Proyecto de Fin de Carrera

Pizarra Web Compartida

Albert Llop

21 de diciembre de 2008

Índice general

Capítulo 1

Desarrollo: Ruby on Rails

Se dispone en estos momentos de un motor completo en Javascript que permitiría todo el proceso necesario para los objetivos de este proyecto. Éste, sin embargo, necesita de un fondo que le aporte todo lo que html estático y javascript no es capaz de hacer, como por ejemplo, tratar con la base de datos. Por las razones comentadas en el primer capítulo, se ha considerado que Ruby on Rails sería la solución para implementar este fondo, y en este capítulo se comenta el proceso seguido, resaltando en cada caso las diferencias de utilizar Ruby on Rails frente a soluciones más clásicas como PHP o ASP.

1.1. Consideraciones previas

Antes de nada, para entender como funciona Ruby on Rails, hay que aclarar ciertos conceptos. Primero, el lenguaje en que se está programando es Ruby. Éste es un lenguaje interpretado, que al igual que lenguajes de scripting típicos como Perl, PHP o Python, se compilan dinámicamente a la hora de la ejecución. De hecho, existen implementaciones de interpretadores de Ruby en varios lenguajes, siendo la implementada en C la *oficial*, pero existiendo otras tan dispares como jRuby, una implementación en Java que permite utilizar cualquier librería de Java.

Ruby es un lenguaje de alto nivel, con una sintaxis que hace que sea muy fácil de entender para gente que no la conoce. Por ejemplo:

```
100.times do
  print "No hablaré en clase".upcase
end
```

La popularidad de Ruby se ha incrementado con el éxito de Ruby on Rails, y ha hecho que se puedan encontrar librerías para prácticamente cualquier cosa. La sencillez del código lo hacen muy adecuado para el mundo del desarrollo web, donde se prioriza la agilidad de desarrollo, antes que una gran eficiencia. Existen múltiples técnicas de cacheo que hacen que el procesado necesario por el lenguaje de scripting sea mínimo, relegando todo el trabajo al servidor web (Apache, por ejemplo), y a unas consultas a la base de datos bien eficientes. Por tanto, lenguajes de más bajo nivel que podrían aportar una mayor eficiencia del código no se consideran adecuados por requerir de un proceso de desarrollo más largo y costoso, para unas ganancias relativamente mínimas. El uso de este tipo de lenguajes se relega a partes muy pequeñas y precisas, normalmente en los culos de botella donde puedan ser útiles. Se considera más beneficioso poder desarrollar más, de forma más sencilla para así evitar bugs indeseables, y utilizar dichas técnicas de cacheo y de gestión de base de datos para lograr la eficiencia.

Rails es un Framework escrito en ruby que permite un desarrollo web ágil y sencillo. Se basa en hacer fácil el trabajo del programador, y en su famoso *Convention over configuration* (convención antes que configuración). Debido a que la mayoría del tiempo, el desarrollo de webs es un proceso repetitivo, es posible establecer una serie de patrones que asumir, y solo modificar en los casos especiales en que *lo normal* no es suficiente.

Está basado en una arquitectura MVC (Modelo Vista Controlador), permitiendo aplicar el patrón en 3 capas estudiado las diversas asignaturas de Ingeniería del Software, así como la mayoría de buenas prácticas, hasta ahora mayoritariamente difíciles de aplicar en el mundo del desarrollo web.

Rails, gracias a la sencillez de Ruby, promueve la práctica del desarrollo ágil de software (Agile software development), una metodología de desarrollo que se basa en aligerar el proceso de desarrollo, alejarse de metodologias *burocráticas*, centrándose en conseguir software de alta calidad de forma muy rápida. Estas características son muy apreciadas en el mundo del desarrollo

web, puesto que como ya se ha comentado, la eficiencia suele ser secundaria, y los procesos, al ser repetitivos en su mayoría, hacen de otras metodologías demasiado pesadas y lentas.

A lo largo de este capítulo se irán remarcando los puntos por los cuales Ruby es tan adecuado al desarrollo web, porqué Rails permite un desarrollo más ágil, y porqué una metodología basada en la escasez de documentación y planificación es posible, y de hecho, beneficiosa, en el contexto de este proyecto (y en el de otros similares de desarrollo de webs).

1.2. Primer ciclo

La metodología de desarrollo ágil defiende un desarrollo iterativo, por ciclos pequeños que generen de forma rápida ciertas funcionalidades. Cada ciclo amplia funcionalidades hasta que al final estén todas incluidas. Cada ciclo debe ser completo, con sus fases de análisis de requisitos, diseño, implementación y revisión. La fase de documentación tiene como resultado este capítulo.

En este primer ciclo, se pretende obtener todas las funcionalidades necesarias para poder crear y editar documentos.

1.2.1. Análisis de requisitos

Gracias a que se ha tratado la implementación del Javascript de forma previa a este ciclo, se conocen ya los requisitos básicos necesarios para poder utilizar el motor de dibujo. De forma detallada, las funcionalidades requeridas para este ciclo son las siguientes:

- Poder crear, editar y borrar documentos, dándoles un título, una descripción y asignándole una cantidad de páginas fija, mediante interacción normal por HTML.
- Tener soporte para documentos, páginas y los elementos contenidas en ellas, de forma que el motor de comunicación en Javascript pueda comunicarse con la aplicación para editar las pizarras.

1.2.2. Diseño

El primer paso natural a la hora de planear un ciclo en Ruby on Rails es planeando la estructura de la aplicación en cuanto a modelos y controladores se refiere. Los modelos serán necesarios para poder guardar los datos en la base de datos, y deben estar relacionados adecuadamente de forma que sea fácil y directo acceder a datos de modelos *cercanos*. También es importante considerar los controladores en los que agrupar las funciones. Sería posible implementar la aplicación entera en un controlador enorme, pero obviamente esto no sería práctico ni productivo.

Modelos

El diagrama de clases es extremadamente sencillo en este ciclo. Solamente son necesarias tres clases, una para representar el documento, otra para las páginas y otra para los elementos. Durante la discusión sobre la implementación del javascript ya se discutió que la descripción de cada elemento se guardaría como una cadena de texto JSON representativa de los elementos que se crean mediante javascript, por lo tanto, un elemento no necesita más variedad de atributos.

El diagrama final puede observarse en la figura 1.1

Controladores

En estos momentos se considera suficiente agrupar todas las funcionalidades que traten con los documentos dentro de un solo controlador, `documents_controller`.

1.2.3. Implementación

Para inicializar una aplicación en rails, hay que crear dicha aplicación, y configurar la conexión con la base de datos. Desde el principio, se le ha dado el nombre `drawme`.

```
rails drawme
```

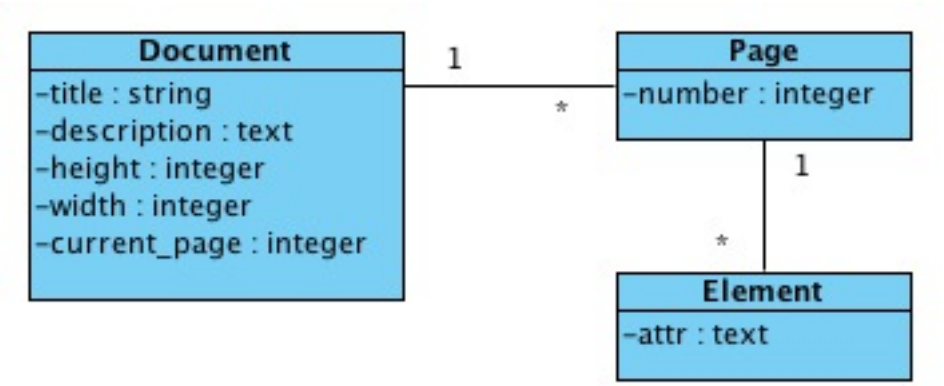

Figura 1.1: Clases del dominio, 1er ciclo

Modelos

Este comando generará los archivos que formen el esqueleto de la aplicación. Una vez hecho esto, hay que generar los modelos que se han planeado:

```
$ script/generate model Document
$ script/generate model Page
$ script/generate model Element
```

Estos comandos generarán los archivos dentro de la carpeta `/app/models`, además de las *migraciones* correspondientes para crear las tablas en la base de datos. Una migración es una acción sobre la base de datos, de cualquier tipo. Dentro de la carpeta `/db/migrations` se encuentran, ordenadas de forma temporal mediante un timestamp. Gracias a esto se pueden ir realizando modificaciones sobre la base de datos de forma ordenada, permitiendo así, por ejemplo, que si en un futuro se pretende realizar una modificación, no se tenga que eliminar la base de datos entera y volverla a crear.

A forma de ejemplo, la migración para la tabla del modelo `Document` será así:

```
class CreateDocuments < ActiveRecord::Migration
  def self.up
    create_table :documents do |t|
      t.string :title
      t.text :description
      t.integer :current_page, :default => 1
      t.integer :height, :default => 600
      t.integer :width, :default => 800
      t.timestamps
    end
  end

  def self.down
    drop_table :documents
  end
end
```

Ya en este punto se pone en práctica el concepto de *convention over configuration*, en que se ha creado el modelo `Document`, y la migración generada llama a la tabla de la base de datos `documents`. Esto siempre es así, un modelo es un nombre en singular, y su tabla es el equivalente en plural. Sabiendo esto, no habrá en ningún momento problemas de nomenclatura, y en caso de que sea necesario llamar a una tabla con un nombre distinto al de su modelo, será necesario añadir una línea en su modelo correspondiente (en este caso `/app/models/document.rb`).

Un documento puede tener múltiples páginas, y según las prácticas habituales para implementar este tipo de relaciones en bases de datos relacionales, hay que añadir una clave externa en la tabla de las páginas. Para hacer esto, en Ruby on Rails se haría lo siguiente:

```
class CreatePages < ActiveRecord::Migration
  def self.up
    create_table :pages do |t|
      t.integer :number, :null => false
      t.references :document

      t.timestamps
    end
    add_index :pages, :document_id
  end

  def self.down
    drop_table :pages
  end
end
```

Por convención, los campos que referencian otra tabla, tienen la nomenclatura del tipo `modelo_id`, y se aprovecha en este caso para añadir un índice sobre este campo, para que cuando sea necesario consultar las páginas de un documento, no sea necesario recorrer la tabla de páginas entera.

La migración para `Element` es igual a la de `Page`.

```
class CreateElements < ActiveRecord::Migration
  def self.up
    create_table :elements do |t|
      t.text :attr, :null => false
      t.references :page

      t.timestamps
    end
    add_index :elements, :page_id
  end

  def self.down
    drop_table :elements
  end
end
```

Para trasladar estas migraciones a la base de datos es necesario ejecutar la línea siguiente en la línea de comandos:

```
$ rake db:migrate
```

Estas migraciones, no obstante, solamente sirven para tratar con la base de datos. Para que la aplicación sepa que un `Document` tiene múltiples `Pages`, es necesario dejar constancia en los modelos:

```
class Document < ActiveRecord::Base
  has_many :pages, :dependent => :destroy
  has_many :elements, :through => :pages
end

class Page < ActiveRecord::Base
  has_many :elements, :dependent => :destroy
  belongs_to :document
end

class Element < ActiveRecord::Base
  belongs_to :page
  belongs_to :document, :through => :page
end
```

Con estas líneas, será posible en un futuro, ejecutar instrucciones como las siguientes:

```
# Añadir una página a un documento
document.pages << Page.new

# Añadir un elemento a una página
page.elements << Element.create(:attr => "...")

# Borrar todas las páginas de un documento
document.pages.clear
```

Cualquier operación que trate las relaciones entre estos elementos será posible de forma más sencilla, siempre tratando con la base de datos de forma transparente.

Controladores y vistas

De forma similar a los modelos, para generar los archivos necesarios para un controlador:

```
$ script/generate controller Documents
```

El archivo generado, utilizando las convenciones, será `/app/controllers/documents_controller.rb`. Dentro de este archivo, cada función implementada tendrá una traducción directa en las capa de vistas, por lo tanto el proceso de implementar acciones en controladores suele ir al mismo ritmo que se van implementando las vistas.

Debido a que es el primer ciclo, y que de momento aún no se tiene muy claro cómo acabará estructurándose la web, se optará por tener lo que en Rails se llama un *Scaffold* con unas pocas modificaciones, para poder editar los documentos. Un Scaffold es una envoltura para un modelo, con las cuatro operaciones básicas que se pueden querer realizar: crear, mostrar, modificar y borrar (`create`, `show`, `update` y `destroy`). Debido a la naturaleza de la web, es necesario otras acciones, previas a estas cuatro. Por ejemplo, antes de crear un documento, es necesario mostrar

un formulario en el que rellenar los datos necesarios. Dicha acción se llamaría `new`. Para poder mostrar un elemento en concreto, primero hay que listar los que hay, y dicha acción se llama `index`. Para poder modificar un elemento primero hay que mostrar como está en este instante, además de proporcionar un formulario con el que poder especificar los cambios. Ésta acción se llamaría `edit`. Para eliminar un elemento solo es necesario saber qué elemento eliminar, por lo tanto es posible ejecutar tanto desde `show` como desde `index`.

Éstas cuatro acciones forman el llamado `CRUD` (Create, Retrieve, Update y Delete), y equivalen a las cuatro comentadas, que junto con las otras tres, forman las siete acciones básicas de un controlador de Scaffold.

Puesto que estas acciones son tan comunes, Rails proporciona una forma de generar un Scaffold sencillo de forma automática, para utilizar como punto de partida.

```
script/generate scaffold Document
```

Gracias a las convenciones de Rails, este Scaffold buscará el modelo `Document` y creará el controlador `documents_controller` con el código adecuado, y las vistas correspondientes.

Este scaffolding, sin embargo, no tiene en cuenta las relaciones entre modelos, ni ningún comportamiento personalizado que se quiera tener. Así, los cambios necesarios para adecuar este scaffold al gusto de la aplicación serían los siguientes:

- En el momento de crear un documento, añadirle un número fijo de páginas vacías, con sus números de páginas correspondientes.
- El atributo `current_page` no debe ser editable, de momento, puesto que se manejará mediante la interfaz javascript.

Con estos pasos se puede considerar la primera funcionalidad objetivo cumplida. La segunda funcionalidad es permitir tratar con la interfaz, mediante la comunicación con ajax y transportando elementos en formato JSON. Las funciones necesarias para ello se pueden resumir en las siguientes:

`add_element` recibiendo para qué página (y por consiguiente, qué documento), y el objeto en JSON, y retornando el identificador del mismo una vez guardado.

`remove_element` recibiendo el identificador del elemento.

`list_elements` recibiendo la página (y por consiguiente, el documento), devolviendo el listado de elementos en formato JSON.

`change_page` recibiendo la página a la que se quiere cambiar.

Cabe destacar de la implementación de estas funciones, la facilidad que da Rails para tratar con peticiones en Ajax. En el caso de la función `list_elements`, por ejemplo, lo lógico sería realizar una búsqueda en la base de datos, teniendo luego un vector de elementos, que luego se deberían traducir a JSON, y mostrarse en la vista.

```
def list_elements
  render :json => Page.find(params[:page_id]).elements
end
```

Debido a que JSON se ha convertido en un formato standard en este tipo de acciones, Rails usa las librerías de Ruby para la conversión de objetos a sus equivalentes en JSON, y mediante una sola línea es posible devolver un objeto exactamente equivalente al que se tiene en ese momento en Ruby, pero que javascript podrá entender.

Con esto, se tienen las dos funcionalidades originales, y ya sería posible empezar a dibujar en la interficie en Javascript, con múltiples usuarios al mismo tiempo con persistencia en una base de datos.

1.3. Segundo Ciclo

Se considera interesante que la siguiente evolución de la aplicación sea la introducción del concepto de Usuario.

1.3.1. Análisis de requisitos

- Poder registrarse en la web de la forma más simple posible.
- Tener un panel sencillo desde el que manejar sus documentos
- Seguridad básica para que no se pueda acceder a elementos que no se debería tener acceso.

1.3.2. Diseño

En cuanto al diagrama de clases del dominio, en este caso aparece un nuevo modelo usuario `User`, el cual tiene varios `Document`'s, y tiene los atributos básicos de nombre de usuario (`login`), `email` y `password`.

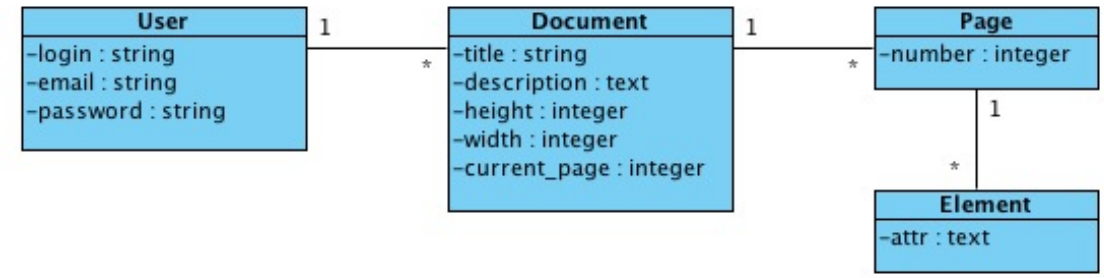

Figura 1.2: Clases del dominio, 2º ciclo

Para manejar todo el proceso de registro, y cualquiera del resto de acciones necesarias, si llegara el caso de recuperar contraseña, dar de baja, o editar los datos de usuario, se cree necesario crear otro controlador diferente, al que se llamará `users_controller`. También, al tener en este caso una parte privada y una parte pública de la web, es necesario tener una serie de acciones que no tienen que ver ni con manejar documentos ni con registrarse. Por ello, y para cualquier otra acción sencilla que se quiera añadir en un futuro referente a lo que sería la web en si (página de contacto, ayuda, información del autor, etc), se ha decidido crear un controlador extra llamado `website_controller`.

1.3.3. Implementación

Registro de usuarios

El proceso de manejo de usuarios es un tema muy delicado en cuanto a seguridad se refiere. Existen sobradas razones para blindar totalmente este proceso, que la contraseña esté guardada de forma encriptada, y que los datos que introduce el usuario estén seguros. Y puesto que la funcionalidad de poder registrarse en una web es algo tan común en el mundo de las webs, existe alguien que se ha encargado de facilitar todo este proceso en forma de plugin para Rails.

Los plugins pueden servir para multitud de propósitos, y suelen solucionar necesidades comunes de las webs que Rails no incluye para no sobrecargar el Framework. En este caso el plugin

se llama `restful_authentication` [1]. Este plugin se encarga de las tareas de registro de usuarios creando un modelo llamado User, tal y como se había planeado, y un controlador con las funciones de registro (`signup`), login y logout. Además, aporta una serie de funciones de ayuda (`helpers` en Rails), que permiten tratar en todo momento con el usuario, y por ejemplo, saber si la persona que está realizando una petición de una página está logueado o no.

Este plugin utiliza la técnica de salted passwords [2] para almacenar las contraseñas en la base de datos. Ésta técnica genera una cadena aleatoria (salt) de 40 caracteres mediante `SHA1`, que se utiliza para generar el llamado salted password, también de 40 caracteres mediante `SHA1`. Se considera el standard de facto en cuanto a registro y guardado de contraseñas se refiere, y es una técnica utilizada en numerosos protocolos criptográficos, como por ejemplo SSL. Esta encriptación dificulta los ataques a contraseñas mediante diccionario de forma exponencial, puesto que por cada palabra *común* de los diccionarios usados, es necesario tener las 2^{160} posibles combinaciones introducidas por el salt de 160 bits. Incluso si la base de datos se comprometiera, y un atacante tuviera acceso a alguno de los campos, tanto el salt, como el password encriptado, aún debería romper la clave mediante el método tradicional, puesto que ya sabría cual es el salt, pero seguiría teniendo que encontrar cual es la clave que, añadida al salt, y encriptándola mediante SHA1, genera el password encriptado.

Este plugin requiere de el campo extra salt, que no se había planteado en un principio en la etapa de diseño.

Panel de control

Mediante el plugin de usuarios es cuestión de controlar si se está logueado o no para mostrar la pantalla inicial de la web, del controlador `website_controller`, o la pantalla con el *panel de control* del usuario, también dentro del mismo controlador. El siguiente paso es evolucionar el Scaffold de documentos para controlar que en todo momento, solo el propietario de los documentos pueda realizar las acciones solicitadas.

Así, por ejemplo, las funciones del `documents_controller` evolucionan de unas líneas como las siguientes:

```
def update
  @doc = Document.find(params[:id])
  @doc.update_attributes params[:document]
  redirect_to :action => "edit", :id => @doc.id
end
```

A algo como lo siguiente:

```
def update
  @doc = Document.find(params[:id])
  redirect_to :action => :index unless @doc.user == current_user
  @doc.update_attributes params[:document]
  redirect_to :action => "edit", :id => @doc.id
end
```

Dichos cambios son mínimos, por la simplicidad en que todos ellos están implementados. La parte de comunicación con Javascript, de momento, se dejará abierta hasta que se decida qué política se aplicará para permitir o no a los usuarios ver o editar sus contenidos.

[1] http://github.com/technoweenie/restful-authentication
[2] http://en.wikipedia.org/wiki/Salt_(cryptography)

1.4. Tercer Ciclo

En estos momentos un usuario puede crear y editar sus documentos, pero aún no existe ningún tipo de seguridad a la hora de trabajar desde la pizarra. La idea original para esta aplicación, es que los usuarios podrían acceder a documentos dependiendo de los permisos que se le hubiera dado, tanto por la parte del grupo como independientemente. Para este ciclo, el objetivo será la introducción del concepto Grupo, de forma que se puedan agrupar usuarios de forma sencilla y rápida, y que se puedan asignar en un futuro, permisos de forma rápida a estos grupos de gente.

1.4.1. Análisis de requisitos

- Poder crear, editar y borrar grupos.
- Un grupo deberá tener un dueño, que será el creador, que siempre tendrá los máximos permisos, y que podrá invitar, echar y promocionar a otros usuarios. Una vez invitado, un miembro puede ser promocionado a administrador, de forma que pueda a su vez, invitar, promocionar y echar a gente. Un administrador no puede eliminar el grupo o echar / degradar a otros administradores, privilegios reservados al dueño del grupo.
- Los usuarios invitados pueden aceptar o rechazar las invitaciones, además de salir del grupo voluntariamente, después de ser invitado. Para evitar una avalancha de invitaciones, solo un usuario puede invitar a un usuario a un grupo.

1.4.2. Diseño

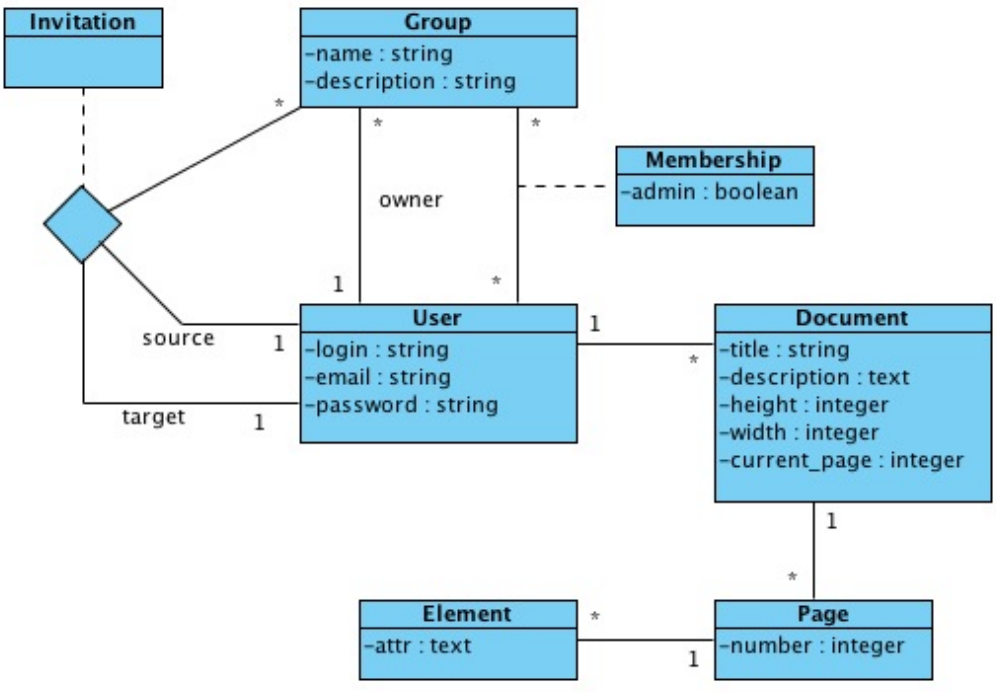

Figura 1.3: Clases del dominio, 3er ciclo

En este paso se hace todo el sistema de clases un tanto más complejo. Un grupo tiene una relación doble con la clase `User`, teniendo un dueño, y múltiples usuarios. Además, puesto que

los miembros pueden ser administradores, existen dos posibilidades. Una sería tener una triple relación, de dueño, administradores y usuarios normales; la otra sería tener una clase asociativa `Membership`, que contenga la información de si este miembro tiene permisos de administrador o no. La segunda opción se cree más eficiente, puesto que siempre será más fácil modificar un atributo de la tabla `Membership` que destruir una relación y crear otra nueva, además de simplificar el proceso de encontrar todos los miembros de un grupo, sean administradores o no.

En cuanto al proceso de realizar invitaciones, es necesario almacenar de alguna forma temporal dichas invitaciones, y la solución natural es una asociación ternaria entre un grupo y dos usuarios, el que invita (source) y el invitado (target).

Ruby on Rails aún no soporta relaciones asociativas, debiéndose por tanto normalizar el diagrama antes de poder traducirse a la estructura de modelos.

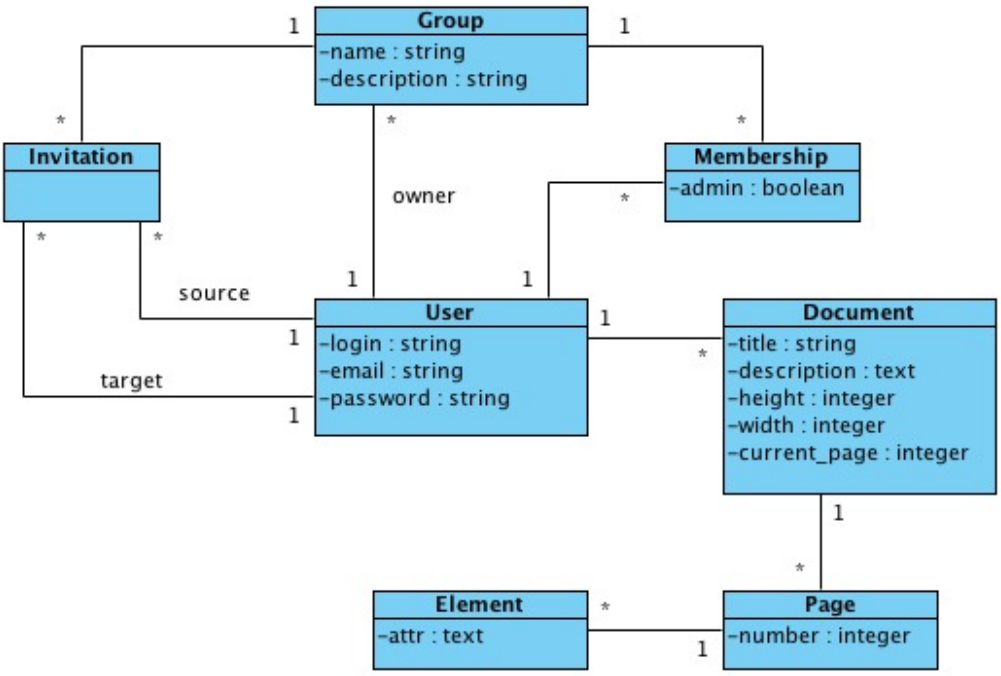

Figura 1.4: Clases del dominio, 3er ciclo, diagrama normalizado

En cuanto a modelos, se cree necesario crear un nuevo controlador para todas las acciones referentes a los grupos

1.4.3. Implementación

A pesar de la nueva complejidad de las clases introducidas (tres nuevas clases, con un total de seis nuevas relaciones), el proceso de traducción de este diagrama a clases es puramente burocrático. El peso de controlar las restricciones dejadas atrás por la normalización recae sobre los controladores, que a la hora de invitar a gente se deberá comprobar que no sea miembro y que no haya sido invitado ya, todo sostenido sobre un Scaffold para los grupos, de la misma forma que se ha hecho con los documentos, pero con un pequeño extra de complejidad.

La forma de organizar todas estas nuevas funcionalidades es mostrando una lista de grupos (acción `index` del Scaffold) dentro del panel de control, de la misma forma que los documentos, y que al ir a editar un grupo se muestre un formulario si eres administrador, o una vista simple

si no se es. De la misma forma, el pequeño formulario para invitar a gente solo aparecerá si el usuario es administrador. En cuanto a los usuarios invitados, cuando se entre en el panel, en caso de que haya sido invitado a algún grupo, se le mostrará un mensaje con el nombre de usuario de la persona que ha invitado y el nombre del grupo, con opción de aceptar o rechazar.

1.5. Cuarto Ciclo

Una vez se tiene el concepto de Grupo introducido en el sistema, está todo preparado para poder generar todo el sistema de permisos para la interfaz Javascript.

1.5.1. Análisis de requisitos

- Individualmente para cada documento, se debe poder definir una lista de usuarios y grupos que pueden *participar* en este documento, definiendo para cada uno de estos si participan en calidad de espectador o de ponente. Un ponente puede *dibujar* en la pizarra, los espectadores solo reciben los cambios hechos en las pizarra.
- Además, se debe poder hacer una pizarra pública, de forma que cualquier usuario pueda participar en forma de espectador.

1.5.2. Diseño

El concepto de permisos es semejante al concepto de invitaciones, siendo la única diferencia que no es algo que se pueda rechazar. El creador del documento puede añadir o quitar conexiones entre un documento y un usuario o grupo, que además deberá constatar si es en forma de ponente o de observador, necesitando pues, una clase asociativa.

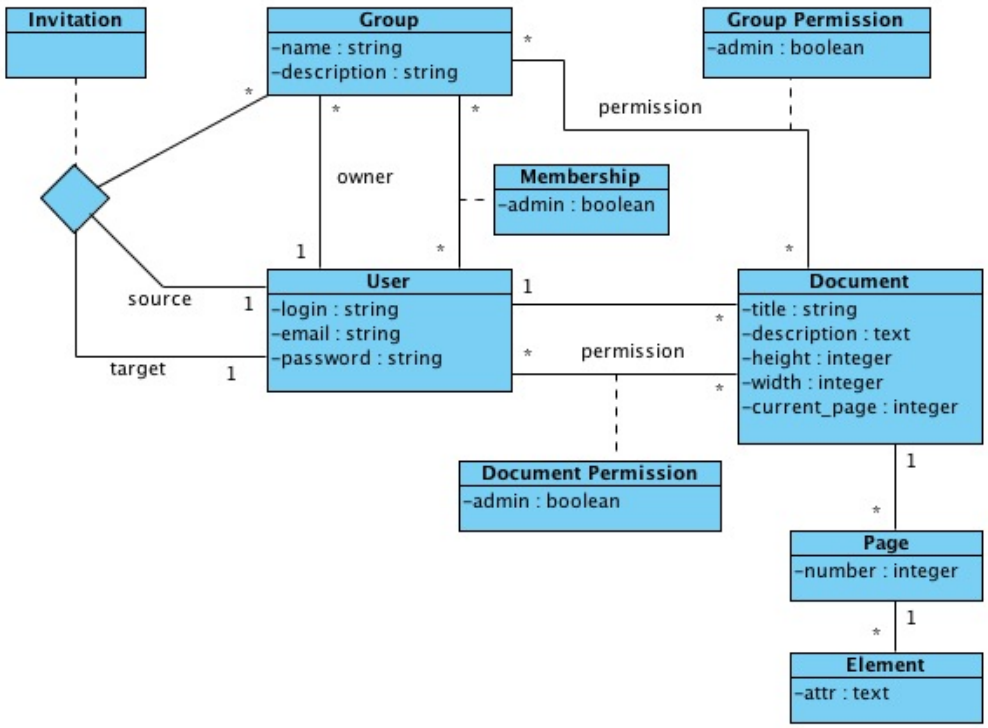

Figura 1.5: Clases del dominio, 4º ciclo

Y por lo tanto, normalizado:

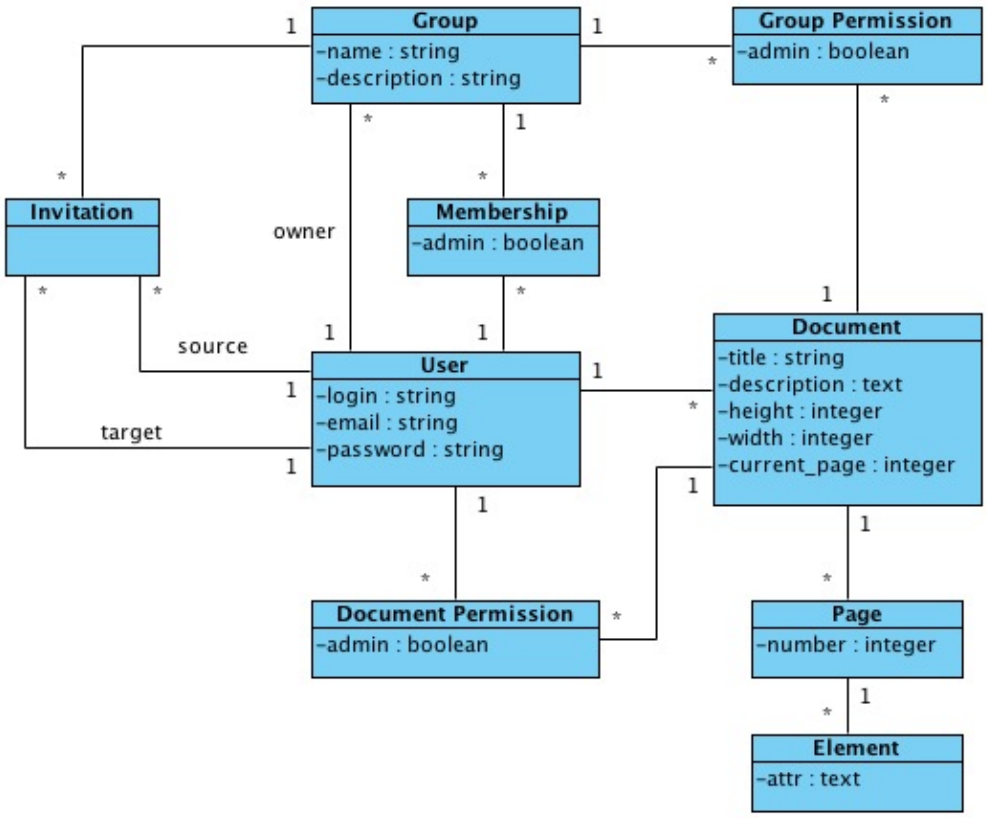

Figura 1.6: Clases del dominio, 4º ciclo, diagrama normalizado

1.5.3. Implementación

De nuevo, traducir estas dos nuevas clases a modelos de Rails es sencillo. El proceso de integrar estas nuevas funcionalidades en el sistema, es más un problema de encontrar una interfaz adecuada para todas estas operaciones, que de implementación de los controladores.

Finalmente el diagrama de clases se ha convertido en algo más grande, e incluso con las facilidades que ofrece Rails para manejar todas estas relaciones, es importante mantener todo bien organizado. Se considera una buena práctica tener la mayor parte del código implementada en los modelos. Esto es así puesto que una función implementada en un modelo siempre es reutilizable, y poniéndose un límite de unas 10 líneas de código en cada función del controlador, hace que se tenga en todo momento unas acciones bien claras y definidas, y por lo tanto, de muy fácil modificación en futuras iteraciones.

Así pues, lo más fácil es implementar funciones como las siguientes:

```
document.can_be_seen_by?(user)
document.can_be_edited_by?(user)
document.can_be_deleted_by?(user)
user.all_accessible_documents
user.invite(user,group)
group.is_member?(user)
group.is_admin?(user)
group.is_owner?(user)
```

```
group.add_user(user)
group.promote(user)
group.unpromote(user)
```

Todas son completamente autoexplicativas gracias a sus nombres, y contribuyen a que los controladores simplifiquen la mayoría de su código dejando solamente las líneas esenciales para que cualquier programador pueda saber de un vistazo qué hace este controlador.

Teniendo estas funciones implementadas, modificar las cuatro acciones de comunicación con el motor para tenerlas en cuenta, no necesita más que una línea extra.

1.6. Quinto Ciclo

En este ciclo se pretende tratar un tema que se ha dejado de banda desde el principio, que es la subida de archivos (pdf's o imágenes) para hacer de fondo en las páginas de los documentos. Debido a la naturaleza compleja de la tarea se ha preferido dejar para las últimas etapas del desarrollo, cuando hubiera una comprensión mayor del funcionamiento de Ruby on Rails.

1.6.1. Análisis de requisitos

- Poder crear páginas en blanco.
- Poder subir imágenes una a una creando páginas para el documento con ellas.
- Poder subir archivos comprimidos con imágenes dentro, que creen una página por cada imagen que se encuentre en el archivo. Las páginas se deberían ordenarán alfabéticamente según el nombre del archivo de la imagen. Como opción básica, se debe poder subir archivos zip, pero idealmente se debería poder subir otros formatos comunes, como rar o gz.
- Poder subir PDF's, que automáticamente convertirían cada página a una imagen, asignándola a una página nueva.

1.6.2. Diseño

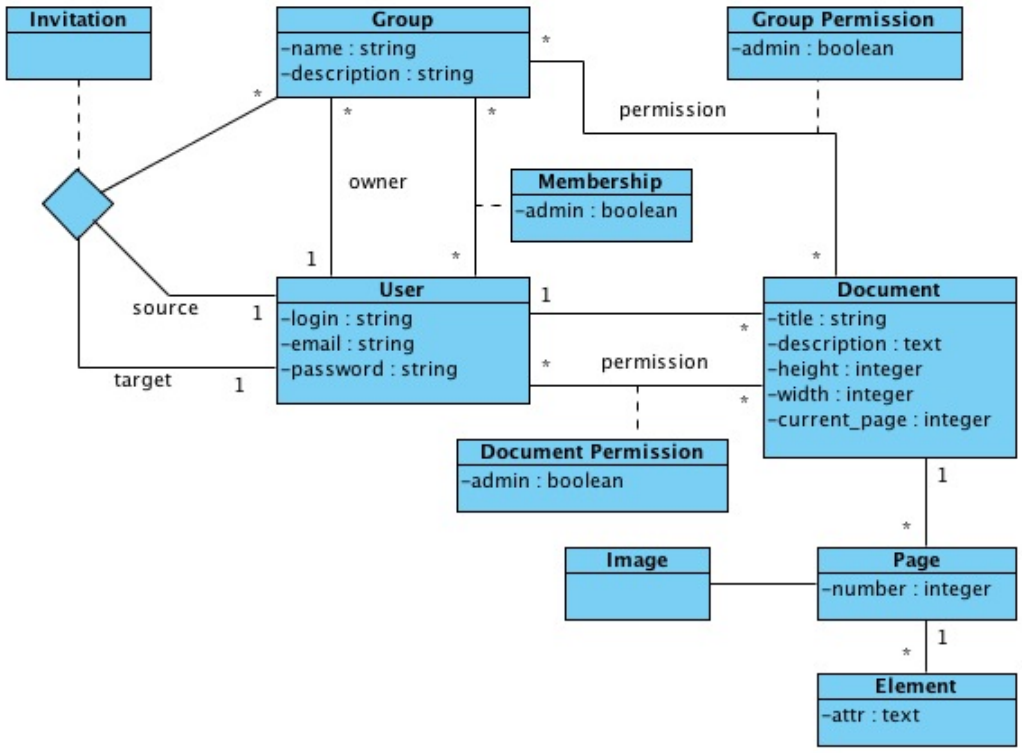

Figura 1.7: Clases del dominio, 5º ciclo

Debido a lecturas en ciclos anteriores se conoce que el proceso de subir imágenes es otro aspecto muy común de las páginas web, y por lo tanto existe un Plugin excelente que da todas

las herramientas necesarias para subir, generar thumbnails y almacenar imágenes en distintos soportes. Estos plugins generalmente requieren de un modelo extra que representará las imágenes, y que, como cualquier otro modelo dentro de Rails, se puede relacionar con el resto. Por tanto, aprovechando este conocimiento, se puede planear ya este modelo, y asignarlo, lógicamente, al modelo `Page`.

Este modelo solo tiene los atributos creados por el plugin, por lo cual se obvian en beneficio de una mayor claridad del diagrama.

1.6.3. Implementación

La forma en que se plantearán estas funcionalidades, serán mostrando una columna lateral en la edición de un documento, separada del formulario de edición del documento y de administración de permisos, mediante el cual se mostrarían miniaturas de las páginas, y se podrían ir añadiendo páginas mediante las opciones disponibles.

Páginas en blanco

Esta es la opción más sencilla, y no necesita más que una acción que cree una página de la misma forma que se creaban anteriormente, cuando se generaba una cantidad de páginas fija. De la misma forma, es posible añadir una opción para generar un número de páginas, establecido por el usuario rellenando un campo de texto.

Subir imágenes una a una

Esta opción es el ejemplo clásico de uso de un plugin para subir archivos. El plugin utilizado en este caso es `attachment-fu` [3] , el cual, añadiendo unas líneas como las siguientes, hace que se relacione directamente con los archivos subidos mediante la interfaz web.

```
class Image < ActiveRecord::Base
  belongs_to :page

  has_attachment  :content_type => :image,
                  :processor => 'MiniMagick',
                  :thumbnails => {:small => '85x>',
                                  :medium => '360x360>',
                                  :big => '800x800>'},
                  :storage => :file_system,
                  :path_prefix => "/document_images/"
end
```

Introduciendo esto en `/app/models/image.rb`, y al crear un objeto de tipo Image, pasándole como parámetro el input en el que se el usuario añade el archivo, hará las siguientes acciones:

- Comprobará que es una imagen, porque se le ha pasado el parámetro `:content_type =>:image`. En caso de no ser un archivo de tipo imagen válido, fallará el momento de crear el objeto, es decir, al realizar un `save` o un `create` sobre `Image`.
- Generará tres thumbnails, con los tamaños especificados usando el formato clásico de `ImageMagick` [4], mediante `MiniMagick` [5].

[3] http://github.com/technoweenie/attachment_fu
[4] http://imagemagick.org/script/command-line-options.php#resize
[5] http://rubyforge.org/projects/mini-magick/

- Guardará las imágenes en la carpeta `/document_images`. Puesto que se le ha especificado claramente que está en la raíz de la aplicación, las guardará ahí, y no en `public`, que es lo normal. `public` es la carpeta que está abierta al público, y si se guardara en `/public/document_images` en vez de en `/document_images`, cualquier persona podría acceder a las imágenes mediante la dirección pública como `http://dominio.com/document_images/..`

  De esta forma las imágenes están fuera del alcance, y se deberán servir mediante un método que lea este archivo y lo transmita, que al ser programado especialmente, puede controlar que el usuario que está reclamando una imagen tenga permisos para ver el documento, y no esté intentando ver documentos del resto.

Con la siguiente instrucción:

```
render :file => image.public_filename(params[:thumb])
```

es posible renderizar dicha imagen, de la misma forma que si accediera desde la dirección pública, pero estando dicha imagen fuera del alcance de todo el mundo, y pudiendo realizar las comprobaciones pertinentes.

Subiendo imágenes en un archivo comprimido

Esta segunda opción añade una complicación básica, en que ninguno de los plugins existentes permiten hacer nada parecido a lo necesario aquí, por lo que debe hacerse todo a mano. La forma en que funcionan las subidas mediante web, en cualquier sistema, es que el servidor genera un archivo temporal conteniendo el archivo en si, y que Ruby on Rails permite acceder de forma normal, igual que se accede a cualquiera de los parámetros rellenados en un formulario, pero en vez de ser un `String`, es un `Tempfile` [6].

Puesto que es un archivo que se deberá descomprimir, y después realizar las acciones necesarias para generar páginas con las imágenes descomprimidas, es necesario algún tipo de organización de directorios para poder trabajar de forma temporal, y sin que pueda haber interferencias entre varios procesos descomprimiendo al mismo tiempo. La carpeta `/document_temp` contendrá una serie de carpetas que se irán creando cuyo nombre será el timestamp del momento de la subida, concatenado con el identificador del documento para el cual se están subiendo imágenes. De esta forma, se puede en un futuro tener un control de carpetas que quizá hayan quedado descontroladas por algún proceso interrumpido de forma inesperada, siempre controlado que por ejemplo, dichas carpetas temporales hayan sido creadas hace más de una hora.

Dicha previsión se cree conveniente para poder blindar el proceso, y que ningún posible error en el código, o subidas de archivos maliciosos puedan ocupar espacio indeseado.

Mediante Ruby, igual que con cualquier otro lenguaje, es posible realizar todo tipo de operaciones de movimiento de archivos y creación de carpetas, por lo que crear dicha carpeta y eliminarla al final no es problema. El siguiente reto es conseguir descomprimir un archivo, en principio, desconocido, y por supuesto, contra más flexibilidad mejor. Existe un descompresor llamado `e`, de Martin Ankerl [7], que al estar escrito en Ruby, es muy fácilmente adaptable a las necesidades de este proyecto. El código utilizado por este descompresor soporta hasta un límite de 32 formatos distintos, siempre y cuando esté el descompresor necesario instalado en el sistema.

Una vez descomprimidas las imágenes, existe el problema de la organización interna del fichero. ¿Qué pasa si existen varias carpetas? En este punto se corre el riesgo de complicar extremadamente la cosa, puesto que no siempre se puede saber cuál es la intención del usuario

[6] http://corelib.rubyonrails.org/classes/Tempfile.html
[7] http://martin.ankerl.com/2006/08/11/program-e-extract-any-archive/

a la hora de subir las imágenes comprimidas en varias carpetas. En este caso, se ha tomado la política de extraer todas las imágenes, independientemente de la carpeta en que estén, y añadirlas de forma ordenada alfabéticamente.

Para hacer esto, de nuevo, aparentemente complejo, no hay más que usar uno de los módulos de la librería básica de Ruby, `Find` [8], el cual permite recorrer recursivamente una estructura de directorios, a partir de la cual generar un vector de archivos, referenciables posteriormente. Mediante este vector, y la función `File.basename`, es posible ordenar los archivos, guardados en el vector `files`, con la siguiente línea:

```
files.sort! {|a,b| File.basename(a) <=> File.basename(b)}
```

Ruby, de nuevo, ya tiene un algoritmo de ordenación implementado, al cual se le puede explicar por qué parámetro ordenar un vector, en este caso, por los nombres de archivo.

Teniendo, pues, un vector con referencias a las imágenes, ya comprimidas y ordenadas alfabéticamente solo queda generar las páginas con sus elementos `Image` generados. Mediante otra línea de texto explicada en la documentación del plugin `attachment-fu`, es fácilmente generable objetos de este tipo a partir de archivos ya existentes en el servidor.

Subiendo archivos PDF

Éste, a pesar de lo que pueda parecer, es un problema muy semejante al proceso anterior. El proceso de convertir de un archivo PDF a imágenes es posible gracias a la herramienta `ImageMagick`, un standard de facto en cualquier servidor web para el proceso de imágenes, que se ha usado tradicionalmente para la generación de thumbnails, redimensionado de imágenes, ademas de otras tareas un tanto más avanzadas como la adición de marcas de agua a imágenes. Entre sus posibilidades, existe la posibilidad de convertir entre cualquiera de los formatos soportados, encontrándose PDF entre ellos. Convertir de un PDF a una serie de imágenes es posible mediante el comando:

```
$ convert file.pdf image.jpg
```

Generando una imagen por página con los nombres `image0.jpg`, `image1.jpg`, etc. Teniendo, pues, estas imágenes descomprimidas en un directorio, no hay más que seguir el mismo proceso que con las imágenes descomprimidas para crear las páginas necesarias.

Otras posibilidades

Una de las posibilidades que se barajó el etapas preliminares del proyecto, fue la posibilidad de subir presentaciones PowerPoint directamente, puesto que es uno de los formatos más usados para presentaciones, y al fin y al cabo el objetivo básico de esta aplicación es facilitar dichas presentaciones cuando deben realizarse a distancia.

Esta tarea, sin embargo, es prácticamente imposible presuponiendo que la aplicación funcionará sobre una plataforma Unix, lo más normal en cualquier servidor web. En caso de estar funcionando en una plataforma Windows, sería posible mediante los componentes OLE32. En cualquier otro caso, este problema no se ha podido resolver.

[8] http://corelib.rubyonrails.org/classes/Find.html

1.7. Sexto Ciclo

La siguiente iteración intentará solucionar un problema aparecido en el ciclo anterior, y requiere de un enfoque distinto para el proceso de generación de páginas en masa a partir de PDF's o archivos comprimidos. Pero previa comprensión de las razones por las que puede haber un problema en este proceso, es necesario entender como funciona una web Ruby on Rails en un entorno *real*.

La forma tradicional de mantener un servidor web sirviendo una aplicación desarrollada en Ruby on Rails, es teniendo una o varias instancias de dicha aplicación sirviendo páginas. Cada instancia de dicha aplicación puede servir solamente una página a la vez, pero puesto que están cargadas todas las librerías, y además se utilizan múltiples técnicas de cacheo, y que por tanto no es necesario recargar código después de cada consulta, dichas páginas se sirven de forma extremadamente rápida.

Esto, sin embargo, quiere decir que mientras una aplicación está ocupada sirviendo una petición, no puede atender a otra. Generalmente, páginas normales de la aplicación tienen un tiempo de generación mínimo, pero ante tareas como las que atañen a este ciclo (conversión de PDF's y descompresión de archivos), significan mantener una de dichas instancias ocupadas durante segundos enteros. Así, por tanto, en cuanto hubiera una cantidad de personas subiendo archivos igual a la cantidad de instancias del servidor, nadie podría navegar, pues todas las instancias estarían ocupadas. No solo eso, sino que tener varios procesos realizando operaciones de actividad intensiva de CPU, y que además suelen necesitar de memoria extra, no es una situación deseable.

Éste es un problema similar al que afrontan servicios similares, como podrían ser, por ejemplo, cualquiera de los servicio de subida de vídeos como Youtube o Vimeo, y la solución para estos problemas, es la de tener un proceso a parte dedicado solamente a realizar dichas tareas. Este proceso, al ser solamente uno, evita situaciones en que el sistema pueda estar saturado tanto por no quedar instancias disponibles a los usuarios para seguir visitando la web, como por tener múltiples procesos intensivos de CPU realizándose al mismo tiempo.

1.7.1. Análisis de requisitos

- Implementar un mecanismo de forma que pueda existir un proceso a parte dedicado a realizar tareas costosas.
- Que el código actual utilice dicho sistema.

1.7.2. Diseño

La dinámica de funcionamiento para esta nueva funcionalidad, será la de ir creando tareas que el proceso aparté irá procesando. Es necesario pues, almacenar dichas tareas en la base de datos, con los datos necesarios para poder ser recuperadas y procesadas a posteriori. Este nuevo modelo, al que se llamará `Task`, aparece en el diagrama de clases que se encuentra en el diagrama 1.8:

1.7.3. Implementación

Este problema, como ya se ha comentado, es muy común en este tipo de servicios, con lo que existe un plugin adecuado, mucho más completo que si esta parte del código se hubiera tenido que programar desde cero. Dicho plugin es `delayed_jobs` [9], fruto del trabajo de la web Shopify

[9] http://github.com/tobi/delayed_job

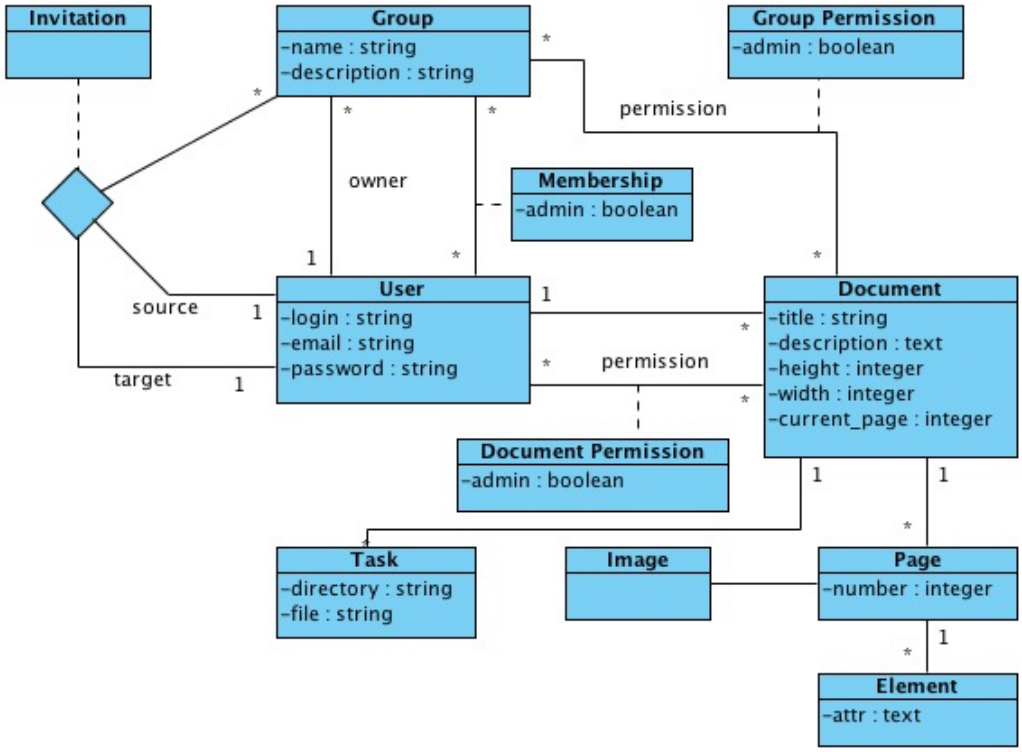

Figura 1.8: Clases del dominio, 6º ciclo

[10], un servicio de hosting de e-commerce, y que ha publicado su sistema de ejecución de tareas de forma pública en modo de plugin.

Este plugin generaliza mucho más de lo que se había pensado inicialmente, y es capaz de almacenar cualquier tipo de modelo, serializándolo para posterior recuperación, ejecutando siempre la función `perform`, de forma que con un solo proceso es posible realizar cualquier tipo de tarea que se quiera *separar* del flujo normal de trabajo. En este caso no es necesario, pero es perfectamente adecuado para posibles necesidades en el futuro.

La forma en que funciona es teniendo una tabla propia de tareas, llamadas `Jobs`, de forma que, en realidad, harían falta dos nuevos modelos para poder desempeñar estos trabajos, pero puesto que el plugin se encarga de almacenar el objeto que debe procesar serializándolo, es posible implementar un modelo que no tenga porqué estar reflejado en la base de datos.

La forma en que Rails hace que un modelo esté automáticamente replicado en la base de datos es mediante el motor `ActiveRecord`. Si en vez de heredar el modelo Task de ActiveRecord, se hereda de un objeto más simple, como un Struct, se tiene un modelo personalizable, que `delayed_jobs` puede serializar de forma muy sencilla, y que puede tener implementadas igualmente funciones. La clase Task quedaría así:

```
class Task < Struct.new(:document_id, :directory, :file)
  def perform
    # procesar el archivo file que está en el directorio directory,
    # y asignar las páginas generadas al documento document_id
  end
end
```

[10] http://www.shopify.com/

Y a la hora de añadir una `Task` a la cola de tareas de `delayed_jobs`:

```
Delayed::Job.enqueue Task.new(document,directory,file)
```

Para ejecutar el proceso que se encarga de ir reclamando `Jobs` e ir ejecutando sus funciones `perform`:

```
$ rake jobs:work
```

Este proceso no se parará aunque las funciones que se ejecuten salten una excepción, sino que la capturará y guardará el mensaje de error para posterior revisión, evitando así que tareas defectuosas o maliciosas puedan perjudicar al resto de elementos a procesar.