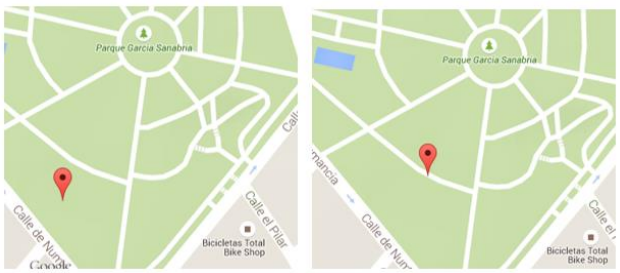


Figura 3.2. Ajuste de punto a ruta peatonal más próxima.

Para solventar el problema, la solución más simple y sencilla es usar un servicio de rutas que permita ajustar un punto geo-localizado a la carretera peatonal (las autopistas entre otros tipos de carreteras no son válidas) más próxima, pudiendo ser accesible desde cualquier punto.

Por comodidad, se permite usar servicios basados en peticiones REST (HTTP) tales como *OpenSource Routing Machine* (OSRM) o *MapQuest Directions API*. Para más información de cómo usar un servicio u otro, véase el fichero *README.md*.

Otro aspecto importante es que los mensajes tendrán una cierta duración, forzando a los jugadores a trabajar de manera rápida y eficiente. Si no logran recolectar el mensaje por completo a tiempo, no podrán leer su contenido. Nótese que también sirve como medida anti-spam[30].

En cuanto a diseño y cambios en la base de datos, se debe poder especificar cuándo un mensaje es replicable o no. Para ello se utilizarán nuevos atributos en los modelos de la base de datos, tal como se muestra en Figura 3.3.

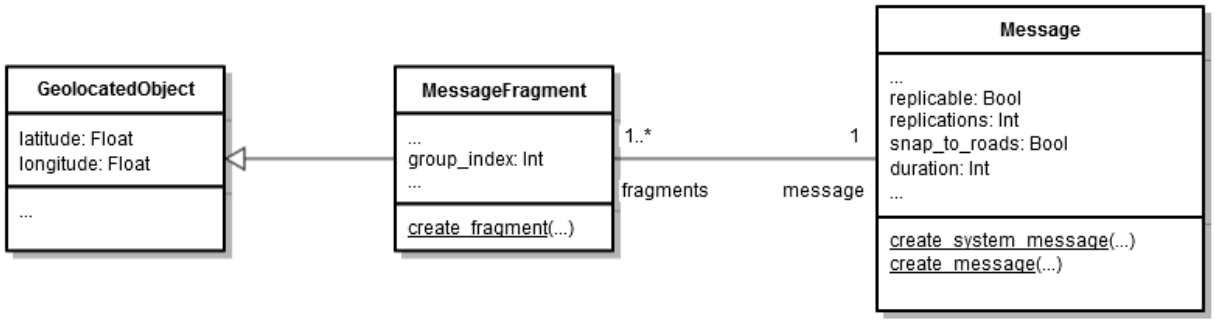


Figura 3.3. Modificaciones en mensajes y fragmentos.

En este punto se involucran las llamadas "mecánicas de juego", que consisten en clases que heredan de `Mechanic` con métodos estáticos que son inicializadas al inicio de la ejecución mediante la clase gestora `MechanicManager` (en el fichero */rb/managers/mechanics.rb*).

Por lo general, al iniciarse, cada una carga un fichero JSON[31] que contiene información sobre el aspecto de la jugabilidad en cuestión.

Puede apreciarse un ejemplo del diseño de las clases referentes a mecánicas de juego en la Figura 3.4.

[30] Contenido no deseado en correos, páginas webs,...

[31] Notación de objetos javascript como una cadena de caracteres.

Nótese que la mayoría de mecánicas permiten cambios de sus dependencias (ficheros JSON) en tiempo real, usando el panel de administración ubicado en la ruta */admin/*.

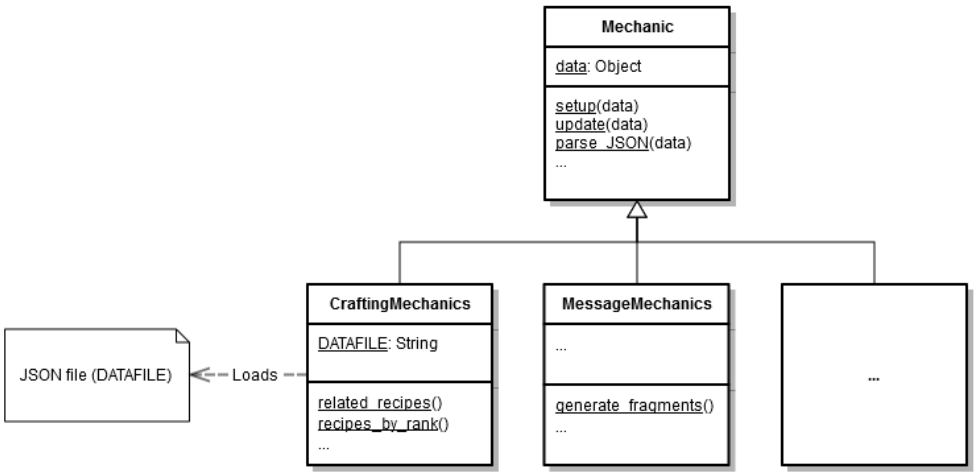


Figura 3.4. Estructura de mecánicas de juego.

Actualmente, son dos las mecánicas que se relacionan directamente con la fragmentación de mensajes:

- `GeolocationMechanics`, ubicado en el fichero */rb/game/mechanics/geolocation.rb*: Principalmente se utiliza para ajustar (si procede) los fragmentos de los mensajes de sistema a las carreteras peatonales más próximas.
- `MessageMechanics`, ubicado en el fichero */rb/game/mechanics/messages.rb*: Esta clase más avanzada permite generar fragmentos cercanos a un usuario teniendo en cuenta aspectos importantes, como el radio de búsqueda de un usuario, los mensajes cercanos, o el máximo número de fragmentos existentes para crear nuevos fragmentos.

En resumidas cuentas:

- Los mensajes de usuarios (con autor) no pueden ser replicables, ya que interesa que los jugadores accedan a contenido generado para el juego: la historia.
- Se pueden crear mensajes de sistema replicables cuyos fragmentos se ajusten a carreteras próximas utilizando el método siguiente:

```
Game::Database::Message.create_system_message(contenido, n_fragmentos)
```

- Los mensajes de sistema se deben ajustar a la carretera peatonal más próxima, mientras que los de usuarios no, ya que podrían especificar posiciones especiales (puntos limpios, restaurantes,...).
- Los clientes realizas peticiones constantemente de búsqueda de fragmentos cercanos. Si en algún momento no hay suficientes, se generarán nuevos fragmentos.
- En caso de que un usuario haya recolectado todos los fragmentos de una zona, no se generarán nuevos fragmentos a no ser que el mensaje que los tenía haya caducado, evitando el sedentarismo.
- Las copias de mensajes se realizarán copiando todos sus fragmentos, creando "grupos de replicación". De esta manera, no se generarán más de un tipo que de otro.

3.4 Gestión de estadísticas del jugador

En el documento de venta del proyecto se estipula claramente que el jugador deberá poder progresar en el juego escalando niveles mientras gana experiencia.

Se ha optado por diseñar un sistema sencillo en el cual los usuarios tienen un perfil, que contiene un número especificando el nivel, y otro valor especificando la experiencia.

En la Figura 3.5 se puede apreciar el modelo de la base de datos para crear este efecto. Se añadirá por tanto un nuevo nodo en la base de datos por cada jugador registrado.

Para complementar el funcionamiento de esta nueva característica, es necesario definir una nueva mecánica, `LevelingMechanics`, que se encargará de calcular la experiencia necesaria para subir al siguiente nivel. En el apéndice A.1. Subida de nivel se puede estudiar el algoritmo utilizado para actualizar todos los datos cuando el usuario gana experiencia, usado en la función `gain_experience`.

La curva de experiencia por niveles se encuentra ilustrada en la Figura 3.6, y se define vía código como una función cargada desde el fichero JSON */data/leveling.json*, por lo que puede ser fácilmente ajustada por un administrador.

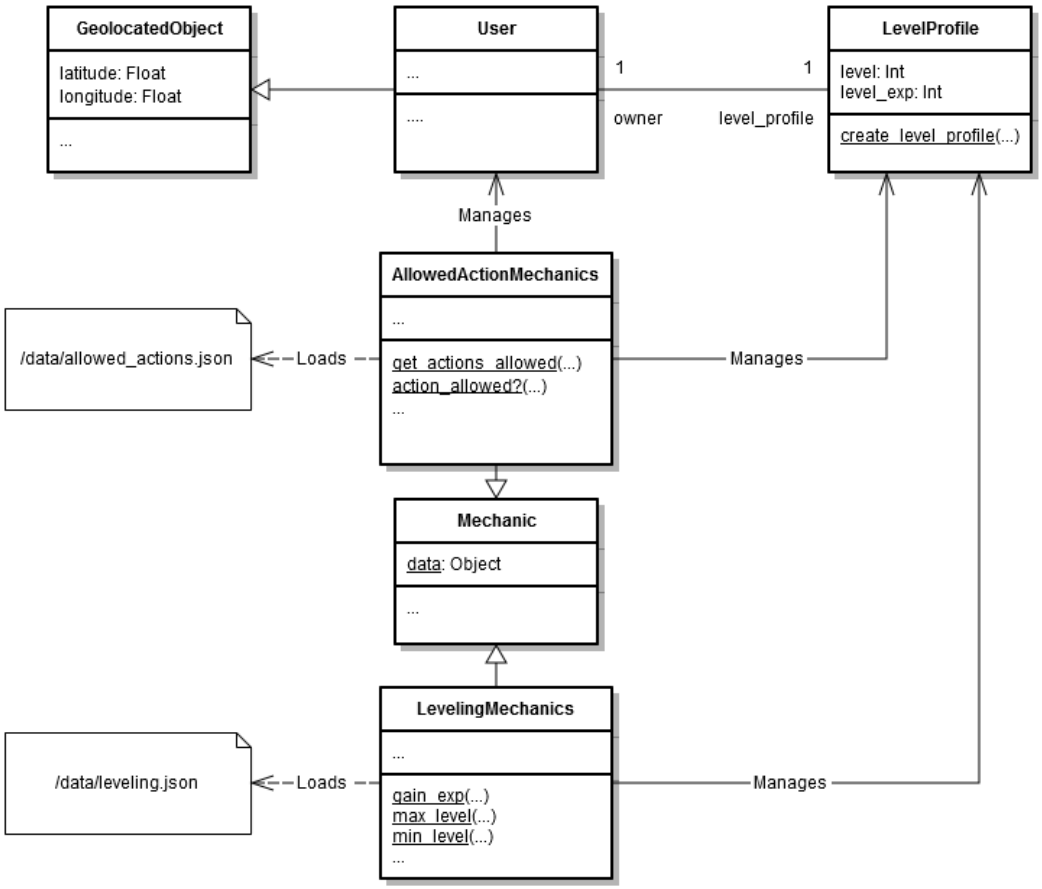


Figura 3.5. Diagrama de clases para la gestión de niveles y experiencia de los usuarios.

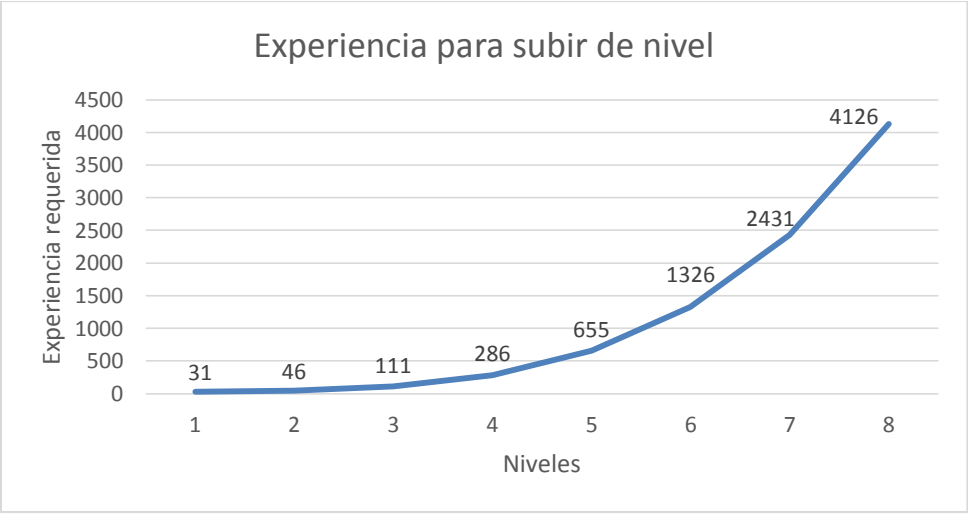


Figura 3.6. Experiencia requerida para subir de nivel.

Cuando el usuario realiza ciertas acciones (recolectar fragmentos, decodificar mensajes,...) gana experiencia, que, en algún momento, le permitirá subir de nivel, lo que supondrá una serie de ventajas, como el acceso a nuevas acciones.

Esto es regulado por otra mecánica, `AllowedActionMechanics`, que define las acciones permitidas por un usuario a un determinado nivel.

Es conveniente mencionar en este apartado el uso de lanzadores de eventos. Las clases pueden implementar o extender el módulo `Evented` (definida en el fichero */rb/generic_utils.rb*), lo que permite lanzar eventos y ejecutar los callbacks o las escuchas que se hayan definido. Para el caso de los usuarios, los eventos definidos se encuentran en el fichero */rb/db/objects/user_events.rb*, y son los siguientes:

- *OnCreate*: Cuando el usuario se registra en el sistema por primera vez.
- *OnRemove*: Cuando el usuario es borrado de la base de datos.
- *OnLevelUp*: Cuando el usuario sube de nivel.

Así, se pueden definir todos los eventos que el programador desee. Para lanzar el evento y llamar a los callbacks, basta con llamar a la función `dispatch(…)` del usuario en cuestión, especificando el nombre del evento y los parámetros extras necesarios para su correcta ejecución. En la Figura 3.7 se detalla un ejemplo de implementación.

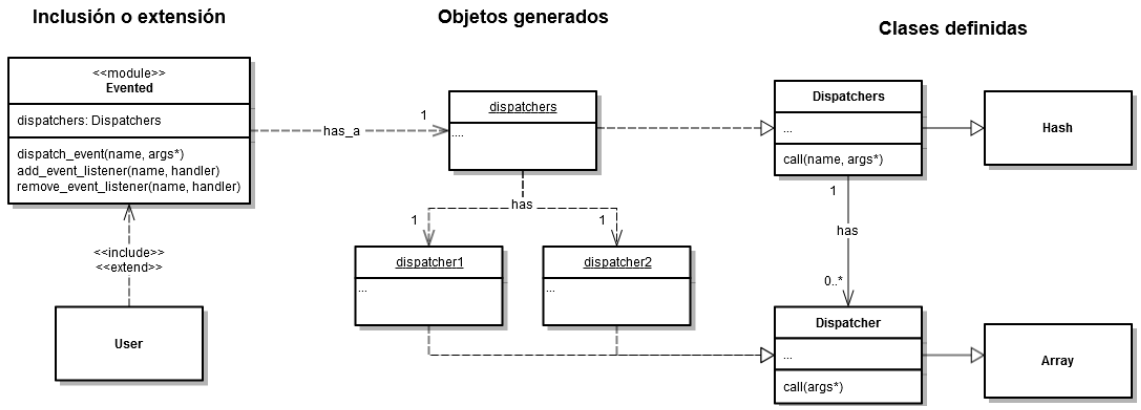


Figura 3.7. Diagrama de clases para implementación de eventos a nivel de clase u objeto.

Para más información sobre los parámetros actuales y temporales del juego, véanse los ficheros de datos JSON */data/allowed_actions.json* y */data/leveling.json*.

Paralelamente a esto, es ideal guardar una serie de estadísticas por usuario; por ejemplo, número de fragmentos recolectados, o número de mensajes completados. Como no es ideal, en un momento tan temprano del desarrollo,

sobrecargar la base de datos, se ha optado por la simple solución de añadir una propiedad por cada estadística que se desea reservar.

Posteriormente, y de manera manual, el programador deberá definir una nueva propiedad en la base de datos de igual manera que en este ejemplo:

```
class User
  # ...
  # Contador de fragmentos recolectados por el usuario.
  # @return [Integer] Cantidad de fragmentos recolectados por el usuario.
  property :count_collected_fragments, type: Integer, default: 0
  # ...
end
```

3.5 Ajuste y configuración de la API

Para compatibilizar lo codificado con la API del proyecto basta con modificar las acciones que realiza la misma (sea REST o WebSocket) generando una salida apropiada.

Es algo trivial, pues se ha hecho a la par mientras se implementaban los distintos aspectos mencionados en este capítulo.

Por ende se han actualizado los siguientes grupos de métodos de la API para que todo funcione correctamente:

- User (REST): Obtiene información relativa al usuario (quién es, perfil de usuario y acciones permitidas por el usuario).
- Users (WebSocket): Se utiliza para manipular información del usuario que cambia continuamente (actualizar geo-localización y obtener jugadores cercanos).
- User Messages (REST): Todo método de interacción entre un usuario y mensajes o fragmentos irán en este grupo.
- Messages (REST): Si permite interactuar con la información de un mensaje o fragmento sin necesitar de un usuario, entonces debe ir en este grupo (obtener el mensaje de un fragmento, obtener mensajes sin autor, y obtener la información de un mensaje).

Para saber cómo actúa cada función y los parámetros de entrada que utiliza, véanse los ficheros */data/rest_methods.json* y */data/websocket_methods.json*.

Capítulo 4.
Gestión de objetos

Aunque no se vaya a implementar aún en el cliente o *front-end*, es necesario añadir en el servidor o *back-end* la gestión de objetos e inventario, para que esté listo y funcional cuando se codifique en el lado del cliente.

A continuación se listan los distintos apartados de este módulo según el orden en el que se desarrollaron los mismos.

Generación e interacción de objetos en el mundo

4.1 Supóngase que los objetos son elementos de la base de datos con una serie de propiedades inherentes a sí mismos. Dichos elementos son representados por una descripción genérica: identificador único, nombre, descripción, imagen representativa, calidad, etcétera. Un usuario podrá poseer objetos en su inventario (explicado más adelante).

No obstante, esos objetos pueden encontrarse de forma natural en el mundo, o deben ser creados por el jugador. Este apartado trata el primer caso, en el que el usuario deberá recolectar recursos o minerales de los diferentes depósitos de objetos.

El funcionamiento es bastante sencillo:

- Se define o se crea un objeto en el fichero */data/items.json*.
- Se especifica si el objeto tendrá depósitos (puede recolectarse en el mundo).
- En caso de tener un depósito asociado, deben definirse todas sus propiedades. El depósito actúa como "clase" para generar instancias.
- Cada vez que un usuario realice peticiones de búsqueda de instancias depósitos en una zona en la que no haya suficientes, se generarán nuevas estancias ajustadas a la carretera más próxima, al igual que ocurre con los fragmentos de los mensajes replicables.

En la Figura 4.1 se detallan los aspectos más importantes de implementación, que son los siguientes.

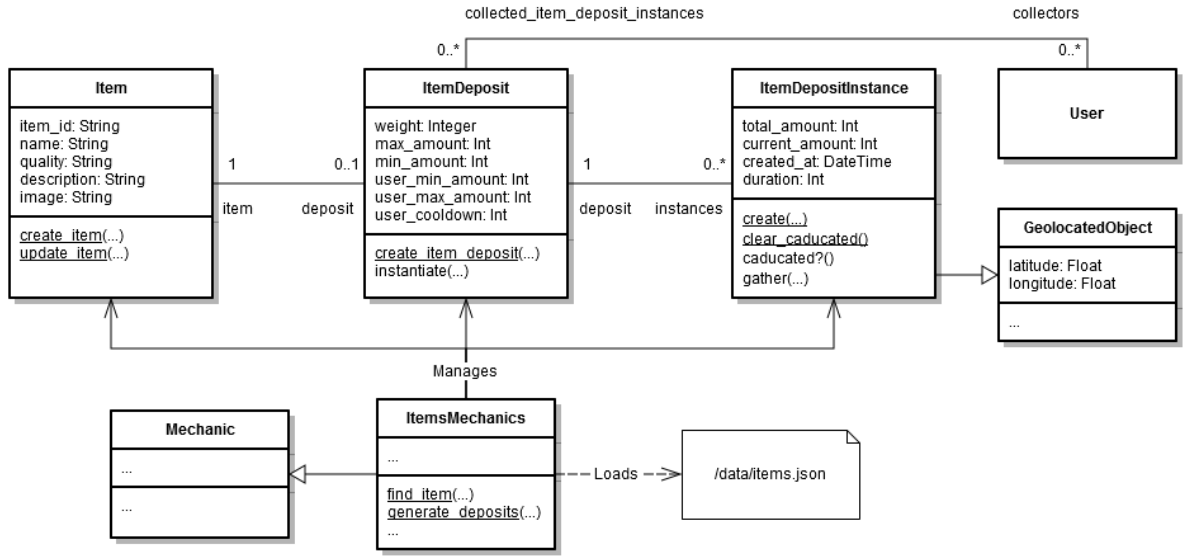


Figura 4.1. Diagrama de clases de objetos y depósitos.

La generación de depósitos se basa en un sistema de pesos. Cada peso asociado representa una probabilidad de que se genere una estancia de depósito de ese tipo; cuanto mayor sea, mayor será la probabilidad. Se ha utilizado la gema pickup[32] para realizar esta tarea.

Mientras se crea la instancia, se deben definir la cantidad de recursos `total_amount` que podrán ser recolectados de la misma, comprendida entre `min_amount` y `max_amount`.

Cuando un usuario recolecte minerales de la estancia del depósito con la acción `collect_item_from_deposit` (siempre y cuando tenga espacio en el inventario) conseguirá minar una cantidad aleatoria entre `user_min_amount` y `user_max_amount`, y se substraerá la cantidad a la variable `current_amount`. Además, el usuario no podrá volver a minar el depósito hasta que transcurran `user_cooldown` segundos, y tampoco podrá recolectarlo a no ser que disponga del nivel adecuado.

[32] Librería de Ruby que permite seleccionar objetos con peso de un grupo de manera aleatoria.

Para una mayor dinámica de juego, las instancias de los depósitos tendrán un tiempo de duración. Si el depósito se queda sin recursos, o si se agota el tiempo, será marcado como caducado, y por consiguiente será borrado.

Nótese que se borrarán de manera automática con el gestor de tareas programadas `TasksManager`, ubicado en */rb/managers/scheduled_tasks.rb*.

4.2 Gestión de inventario

El almacenamiento de objetos por parte de los usuarios se realiza por medio de inventarios o mochilas.

Cada usuario tendrá una mochila personal e intransferible, con una cierta capacidad (`slots`) para almacenar pilas de objetos.

Una pila o cúmulo podrá contener como máximo `max_amount` unidades, y serán representadas por medio de relaciones Neo4j. Esto significa que, si un usuario tiene el inventario vacío, su mochila tendrá cero relaciones con objetos. En cambio, si está completo, tendrá `slots` pilas.

Además, existe una clase gestora de mecánicas, `BackpackMechanics`, que gestiona todos los aspectos relativos a este apartado. Por ejemplo, se contemplan características como el tamaño base, nuevos huecos por nivel, etcétera.

En la Figura 4.2 se detallan los aspectos de implementación mencionados.

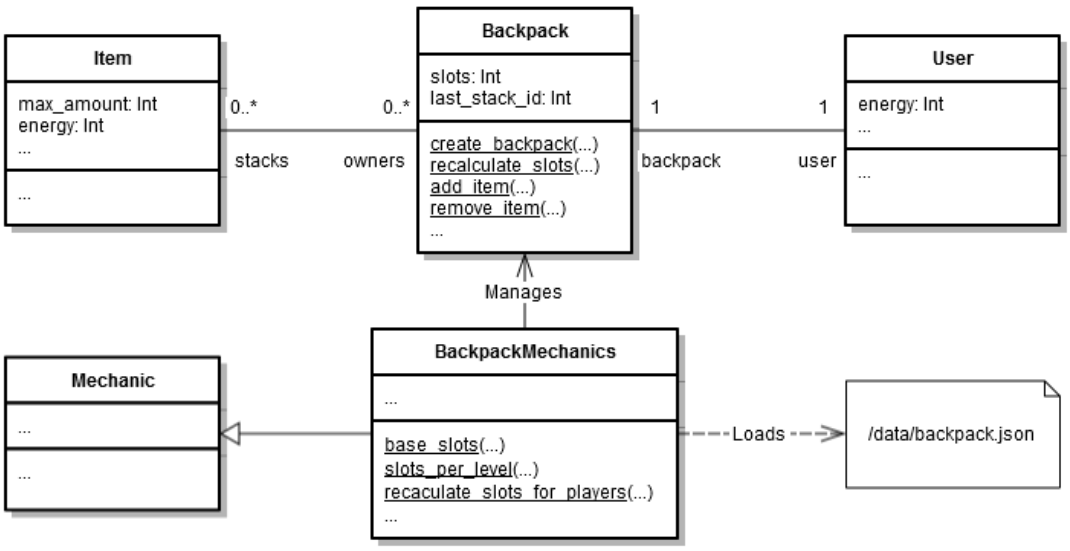


Figura 4.2. Diagrama de clases de inventario.

Por consiguiente, cada vez que se añada un nuevo objeto al inventario, sea por recolección o creación, se establecerá un enlace o relación entre la mochila de un jugador y el objeto en sí, conteniendo propiedades como fecha de creación, o cantidad de objetos en la pila. La clase que representa estos enlaces es `BackpackItemStacks`, y se encuentra en */rb/db/relations/backpack_item.rb*.

También se debe tener en cuenta que un usuario tiene un límite de objetos, así que es necesario que pueda gestionar el contenido de su propia mochila. En lugar de borrar objetos y perderlos para siempre, estos pasan a convertirse en "energía", que posteriormente podrá ser utilizada para realizar otras acciones, como por ejemplo, alimentar balizas, comprar objetos o recetas (no implementado), etcétera.

La mochila proporciona una gran cantidad de métodos para realizar tareas rutinarias y tediosas, alejando al programador de la dura tarea de trabajar con muchas relaciones de Neo4j. Entre ellos, los más destacados son los siguientes:

- `add_item(item, amount)`: Permite añadir nuevos objetos al inventario, siempre y cuando haya espacio. En primer lugar, se partirán los objetos en las pilas sin completar. Posteriormente, la cantidad restante se colocará en un nuevo hueco. El algoritmo se encuentra descrito en A.2. Añadir objetos al inventario.
- `exchange_stack_amount(stack_id, amount)`: Dado un identificador de un cúmulo de objetos, se intercambiará la cantidad especificada por energía siempre que haya suficientes unidades.
- `split_stack(stack_id, restack_amount, target_stack_id?)`: Intenta crear una partición de la pila especificada en otro cúmulo de tamaño `restack_amount`. En caso de indicar otro identificador de pila, se intentará añadir en dicho cúmulo. El algoritmo se encuentra descrito en A.3. Dividir pilas de objetos.
- `fits?(item, amount)`: Comprueba si una cierta cantidad de objetos cabe en el inventario. Esta comprobación se usa bastante, sobre todo para añadir objetos.
- `has?(item, amount)`: Comprueba si la mochila posee una cierta cantidad de objetos. Esta comprobación se usa bastante, sobre todo para eliminar objetos.

Construcción de nuevos objetos o “craft”

Una vez el jugador ha recolectado materias primas, es esencial poder interactuar con ellas. Para ello, se ha implementado un sistema de fabricación de objetos llamado “crafting”.

4.3 El proceso es sencillo: dado un usuario, una receta y una cantidad compatible de recursos, podrá generar nuevos objetos que puedan ser de cierta utilidad.

Una receta se caracteriza por los siguientes aspectos:

- Debe tener un identificador de receta único.
- Debe tener una entrada, compuesta por un conjunto de pares de objetos y sus respectivas cantidades.
- Debe tener una salida, o lo que es lo mismo, un par de objeto producido y su cantidad.

Se puede visualizar lo anteriormente explicado en la Figura 4.3.

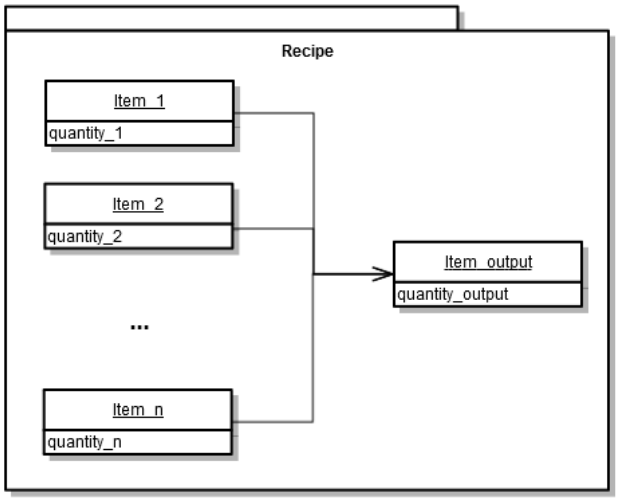


Figura 4.3. Esquema de fabricación de objetos.

La clase de mecánicas de juego `CraftingMechanics` se encarga de gestionar estos datos. En el fichero */data/crafting.json* se describen las distintas recetas organizadas por rangos.

Actualmente, los rangos se utilizan para desbloquear grupos de recetas por nivel, gestionada por la clase de mecánicas de juego `AllowedActions`: en el fichero */data/allowed_actions.json* se especifican las recetas a las que un

usuario tiene acceso con la forma `craft_item_<rank_id>` Así, un usuario podrá crear objetos acomodados a su nivel.

A nivel de codificación, la tarea de iniciar la producción de una receta conlleva los siguientes pasos:

1. Comprobar que el usuario tenga acceso a la receta en cuestión.
2. Eliminar los objetos de entrada del inventario del usuario.
3. Añadir los nuevos objetos producidos.

Si en algún momento se produce un fallo en uno de estos pasos, sea porque no tiene acceso a la receta, no dispone de los recursos necesarios, o no tiene espacio para añadir la producción, se deshará por completo la transacción, dejando al usuario en el estado en el que se encontraba antes de iniciar el proceso.

4.4 Ajuste y configuración de la API

Al igual que ocurrió en el capítulo anterior, la conexión con la API de comunicación con el cliente es algo trivial, pues sólo es necesario añadir los métodos a emplear y formatear la salida de manera correcta.

Los grupos de métodos de la API implicados son los siguientes:

- User Items (REST): Aquí se hayan los métodos relacionados con la interacción entre usuarios y objetos, incluidos depósitos; búsqueda de depósitos cercanos, recolección de depósitos, obtener información de la mochila del jugador, fabricar un objeto..., entre otras.
- Items (REST): Incluye métodos REST que están relacionados con objetos en los que no es necesario especificar un usuario. Entre ellos se encuentran el poder listar todos los objetos del sistema, obtener la descripción de un objeto, listar todas las recetas de fabricación, y buscar recetas relacionados con un determinado objeto (sea entrada o salida).

Para saber cómo actúa cada función y los parámetros de entrada que utiliza, véanse los ficheros */data/rest_methods.json* y */data/websocket_methods.json*.

Capítulo 5. Gestión de eventos sociales

El apartado de eventos sociales es muy ambicioso, pues en un proyecto de estas características, que busca reunir a las personas para realizar obras de caridad y hacer el mundo de un lugar mejor.

Por tanto, para este trabajo de fin de grado no se busca completar todo el juego, sino hacer una pequeña base; las balizas o "beacons", que son marcadores o nodos geo-localizados que indican puntos de energía positiva. Por ejemplo, tiendas que venden productos ecológicos, restaurantes que utilizan alimentos del kilómetro cero[33], entre otras muchas posibilidades.

A continuación se describirán los aspectos del desarrollo de este apartado.

5.1 Gestión de balizas

Como se ha mencionado anteriormente, las balizas son puntos que indican un lugar de energía positiva. Actualmente no proporcionan mucho contenido al juego, pero sí que son fundamentales para el desarrollo de nuevas características de Progrezz.

Las balizas deben cumplir una serie de condiciones de gran importancia para su correcta implementación, listadas en los sub-apartados de este capítulo.

5.1.1 Geo-localización

Las balizas deben estar geo-localizadas. A priori, bastaría con decir que la clase `Beacon` ubicada en */rb/db/objects/beacon.rb* debe heredar de `GeolocatedObject`. No obstante, es posible que en un futuro se añadan nuevas características basadas en objetos geo-localizados (es más, en un momento se pensó incluir en el desarrollo el poder añadir maquinaria que permitiese

[33] Filosofía que promueve el consumo de alimentos locales, comarcales y estacionales.

recolectar recursos a distancia). Por tanto, es conveniente implementar la clase `ItemGeolocatedObject`, que además de tener un contador de duración o tiempo de vida (idéntico al de las instancias de los depósitos de objetos).

Es importante mencionar también que define la estructura de nuevas relaciones en la base de datos, que conectan usuarios con objetos que hayan desplegado.

En la Figura 5.1 se representan estos detalles de implementación mediante un diagrama de clases.

5.1.2 Nivel

Al igual que los jugadores, las balizas tendrán acceso a una lista de niveles y características por cada vez que progrese en uno. Para implementar esta característica, se ha reutilizado la clase `LevelProfile`. No obstante, los niveles no están comprendidos en el mismo intervalo, ya que en este caso van desde el cero hasta el tres, y las mejoras son completamente distintas.

Es por eso que se ha tenido que crear una nueva mecánica de juego que permita la correcta gestión, `BeaconMechanics` (en */rb/game/mechanics/item_mechanics/beacon_mechanics.rb*), cuya estructura es casi idéntica a `LevelingMechanics`, aunque variando en algunos aspectos. El fichero de datos ajustables se encuentra ubicado en */data/beacons.json*.

No obstante, las balizas no pueden realizar acciones, ya que son elementos estáticos. Por tanto, para ganar experiencia, deben recibir energía cedida por usuarios que consideren que el punto representa en su totalidad una localización de energía limpia.

Estos son algunos de los detalles más importantes referentes al nivel de las balizas:

- El nivel mínimo es el 0, mientras que el máximo es el 3. La experiencia requerida para escalar niveles puede ser manualmente ajustada en el fichero JSON.
- La cantidad de tiempo que se añade a la duración de la baliza, por cada unidad de energía cedida, aumenta con cada nivel. También puede ser ajustada manualmente.

- Las balizas tienen un radio de acción y un peso aplicado a la zona circundante, sensibles al nivel actual. Se hablará de esto con más detalle en los siguientes apartados.

En la Figura 5.1 se representan estos detalles de implementación mediante un diagrama de clases.

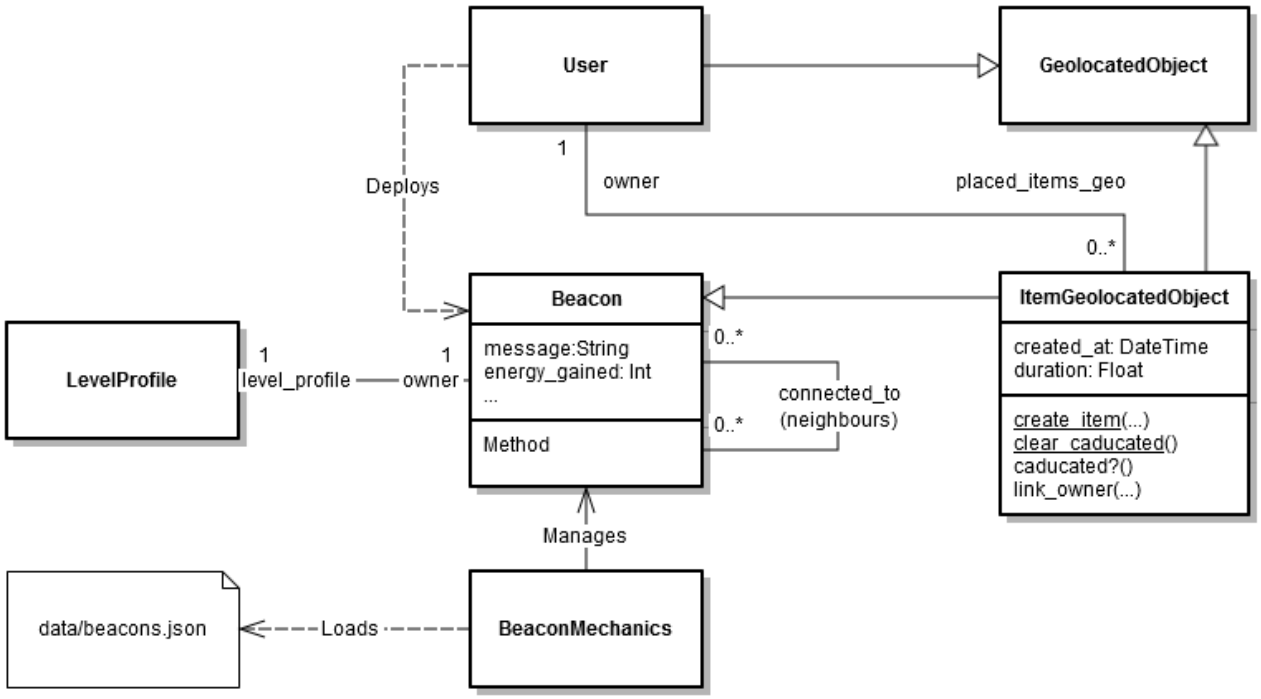


Figura 5.1. Diagrama de clases de las balizas.

5.1.3 Duración

Es posible que un punto de energía deje de serlo en un momento determinado. Por ello, el tiempo de las balizas debe poder expirar. Esto implica que los usuarios tengan que actuar correctamente, y si a una baliza ya no le corresponde estar en su sitio, no deben alimentarla con energía.

Por tanto, una baliza tendrá una duración base de 10080 minutos (7 días), y se añadirán minutos extra de duración en función de la energía añadida. En cuanto expire la baliza será eliminada definitivamente.

Nótese que estos aspectos pueden ser ajustados cómodamente para balancear el contenido ofrecido por el juego.

5.1.4 Fabricación

Al igual que la mayoría de objetos, las balizas deben poder ser construidas por los jugadores a un nivel moderadamente alto, ya que se considera una mecánica de juego bastante avanzada.

No se han definido los materiales necesarios para construirla, pues esto es tarea de los administradores y guionistas del juego. No obstante, deberán tener un coste elevado, pues incorporan una serie de mejoras substanciales para los usuarios, descritas en la sección 5.1.5 Mejoras para el jugador.

5.1.5 Mejoras para el jugador

Las balizas son una mecánica simple. Por desgracia, a simple vista, no son atractivas para la mayoría de jugadores. Es por ello que surge la necesidad de incorporar ciertas mejoras o beneficios para los jugadores que interactúen con las balizas.

Actualmente se ha implementado sólo una mejora, y es que la probabilidad de que aparezcan depósitos de minerales raros aumenta de manera drástica cuando hay balizas cerca.

Para llevar a cabo esta tarea basta con, en el momento de generar depósitos, comprobar si hay balizas cercanas, y si su radio de acción incluye el punto del depósito. En ese caso, se incrementa el peso de todos los depósitos de manera uniforme, haciendo que los que tengan un peso menor tengan más probabilidad de aparecer.

Esto se debe a que la probabilidad de que aparezca un depósito de un determinado tipo es la relación entre su peso y la suma de todos los pesos de los depósitos. Se puede ver un ejemplo simplificado en la Tabla 5.1.

Peso depósito 1	Peso depósito 2	Probabilidad depósito 1	Probabilidad depósito 2
15+0= **15**	85+0= **85**	15/(15+85)= **0.15**	85/(15+85)= **0.85**
15+10= **25**	85+10= **95**	25/(25+95)= **0.21**↑	95/(25+95)= **0.79**↓
15+20= **35**	85+20= **105**	35/(35+105)= **0.25**↑	105/(35+105)= **0.75**↓
15+30= **45**	85+30= **115**	45/(45+115)= **0.28**↑	115/(45+115)= **0.72**↓

Tabla 5.1. Ejemplo de aumento de pesos para incrementar probabilidades.

La alteración de pesos se realiza desde la clase `BeaconMechanics` con el método `add_ponderation`, que es llamado desde `generate_nearby_deposits` de `ItemsMechanics`.

Se ha pensado también que para un futuro sería conveniente premiar al usuario que ha colocado la baliza, cediéndole energía o experiencia con cada mejora. Queda abierto para futuras mejoras o actualizaciones.

5.1.6 Triangulación

La triangulación de balizas se detalla con mayor profundidad en el punto 5.2 Triangulación de balizas.

5.2 Triangulación de balizas

Una de las partes probablemente más vistosas de este capítulo es la triangulación o conexión de balizas. Muchas balizas interconectadas entre sí son capaces de crear portales o vórtices de energía que, por desgracia, no serán implementados como parte de este trabajo.

Las balizas sólo se conectarán como máximo a otras tres balizas para formar lazos, siempre y cuando las otras tengan al menos un espacio de conexión. Para implementar esta característica en la aplicación, se ha definido una relación bidireccional que permita crear vecindad entre dichos nodos.

Por tanto, cuando un usuario despliega una baliza, se intentará unir a aquellas más próximas cuyos radios de acción cubran la posición de la desplegada, o viceversa.

A la hora de buscar balizas cercanas se buscan primero las balizas en el cuadrado contenedor del círculo. Posteriormente, se comprueba que balizas del cuadrado caen dentro del círculo de búsqueda, mejorando drásticamente la eficacia frente a otros métodos, como la iteración sobre todas las balizas para encontrar las que se encuentre en un círculo alrededor.

El proceso de actualización de vecindad puede ser complejo y costoso. La pregunta clave es: ¿cuándo es necesario hacer estas comprobaciones? La respuesta se explica en los siguientes casos:

- Cuando se crea una nueva baliza es necesario comprobar si es posible conectarla con vecinas disponibles.
- Cuando se destruye una baliza porque ha caducado, se deben borrar los enlaces con todas las vecinas. Además, es necesario actualizar los enlaces de todas las balizas afectadas, pues cabe la posibilidad de que puedan conectarse con balizas cercanas con las que no pudieron relacionarse porque estaban completas en capacidad de conexiones.
- Cuando sube de nivel, ya que su radio de acción aumenta, por lo que es posible que se descubran nuevos elementos a su alcance.

Entonces, se puede valorar la posibilidad de extender el módulo `Evented`, al igual que se hizo con los usuarios, para lanzar el evento de creación, borrado o subida de nivel para actualizar los vecinos cercanos, creando una estructura modular y usable.

El algoritmo de actualización de vecinos se describe con un mayor grado de detalle en el apéndice A.4. Actualizar los vecinos de una baliza.

Se puede apreciar en la Figura 5.2 una imagen del panel de administración para comprobar todas las balizas relacionadas con la baliza dada, mediante un algoritmo de búsqueda en amplitud, en un sencillo mapa de Open Street Maps[34].

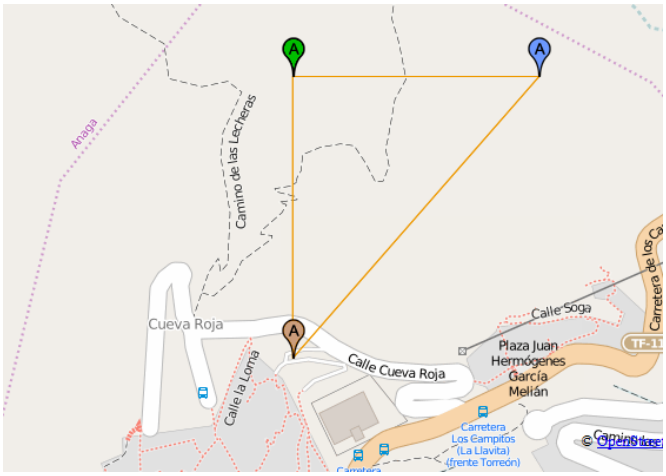


Figura 5.2. Balizas interconectadas.

Recordar que las balizas no son como los fragmentos de mensajes del sistema o como los depósitos, y no es necesario ajustarlos a la carretera más próxima.

[34] Servicio de mapas libre y gratuito.

Ajuste y configuración de la API

Al igual que en casos anteriores, el ajuste de la API de comunicación con el cliente es algo trivial, por lo que sólo es necesario añadir las funciones necesarias y formatear la salida según convenga.

5.3 Los grupos de métodos de la API implicados son los siguientes:

- User Beacons (REST): Aquí se concentran todas las funciones relacionadas con la interacción entre usuarios y balizas, que son las siguientes:
  - `user_get_nearby_beacons`: Búsqueda de balizas cercanas a un usuario. Dependerá del radio de visión del mismo. Devolverá al cliente toda la información de las balizas, incluido los identificadores de los elementos interconectados.
  - `user_deploy_beacon`: Despliega una baliza, siempre y cuando disponga de suficientes en el inventario. Debe especificar un mensaje, y se desplegará en la posición geo-localizada del jugador.
  - `user_yield_energy`: Cede energía a una baliza, ideal para hacer que suba de nivel para mejorar sus capacidades.
  - `user_get_deployed_beacons`: Muestra toda la información de las balizas desplegadas por el usuario, incluidas las referencias a las vecinas.

Para saber cómo actúa cada función y los parámetros de entrada que utiliza, véanse los ficheros */data/rest_methods.json* y */data/websocket_methods.json*.

Capítulo 6.
Conclusiones y líneas futuras

Este trabajo ha servido para muchos fines: educación, aprendizaje, mejora de habilidades,... Pero el más importante y destacado ha sido trabajar en un proyecto social.

Posterior a su realización se han obtenido una serie de conclusiones teóricas y prácticas, listadas a continuación:

- Un proyecto social como lo es Progrezz necesita de mucho trabajo para ser desarrollado; esto es, apoyo de la comunidad. No obstante, a no ser que el proyecto esté medianamente avanzado y sea atractivo, será difícil atraer a dicha comunidad de desarrolladores, por lo que queda claro que este será un proyecto de larga duración.
- Para que un proyecto de tal calibre tenga un futuro esperanzador, es necesario dedicarle gran cantidad de horas de diseño y análisis, pues siempre se añadirán módulos y características al mismo. Esto significa que debe poder mantenerse, evitando al máximo tener que realizar cambios en el proyecto.
- Se han aplicado muchos conocimientos de las asignaturas cursadas en la carrera; desde programación e informática básica, pasando por las relativas a gestión de sistemas y bases de datos, hasta aquellas que requieren de más tiempo y dedicación sobre construcción de sistemas y aplicaciones, como programación de aplicaciones interactivas (programación orientada a eventos), diseño y análisis de algoritmos, etcétera.
- La jugabilidad y usabilidad de un videojuego es uno de los factores más importantes para que las personas mantengan contacto con el mismo. Es por eso que ha surgido la necesidad de implementar y habilitar una serie de herramientas para que los administradores, guionistas, desarrolladores y gestores puedan actualizar los parámetros de juego según convenga.

- Personalmente, he descubierto que los diagramas UML pueden ser de gran utilidad a la hora de expresar ideas o conceptos del desarrollo. Tal como se ha podido observar, dichos diagramas representan conceptos complejos de manera sencilla e intuitiva, como las distintas estructuras de la base de datos.
- Esta experiencia me ha dado la oportunidad de conocer y trabajar con gente de gran talento, ampliando mis horizontes, permitiéndome aplicar y aprender conocimientos para conseguir un producto desarrollado como si de un trabajo profesional se tratase.

En conclusión, se han implementado todos los apartados y tareas descritas en el proyecto del Trabajo de Fin de Grado, por lo que la estimación inicial de tiempo de cada una de las tareas puede considerarse bastante acertada.

En un futuro, el proyecto seguirá en desarrollo y mantenimiento por la comunidad de desarrolladores que quieran aportar contenido. No obstante, no existe una plantilla fija de trabajadores que permitan un desarrollo constante y lineal del proyecto, sino más bien una serie de voluntarios (programadores, guionistas, directores y administradores) que pueden ser de utilidad para mantener el proyecto.

El próximo punto a desarrollar en el back-end es mejorar e implementar los módulos restantes de gestión de eventos sociales, ya que no ha habido tiempo de implementarlos como parte de este trabajo.

En lo que respecta a mi persona, se tratará de dedicar algo de tiempo de ocio con el objetivo de desarrollar nuevos componentes, mejorar los existentes, y corregir los fallos encontrados.

Finalmente, terminar este apartado con una bella y motivadora oración del proyecto:

"Progrezz estará en una fase beta permanente, porque hoy, lo único constante es el cambio." – Jorge García del Arco.

Capítulo 7.
Summary and Conclusions

This chapter describes the fundamental aspects of this inform in English.

It includes an extended summary about the project's development and a list of conclusions extracted from it.

Summary

7.1 Serious games or applied games are a small subset of videogames. Their first goal is not offering entertainment as the rest of videogames, but offering features to enhance some aspects of the final users, such as education (e-learning), health services...

Unwanted or annoying tasks can be performed comfortably (and even fun) by using gamification. For example, by including a score board in a multiplication learning game for students.

Progrezz is an open source (*MIT* licensed), geo-located and social serious game developed by an open community. The project were only an alpha and early prototype before this grade dissertation. It only allowed to share pre-built messages in some predefined geo-located points.

Said this, the main objective of this project is to design, analyze and implement a set of features to build a stable and usable prototype of the back-end or server application of Progrezz.

The structure of this document is almost the same as the hierarchy of modules to develop, being a total of four.

The first module consist on preparing the content of the existing prototype. It used Ruby-Sinatra as a web-app framework, with PostgreSQL as a database engine, connected to the code via the *DataMapper ORM* (Object-Relational Mapper). Besides that, it was poorly implemented (it was developed on a hack-a-thon in a weekend), meaning that this project should be started from scratch.

Modularity is a required feature, so the Sinatra Application has been totally rebuilt. Also, the database engine has been replaced by *Neo4j*, a Graph-based database managing tool, connected to *Ruby* with the *neo4jrb* wrapper.

The database model has been remodeled too. There were too many non-related tables, increasing the complexity of the prototype. With the relationship system of *Neo4j*, this can be fixed easily by redefining those model classes. No much changes are required, because *neo4jrb* is almost a clone of *ActiveRecord* or *DataMapper* (same function names, similar functionality…). At this point, the prototype features have been kept and improved, so the gameplay haven't been modified at all.

The second module contains all the tasks relative to managing players. The most important was to enable a security system to allow users to register and login. The *OAuth* standard (*OmniAuth Ruby* library) has been used and implemented to avoid security breaches, including login services from Facebook, Twitter, Google+, Steam and GitHub. This, combined with the `AuthManager`, allows the correct management of authentication and users registration.

The next step was to implement a system so the users could interact with each other. Anonymity is always required for all actions to keep the mysterious aura of the game, so there is no chat system to implement, but a geo-located, anonymous and persistent message system to notify other nearly users.

Then, because some lore is necessary, an automatic system to leave system or lore messages (split in fragments) was implemented. A routing machine like *MapQuest Directions* or *OSRM* (*Open Source Routing Machine*) is required to snap the geo-located points to the nearest pedestrian road to avoid the inaccessibility of a fragment. When a user logs-in and makes a "search fragment" request to the back-end, the back-end will add more fragments if these are not enough.

Stats are also needed for the gameplay. Following the simplest model, all users have a level profile containing a level and an experience bar. User actions will increment the experience bar. When it's big enough, the leveling system will add a new level to his profile.

A tweak on the REST and WebSocket API was require to allow the user to access all this changes. This implies changing some code, and formatting the output of the server response to fit the scheme of the functions responses.

The third module is related to item management. As a role playing inspire game (RPG), Progrezz initial prototype lacks an inventory system.

The first step was to implement a system that can "leave" deposits or groups of items in the world. Similar to the automatic fragment system, and based on the items' game parameters, this can instance and geo-locate deposits so online users may get closer and gather them.

A backpack system can be implemented by adding relations between users and items nodes. Each relation represents a stack on the user's backpack or inventory, including the amount of that stack. Relations' management can be repetitive; backpacks' instance methods were implemented to encapsulate this kind of tasks (adding or removing stacks of items, splitting stacks…).

Those gathered items are not only decorative: they are used to build or craft new items by using recipes or schemes. Each recipe requires an input (a list of items and their quantities) to build an output (item and its quantity). The recipes have been splitted in ranks, forcing users to level up to unlock new recipes' ranks.

As it was said in the previous module, this one needed some modifications on the back-end's API system.

The fourth and last module includes all systems related to social events. This part is the thickest of the development, so only the beacon subsystem was implemented.

Beacons are geo-located nodes that refers to a place which generates "positive energy" with solidarity actions. For example, a km0 restaurant that recycles may have a beacon marking its location.

Alive beacons mean a significant improvement for players (for example, rare deposits have more chances to spawn). Also, beacons may also connect to each other by using the implemented `BeaconsManager`'s triangulation algorithm.

Finally, this module required some modifications on the source code of the back-end REST API.

Conclusions

This project has served me in many aspects: education, learning, knowledge improvement ... But the most important was been being able to work on a social project.

7.2 After its completion I have obtained some theoretical and practical conclusions, listed below:

- A social project like Progrezz needs a lot of work to become a serious project; that is to say a lot of community support. However, unless the project is fairly advanced and attractive, it will be hard for developers to join and help, so it is clear that this will be a long-term project.
- For a project of this magnitude it is necessary to invert many hours of design and analysis so it can have a hopeful future, because features and modules will be added continuously. This means that it should be maintainable, avoiding having to make changes to the project source code.
- A lot of knowledge acquired from the subjects studied during the grede have been applied in Progrezz's development; from basic computer programming subjects, to those concerning management systems and databases, and to those that require more time and effort on building systems and applications, such as interactive applications programming (specially event-driven programming), design analysis algorithms and so on.
- The gameplay and usability of a video game is one of the most important factors for people to keep contact with it. It is necessary then to implement and enable a set of tools so administrators, writers, developers and managers can update the game parameters as the project needs it.
- Personally, I have found that the UML diagrams can be very useful in expressing ideas and concepts. These diagrams represent complex concepts in a simple and intuitive manner, and, in this case, they have served to represent the different structures of the database.
- This experience has given me the opportunity to meet and work with people of great talent, expanding my horizons, allowing me to apply and learn skills to get a product developed as if it were a professional job.

In conclusion, all the sections and tasks described in the grade dissertation's draft have been implemented, so the initial estimation of time of each of the tasks can be considered correct.

Capítulo 8. Presupuesto

En este apartado se tratará de aproximar cuánto hubiese costado el desarrollo de este proyecto si se hubiese tratado de un trabajo profesional remunerado.

8.1 Coste del proyecto

Hay que tener en cuenta dos partes en el coste del proyecto:

- Desarrollo: Se deberá mantener un trabajador o grupo de trabajadores formados (ingenieros informáticos) para que puedan diseñar, analizar e implementar todas las características del proyecto. Se estiman 280 horas de trabajo distribuidas a lo largo de 3 meses. No es necesario pagar licencias de terceros ni costes extra. En la Tabla 8.1 se detallan los costes de este apartado.

Puesto	Horas	Meses	Coste €/h	Coste €/mes	Coste total
Ing. Informático	280 horas	3 meses	15 €/hora	1400 €/mes	4200 €
Total	**280 horas**	**3 meses**	**15 €/hora**	**1400 €/mes**	**4200 €**

Tabla 8.1. Tabla resumen del coste de desarrollo.

- Mantenimiento: Una vez desarrollado, se deberá mantener un servidor dedicado para correr la aplicación durante un año. Se ha seleccionado una máquina de gama baja (AMD Opteron™ 1381HE, 4 GB RAM y 2x500 GB HDD) de la empresa STRATO. Un trabajador deberá mantener y revisar el servidor al menos 3 horas por semana. En la Tabla 8.2 se detallan los costes de este apartado.

Sección	Horas	Meses	Coste €/h	Coste €/mes	Coste total
Ing. Informático	144 horas	12 meses	15 €/hora	180 €/mes	2160 €
Servidor web	8760 horas	12 meses	0.054 €/hora	39 €/mes	468 €
Total	**8760 horas**	**12 eses**	**15.054 €/hora**	**219 €/mes**	**2628 €**

Tabla 8.2. Tabla resumen del coste de mantenimiento.

Apéndice A. Algoritmos

A.1.Subida de nivel

```
# Fichero /rb/game/mechanics/leveling.rb, línea 59
# Daniel Herzog Cruz, 1 de mayo de 2015
# Este algoritmo permite actualizar los datos de la DB para cuando un
# usuario realiza un acción, ganando la experiencia oportuna.
LevelingManagement::gain_exp(user, action_name):
  output := {}
  current_level := user.level_profile.level
  current_exp := user.level_profile.level_exp

  # Si ya tiene el nivel máximo, no hacer nada
  if(current_level >= MAX_LEVEL)
    return output

  # Extraer experiencia realizada, y añadir a la experiencia actual
  action_exp := DATA["exp"]["exp_per_action"][action_name]
  current_exp := current_exp + action_exp
  exp_for_next_level := _next_level_required_exp( current_level + 1 )

  # Acomodar "overflow" de experiencia, y subir niveles necesarios
  while(current_exp >= exp_for_next_level)
    # Incrementar el nivel y añadir a la salida
    current_level := current_level + 1
    output[:new_level] := current_level

    if(current_level < MAX_LEVEL)
      # Si no es nivel máximo, restar la cantidad añadida de exp
      current_exp := current_exp - exp_for_next_level
      # Recalcular experiencia para el siguiente nivel
      exp_for_next_level := _next_level_required_exp( current_level + 1 )
    else
      # Si ya es nivel máximo, dejar la experiencia actual como 0.
      current_exp := 0

  # Generar parte de la salida
  output[:exp_gainer] := action_exp
  # Actualizar la base de datos
  user.level_profile.update(level: current_level, exp_level: current_exp)
  # Y retornar
  return output
```

A.2. Añadir objetos al inventario

```
# Fichero /rb/db/objects/backpack.rb, línea 90
# Daniel Herzog Cruz, 18 de mayo de 2015
# Este algoritmo permite actualizar los datos de la DB para cuando un
# usuario añade un objeto a su inventario.
Backpack::add_item(item, amount):
  output := { added_amount: 0, desired_add_amount: amount, info: "" }

  # Comprobar si se puede añadir a un stack existente
  for_each(stack in this.stacks.match_to(item))
    empty_space := stack.emtpy_space()
    # Si hay espacio, añadir lo que se pueda
    if(empty_space >= 0)
      # La cantidad a añadir es el espacio mínimo o la cantidad deseada
      add_amount =: min(empty_space, amount)
      stack.add_item(add_amount)
      amount := amount - add_amount
      output[:added_amount] := output[:added_amount] + add_amount

  # Si aun queda cantidad, se añade a otro stack
  if(amount > 0)
    # Si no queda espacio, informar al usuario
    if(this.stacks.count == this.slots)
      output[:info] := "Could not add all items: backpack is full."
    else
      output[:added_amount] := output[:added_amount] + amount
      this.create_relation(item, amount)
  # Retornar salida
  return output
```

A.3. Dividir pilas de objetos

```
# Fichero /rb/db/objects/backpack.rb, línea 196
# Daniel Herzog Cruz, 18 de mayo de 2015
# Este algoritmo permite actualizar los datos de la DB para cuando un
# usuario divida un stack de objetos de su inventario.
Backpack::split_stack(stack_id, restack_amount, target_stack_id = null):
  this.check_if_free_slots()
  stack := this.get_stack(stack_id)

  # Comprobar si hay un stack objetivo (o no)
  if(target_stack_id == null)
    stack.update(amount: stack.amount - restack_amount)
    new_stack := create_stack(stack.item, restack_amount)
  else
    stack_to := this.get_stack(target_stack_id)
    # Si no hay espacio, generar un error
    if(!stack_to.fits? restack_amount)
      raise Error
    # Realizar cambios
    stack.update( amount: stack.amount - restack_amount )
    stack_to.update( amount: stack_to.amount + restack_amount )

  # Borrar el stack viejo si está vacío
  stack.remove_if_empty()

  # Preparar y devolver la salida
  return { old_stack: stack.to_hash, new_stack: new_stack.to_hash }
```

A.4. Actualizar los vecinos de una baliza

```
# Fichero /rb/db/objects/beacon.rb, línea 196
# Daniel Herzog Cruz, 29 de junio de 2015
# Este algoritmo permite actualizar conexiones con balizas cercanas.
# 1. Primero, se buscará las balizas en el máximo radio de conexión.
# 2. Se ordenarán por distancia a la baliza actual.
# 3. Para cada baliza, se comprobará si está lo suficientemente cerca.
# 4. De ser así, se connectará con la nueva baliza.
# 5. Se repetirán los pasos 3 y 4 hasta que se agoten las balizas o se
#     llegue al máximo número de conexiones.
Beacon::update_neighbours():
  # Comprobar si la baliza está muerta
  if(this.caducated?)
    this.remove()
    return

  # Buscar en el máximo radio posible
  max_radius := Game::Mechanics::BeaconMechanics.max_radius()
  radius     := this.action_radius

  # Comprobar si ha muerto alguna baliza vecina
  this.check_neighbours

  # Comprobar si ya ha suficientes balizas
  neighbours_count := this.neighbours.count
  if(neighbours_count >= this.max_connections)
    return

  geo := this.geolocation
  # Si no, buscar balizas cercanas en la BD
  beacons := Game::Database::Beacon.search_by_radius( geo, max_radius )
  # Eliminarse a sí mismo, ya que no puede añadirse como vecina.
  beacons.delete(this)
  # Ordenarlas por cercanía a la posición de la baliza
  beacons.sort! { |a, b| distance(geo, a) <=> distance(geo, b) }

  # Para cada una, intentar asociarla si está lo suficientemente cerca
  for_each(b in beacons)
    # Seleccionar el radio más grande
    current_radius = (radius > b.action_radius)? radius : b.action_radius

    # Si está cerca, intentar conectar (connect_to) e intentar salir
    if distance( geo, b.geolocation) <= current_radius
      if(this.conect_to(b))
        neighbours_count := neighbours_count + 1
      # Salir si ya hay suficientes conexiones
      if(if neighbours_count >= this.max_connections)
        break
```

Apéndice B. Repositorios

B.1. Repositorios del proyecto

- Perfil del grupo "Team Progrezz", GitHub: https://github.com/teamprogrezz
- Repositorio del proyecto "progrezz-server", GitHub: https://github.com/teamprogrezz/progrezz-server

Bibliografía

[1] Colaboradores de OpenStreetMaps. (s.f.). *Open Street Maps*. Recuperado el 5 de junio de 2015, de http://www.openstreetmap.org

[2] Driessen, V. (s.f.). *A successful git branching model*. Recuperado el 5 de junio de 2015, de http://nvie.com/posts/a-successful-git-branching-model/

[3] ECMA-404. (s.f.). *The JSON Data Interchange Standard*. Recuperado el 5 de junio de 2015, de http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

[4] Gliffy Team. (s.f.). *Gliffy*. Recuperado el 5 de junio de 2015, de https://www.gliffy.com/

[5] MapQuest. (s.f.). *MapQuest Directions*. Recuperado el 5 de junio de 2015, de http://www.mapquest.com/

[6] Neo4j. (s.f.). *Sitio web Neo4j*. Recuperado el 29 de Mayo de 2015, de http://neo4j.com/

[7] Neo4j Team. (s.f.). *Sitio web de Neo4j*. Recuperado el 29 de Mayo de 2015, de http://neo4jrb.io/

[8] neo4jrb. (s.f.). *Wiki de neo4jrb*. Recuperado el 5 de junio de 2015, de https://github.com/neo4jrb/neo4j/wiki

[9] OmniAuth Team. (s.f.). *OmniAuth website*. Recuperado el 2 de junio de 2015, de http://intridea.github.io/omniauth/

[10] OSRM Team. (s.f.). *OpenStreet Routing Machine*. Recuperado el 5 de junio de 2015, de http://project-osrm.org/

[11] Ruby. (s.f.). *Ruby doc 2.2.0*. Recuperado el 5 de junio de 2015, de http://ruby-doc.org/core-2.2.0/

[12] Sinatra Team. (s.f.). *Ruby Sinatra readme*. Recuperado el 5 de junio de 2015, de http://www.sinatrarb.com/intro.html

[13] Wikipedai. (s.f.). *Distribución de probabilidad discreta*. Recuperado el 5 de junio de 2015, de http://es.wikipedia.org/wiki/Distribuci%C3%B3n_de_probabilidad

[14] Wikipedia. (s.f.). *Callback (informática)*. Recuperado el 5 de junio de 2015, de http://es.wikipedia.org/wiki/Callback_%28inform%C3%A1tica%29

[15] Wikipedia. (s.f.). *Class diagram (UML)*. Recuperado el 5 de junio de 2015, de http://en.wikipedia.org/wiki/Class_diagram

[16] Wikipedia. (s.f.). *Craft*. Recuperado el 5 de junio de 2015, de http://en.wikipedia.org/wiki/Craft

[17] Wikipedia. (s.f.). *Front-end y back-end*. Recuperado el 5 de junio de 2015, de http://es.wikipedia.org/wiki/Front-end_y_back-end

[18] Wikipedia. (s.f.). *Juego serio*. Recuperado el 5 de junio de 2015, de http://es.wikipedia.org/wiki/Juego_serio

[19] Wikipedia. (s.f.). *ORM*. Recuperado el 2 de junio de 2015, de http://es.wikipedia.org/wiki/Mapeo_objeto-relacional

[20] Wikipedia. (s.f.). *Triangulación*. Recuperado el 5 de junio de 2015, de http://es.wikipedia.org/wiki/Triangulaci%C3%B3n