Ruby es un lenguaje de programación interpretado y orientado a objetos. La filosofía de Ruby es *don't repeat yourself (DRY)* - no te repitas. La idea de ruby es que usted no necesita repetir lo que ya ha definido en otro lugar. Esto hace a ruby muy compacto.

Rails es un armazón de librerías diseñado específicamente para crear aplicaciones de web. Como está hecho en ruby, es compatible con la filosofía de DRY. En vez de configuración, Rails prefiere convención y anotaciones. Esto proviene principalmente de las frustraciones con plataformas que obligan a uno a repetir en archivos de configuración XML una historia que ya se ha dicho en código.

Acerca del Curso

Este curso pretende proporcionar al usuario información suficiente para hacer aplicaciones de web profesionales utilizando Ruby on Rails. Está enfocado a personas que ya sepan algo de programación en otros lenguajes, lo cual quiere decir que durante el curso haré referencia a conceptos y librerías familiares a usuarios de otros lenguajes y productos.

El curso se enfoca a crear una libreta de direcciones utilizando un servidor MySQL. Durante el curso también trato de enseñar conceptos relacionados como pruebas de unidad, metodologías ágiles y control de versiones.

Contenido

Introducción

El internet ha madurado bastante desde los días de los start-ups. Ruby on Rails y Web 2.0 son el resultado de dicho proceso de maduración.

Condiciones Económicas

Los años 1997 hasta el 2001 fueron lo que se llamó el *dot com bubble*. Los *internet start-ups* - compañías pequeñas con fondos iniciales enormes proporcionados por compañías de capital riesgoso muy entusiastas - crearon una revolución basada en las expectativas de la nueva tecnología, y aunque muchas compañías fueron muy exitosas y lograron convertir estos fondos iniciales en una compañía redituable a largo plazo, muchas otras dejaron de existir en la caída de la burbuja del NASDAQ. Como toda burbuja tecnológica, esto es parte de un ciclo económico.

"You never leave a recession on the same technology that you entered it."
"Nunca sales de una recesión con la misma tecnología con la que entraste. "
Chairman Emeritus of the Board
Intel Corporation

Algo interesante que suele ocurrir en las partes bajas de los ciclos económicos es que hay una reorganización y re-evaluación de las tecnologías disponibles (lo que funcionó y lo que no valió la pena), y durante estas re-evaluaciones nuevas tecnologías y nuevas maneras de solucionar problemas aparecen, lo cual transforma cualitativamente la tecnología y recomienza el ciclo económico.

Cambios en Metodología

Durante estos años de la recesión en tecnología, los procesos también fueron re-evaluados, y las metodologías ágiles de desarrollo aparecieron, con personajes como Martin Fowler de ThoughtWorks y Dave Thomas de Pragmatic Programmers.

Éstas metodologías ponen énfasis en producir software de alta calidad en corto tiempo mediante comnunicación abierta. Hablaremos mucho más de metodologías ágiles más tarde.

Web 2.0

Este cambio de las expectativas de calidad en la tecnología de internet está siendo llamado, bien o mal, Web 2.0. Hay muchas opiniones al respecto, pero lo que todo mundo parece comprender es que la más reciente generación de aplicaciones en la red es muy diferente a las anteriores. Tim O'Reily trata de explicar Web 2.0 como una serie de principios y prácticas, que incluyen ciertos patrones de diseño dinámicos, como RSS, tags, colaboración, AJAX, etcétera.

Todos estos principios y cambios en expectativas de calidad, tanto en el software producido como en la metodología para producirlo requieren de un dinamismo mucho mayor a lo que era necesario antes con lenguajes compilados como Java. Así que más y más desarrolladores, sufriendo los problemas de la tecnología existente y bajo presion por la nueva metodología y expectativas, comenzaron a buscar alternativas.

Durante esta búsqueda de alternativas, un programador Danés llamado David Heinemeier Hansson utilizó el lenguaje Ruby para crear una aplicación llamada Basecamp. Las librerías reutilizables que salieron de este proyecto es lo que hoy conocemos como Ruby on Rails.

Instalando Rails

Este capítulo explica cómo puede usted instalar Rails en su plataforma de desarrollo.

Rails es un proyecto de fuente abierta, y como tal, hay varias maneras de instalarlo y obtenerlo dependiendo de su plataforma y sus necesidades.

En este artículo cubriremos dos maneras de instalar rails:

- Para principiantes
- Para usuarios avanzados y ambientes de prueba, pre-producción y producción

Para Desarrolladores

Si usted es un desarrollador de software, la manera más sencilla de utilizar rails es mediante una solución todo-en-uno que no afecte su ambiente actual.

Windows

Si usted utiliza Windows, puede instalar la plataforma Rails completa (incluyendo una version de mysql) utilizando proyecto Instant Rails. Éste proyecto incluye apache, ruby, rails, y el servidor de mysql, todo listo para ejecutarse.

Macintosh

Si utiliza Macintosh OS X, puede utilizar el proyecto Locomotive. Locomotive incluye ruby, rails y sqllite (en vez de un servidor de mysql). Si usted tiene un servidor mysql en otra máquina, locomotive incluye las librerías de mysql para conectarse.

Estos ambientes de desarrollo son adecuados para comenzar a escribir aplicaciones en Rails en su propia máquina, pero no son adecuados para ambientes multiusuario. Así que una vez que usted tenga su programa listo para que otras personas lo utilizen (ya sea en modo de prueba, pre-producción o producción), necesitará comprender como instalar un ambiente de rails desde cero.

Componentes de Rails

Para efectos de instalación, Rails se puede dividir en los siguientes componentes:

- **Lenguaje Ruby**. El lenguaje en el cual fué escrito Rails.
- **Las librerías de Rails**. Rails se compone de varias librerías escritas en ruby.
- **La base de datos.** Una base de datos relacional basada en SQL es indispensable para un sistema de desarrollo de aplicaciones web moderno. La mayor parte de los ejemplos utilizan MySQL.

Ruby

Ruby es un lenguaje de programación interpretado. Ruby funciona con cualquier sistema operativo moderno, pero se siente más "en casa" en sistemas tipo Unix.

La manera en que usted instale ruby depende mucho de su tipo de máquina y sus necesidades. Es recomendable que lea todo lo relevante a la plataforma que desee instalar antes de que haga una decisión acerca de cómo instalarlo.

Ruby en Unix, Linux y Mac OS X

Si usted tiene una máquina que proviene de la familia Unix (incluyendo Mac OS X), es probable que ya tenga Ruby incluido. Rails requiere de ruby 1.8.2 o 1.8.4 (pero 1.8.3 no funciona).

Para verificar su versión de ruby, ejecute el comando "ruby -v". Si su versión es adecuada, usted puede proceder a la sección de Gems.

Si su versión de ruby no es compatible con rails, usted tiene dos opciones:

1. Checar si su sistema operativo necesita actualización y la versión actualizada incluye un nuevo Ruby.
   En la actualidad los sistemas Linux tienden a tener los más nuevas instalaciones de Ruby en binario.
   En sistemas Solaris, algunos usuarios utilizan los paquetes binarios de CSW blastwave para mantener sus sistemas solaris actualizados sin tener que instalar un compilador en la máquina.
   En Mac OS X, depende de su versión del sistema operativo. El wiki de RubyGarden tiene información acerca de como instalar los binarios.
2. Instalar ruby en /usr/local, ya sea en binario o compilado.

Nota: Muchos administradores de sistema Unix (en especial de Linux) prefieren instalar los sistemas de web compilando los archivos fuente. Esto permite el control absoluto de su ambiente, pero requiere de mayor conocimiento de su sistema, así como un compilador.

Si usted decide instalar un sistema ruby desde código fuente, baje el código fuente del sitio oficial de Ruby (en Mac OS X también necesita instalar las herramientas XCode, que normalmente vienen incluidos en sus discos del sistema operativo), y utilice la manera estándar de compilar programas en Unix:

```
$ cd ruby-1.8.4-src
$ ./configure --prefix=/usr/local
$ make
$ make install
```

Después de instalarlo, añada el directorio /usr/local/ruby/bin a su variable PATH.

Recuerde: Si su máquina ya incluía ruby, el directorio de su nueva copia de Ruby debe estar **antes** de los demás directorios en su variable PATH.

Ahora verifique que su instalación funciona ejecutando ruby -v y verificando la nueva versión.

Ruby en Windows

Como mencioné anteriormente, ruby se encuentra "más en casa" en un sistema Unix. El problema principal de Windows es que éste sistema operativo no viene con un compilador - los usuarios de Windows se sienten más comodos con paquetes en binarios y con instalador.

Si usted no tiene deseo de aprender a manejar un sistema Unix, usted puede utilizar el One Click Ruby Installer for Windows. Este es un instalador que incluye ruby, muchas librerías en binario, y el libro "*Programming Ruby: The Pragmatic Programmer's Guide*" en un archivo de ayuda. Una vez instalado, sencillamente ejecute una línea de comando y escriba "ruby -v" para verificar su versión.

Pero si usted quisiera instalar un subsistema tipo Unix bajo Windows (o si usted está obligado a usar Windows por su trabajo pero prefiere usar Linux en casa), usted puede instalar Cygwin, que es una recompilación de las utilerías GNU bajo windows. Durante la instalación, seleccione Ruby para que se encuentre incluido.

Ruby Gems

RubyGems es un manejador de librerías de Ruby que hace muy sencillo instalar librerías. Gems esta a punto de convertirse en parte de ruby. Pero si gems no está incluido en su versión de ruby (lo cual puede verificar ejecutando **gem -v** para ver la versión de gems instalada), lo necesita instalar. Como usted ya tiene ruby, el procedimiento para instalarlo es el mismo para todos los sistemas operativos:

```
# cd rubygems
# ruby setup.rb
# gem -v
```

Ahora que tiene ruby gems, puede proceder a instalar rails.

MySQL

Ahora necesita una base de datos. Si usted utiliza Unix, es importante instalarla antes de instalar el paquete mysql de gems porque las librerías de cliente de mysql son necesarias.

Para instalar mysql, baje la versión para su plataforma del sitio de internet de MySQL.com. Una vez que la instale, puede proceder a instalar rails.

Rails

Ahora sí podemos instalar rails. Para instalar rails ejecute lo siguiente

```
$ sudo gem install rails --include-dependencies
$ sudo gem install mysql --with-mysql-dir=/usr/local/mysql
```

El parámetro with-mysql-dir sólo es necesario si su mysql no instala correctamente (porque el compilador no encontró las librerías de mysql). Asegúrese de que el directorio queutiliza en el parámetro sea el directorio donde instaló mysql.

Finalmente, ejecute el comando **rails --version** para verificar que su versión de rails fué instalada correctamente.

Nuestra primera aplicación

Creando un Esqueleto

Uno comienza con su aplicación de Rails creando un esqueleto. Crear un esqueleto es muy facil:

```
$ rails direcciones
 create
 create app/apis
 create app/controllers
 create app/helpers
 create app/models
 create app/views/layouts
 create config/environments
 create components
 create db
 create doc
 create lib
 create log
 create public/images
 create public/javascripts
 create public/stylesheets
 create script
 create test/fixtures
 create test/functional
 create test/mocks/development
 create test/mocks/test
 create test/unit
....
```

Facil, no? Ahora veamos lo que creamos. Para esto necesitamos ejecutar un servidor. Rails puede ser configurado con apache y FastCGI. Pero nuestro esqueleto tambien cuenta con un pequeño servidor web. Así que lo ejecutamos:

```
$ cd direcciones
$ ruby script/server
=> Rails application started on http://0.0.0.0:3000
=> Ctrl-C to shutdown server; call with --help for options
[2005-12-06 23:27:44] INFO WEBrick 1.3.1
[2005-12-06 23:27:44] INFO ruby 1.8.2 (2004-12-25) [i386-mswin32]
[2005-12-06 23:27:45] INFO WEBrick::HTTPServer#start: pid=3972 port=3000
```

Note el puerto que el servidor nos ha dedicado, 3000. Este es el puerto que vamos a accesar utilizando nuestro navegador favorito. La direccion es localhost, o 127.0.0.1. Así que en nuestro navegador iremos a *http://127.0.0.1:3000/* . Si todo salió bien, veremos una pantalla de felicitaciones con instrucciones para configurar Apache y pasos siguientes.

Ahora que tenemos nuestra aplicación de rails lista, podemos comenzar a ver como está compuesta nuestra nueva aplicación.

Tu ambiente

Rails 1.1 tiene un enlace llamado "About your Application's environment" que le permite verificar información acerca del ambiente. Esta información se encuentra en el URL /rails_info/properties.

Anatomía de una Aplicación Rails

Una aplicación rails se encuentra en el subdirectorio *app* y se compone de los siguientes directorios:

- apis - las librerías que su programa requiere fuera de Rails mismo.
- controllers - Los controladores
- helpers - Los helpers
- models - Los modelos. Basicamente las clases que representan los datos que nuestra aplicación manipulará.
- views - Las vistas, que son archivos rhtml (como JSP o ASP).
- views/layouts - Los diseños. Cada controlador tiene su propio diseño, donde se pondran las cabeceras y pies de página.
- config - Archivos de configuración. La configuración de la base de datos se encuentra aquí.
- script - Utilerías para generar código y ejecutar el servidor.
- public - La raíz del directorio virtual. Cualquier contenido que usted encuentre aquí será publicado en su aplicación directamente. En una nueva aplicación, usted puede encontrar las páginas web para los errores 404 y 500, las imágenes, javascripts, estilos, etcétera.
- test - Los archivos para las pruebas funcionales y de unidad.

Llamando código

Seguramente querremos que nuestro directorio tenga una pantalla principal. Conceptualmente, llamaremos a esto el módulo principal. Así que lo generamos:

```
$ ruby script/generate controller principal
 exists app/controllers/
 exists app/helpers/
 create app/views/principal
 exists test/functional/
 create app/controllers/principal_controller.rb
 create test/functional/principal_controller_test.rb
 create app/helpers/principal_helper.rb
```

Ahora puede usted visitar http://127.0.0.1:3000/principal para ver

> **Unknown action**
> No action responded to index

El archivo que fue creado se llama **controllers/principal_controller.rb**. Vamos a poner nuestro código ahí:

```
class PrincipalController < ApplicationController
  def index
     mensaje = "Hola a todos...<br>"
     ["Pedro", "Pablo", "Juan"].each do |nombre|
       mensaje += "y a "+nombre+" tambien<br>"
     end
     render_text mensaje
   end
end
```

Cuando visitamos de nuevo, vemos nuestro resultado:

> Hola a todos...
> y a Pedro tambien
> y a Pablo tambien
> y a Juan tambien

Finalmente, queremos que la raíz de esta aplicación sea también nuestro controlador. Así que eliminamos el index.html de bienvenida:

$ rm public/index.html

Si vemos la página de nuevo, ahora dirá:

> **Routing Error**
> Recognition failed for "/"

Esto es porque todavía no le hemos dicho a la aplicación que nuestro controlador "por defecto" es el principal. Podemos hacer esto definiendo una "ruta". Así que modificamos la configuración de rutas, en el archivo **config/routes.rb**:

```
map.connect '', :controller => "principal"
```

Ahora cuando visitamos la página, vemos el resultado de nuestro controlador principal.

Si tiene dudas, puede ver la película.

Teoría: El Paradigma MVC

MVC son las siglas de Modelo, Vista y Controlador. Es el patrón de diseño de software múy común en programas interactivos orientados a objetos.

Bajo el patrón de diseño de MVC, dentro de un programa, cada dominio lógico de edición (por ejemplo datos personales, cuentas bancarias, artículos noticiosos, etcétera) necesita tres partes:

1. El modelo, que es una representación de objetos del dominio lógico. En el ejemplo de datos personales, objetos como Persona, Dirección y Teléfono.
2. La vista, que es una o varias piezas de código que formatean y muestran todo o parte del modelo en el dispositivo interactivo (típicamente la pantalla). La vista típicamente es notificada de cambios en el modelo.
3. El controlador, que interpreta la información que el usuario provee para hacer cambios en el modelo.

La idea es que, al aplicar esta separación, se hace posible crear más de una vista para el mismo modelo (digamos, una vista abreviada y una vista detallada), y reutilizar el modelo (y el código que guarda el modelo de manera permanente) para escribir utilerías relacionadas, o incorporar datos del dominio original en programas más grandes.

Teoría: Pruebas de Unidad

Introducción

En cualquir sistema complejo, es crítico probarlo praa saber que funcionará una vez que se lance en vivo. Muchos grupos de trabajo se limitan a dedicar una o dos personas a probar el sistema. Éste método tiene algunos problemas

1. El programador está separado de la prueba. Esto impide que el programador proporcione información acerca de condiciones extremas de prueba, y el aislamiento también provoca que el programador siga cometiendo los mismos errores por no tener información acerca de como se prueba su sistema.
2. Una prueba que se limita a utilizar el sistema como un usuario no cubre problemas de diseño de objetos. Mientras el programa crece, el código espagueti se hace más y más complejo. El problema se hace peor exponencialmente mientras el sistema se mantenga en esta situación, hasta que arreglarlo se vuelve más caro que comenzar de nuevo.
3. Como humanos, nos es fácil olvidar maneras alternativas de utilizar las mismas funciones. Por ejemplo, ¿utilizas Archivo/Guardar, el boton para guardar o Control-S? ¿Si es Archivo/Guardar, con el teclado o con el mouse? ¿Còmo sabes que los dos funcionan? Todas estas condiciones parecen multiplicarse diario.

Lo que las pruebas de unidad buscan es evitar dichos problemas para que la calidad del sisterma sea mayor. Es un hecho comprobado que los sistemas que están diseñados utilizando metodologías que ponen la prueba como parte integral del diseño terminan siendo más confiables.

¿Qué es una Prueba de Unidad?

Una prueba de unidad pretende probar cada funciòn en un archivo de programa simple (una clase en

terminologìa de objetos). Las librerías de pruebas de unidad formalizan este trabajo al proporcionar clases para pruebas.

La prueba de unidad ayuda a que el mòdulo se haga independiente. Esto quiere decir que un mòdulo que tiene una prueba de unidad se puede probar independientemente del resto del sistema. Una vez que un gran porcentaje de su programa cuente con pruebas de unidad,

Note que las pruebas de unidad no sustituyen a las pruebas funcionales del departamento de control de calidad, pero a la larga reducen la cantidad de errores de regresión en futuras versiones del producto.

Una buena prueba de unidad:

- No requiere de mucho còdigo para prepararla.- Una prueba de unidad cuya preparaciòn toma 3 minutos (conectàndose a la base de datos, levantando un servidor de web, etcètera) se debe considerar demasiado pesada para ejecutarse cada vez que se hagan cambios.
- Prueba sólo un método. Aunque no siempre es necesario hacer "una prueba por método", sí es importante asegurar que cubrimos toda la clase. Muchas veces se requiere más de un método de prueba por cada método en la clase, para poder probar diferentes datos y casos extremos (por ejemplo, ¿qué pasa cuendo el usuario no existe?).
- Verifica los resultados con aserciones. De preferencia se debe de checar sólo un resultado. Se pueden checar varias partes del mismo resultado, pero diferentes condiciones deben probarse en un método separado.
- Deja los datos de la misma manera en que los encontró. Cuando la prueba comienza, los datos se preparan si es necesario, pero cuando la prueba termina, es importante que el sistema borre los datos después de la verificación. Es por esto que muchas librerías de pruebas unitarias tienen métodos llamados setup() y teardown() o similares.

 **Pruebas de Unidad y Diseño de Software**

Las pruebas de unidad influencían el diseño (para bien). Esto quiere decir que el aplicar pruebas de unidad a un sistema previamente escrito será difícil. Cuando usted se decida a añadir pruebas de unidad a un sistema existente, asegúrese de escojer sus batallas y hacer pruebas de regresión del sistema. Si su sistema es mediano o grande, es conveniente planear sus pruebas y aplicar los cambios necesarios a traves de varias versiones futuras. No lo intente todo al mismo tiempo.

Cobertura

Otro concepto relacionado a las pruebas de unidad es el de cobertura. ¿Cómo sabemos cuantas líneas del programa están siendo probadas (y más importante, las que no se probaron)? Una vez que usted tiene pruebas de unidad, probablemente le interesará tener un reporte de las líneas que no se han probado. Una utilería gratuita para proporcionar estos reportes en ruby se llama, apropriadamente, rubycoverage. Rubycoverage no viene con rails - usted lo debe instalar utilizando gems.

Desarrollo Basado en Pruebas (o Metodología "Prueba Primero")

Mencionamos anteriormente que las pruebas de unidad influencían positivamente el diseño de su programa. Es por esto que muchas metodologías ágiles son partidarias de lo que se llama "probar primero".

La idea es que al escribir la prueba primero (o al menos al mismo tiempo que el módulo original), a la larga es más sencillo mantener el programa y el diseño nunca se perjudica al punto que crea dificultades en el control de calidad.

La importancia de Pruebas de Unidad en Rails

En ruby on rails, y en todos los lenguajes interpretados en uso hoy día, la idea de las pruebas de unidad es central e importantísima. Como casi todos los errores en un programa interpretado sólo ocurren cuando el programa corre (porque no hay etapa de compilación), en esta clase de lenguajes es **esencial** escribir las pruebas de unidad al mismo tiempo que el código y asegurar una cobertura adecuada.

Las pruebas de unidad y el desarrollo basado en pruebas son importantes en todos los lenguajes, pero son aún más importantes en Ruby y Rails. No lo olvide.

Por este motivo, durante el curso veremos cómo podemos escribir nuestras pruebas primero (o al menos al mismo tiempo) que el resto del sistema.

Pruebas de Unidad y Rails

Ruby on Rails utiliza la librería Test::Unit como estándar para crear pruebas de unidad. Cuando usted utiliza los generadores de Rails para crear cualquier módulo, ya sea modelos o controladores, rails automáticamente crea una prueba de unidad para dicho controlador o modelo en el directorio **test**, incluyendo una que otra prueba básica. El diseño MVC de Rails hace más fácil escribir pruebas de unidad. Cuando usted extienda el programa sólo tiene que mantener las pruebas.

Además, el comando "rake" en una aplicación de rails ejecuta las pruebas. De esta manera usted puede añadir las pruebas de unidad y ejecutarlas con un sólo comando.

Configurando la Base de Datos

Creando una base de Datos en MySQL

Ahora vamos a preparar sus bases de datos. Rails utiliza configuraciones por separado para desarrollo (development), prueba (test) y producción. La idea es que usted desarrolla y prueba en una base de datos, y ejecuta las pruebas de unidad apuntando a la base de datos de prueba. En general, rails no toca el esquema de la base de datos de producción.

La recomendación es que utilice usted los sufijos "_dev" para desarrollo, "_test" para prueba y "_prod" para producción, y que su nombre básico de la base de datos sea el mismo.

Preparando la base de datos de Desarrollo

Para preparar su base de datos de desarrollo en MySQL, ejecute los siguientes comandos:

```
$ mysql -h miservidor.mydominio.com -u root -p
password: **********
mysql> create database direcciones_dev;
Query OK, 1 row affected (0.53 sec)
mysql> grant all privileges on direcciones_dev.* to 'direcciones_dev'@'%'
    -> identified by 'direcciones';
Query OK, 0 rows affected (0.80 sec)
mysql> Control-C
Aborted
```

Configurando su Aplicación

Para configurar su base de datos, usted necesita editar el archivo  **config/database.yml**

para apuntar a su servidor:

```
development:
  adapter: mysql
  database: direcciones_dev
  host: miservidor.mydominio.com
  username: direcciones_dev
  password: direcciones

test:
  adapter: mysql
  database: direcciones_test
  host: miservidor.mydominio.com
  username: direcciones_test
  password: direcciones

production:
  development
```

Ahora su sistema se puede conectar a la base de datos en la base de datos de desarrollo. Haga lo mismo con la base de datos de prueba (crear la base de datos, su usuario, y los privilegios en MySQL).

Una vez que su programa esté listo, su administrador de sistema puede configurar la base de datos del sistema de producción dentro de este mismo archivo. Un sistema se puede poner en modo de producción cambiando el archivo **config/environment.rb** lo dice.

Creando un Módulo con Modelo, Controlador y Vista

Planeando nuestro modelo

Hemos mencionado que lo que queremos hacer en este curso es escribir una libreta de direcciones. Así que comenzaremos a planear nuestro modelo.

 **Diferentes Maneras de crear un Modelo**

Hay tres maneras de crear un modelo en un sistema:

- **Pantallas:** No es muy recomendable, pero puede uno basarse en un boceto de las pantallas.
- **Modelo de Objetos:** Si usted está acostumbrado a utilizar Java y/o a hacer programas que no utilicen bases de datos, puede ser que usted utilice un modelo de classes y objetos.
- **Modelo de base de datos:** Si usted crea muchas aplicaciones de bases de datos (o es administrador de bases de datos), usted probablemente crea su modelo de base de datos primero.

El modelo de rails es una combinación de modelo de base de datos y modelo de objetos. En el paradigma de diseño típico, usted modela las tablas y relaciones de bases de datos y después crea el código del modelo. Ésta es la manera recomendable de diseñar el programa.

Creando la tabla en la base de datos.

La mejor manera de crear la tabla es utilizando una migración. Una migración es un concepto de ActiveRecord que le permite mantener las modificaciones al esquema de su base de datos

sincronizadas con la version actual de su código. Para crear una migración, usted debe hacer lo siguiente:

```
$ cd direcciones
$ ruby script/generate migration crea_tabla_personas
     create  db/migrate
     create  db/migrate/001_crea_tabla_personas.rb
```

Rails ahora ha creado una clase llamada CreaTablaPersonas (en el archivo db/migrate/001_crea_tabla_personas.rb, que contiene los métodos self.up y self.down:

```
class CreaTablaPersonas < ActiveRecord::Migration
  def self.up
  end

  def self.down
  end
end
```

Una migración define los métodos self.up y self.down para hacer y deshacer respectivamente sus cambios al esquema. Para crear nuestra tabla, definimos la tabla con un comando de ruby en self.up. También escribiremos el código para deshacer nuestros cambios (en este caso, eliminar la tabla) en self.down:

```
class CreaTablaPersonas < ActiveRecord::Migration
  def self.up
    create_table :personas do |tabla|
      tabla.column :nombre, :string, :limit => 80 :null => false
      tabla.column :apellido_paterno, :limit => 80, :string
      tabla.column :apellido_materno, :string, :limit => 80
      # El nombre que debemos desplegar en pantalla
      tabla.column :desplegar, :string, :limit => 80
      tabla.column :sexo, :string, :limit => 10, :default => 'Hombre', :null =>
false
      tabla.column :notas :text
    end


  end

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

El resultado de ejecutar la migración será el siguiente SQL (en mysql):

```
CREATE TABLE personas (
 id int NOT NULL auto_increment,
 nombre varchar(80) default NOT NULL,
 apellido_paterno varchar(80) default NULL,
 apellido_materno varchar(80) default NULL,
 desplegar varchar(80) default NULL,
 sexo varchar(10) default 'Hombre',
 notas TEXT NULL,
 PRIMARY KEY (id)
);
```

Note que el campo identificador (id) se crea automáticamente.

Cuando su código esté listo, simplemente ejecute el comando:

```
$ rake migrate
```

Y el sistema se conectará a la base de datos y ejecutará la migración.

Rails mantiene una tabla en la base de datos llamada schema_info que contiene la "versión" actual de la base de datos. Si usted ejecuta el comando rake migrate VERSION=x, la base de datos cambiará a la versión X, ya sea ejecutando los métodos "up" de las migraciones que no se han hecho, o ejecutando los métodos "down" de las versiones que hay que revertir.

Migraciones Irreversibles

De vez en cuando usted necesitará hacer una migración irreversible (por ejemplo, si necesita eliminar una columna de la base de datos). Cuando esto ocurra, ¡**no deje self.down vacío**! Utilice "raise IrreversibleMigration" para que el sistema sepa que los cambios en self.up son irreversibles.

Creando un Andamio

Ahora que tenemos nuestro modelo en la base de datos, podemos decirle a rails que cree nuestro modelo, controlador y vista con un sólo comando, ejecutando el programa generate con el nombre de nuestra tabla en MySQL:

```
$ ruby script/generate scaffold persona
 exists app/models/
 exists test/unit/
 exists test/fixtures/
 create app/models/persona.rb
 create test/unit/persona_test.rb
 create test/fixtures/personas.yml
...
```

Para ver como se creó el modelo, veamos el código de persona.rb:

```
class Personas < ActiveRecord::Base
end
```

Es algo que puede escribir usted mismo, pero usar los generadores le ahorra tiempo y evita errores.

¿Y las propiedades?

Notará que su clase Personas no tiene propiedades equivalentes a los campos de la base de datos. Esto es porque Ruby automáticamete crea propiedades para los campos de la tabla de Personas en la base de datos. Recuerde, Ruby utiliza el principio "*No te Repitas*".

Esto es muy útil, especialmente porque cuando estamos comenzando en nuestro proyecto muchas veces tenemos que cambiar los nombres de los campos hasta que los prototipos y el código se estabilice.

El generador ha creado lo que llamamos un andamiaje (scaffolding).

¿Andamio? ¡Si no soy albañil!

En construcción, un andamio (scaffold) es una construcción muy sencilla que hace posible la construcción del edificio real. De la misma manera, un el generador crea una construcción sencilla en Rails que nos permite añadir registros a la base de datos de inmediato, y nos sirve de andamiaje, haciendo posible construir el programa real.

Ahora podemos comenzar nuestro servidor y navegar a http://127.0.0.1:3000/personas para ver nuestro andamio y crear algunos registros de prueba.

Como verá su andamio contiene un listado, vista, actualización, y borrado de registros. Esto es lo que le llaman en inglés "CRUD", por las siglas de "create/replace/update/delete". Su listado también le muestra páginas. Trate de añadir más de diez registros para verlo funcionar.

Singular/Plural: Note que el estándar de Rails es utilizar **plural** para los nombres de las tablas

en la base de datos (*create table personas*), y **singular** para ejecutar los generadores. El generador pluralizará el nombre cuando lo considere adecuado.

Rails pluralizó automáticamente el nombre "persona" que le dimos al generador. La estructura plural/singular queda como sigue:

Tabla (en Base de Datos)	PERSONAS	Plural
Modelo (en app/models)	persona.rb	Singular
Vista (en app/views)	personas/*.rhtml	Plural
Controlador (en app/controllers)	pesonas_controller.rb	Plural
Prueba (en tests/functional)	personas_controller_test.rb	Plural

La idea de la regla es que la única mención de singular es cuando creamos el modelo.

Si esto no es adecuado a su proyecto, usted puede cambiar algunas de estas reglas cambiando el archivo **config/environment.rb**. Algunos de los cambios que le interestarán son:

- **ActiveRecord::Base.pluralize_table_names** (true/false) le permite decirle a rails que pluralize o no los nombres de las tablas. Si su administrador de base de datos no quiere que las tablas estén en plural, escriba ActiveRecord::Base.pluralize_table_names = false.
- **ActiveRecord::Base.primary_key_prefix_type** (:table_name/:table_name_with_underscore) le permite definir la manera en que ActiveRecord calculará el campo identificador por defecto. En nuestro ejemplo, con el valor :table_name el campo identificador sería personaid, pero con la opción :table_name_with_underscore el campo identificador sería persona_id.

El Inflector

ActiveRecord utiliza una clase llamada Inflector para especificar las reglas de pluralización. Usted puede modificar y añadir nuevas reglas de pluralización para su aplicación también con **config/environment.rb**. Simplemente agregue otras reglas utilizando expresiones regulares para definir inflecciones del plural, singular, irregular, o imposible de contar (no tiene plural) como sigue:

```
Inflector.inflections do |inflect|
  inflect.plural /^(ox)$/i, '\1en'
  inflect.singular /^(ox)en/i, '\1'
  inflect.irregular 'person', 'people'
  inflect.uncountable %w( fish sheep )
end
```

Pero mi Administrador de Base de Datos es muy Estricto....

Si su administrador de DB es muy estricto (o la base de datos pertenece a otro programa y no se puede cambiar), usted puede cambiar el nombre de la base de datos en el modelo persona.rb cambiando el método table_name. Por ejemplo, trabajamos en un banco enorme y tienen estándares de esos que obligan a utilizar nombres ilegibles (pobrecitos!) de sólo ocho caracteres para que sean compatibles con su computadora mainframe de 1970 (ouch!), y el nombre de la tabla de persona debe ser "DC_PERSN":

```
class Personas < ActiveRecord::Base
  def self.table_name() "DC_PERSN" end
end
```

Listo!

Control de Versiones con Subversion

Subversion es el sistema de control de versiones que utilizaremos a través de este curso. No depende ni tiene nada que ver con Ruby o Rails, de hecho usted puede utilizar subversion con cualquier lenguaje de programación (incluso con cualquier colección de archivos para los cuales usted desee mantener un historial de versiones).

Nota: Esto es sólo una introducción básica a subversion para que pueda comprender los comandos de subversion que ejecutaré de vez en cuando en el curso de rails. El tema de control de versiones en sí mismo es un tema enorme. Para una mucha mejor explicación en español de cómo funciona subversion, el libro en línea "Control de Versiones con Subversion" es mucho más adecuado (y también es gratuito).

Programación y Versiones

Si usted ha escrito programas con anterioridad seguramente le resultará familiar la historia de Juan López, programador:

Ayer como a las 10:00pm mi programa funcionaba perfectamente. Estaba yo haciendo cambios pequeños, nada más para limpiar el código, y me quedé hasta las 2 de la mañana haciendo mejoras estéticas pequeñitas.

Ahora son las 7:00am y tengo una demostración con el cliente en una hora, ¡¡¡y el programa no funciona!!! No se que le hice, pero ojalá hubiera hecho una copia a las 10:00pm.

Este es uno de los muchos motivos por los que todo programador necesita un sistema de control de versiones. Con un sistema de control de versiones, nuestro hipotético Juan López podría haber vuelto a la versión que funcionaba perfectamente a las 8 de la noche y no hubiera perdido el cliente.

La otra ventaja de un sistema de control de versiones es la posibilidad de comparar. Con un sistema de control de versiones, Juan hubiera podido hacer una comparación de los archivos de su versión con la versión de las ocho de la noche, para ver únicamente las diferencias. Como sólo necesitaría examinar las diferencias, podría haber encontrado el error rápidamente y no tendría que perder muchos de sus cambios estéticos para recuperar el funcionamiento de su programa.

Subversion

En este curso utilizaremos el sistema de control de versiones llamado subversion. Subversion es un sistema diseñado para reemplazar otro sistema de control de versiones llamado CVS. CVS es el sistema de control de versiones usado en gran parte del mundo de la programación de fuente abierta. Varios de los comandos que utilizaremos aquí también son válidos en CVS.

Instalación en Unix

Si usted está en un sistema Unix, primero revise su versión de subversion - algunas distribuciones de Unix ya lo tienen. Trate de ejecutar **svn --version** para ver si el programa responde. SI no lo tiene, puede descargar los códigos fuentes o los programas binarios de la página de web de subversion. Una vez instalado y configurado puede usted continuar para crear su repositorio.

Instalación en Windows

Para los usuarios de Windows hay dos opciones: Usted puede utilizar los binarios de línea de comando, o una extensión para Windows Explorer llamada TortoiseSVN, que le proporcionará

menús de contexto. Recuerde que todas las demás plataformas utilizan línea de comando. Este capítulo explica cómo utilizar la línea de comando. En tortoiseSVN esto normalmente quiere decir hacer click con el botón derecho en Windows y seleccionar el comando con el mismo nombre en el menú de contexto.

Creando un Repositorio

Un repositorio es un área en su sistema donde el sistema de control de versiones va a guardar su programa. Su programa se guarda en una serie de archivos que guardan toda la historia de su programa. Usted puede accesar toda la historia de las versiones de su programa.

Para crear un repositorio en subversion, usted puede utilizar el comando svnadmin create:

```
$ svnadmin create /datos/repositorios/direcciones
```

Esto creará un repositorio llamado direcciones en el directorio datos/repositorio (que primero debe existir). De preferencia (pero no obligatoriamente) usted debe hacer esto en un servidor, para que se pueda conectar a el repositorio desde otras máquinas.

Importando nuestro código

Ahora que su repositorio ha sido creado, podemos importar nuestro contenido. Con anterioridad hemos creado una aplicación de rails que va a ser una libreta de direcciones, y creamos nuestro modelo de personas junto con el andamiaje. Todo funciona bien hasta ahora, así que lo que queremos es que esta sea la primera "revisión" de nuestro programa. Así que importamos nuestro código fuente del programa de direcciones especificando donde vive nuestro archivo:

```
$ cd direcciones
$ svn import file:///datos/repositorios/direcciones
```

¿Porqué "file://"/?
Subversion utiliza URNs para especificar donde vive su repositorio. Cuando usted utiliza svn sin un servidor, el prefijo es file://, para acceder al sistema de archivos local. En un ambiente multiusuario es típico utilizar urns con http:// o ssh://.

A continuación svn abrirá un editor de textos (en Unix, el editor de textos es típicamente vi, o el editor especificado en la variable $EDITOR), que contiene lo siguiente:

```
--This line, and those below, will be ignored--
A    .
```

La "A" junto al punto decimal quiere decir que estamos Añadiendo el directorio actual (representado por un punto). Es la manera en la que svn nos dice que va a ocurrir una vez que salgamos del editor de textos. El motivo por el cual el editor de textos es mostrado es para que usted describa los cambios que le ocurrieron al repositorio (para el archivo histórico).

Usted ahora puede escribir una descripción de lo que le está haciendo al repositorio. En este caso, estamos apenas comenzando, así que escribimos algo como lo siguiente:

```
Importando el sistema de direcciones
--This line, and those below, will be ignored--
A    .
```

Ahora guarde el archivo y salga del editor. A continuación verá que subversion añade todos los archivos en su directorio al repositorio:

```
Adding         test
Adding         test/unit
Adding         test/unit/persona_test.rb
Adding         test/test_helper.rb
```

```
Adding         test/functional
Adding         test/functional/personas_controller_test.rb
Adding         test/fixtures
Adding         test/fixtures/personas.yml
Adding         test/mocks
Adding         test/mocks/test
Adding         test/mocks/development
Adding         app
Adding         app/helpers
Adding         app/helpers/application_helper.rb
Adding         app/helpers/personas_helper.rb
Adding         app/models
Adding         app/models/persona.rb
Adding         app/controllers
...
Adding         public/favicon.ico
Committed revision 1.
```

Nuestro repositorio ahora ha sido importado de este directorio. Ahora necesitamos sincronizar con el repositorio para que pueda mantener nuestros cambios. Para hacer eso necesitamos primero jalar una copia que provenga del repositorio, así que cambiamos el nombre del directorio de nuestro programa (lo borraremos después que nos aseguremos que todo funciona bien) y ejecutamos el comando svn checkout para que el sistema utilice nuestro repositorio:

```
$ cd ..
$ mv direcciones direcciones.borrame
$ svn checkout file:///datos/repositorios/direcciones
A    direcciones/test
A    direcciones/test/unit
A    direcciones/test/unit/persona_test.rb
A    direcciones/test/test_helper.rb
A    direcciones/test/functional
A    direcciones/test/functional/personas_controller_test.rb
A    direcciones/test/fixtures
A    direcciones/test/fixtures/personas.yml
A    direcciones/test/mocks
A    direcciones/test/mocks/test
A    direcciones/test/mocks/development
...
Checked out revision 1.
$ cd direcciones
```

Modificando Archivos

Ahora tenemos una copia que proviene del repositorio. Esta copia será actualizada cuando nosotros ejecutemos el comando svn commit desde el directorio de direcciones. Podemos actualizar subdirectorios independientemente, depende de donde ejecutemos el comando commit. Además, cambios a varios archivos en el repositorio se realizan atómicamente (se toman en cuenta como uno). Por ejemplo, haga algunos cambios a su archivo doc/README_FOR_APP con un editor de texto (este archivo es útil para describir de que se trata su programa y como instalarlo) y después ejecute svn commit:

```
$ svn commit
```

Ahora verá que su editor abre y muestra que ha habido un cambio al archivo:

```
--This line, and those below, will be ignored--
M    doc/README_FOR_APP
```

Escriba de que se trató el cambio (por ejemplo, "Mejor explicación de nuestro programa") y salga

del editor. Subversion guardará el cambio.

```
Sending        doc/README_FOR_APP
Transmitting file data .
Committed revision 2.
```

Ahora el repositorio se encuentra en la revisión 2.

¿Qué significan A y M?
Subversion utiliza letras para explicarle qué le ocurre a los archivos. Las letras son como sigue:

A Este archivo ha sido añadido (Added)

M Este archivo ha sido modificado (Modified)

D Este archivo ha sido elimidado (Deleted)

Historial de un archivo

Para ver la historia de un archivo, utilice el comando svn log:

```
$ svn log  doc//README_FOR_APP
------------------------------------------------------------------------
r2 | Owner | 2006-01-21 15:39:14 -0800 (Sat, 21 Jan 2006) | 2 lines
Mejor explicacion del programa
------------------------------------------------------------------------
r1 | Owner | 2006-01-21 15:28:01 -0800 (Sat, 21 Jan 2006) | 3 lines
Importando el sistema de direcciones
------------------------------------------------------------------------
```

El sistema le explica los cambios que incluyeron este archivo.

Añadiendo y eliminando archivos

Cuando usted añade y elimina archivos dentro de su sistema, el sistema de control de versiones no efectúa los cambios hasta que usted especifique el comando **svn add** o **svn rm** (añadir y eliminar, respectivamente). Así usted no se necesita preocupar en eliminar un archivo accidentalmente, y de la misma manera no añadirá un archivo por error "ensuciando" la historia de su repositorio.

Por ejemplo, hagamos un archivo de prueba en el directorio **doc**. Digamos que su nombre es prueba.txt

```
$ echo "Hola como estas" > doc/prueba.txt
$ svn add doc/prueba.txt
A         doc/prueba.txt
$ svn commit
Adding         doc/prueba.txt
Transmitting file data .
Committed revision 3.
```

Ahora lo podemos eliminar:

```
$ svn rm doc/prueba.txt
D         doc/prueba.txt
$ svn commit
Deleting       doc/prueba.txt
Committed revision 4.
```

Actualizando el Repositorio

Otros programadores (o usted mismo en otra computadora) pueden cambiar su repositorio. Al principio de cada sesión de trabajo, usted puede ejecutar el comando svn update para sincronizar de nuevo con el repositorio:

```
$ svn update
U doc/README_FOR_APP
```

En este ejemplo, alguien cambió README_FOR_APP y ahora ya tengo sus cambios.

Comparando versiones

Usted puede comparar versiones utilizando el comando **svn diff**. Si usted no especifica sólo el nombre del archivo, el sistema comparará la copia del archivo en su disco duro con la copia más nueva del repositorio.

Por ejemplo, algunos cambios a la versión en disco duro de mi doc/README_FOR_APP. Ahora ejecutaré svn diff:

```
$ svn diff doc/README_FOR_APP
Index: doc/README_FOR_APP
===================================================================
--- doc/README_FOR_APP  (revision 2)
+++ doc/README_FOR_APP  (working copy)
@@ -1,2 +1,3 @@
-Programa de ejemplo para el curso de rails.
+Esta es una nueva línea
```

En este listado, las líneas que se eliminaron comienzan con el signo de menos y las que se añadieron con el signo de más. Las arrobas me dicen aproximadamente donde (línea y columna) se encuentra el cambio.

Finalmente, no quiero quedarme con este cambio, así que mejor ejecuto svn revert para que mi copia se elimine y se reemplace con la del repositorio.

```
$ svn revert doc/README_FOR_APP
Reverted 'doc/README_FOR_APP'
```

Esto es sólo una introducción, y subversion tiene muchos otros comandos. Le recomiendo que revise la documentación y aprenda lo más que pueda de este excelente sistema.

Pruebas de Unidad

Preparando un Ambiente de Prueba

Como vimos cuando estabamos configurando la base de datos, usted necesita tener una base de datos separada para prueba. Si usted ha seguido el tutorial y no creó una base de datos de prueba, la puede crear ahora (vacía) simplemente ejecutando el comando CREATE DATABASE direcciones_test; (o el nombre que tenga su base de datos).

Probando el Directorio

Ahora sí estamos listos para crear las pruebas de nuestro directorio. Rails ya añadió algunas pruebas a nuestro sistema esqueleto, así que intentaremos ejecutarlas:

```
$ rake
(in /home/david/play/direcciones)
/ruby/bin/ruby -Ilib;test "/ruby/lib/ruby/gems/1.8/gems/rake-
0.6.2/lib/rake/rake_test_loader.rb"
"test/unit/persona_test.rb"
Loaded suite /ruby/lib/ruby/gems/1.8/gems/rake-0.6.2/lib/rake/rake_test_loader
Started
.
Finished in 0.2 seconds.

1 tests, 1 assertions, 0 failures, 0 errors
/ruby/bin/ruby -Ilib;test "/ruby/lib/ruby/gems/1.8/gems/rake-
0.6.2/lib/rake/rake_test_loader.rb"
"test/functional/personas_controller_test.rb"
Loaded suite /ruby/lib/ruby/gems/1.8/gems/rake-0.6.2/lib/rake/rake_test_loader
Started
........
Finished in 0.751 seconds.

8 tests, 28 assertions, 0 failures, 0 errors
```

Qué bueno, pasamos! Pero qué se probó? Veamos las pruebas que tenemos. Los programas en el directorio test son:

```
$ find test -name *.rb
test/functional/personas_controller_test.rb
test/test_helper.rb
test/unit/persona_test.rb
```

Pruebas de Unidad vs Funcionales

En el esqueleto creado por rails, las pruebas se dividen en pruebas de unidad (que idealmente prueban cada método de nuestro programa), y pruebas funcionales, que prueban que su programa haga lo que usted espera.

En general, las pruebas funcionales prueban el controlador, y las pruebas de unidad prueban el modelo y las clases de soporte

Cuando usted creó el módulo MVC de personas, el generador de código preparó esqueletos para sus pruebas de unidad.

Para ver de qué se compone una prueba básica, veamos el archivo **test/unit/persona_test.rb**:

```
require File.dirname(__FILE__) + '/../test_helper'

class PersonaTest < Test::Unit::TestCase
  fixtures :personas

  # Replace this with your real tests.
  def test_truth
    assert_kind_of Persona, personas(:first)
  end
end
```

Una prueba de unidad en ruby es una clase que hereda de Test::Unit::TestCase. Dentro de esta clase nosotros escribimos las pruebas.

Adornos (Fixtures)

A veces nuestras pruebas de unidad requieren de registros de prueba en la base de datos. En terminología rails, estos registros se llaman "fixtures" o adornos. Se llaman así porque son útiles primariamente para que la prueba tenga datos con los cuales trabajar.

Los adornos para nuestra tabla se encuentran en el folder **test/fixtures/personas.yml**. Como podrá imaginar, cuando creamos nuesto módulo utilizando el generador rails nos hizo el favor de incluir este archivo de adorno para la tabla:

```
# Read about fixtures at http://ar.rubyonrails.org/classes/Fixtures.html
first:
  id: 1
another:
  id: 2
```

Lo que tenemos que hacer ahora es añadir nuestros datos iniciales. El llenado de los adornos es muy sencillo, y se hace de acuerdo a la descripción de la tabla que diseñamos con anterioridad:

```
# Read about fixtures at http://ar.rubyonrails.org/classes/Fixtures.html
personaje_novela:
  id: 1
  nombre: Alonso
        apellido_paterno: Quijano
        apellido_materno: Cervantes
        desplegar: Alonso Quijano
        sexo: Hombre
        notas: Nacio en un lugar de la Mancha de cuyo nombre no quiero
acordarme.

actor_famoso:
  id: 2
  nombre: Pedro
        apellido_paterno: Infante
        apellido_materno: Cruz
        desplegar: Pedro Infante
        sexo: Hombre
        notas: El Idolo de Guamuchil
```

Note que en el formato YML podemos nombrar nuestros registros para describir su naturaleza, que nos ayudará para que nuestras pruebas sean más claras. En este caso, cambiamos los nombres "first" y "another" por "personaje_novela" y "actor_famoso". Como cambiamos el nombre de nuestro primer registro, nuestro método "test_truth" en el listado anterior también necesita cambiar.

Utilizando Adornos

Estas pruebas son un poco triviales, pero hagamos un par de pruebas de unidad para comprobar que esto funcione:

```
def test_actor_famoso
  actorazo = personas(:actor_famoso)
  assert_equal "Infante", actorazo.apellido_paterno
end
```

Desarrollo Basado en Pruebas

Con anterioridad vimos brevemente que Ruby soporta Desarrollo basado en Pruebas, pero no explicamos específicamte la mecánica del desarrollo basado en pruebas. Ahora veremos un ejemplo práctico de éste desarrollo, que le permitirá comprender el proceso de pensamiento que utiliza un desarrollador que se basa en pruebas.

Cambio de Metodología

Lo que vamos a presentar a continuación representa un cambio de metodología y tren de pensamiento de lo que muchos desarrolladores están acostumbrados.

Acostumbrarse a probar primero no es sencillo, pero vale la pena intentarlo.

Apellidos en Español

En la lengua de Cervantes, los apellidos tienen reglas un poco especiales. Necesitamos implementar un método llamado apellidos que desplegará los apellidos de la siguiente manera:

- Si hay apellido ambos apellidos, "Paterno Materno" (separados por espacios)
- Si hay apellido paterno únicamente, "Paterno".
- Si no hay apellido paterno, "Materno".

Como estamos desarrollando basados en nuestras pruebas, primero tenemos que preparar una prueba que falle, y datos de prueba:

Preparando Datos de Prueba

Para los datos de prueba, añadimos a **personas.yml** registros que cubran los casos adecuados:

```
sin_appellidos:
  id: 3
  nombre: Maria
  desplegar: Maria
  sexo: Mujer

solo_apellido_paterno:
  id: 4
  nombre: Juan
  apellido_paterno: Gomez

solo_apellido_materno:
  id: 5
  nombre: Pedro
  apellido_materno: Paramo

ambos_apellidos:
  id: 6
  nombre: Juan
  apellido_materno: Torres
  apellido_paterno: Mendoza
```

Código de Prueba

Ahora que tenemos información de prueba, podemos crear nuestras pruebas. Todavía no tenemos el código, pero sabemos lo que esperamos en los diferentes casos. Así que añadimos a nuestro archivo **test/unit/persona_test.rb** los métodos de prueba que necesitamos. Comenzamos los métodos con test para que la clase TestCase los ejecute:

```
class PersonaTest < Test::Unit::TestCase
  fixtures :personas

  def test_ambos_apellidos
    bien_nacido = personas(:ambos_apellidos)
    assert_equal( bien_nacido.apellido_paterno+' '+bien_nacido.apellido_materno,
bien_nacido.apellidos )

  end

  def test_solo_paterno
    persona = personas(:solo_apellido_paterno)
    assert_equal( persona.apellido_paterno, persona.apellidos )
  end

  def test_solo_materno
    persona = personas(:solo_apellido_materno)
    assert_equal( persona.apellido_materno, persona.apellidos )
  end

  def test_sin_apellidos
    persona = personas(:sin_apellidos)
    assert_equal( "", persona.apellidos )
  end



end
```

Ahora podemos escribir un metodo "cabo" (un método básicamente vacío), que ataremos con el código adecuado más tarde. En nuestro archivo **app/models/persona.rb**.

```
def apellidos
  ""
end
```

La Prueba que Falla

Ahora cuando ejecutemos rake, veremos que la única prueba que funciona es la de sin_apellidos. Funciona de pura casualidad, porque la implementación de nuestro método está vacia:

```
$ rake TEST=test/unit/persona_test.rb
(in /cygdrive/c/home/david/play/direcciones)
/usr/bin/ruby -Ilib:test
  "/usr/lib/ruby/gems/1.8/gems/rake-0.7.0/lib/rake/rake_test_loader.rb"
        "test/unit/persona_test.rb"
Loaded suite /usr/lib/ruby/gems/1.8/gems/rake-0.7.0/lib/rake/rake_test_loader
Started
F.FF.
Finished in 0.611 seconds.

  1) Failure:
```

```
test_ambos_apellidos(PersonaTest) [./test/unit/persona_test.rb:13]:
<"Mendoza Torres"> expected but was
<"">.

  2) Failure:
test_solo_materno(PersonaTest) [./test/unit/persona_test.rb:24]:
<nil> expected but was
<"">.

  3) Failure:
test_solo_paterno(PersonaTest) [./test/unit/persona_test.rb:19]:
<"Gomez"> expected but was
<"">

5 tests, 5 assertions, 3 failures, 0 errors
rake aborted!
Test failures

(See full trace by running task with --trace)
```

Note que podemos ejecutar sólo una prueba simplemente añadiendo el nombre del archivo ruby que contiene su prueba.

Pero lo importante que hay que tener en cuenta es que, gracias a nuestras pruebas de unidad (que fueron creadas antes del código basadas en el diseño), tenemos las siguientes ventajas:

- Sabemos en todo momento qué condiciones están cubiertas por nuestro código (gracias a que los nombres de las pruebas son descriptivos)
- Tenemos una buena idea de qué tanto avance hemos hecho en nuestro código (por ejemplo, en este momento, una de cuatro pruebas pasan).
- Otras personas pueden comenzar a integrar con nuestro método apellido, y nuestras pruebas manejan las expectativas. Esto es útil en especial cuando usted trabaja en equipo
- Usted puede añadir el comando rake a un proceso nocturno para saber cuando hay cambios que rompen las pruebas. Cuando el código cambie en una manera que no es compatible con el resto del sistema, sus pruebas se romperán, y usted lo sabrá antes de que se enteren sus usuarios.
- Cuando usted descubra algún problema causado por **límites** (valores fuera de lo común), puede añadir otra prueba al área adecuada y después correjir el error, lo cual le proporciona una regresión automática.

Implementando la función

Finalmente, podemos implementar la función para los apellidos, y continuar arreglando problemas hasta que nuestro comando de prueba muestre el resultado deseado...

```
$ rake TEST=test/unit/persona_test.rb
...
5 tests, 5 assertions, 0 failures, 0 errors
```

Y ahora si podemos decir que nuestro soporte de apellidos en español está completo, y tenemos cinco pruebas que lo demuestran.

Teoría de Objetos

Objetos

Un objeto es un tipo de datos que incorpora datos y código (comportamiento) en un solo "paquete". Ántes de la era de la orientacion a objetos, el código y los datos eran dos cosas separadas. La orientación por objetos nos permite representar de un modo mucho más conveniente al mundo real.

¿Cómo podemos modelar un objeto del mundo real con objetos "de computadora"? Veamos un ejemplo:

Ejemplo de un Objeto

Supongamos que estamos haciendo un juego acerca de un Acuario. Para este juego queremos diseñar un Bonito Delfín que va a brincar con el aro y jugar con la pelota, pero queremos en el futuro extender el juego para cualquier tipo de animal marino porque va a ser una simulación bien picuda. Tal como en el mundo real, necesitamos comenzar con un objeto llamado "**pescado**" (Para los amantes de la biologia: Ya se que el delfín es un mamífero y no un pescado, pero éste es un ejemplo). El objeto pescado tiene sus datos, como son alto, largo, peso y color. Pero el pescado tambien puede hacer otras cosas, como por ejemplo nadar y sumergirse. Entonces primero hacemos nuestro objeto pescado, que tiene la siguiente forma:

```
class Pescado

  def Pescado
    @largo = 200
    @alto  = 50
    @ancho = 50
    @peso  = 350
  end

  def nadar
    # Codigo...
  end

  def sumergirse
    # Codigo...
  end

end
```

Este código le dice a la computadora: Pescado es una clase (una clase es una definición de un objeto). Tiene los campos Largo, Alto, Ancho y Peso, que son números de punto flotante. Sus métodos públicos (lo que todo mundo sabe que un pez puede hacer) son Nadar y Sumergirse.

Si esto parece complejo, por favor sigan leyendo. Muy pronto todo quedará claro.

Ahora el Delfín. He decidido para este ejemplo que hay dos clases de delfines, los entrenados y los salvajes (no entrenados). Asi que comencemos con el delfin entrenado:

```
class Delfin < Pescado
  def tomar_aire
    # Codigo..
  end

  def haz_ruido
    # Codigo..
  end
```

```
end
```

Ahora bien, este codigo le dice a la computadora: El Delfin es una clase que hereda de Pescado (o "Un Delfin es un Pescado"). Esto quiere decir que **un Delfin puede hacer todo lo que un pescado puede hacer** (tiene largo, alto, ancho, peso y además puede nadar y sumergirse). Entonces yo ya terminé mi delfin, y no tuve que implementar de nuevo las funciones para nadar y sumergirse, que el pescado (y ahora tambien el delfin) puede hacer. Ahora vamos a entrar en la materia del jueguito, el delfin entrenado:

```
class DelfinEntrenado < Delfin
  def juega_con_pelota(segundos)
    # Codigo
  end
  def brinca_el_aro(circunferencia)
    # Codigo
  end
end
```

El DelfinEntrenado es una clase que hereda no de Pescado, sino de Delfin (o "Un delfín entrenado sigue siendo un delfin, pero..."). Al igual que la vez anterior, un delfín entrenado puede hacer todo lo que un delfin puede hacer, pero además puede jugar con pelota y brincar el aro.

¿Porqué hacer tres objetos nada más para representar un delfín? Supongamos que ahora quiero hacer un tiburón...

```
class Tiburon < Pescado

  def Tiburon
    @dientes = 200
    @bocon = true
    @come_hombres = false
  end

  def enojate
  end

  def come_persona(sabroso)
  end

  def espanta_visitantes
  end

end
```

Gracias a la orientación a objetos, ahora estoy "re-utilizando" el código que usé para implementar mi delfín. Ahora, si mañana descubro que los pescados pueden nadar de una manera especial, o quiero hacer la simulación detallada y quiero hacer que hagan algo más, como nacer y morirse, todos los objetos de tipo pescado (tiburon, delfín, delfín entrenado, pez dorado) van a nacer y morir igual que el pescado, porque la implementación aplica al tipo y a todos los tipos de la misma clase. Ahora bien, la "jerarquía" de los objetos que hemos hecho es como sigue:

```
Object
   +----- Pescado
             |
             +----- Delfin
             |          |
             |          +----- Delfin Entrenado
             |
             +----- Tiburon
```

La jerarquía de objetos es una especie de "árbol genealógico" que nos dice qué objetos se "derivan"

de otros objetos. Como conceptualizamos esta jerarquía? Es de hecho bastante sencillo una vez que nos acostumbramos al árbol. Por ejemplo, supongamos que queremos saber que interfaces soporta el "Delfín Entrenado". Leemos todos los "padres" de la siguiente manera: "Un Delfin ES UN Pescado, que ES UN Objeto". Esto quiere decir que un objeto de la clase DelfinEntrenado soporta todas los metodos que objetos de la clase Delfin y de la clase Pescado.

Tipos en Ruby

Los lenguajes orientados a objetos son típicamente de tipos fuertes o tipos débiles.

Un lenguaje de tipos fuertes revisa las asignaciones de valores en el programa para ver si son del mismo tipo antes de la ejecución (al tiempo de compilar). Un lenguaje de tipos débiles sólo revisa al tiempo de ejecución.

La evaluación de tipos en ruby es un poco especial. En ruby, los tipos están definidos como las capacidades de un objeto, en vez de estar definidos arbitrariamente por naturaleza. En otros lenguajes, esto se llama polimorfismo, y lo consiguen mediante interfaces. En Ruby, las inerfaces no son necesarias.

Los rubyistas le llaman "evaluación tipo Pato", por el dicho en Inglés que va *"Si camina como un pato, y habla como un pato, es un pato"*.

Esto quiere decir que ruby intentará ejecutar los métodos de su objeto incluso cuando los típos sean diferentes. En nuestro ejemplo anterior, si le añadieramos al Tiburón un método brinca_el_aro:

```
class Tiburon < Pescado
  def brinca_el_aro(circunferencia)
  end
end
```

Nuestro código intentaría ejecutarlo incluso cuando fué diseñado para un DelfínEntrenado.

Mapeo de Objetos a Relaciones

El Modelo Relacional vs El Modelo de Objetos

Las bases de datos SQL funcionan en modo relacional. Esto quiere decir que utilizan una Matriz de celdas para representar los datos. En cada renglón de una matriz, existe un registro individual. Cada columna de la matriz representa uno de los valores de ese registro, y tiene un tipo específico (numerico, caracteres, etcetera):

Tabla: Personas

id	Nombre	Apellido	Sexo	Nacido En
1	Elvis	Presley	M	Nashville, Tenesee
2	Albert	Einstein	M	Württemberg, Alemania

En el modelo relacional, dos tablas pueden estar relacionadas a traves de una clave de identidad, entre otras cosas para poder representar valores repetidos, o valores que no pertenecen a la naturaleza de lo que representa la tabla:

Tabla: Telefonos

id	persona_id	Pais	Area	Telefono
1	2	52	777	334-2340
2	2	49	30	495-2030
3	1	1	615	394-2304

A traves de una de las columnas, la tabla se relaciona.

Sin embargo, en el modelo de objetos, los registros son instancias de un tipo de objeto. En el mundo de programacion orientada a objetos no hay diferencia innata entre un tipo que es parte del lenguaje (por ejemplo String) y un tipo que representa registros para la base de datos (como Persona o Telefono). Además, en el modelo de objetos los tipos tienen referencias directas:

- Persona
    - nombre: String
- apellido: String
- sexoMasculino: boolean
- nacidoEn: String
- telefonos: Telefono[]
        - pais: String
- area: String
- telefono: String

Mapeo de objetos quiere decir crear una manera de convertir un diagrama de objetos para que pueda ser automáticamente mantenido por la base de datos. Los sistemas de mapeo de objetos típicamente saben como mapear los tipos innatos del lenguaje, como string e int. Estos tipos son especiales ya que son tipos de columnas en el mundo de las bases de datos relacionales.

Aproximaciones al problema

Ahora que sabemos que el mapeo de objetos consiste en crear un mapa de los tipos de objetos a las tablas, columnas y tipos de datos, podemos imaginar dos tipos de aproximaciones:

- **De Objeto a Tabla**.- Utilizando este método, usted crea el objeto primero y despues añade metadatos al objeto (ya sea como comentarios, con algún archivo de configuración o con definiciones en el objeto mismo) que especifican los nombres de archivos y el estilo de mapeo que se utilizará.

  Este método es preferido por los *arquitectos de software*, ya que permite diseñar el modelo de objetos independientemente del sistema que se utilice para guardar estos objetos.

  Aunque tiene la ventaja de que produce un diseño aparentemente más "limpio" desde el punto de vista de los arquitectos, el problema de estos diseños es que a los administradores de bases de datos no les gusta porque los sistemas diseñados así resultan bastante lentos para accesar la base de datos.
- **De Tabla a Objeto**.- Este método es favorecido principalmente por los administradores de bases de datos, porque permite optimizar las tablas desde la fase de diseño. Una implementación de este método es la librería Hibernate (para Java y .Net).

  Aunque tiene la ventaja de que produce un diseño más eficiente, el problema de este método es que puede provocar que las definiciones semánticas de los objetos parezcan arbitrarias.

ActiveRecord

Ruby utiliza la librería ActiveRecord para mapear objetos a relaciones. Sus clases heredan de ActiveRecord::Base. Si usted ya ha seguido el tutorial acerca de cómo crear un módulo MVC, usted sabrá que el método que utiliza es el método de tabla a objeto. Sin embargo, usted puede cambiarle de nombre a las columnas y las tablas para crear un modelo semántico más claro.

Definiendo nombres de columnas y tablas

En ActiveRecord los nombres de las columnas se pueden cambiar redefiniendo el método table_name() del objeto, de la manera siguiente:

```
class Personas < ActiveRecord::Base
  def self.table_name() "GENTES" end
end
```

Para cambiar nombres de columnas (y para hacer campos calculados), usted puede simplemente definir métodos para obtener y cambiar los valores en un patrón de fachada:

```
class Personas < ActiveRecord::Base

  def nombre()
    read_attribute("G_NOMBRE")
  end

  def nombre=(elnombre)
    write_attribute("G_NOMBRE", elnombre)
  end

  def nombre_completo()
    nombre() +" "+ apellido()
  end

end
```

En este ejemplo, el programador ha añadido una fachada para que la columna "G_NOMBRE" responda a la propiedad "nombre" en el objeto. Además, se ha añadido una propiedad "nombre_completo" que formatea el nombre y apellido.

Nota: Si usted tiene control del diseño de base de datos así como de los objetos, es más

recomendable utilizar los nombres de las tablas y columnas de tal manera que código de mapeo no sea necesario, como lo vió en el tutorial de módulo MVC.

De esta manera su código es más sencillo de mantener.

Definiendo Relaciones Entre Objetos

Vimos hace unos momentos que en el modelo relacional, las relaciones entre dos tipos son representadas utilizando columnas relacionadas. Ahora veremos este y varios otros tipos de relaciones.

Relaciones Circulares

Las relaciones de pertenencia se representan con has_one o con has_many, lo cual le permite especificar que la relación es de *uno a uno* o de *uno a muchos*, respectivamente. La relación opuesta se puede representar con belongs_to, en la clase de destino. El resultado se ve muy natural:

```
class Persona < ActiveRecord::Base
  has_one :detalle
  has_many :telefono
end

class Detalle < ActiveRecord::Base
  belongs_to :persona
end

class Telefono < ActiveRecord::Base
  belongs_to :persona
end
```

Esto requiere que los siguientes nombres de columnas por defecto:

Tabla	Columna
persona	id
detalles	persona_id
telefonos	persona_id

Por supuesto, si es necesario usted puede cambiar los nombres de columnas por defecto por medio de modificar las opciones de belongs_to. En el siguiente ejemplo cambiaremos el nombre de la columna que conecta a la persona en la clase de detalles:

```
class Detalle < ActiveRecord::Base
  belongs_to :persona, :foreign_key=>"humano_id"
end
```

Relaciones Circulares

La posibilidad de cambiar opciones en belongs_to permite, entre otras cosas crear relaciones circulares, por ejemplo:

```
class Persona < ActiveRecord::Base

  belongs_to :padre, :class_name=>"Persona", :foreign_key=>"padre_id"
  belongs_to :madre, :class_name=>"Persona", :foreign_key=>"madre_id"

end
```

Asociaciones con tabla de Pivote

Un tipo de relaciones muy útil en el mundo de SQL es la relación de "muchos a muchos", lo cual se logra con lo que se llama una tabla de pivote. Supongamos que tenemos una relación de Personas y Compañias. Una persona puede ser cliente de varias compañías. Y las compañías pueden tener varios clientes. La tabla de pivote es como sigue:

companias_personas	tipo
persona_id	int
compania_id	int

Esta relación se puede representar con has_and_belongs_to_many de la manera siguiente:

```
class Compania < ActiveRecord::Base
  has_and_belongs_to_many :personas
end

class Persona < ActiveRecord::Base
  has_and_belongs_to_many :companias
end
```

Definiendo Jerarquías de Objetos y Herencia

Las jerarquías de objetos en sistemas de mapeo relacional se pueden representar automáticamente con una sola tabla. Para lograrlo, su tabla necesita tener un campo llamado type, que representará el tipo del objeto, y los campos de la jerarquía completa deben estar definidos.

Por ejemplo, para representar los campos de la jerarquía de objetos de nuestro ejemplo en la sección teoría de objetos (una jerarquía de pescado->Tiburón->Delfín), podríamos definir una tabla como sigue:

Tabla: Pescados

nombre	tipo
id	integer
type	varchar
nombre	varchar
largo	integer
ancho	integer
ancho	integer
peso	integer
Propiedades del Delfín..	
mamarias	integer
Propiedades del Tiburón..	
come_hombres	boolean
dientes	integer
bocon	boolea

Con este tipo de tabla, ActiveRecord simplemente utilizará los campos adecuados y regresará el tipo adecuado dependiendo del campo "type".

Árboles

Un árbol es una representación donde los objetos tienen un sólo padre. Para definir un árbol en ActiveRecord, utilizamos el código acts_as_tree y definimos un campo llamado parent_id que

permita valores nulos:

Tabla: Categorias

nombre	tipo
id	integer
parent_id	integer NULL
orden	integer
nombre	varchar

Con esto, podemos definir la tabla, e incluso podemos utilizar opciones para mantener el orden de los nodos hijos, así como para mantener una cuenta de hijos.

```
class Categoria < ActiveRecord::Base
  acts_as_tree :order => "orden", :counter_cache => true

end
```

Modificando el Módulo Generado

Tenemos nuestro código generado, pero más que nada esto fué para que nuestros usuarios lo vieran e hicieran algunas decisiones en cuanto a los campos que faltan, etcétera. Ahora analizaremos cómo funciona una vista para comenzar a hacer las modificaciones.

Vistas en nuestro módulo

Cuando generamos el módulo de personas, se generó un módulo de controlador y varias vistas. Repasemos lo que fué generado:

```
app/views/layouts/personas.rhtml
app/views/personas/_form.rhtml
app/views/personas/edit.rhtml
app/views/personas/list.rhtml
app/views/personas/new.rhtml
app/views/personas/show.rhtml
```

Recordemos que ruby utiliza **convención** en vez de **configuración**. La vista para el módulo de personas siempre comenzará en el esquema (layout) de personas. Cuando vemos el texto dentro del archivo **layouts/personas.rhtml** veremos lo siguiente:

```
<html>
<head>
  <title>Personas: <%= controller.action_name %></title>
  <%= stylesheet_link_tag 'scaffold' %>
</head>
<body>

  <p style="color: green"><%= flash[:notice] %>

  <%= @content_for_layout %>

</body>
</html>
```

Un archivo RHTML es una combinación de HTML y ruby. El código ruby vive dentro de los delimitadores **<% %>**, o **<%= %>** si desea el resultado de una función desplegado en el HTML.

Veamos la última parte del esquema de personas:

```
  <%= @content_for_layout %>
```

Este comando le dice a rails que es aquí donde puede insertar el contenido del esquema. El esquema es como un formato general para este controlador.

Despachando peticiones en Rails

El resultado de una peticion a nuestra aplicación depende del URL. Cuando un usuario pide **http://miservidor/persona/list**, Rails deduce que el controlador se llama persona, y la acción dentro del controlador es **list**. Así que rails hará lo siguiente:

- Ejecutar el método **list** en **app/controllers/personas_controller.rb**
- Utilizar **views/personas/list.rhtml** para evaluar la presentación de los resultados del método list.
  Una vez evaluados, los resultados se ponen en una variable llamada **@content_for_layout**
- Utilizar **views/layouts/personas.rhtml** como el esquema para desplegar los resultados del controlador. En algún lugar de este esquema, mostraremos la variable content_for_layout, como vimos con anterioridad.

Esto quiere decir que, si usted desea cambiar de nombre al método list para traducirlo a **listar**, debe hacer lo siguiente:

- Cambiar de nombre al método **list** en **app/controllers/personas_controller.rb**
- Cambiar de nombre al archivo **views/personas/list.rhtml**.

Comandos con formularios

Gran parte de lo que hacemos en Rails es llenar formularios. Los formularios en Rails son compartidos utilizando el archivo _form.rhtml. Veamos una de las vistas que requieren el formulario, **app/views/personas/edit.rhtml**:

```
<%= start_form_tag :action => 'update', :id => @persona %>
  <%= render :partial => 'form' %>
  <%= submit_tag 'Edit' %>
<%= end_form_tag %>

<%= link_to 'Show', :action => 'show', :id => @persona %> |
<%= link_to 'Back', :action => 'list' %>
```

Note que el módulo de edicion prepara la forma para que llame el comando **update** y despues emite un contenido "parcial" llamado "form". Dentro del contenido parcial **_form.rhtml**, podemos ver lo siguiente:

```
<%= error_messages_for 'persona' %>

<!--[form:persona]-->
<p><&t;label for="persona_nombre">Nombre</label>

<%= text_field 'persona', 'nombre'  %></p>

 (mas y mas campos..)
<!--[eoform:persona]-->
```

Rails tiene comandos para todos los controles de forma del HTML, y tien algunos controles extra. Cuando usted estudie el HTML generado, podrá hacer sus propios controles complejos.

Parciales

Como lo hemos visto con el ejemplo de la forma, los parciales son muy útiles. Un "parcial" es un fragmento de RHTML que usted puede re-utilizar en varios contextos, ya sea en varios métodos, en varios controladores, o varias veces en la misma vista.

Usted puede crear un parcial simplemente añadiendo el archivo (su nombre debe comenzar con un simbolo de subrayado) y llamando el parcial con **render :partial=>'miparcial'** (para un archivo llamado _miparcial.rhtml). Por defecto, el parcial será buscado dentro del directorio de la vista actual, pero usted puede cambiar el directorio simplemente mencionandolo, por ejemplo **render :partial=>'directorio/miparcial'** (en este caso, el archivo sería **app/views/directorio/_miparcial.rhtml**).

El parcial tendra acceso a todas las variables de instancia del controlador y la vista que lo llamó.

Ideas para usos de parciales

Además de las formas, he aquí algunos ejemplos para uso de parciales:

- **Navegación**. En vez de repetir un area de navegacion comun por cada pagina, usted puede llamar un parcial para la navegacion comun.
- **Cabeceras/Pieceras**. Usted puede crear cabeceras y pieceras una sola vez y simplemente llamarlas dentro de sus esquemas.
- **Registros complejos dentro de tablas**. Si usted necesita desplegar muchos registros y el tipo de registro es muy complejo (digamos que tiene muchas clases y javascript que debe de funcionar perfectamente), puede crear un parcial para desplegar un registro individual y llamar el parcial dentro del ciclo.
- **Canastas para compras**. Es importante que la canasta que muestra la orden en un programa de comercio sea uniforme. En vez de repetir el código, usted puede llamar un parcial.

Cambiando el Listado

Ahora hay que hacer que el listado se vea mejor. Primero veamos el código generado en **views/personas/list.rhtml** (o listar, si ya le cambió el nombre).

Leyendo el código generado descubriremos el uso de metadata en las columnas de registros de ActiveRecord:

```
<table>
  <tr>
  <% for column in Persona.content_columns %>
    <th>< column.human_name %><th>
  <% end %>
  <tr>

<% for persona in @personas %>
  <tr>
  <% for column in Persona.content_columns %>
    <td><%= h persona.send(column.name) %><td>
  <% end %>
    <td>< link_to 'Show', :action => 'show', :id => persona %><td>
    <td>< link_to 'Edit', :action => 'edit', :id => persona %><td>
    <td>< link_to 'Destroy', { :action => 'destroy', :id => persona },
                :confirm => 'Are you sure?'
 %><td>
  <tr>
<% end %>
```

```
<table>
```

Note que el código generado es dinámico. Cuando usted añade columnas a su objeto Persona, la tabla tendrá esas columnas tambien. También note que para crear un enlace a alguna acción sobre la persona en cada renglón, se utiliza el comando link_to.

Evidentemente, esto sólamente funciona para prototipos rápidos, porque el dinamismo sólo aplica a las columnas y no a los métodos calculados (como el campo calculado "apellidos"). Así que cambiamos el listado para que sea más estático:

```
<% for persona in @personas %>
  <tr>
    <td><%= h persona.nombre %><td>
    <td><%= h persona.apellidos %><td>
                ... (mas campos) ..
    <td>< link_to 'Show', :action => 'show', :id => persona %><td>
 </tr>
```

Obviamente es aquí donde usted puede hacer el HTML tan artístico como quiera. Hablaremos de HTML y CSS más tarde, pero esto le da una buena base para comprender cómo funciona la generación de vistas en Rails y cómo usted puede cambiar el HTML generado y planear su sitio. Por ahora cambie la tabla para que tenga sólo los registros que usted desee.