UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

SISTEMA DE DIAGNÓSTICO AMBIENTAL: COMPARATIVO DE DOS IMPLEMENTACIONES

REPORTE DE TRABAJO PROFESIONAL

QUE PARA OBTENER EL TÍTULO DE:

LICENCIADO EN CIENCIAS DE LA COMPUTACIÓN

P R E S E N T A :

RAÚL MARTÍNEZ HERNÁNDEZ

TUTORA:
ELISA VISO GUROVICH

2017

Hoja de Datos del Jurado

1. Datos del alumno
Martínez
Hernández
Raúl
55 06 78 54
Universidad Nacional Autónoma de México
Facultad de Ciencias
Ciencias de la Computación
304141131

2. Datos del tutor
Dra.
Elisa
Viso
Gurovich

3. Datos del sinodal 1
Dr.
José de Jesús
Galaviz
Casas

4. Datos del sinodal 2
M. en C.
María Guadalupe Elena
Ibargüengoitia
González

5. Datos del sinodal 3
Dra.
Amparo
López
Gaona

6. Datos del sinodal 4
Dr.
Canek
Peláez
Valdés

7. Datos del trabajo escrito
Sistema de Diagnóstico Ambiental
Comparativo de dos implementaciones
73 p
2017

Índice general

Introducción

Preámbulo

Cuando uno enfrenta un problema es común que se pregunte, para empezar, si tiene solución o no y en caso afirmativo se procede a desarrollar una primera forma de resolverlo. A continuación se contempla la posibilidad de que esta solución no sea única, sino que pueda haber más de un método para llevar a cabo un procesamiento de datos equivalente y, de ser así, uno trabaja en la búsqueda y elaboración de estas rutas alternas. Por último, una vez que se cuenta con un arsenal de métodos para resolver el problema, lo último que queda es elegir cuál de todos es el *mejor*.

Como bien sabe cualquier persona entrenada en ciencias de la computación, ninguno de los pasos anteriores es trivial, dependiendo del entorno en que se encuentre el problema (teórico o práctico) se procederá de una u otra forma antes de dar el siguiente paso.

Por otra parte, se usó el término "mejor" de manera un tanto ambigua, debido a que su interpretación depende de la prioridad que se le dé a cada una de las características que puede tener la solución, mismas que varían nuevamente dependiendo si se trata de un problema en el ámbito de la teoría o de la práctica. Un ejemplo clásico de esto es decidir entre una solución que consume mucha memoria y tiene un tiempo de ejecución muy bajo contra una solución que sea más lenta pero cuyo uso de memoria es mínimo.

Este trabajo presenta un ejemplo concreto de un problema práctico, para el cual se desarrolló una primera solución con ciertas características en mente, pero que resultaron no ser óptimas cuando el sistema se puso en marcha y en manos de los usuarios. Debido a ello se tuvo que retomar el análisis, encontrar los puntos débiles del desarrollo anterior y usar estos conocimientos para crear una nueva implementación que difirió mucho en estructura y forma de la primera, pero que efectivamente daba una solución equivalente y con características más adecuadas al uso que realmente se le dio al sistema.

Descripción del problema

El Instituto Nacional de Bellas Artes (en adelante abreviado INBA) cuenta con múltiples entidades para cubrir todas las etapas de la enseñanza y difusión de las artes en México. Naturalmente, los edificios donde opera fueron construidos en diferentes periodos de tiempo, por lo que tuvieron diferentes especificaciones y requerimientos en cuanto al equipo eléctrico (por ejemplo luminarias y aires acondicionados) e hídrico (regaderas, sanitarios, lavamanos, entre otros) que adquirieron e instalaron, mismas que fueron cambiando con el tiempo como consecuencia de mantenimiento y actualizaciones. Además, como en cualquier lugar de trabajo, la operación diaria implica la generación de residuos tanto orgánicos (principalmente desechos de alimentos) como inorgánicos (papel, plástico, aluminio, entre otros), y cada entidad ha desarrollado sus propias prácticas para desecharlos o reciclarlos.

El INBA se halló a sí mismo con un conjunto de entidades cuyas prácticas en materia de sustentabilidad ambiental eran muy variadas (en algunos casos, completamente inexistentes) por lo que decidió hacer un llamado a todos sus integrantes a realizar un esfuerzo conjunto y coordinado, al cual le llamó INBA Sustentable.

La iniciativa INBA Sustentable busca, por un lado, estandarizar el tipo de equipos que utilizan todas las entidades del INBA y por el otro, minimizar el consumo de bienes y generación de desperdicios que inevitablemente ocurren como consecuencia de las actividades del día a día.

Planteamiento general de la solución

Para alcanzar los objetivos propuestos por INBA Sustentable, es indispensable contar primero con un estándar que indique cuáles son los rangos aceptables de consumo de bienes y generación de desperdicios en cada área. Después, se requiere realizar un inventario exhaustivo de todos los equipos existentes en los distintos edificios, con especial énfasis en sus datos de consumo (por ejemplo kilowatts por hora y litros por segundo) y su estado (si está descompuesto o no, cuántas horas al día se usa, por mencionar algunos). A continuación, se debe implementar un mecanismo para registrar regularmente la generación y separación de basura.

Teniendo el inventario y un cierto número de registros, el siguiente paso es evaluar su nivel de conformidad con el estándar de sostenibilidad ambiental establecido previamente. Con este análisis se puede crear un plan de acción que contenga aproximaciones de la inversión requerida, sus beneficios y la prioridad con que se debe atender cada área.

Se propuso el desarrollo de un sistema para automatizar todo el proceso, dividiéndolo en las siguientes etapas:

1. Preparación: Donde se alimenta al sistema una lista de entidades del INBA (organizada por edificios, pisos, espacios y demás) y formularios que describen por un lado los equipos que se deben inventariar y por el otro las características de los residuos generados.

2. Levantamiento: Donde una brigada de personal calificado visita cada entidad y registra los equipos que hay en cada espacio, así como el uso que se les da. En esta etapa también se realiza la captura periódica de los datos de generación de desperdicios.

3. Análisis y reporte: Donde personal especializado hace consultas al sistema para evaluar a cada entidad y emitir el diagnóstico correspondiente.

Desafortunadamente (por motivos fuera de nuestro control) al momento de arrancar el proyecto y comenzar a diseñar el sistema, no se contaba con los formularios requeridos para la etapa de preparación, ni se contaría con ellos por un periodo prolongado de tiempo. Peor aún, una vez que se tuvieran, era muy probable que sufrieran cambios constantes y, además, si en el futuro se decidía adaptar y vender el sistema a otros clientes distintos del INBA, seguramente desearían utilizar un conjunto diferente de formularios.

En consecuencia, no era sensato construir la capa de datos del sistema estáticamente, por lo que la flexibilidad fue una prioridad desde la concepción del sistema.

Capítulo 1

Sistema reflexivo de base de datos orientada a objetos en Ruby

1.1. Motivación

Enfrentados con un tiempo de desarrollo apenas suficiente, llegamos pronto a la conclusión de que usar un *framework* sería de gran ayuda. Generamos una lista con las opciones más populares (incluyendo las ventajas y desventajas de cada una) e inmediatamente sobresalió *Ruby on Rails*, *framework* de desarrollo web ágil basado en Ruby.

Ruby es un lenguaje de programación multiparadigma (con especial tendencia a la orientación a objetos), reflexivo, dinámicamente tipificado e interpretado (se ejecuta en una máquina virtual), creado con la facilidad de uso como prioridad (para mayor claridad de los ejemplos de código y la notación utilizada en este trabajo, puede consultar una breve introducción del lenguaje en el Anexo 1 de este documento).

Ruby on Rails extiende la filosofía de Ruby, por lo que no solo busca ser fácil de usar, sino también minimizar el tiempo de desarrollo y la complejidad de la solución, lo cual logra al adherirse estrictamente a las filosofías "Convención sobre Configuración" (*Convention over Configuration*) y "No te Repitas" (*Don't Repeat Yourself*). Más aún, *Ruby on Rails* incluye de fábrica el ambiente completo para crear y probar un sistema web sin requerimientos adicionales, por lo que montar el proyecto en una computadora nueva es muy rápido (factor que suele subestimarse en el desarrollo de sistemas).

1.2. Descripción general

Fundamentalmente, el producto que se desarrolló es un sistema de captura, almacenamiento y análisis de información, por lo que todos sus componentes (desde el almacenamiento hasta la interfaz de usuario) dependen fuertemente de la estructura de la base de datos misma que, como ya se mencionó, estaba indefinida y además era susceptible de cambiar frecuentemente.

Por lo tanto, debíamos idear una estructura genérica de base de datos, sobre la cual se pudiera construir el resto del sistema, se pudiera almacenar cualquier tipo de objeto (cuando se contara con su definición) y se pudiera modificar fácilmente cuando surgieran cambios.

Ruby on Rails ya incluye un sistema de Mapeo Objeto-Relacional (*Object-Relational Mapping*), pero éste requiere que ya se conozca la definición de las tablas que se quieren almacenar (columnas, tipos de dato, restricciones de integridad y demás) por lo que no era factible usarlo directamente.

Se realizaron varias iteraciones de modelos genéricos de base de datos, pues en cada paso se detectaban elementos que tenían la misma estructura pero distintos nombres, ocasionando que cada modelo fuera más compacto que el anterior. En algún punto detectamos que el modelo Entidad-Relación creado ilustraba perfectamente el paradigma de desarrollo Orientado a Objetos, por lo que decidimos usar la misma terminología y dar los nombres adecuados a las tablas: Clase, Atributo, Herencia, Objeto y demás.

Esta estructura satisfacía los requerimientos planteados y más aún, podía utilizarse no solo en este proyecto en específico, sino en cualquier otro que surgiera en el futuro. Por esta razón, decidimos separar completamente la base de datos orientada a objetos (y la capa mínima necesaria de software para interactuar con ella) en su propia *gem*. Una *gem*[1] es la forma estándar de Ruby para crear y empaquetar componentes reutilizables que después pueden integrarse fácilmente a un proyecto de *Ruby on Rails* mediante el comando *bundle*.

El diseño de las capas restantes (interfaz y lógica) fue mucho más sencillo gracias a que *Ruby on Rails* indicaba cómo debían separarse y comunicarse los diversos componentes del sistema web, pues requiere que éstos se construyan de acuerdo al patrón de diseño MVC (Modelo-Vista-Controlador).

La suma del modelo de datos y la *gem* de Ruby que dicta cómo utilizarlo (conjunto que en adelante llamaremos *núcleo*) es el resultado más valioso de este sistema y el objeto principal de estudio de este capítulo, pues es aquí donde se aplicaron las técnicas de programación que interesan al tema central

[1]Este es el nombre técnico que le dio el creador de Ruby, por lo que aparecerá tal cual.

de este trabajo. Se hará una breve exposición de los principales módulos de las otras capas, meramente por completez y por describir cómo se beneficiaron de las bondades del núcleo.

1.3. Módulos

1.3.1. Núcleo

Se encarga de almacenar y controlar el acceso a los datos. Consta de dos componentes principales: la base de datos y el código en Ruby que permite interactuar con ella.

Base de datos

La figura de la página 9 muestra el diagrama Entidad-Relación utilizado por el sistema. Con tan solo 9 tablas, permite almacenar cualquier tipo de información debido a que es una representación del paradigma orientado a objetos.

Las tablas se pueden agrupar en dos conjuntos: las que almacenan la descripción de los datos y las que almacenan los datos en sí.

Las que almacenan la descripción de los datos:

- *z_classes*: Modela una clase. Contiene el nombre y un booleano que dice si es abstracta.
- *z_class_attributes*: Modela los atributos de una clase.
- *z_inheritance*: Almacena la información de herencia, que puede ser múltiple (es decir, una clase puede tener muchos padres e hijos).
- *z_relationships*: Modela las relaciones de pertenencia entre clases.
- *z_rule_types*: Contiene las reglas de propagación de cambios para las relaciones de pertenencia.

Las que almacenan información son:

- *z_objects*: Modela un ejemplar de una clase.
- *z_classes_z_objects*: Almacena la información necesaria para saber de qué clases es ejemplar el objeto. Observe que la pluralización de "clase" es intencional, lo cual implica que el sistema permite que un mismo objeto sea ejemplar de múltiples clases a la vez.
- *z_class_attributes_z_objects*: Almacena los valores concretos de cada atributo que tiene el objeto.
- *z_objects_z_relationships*: Contiene la información de las relaciones concretas de pertenencia del objeto.

En secciones posteriores se hará más evidente la función de cada tabla y sus columnas.

Capa de código para la interacción con la base de datos

Esta capa consta de numerosas clases en Ruby que pueden agruparse en tres categorías: las que representan las tablas de la base de datos, las que conforman un *parser* (mismo que se describe más adelante) y las que sirven para conectar al núcleo con otros proyectos de Ruby (es decir, las que hacen que sea una *gem*).

En la primer categoría se tiene una clase por cada tabla de la base de datos, cuyo nombre difiere solo en notación (utilizan *CamelCase*).

Todas ellas heredan de un componente fundamental de *Ruby on Rails* llamado *ActiveRecord* que les otorga métodos como *save* y *delete*, cuya función es generar y ejecutar automáticamente sentencias SQL que corresponden a la acción deseada.

Además de servir como puente para la base de datos orientada a objetos, las clases de esta categoría proveen numerosos métodos para simplificar el manejo del concepto que representan. Por ejemplo, *ZClass* contiene un método *all_descendants()* que devuelve una lista con todas las clases que heredan (directa o indirectamente) de una *ZClass* particular (se puede consultar la referencia de todos los métodos disponibles en *ZClass*, y de todas las demás clases del núcleo en el Anexo 2 de este documento).

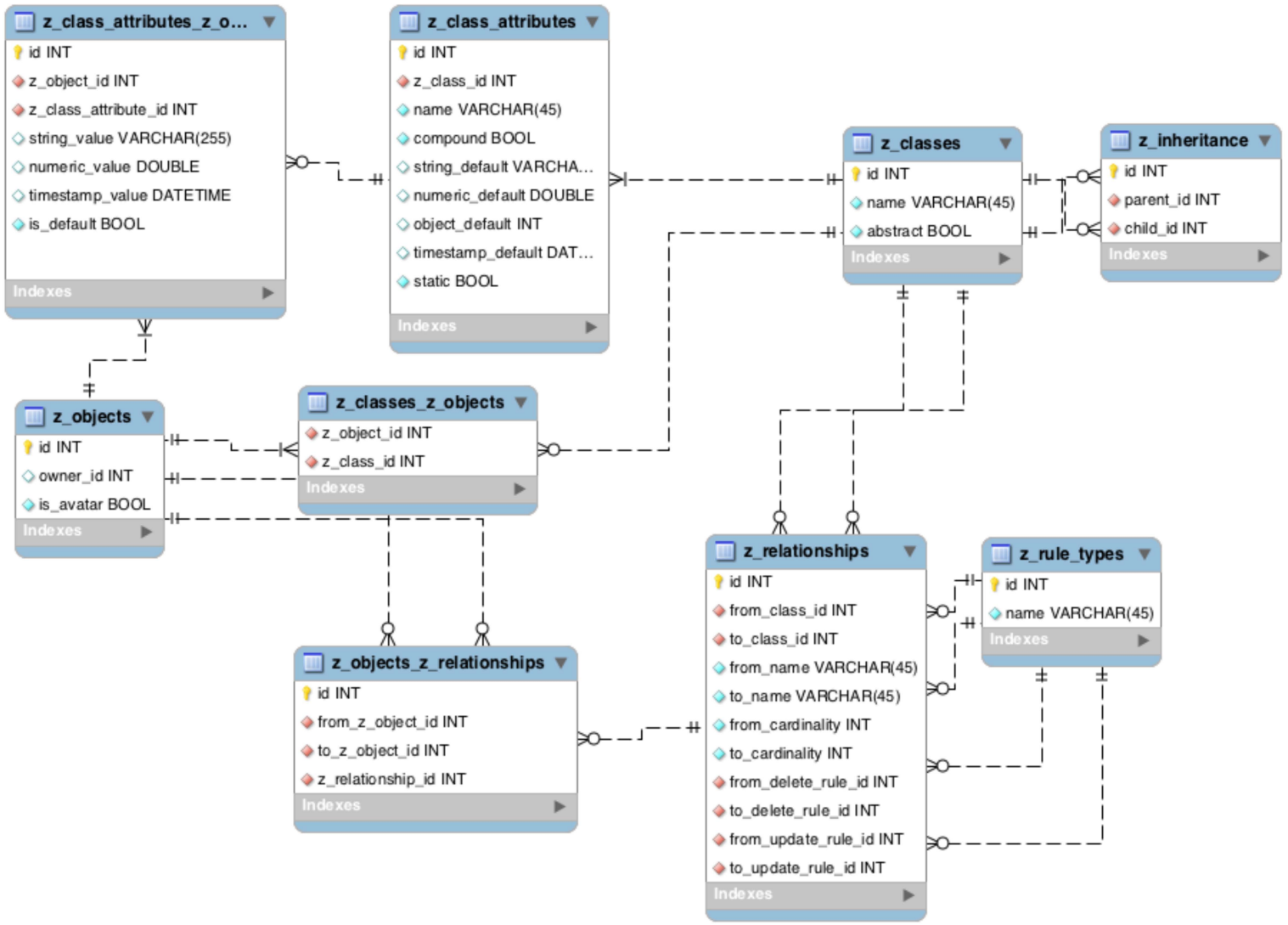

Figura 1.1: Diagrama Entidad-Relación de la base de datos utilizada por el sistema en Ruby.

La segunda categoría consta esencialmente de dos clases, *Node* y *Parser*, que en conjunto permiten procesar archivos de texto plano para generar objetos de tipo *ZClass* y guardarlos en la base de datos. En otras palabras, estos componentes proveen un punto de entrada básico para llenar la base de datos orientada a objetos de manera simple.

Naturalmente, los archivos de texto que sirven de entrada para esta parte deben adherirse a un formato bien definido[2], susceptible de ser interpretado por una computadora. Dicho formato se definió junto con el usuario final de la aplicación y se le nombró UXF. La intención era que el usuario pudiese generar la descripción de los artículos que quería almacenar en el sistema de manera rápida y simple, sin requerir ningún conocimiento ni herramienta especializada.

La tercer categoría está conformada por todos los archivos que Ruby requiere para describir una *gem*. Aunque el código que contienen es extenso e indispensable para que el sistema funcione, no aporta nada al tema de este trabajo, por lo que no profundizaremos en sus detalles.

1.3.2. Espacios

Los espacios modelan ubicaciones del mundo real y definen las acciones que se pueden realizar en éstas.

Los atributos de los espacios se ajustaban a las necesidades del cliente y se modelaron utilizando las clases del núcleo. Algunos de los más comunes eran nombre, descripción y tipo (cocina, pasillo, oficina y demás).

La lógica programada para los espacios se ocupa de dos características principales:

1. Un espacio puede contener otros espacios, pero éstos respetan una jerarquía definida por el usuario.

   Por ejemplo, una jerarquía puede indicar que en el nivel más alto existen entidades, cada entidad consta de edificios, cada edificio tiene plantas y por último cada planta tiene cuartos. Naturalmente, se escribieron validaciones para evitar que el usuario intente capturar conjuntos de espacios que violen la jerarquía (por ejemplo, que intente definir un edificio que contenga otros edificios).

2. Todo espacio tiene un inventario de n artículos (n puede ser 0) de las clases definidas por el usuario.

[2]Un pequeño lenguaje diseñado especialmente para ello.

1.3.3. Usuarios

Modela a las personas que usan el sistema y contiene la lógica necesaria para manejar su interacción con el mismo, a saber, la autenticación (quiénes pueden ingresar al sistema) y la autorización (una vez dentro, qué puede hacer cada quién).

Los atributos de los usuarios se ajustaban a las necesidades del cliente y se modelaron utilizando las clases del núcleo. Algunos de los más comunes eran nombre, apellidos, correo, teléfono, entre otros.

Para la lógica utilizamos una *gem* llamada *Devise* que contiene todos los métodos necesarios para iniciar sesión, recuperación de contraseña olvidada, caducidad de sesiones, cerrar sesión y demás.

El único código a la medida que escribimos fue para indicar que un usuario está asociado a un espacio, por lo que solamente puede capturar el inventario de ese lugar o de los espacios que contiene (por ejemplo, se puede asociar un usuario a la entidad "Facultad de Ciencias", con lo que se impide que intente capturar el inventario de otras facultades).

Adicionalmente se incluyó la opción de asociar a cada usuario una o más clases abstractas, de tal forma que sólo puede crear objetos construidos a partir de descendientes de las clases asociadas (por ejemplo, si se asocia el usuario a la clase "equipo de cómputo" sólo estará autorizado para crear objetos de clases que se deriven de ella).

1.3.4. Captura

Contiene los elementos necesarios para que los usuarios puedan ingresar nueva información al sistema, es decir, *qué* quiere capturar (según las clases disponibles en el núcleo) y *dónde* (utilizando el módulo de espacios).

Este módulo consta únicamente de vistas (las pantallas que el usuario ve y usa) y controladores (aquellos que dictan qué sucede cuando el usuario manipula las vistas) ya que el resto de la lógica se encuentra en otros módulos.

1.3.5. Consulta

Provee mecanismos para que los usuarios puedan extraer información del sistema.

Al igual que el módulo de captura, este otro módulo contiene solo vistas y controladores.

En general, las pantallas de este módulo ofrecen al usuario controles para que ingrese parámetros de búsqueda y que obtenga los resultados que desee, ya sea en forma tabular (donde se presentan los artículos de manera individual) o gráfica (donde se condensan y presentan los resultados mediante funciones de agregación).

La Figura 1.2 presenta la división general del sistema en módulos, así como la capa del patrón MVC a la que pertenecen. Tome en cuenta que, si bien los módulos "espacios", "usuarios", "captura" y "consulta" se comunican entre sí, no se agregaron conexiones entre ellos para mejorar la presentación.

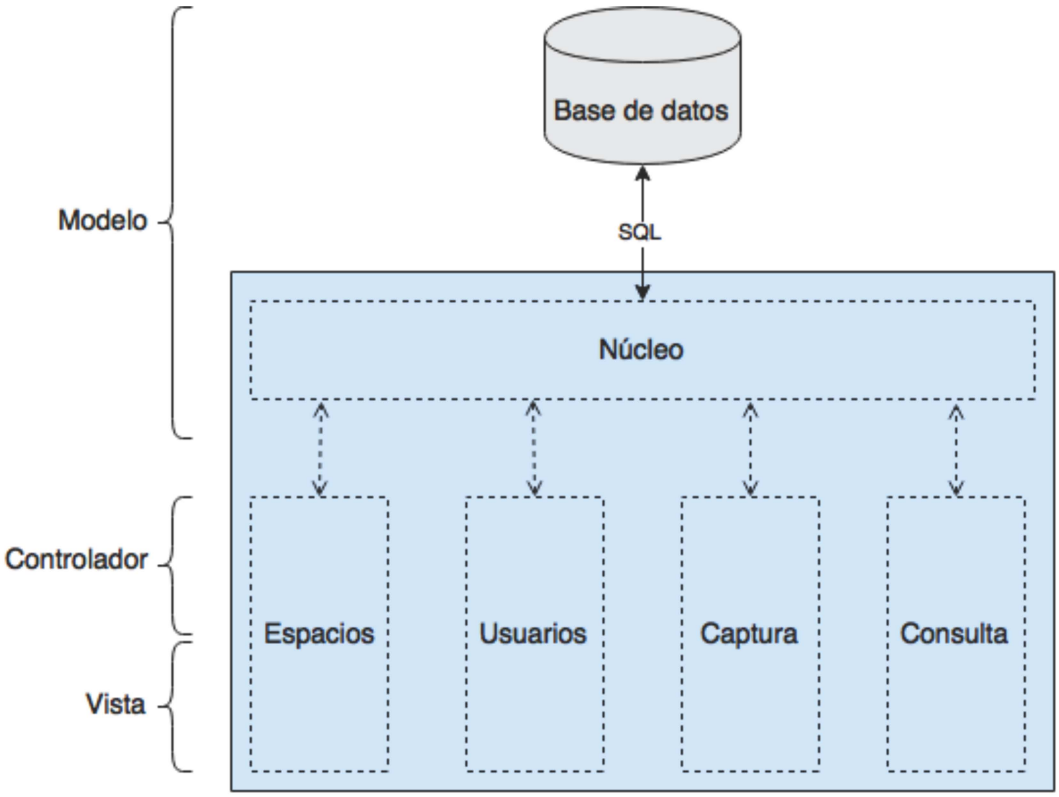

Figura 1.2: Módulos, capas e interacciones del sistema en Ruby.

1.4. Flujo de datos

1.4.1. Preparación

Esta fase comienza con un servidor Linux "nuevo" (es decir, que puede o no tener instalado el software base para ejecutar el sistema) y termina cuando el sistema está listo para ser utilizado por los usuarios.

Está fuera del alcance de este documento describir la descarga e instalación del software base (Ruby, MySQL y demás) por lo que continuaremos la explicación suponiendo que ya se cuenta con ellos. Más aún, daremos por hecho también que ya se copió el código del sistema al servidor que lo hospedará.

La instalación del sistema requiere varios archivos de entrada, mismos que se listan a continuación:

- Archivo con extensión *csv* (valores separados por coma) que contiene todos los espacios donde el cliente quiere que se lleve a cabo el levantamiento de información. Cada línea contiene la descripción de un único espacio.

- Imágenes con los planos arquitectónicos de los espacios. Se permiten archivos con extensión *jpg* o *png*.

- Conjunto de archivos con extensión *uxf* que describen los artículos que al cliente le interesa que se almacenen en el inventario. Recordemos que un *uxf* es un archivo de texto plano que permite describir una sola clase de manera simple.

Todos estos archivos se solicitan al cliente pues no se requiere ningún conocimiento técnico para poder generarlos (por supuesto, se ofreció orientación cuando se presentaban dudas al respecto).

Una vez que se tienen los archivos de entrada y se colocan dentro de una carpeta (siguiendo ciertos lineamientos de nomenclatura y organización), es necesario conectarse a MySQL para crear la base de datos (vacía) junto con un usuario que tenga todos los permisos sobre ella. Los datos de acceso (servidor, usuario, contraseña y demás) se deben copiar en un archivo de configuración de *Ruby on Rails* llamado apropiadamente *database.yml*.

A continuación se debe invocar al proceso de configuración inicial del sistema mediante la línea de comandos. Como único parámetro recibe la ruta absoluta al directorio donde se encuentran los archivos de entrada, con lo que procede a ejecutar los siguientes pasos:

1. Se conecta a la base de datos y verifica si las tablas ya existen. Si no existen las crea y si ya existen las limpia.

2. Crea las *ZClass* básicas del sistema (usuario, espacio y demás) con los atributos y relaciones que les corresponden.

3. Explora recursivamente la jerarquía de directorios en búsqueda de archivos con extensión uxf.

   Por cada uno de ellos, invoca al método *parse()* de la clase *Parser*, el cual creará la *ZClass* que describe al artículo tal como está definido en el archivo.

4. Procesa el archivo con extensión *csv* línea por línea, creando el *ZObject* correspondiente a cada espacio.

5. Recorre el subdirectorio con los planos arquitectónicos, asociando cada plano al espacio que le corresponde.

Naturalmente, si en alguno de los pasos encuentra un error (como un archivo inexistente o un archivo mal formado) lo reporta en la consola y termina.

Una vez que el comando de configuración inicial termina exitosamente, solíamos realizar pruebas (primero internas y después con el usuario) para verificar que todo estuviese correcto y que se podía proceder con la fase de levantamiento.

1.4.2. Levantamiento

Comienza cuando los usuarios están satisfechos con los resultados de la fase de preparación y termina una vez que se ha capturado el inventario en todos los espacios que al cliente le interesan.

Para esta fase se contaba con la ayuda de una "brigada de levantamiento", que consistía de un grupo de personas capacitadas para utilizar el software. Estas personas se trasladaban a los lugares donde se debía realizar el inventario y contaban con dispositivos móviles que les permitían utilizar el sistema de manera remota.

Apoyándose en los planos arquitectónicos, visitaban cada uno de los espacios y capturaban los datos de los artículos que ahí encontraban de acuerdo al siguiente flujo:

1. El usuario elige la opción de captura.
2. El sistema consulta la base de datos y obtiene la lista con los espacios de más alta jerarquía (por ejemplo, la lista de las facultades si se tratara de Cuidad Universitaria).
3. El sistema traduce los resultados de la base de datos a sus correspondientes *ZObject*.
4. Utilizando los *ZObject*, el sistema dibuja en la interfaz un selector para que el usuario indique en dónde se encuentra.
5. El usuario elige alguno de los espacios. Su selección se comunica al servidor mediante JavaScript asíncrono (mejor conocido como AJAX[3]).
6. El sistema obtiene la lista de los subespacios contenidos en el espacio que eligió el usuario.
7. Los pasos 3-6 se repiten tantas veces como sea necesario hasta que el usuario haya indicado correctamente su ubicación actual.
8. De manera similar a la selección del espacio, el usuario interactúa con el sistema para elegir el tipo de artículo que desea capturar (por ejemplo, primero elige "equipo eléctrico", después "equipo de cómputo" y por último "laptop").
9. Haciendo uso de los métodos de *ZClass*, el sistema dibuja un formulario que permite capturar toda la información requerida para describir al artículo deseado, con las siguientes consideraciones:
    - Dado que cada *ZClassAttribute* sabe de qué tipo de dato es, el sistema es capaz de dibujar un campo de captura adecuado. Por ejemplo, si es un atributo de tipo fecha, mostrará un selector de tipo calendario.
    - Dado que cada *ZClassAttribute* indica si es obligatorio o no, el sistema genera código de validación en JavaScript para impedir que el usuario omita la captura de dicho atributo en caso de que sí lo sea.

[3]Asynchronous JavaScript And XML, mecanismo utilizado en programación web para modificar parte de una página, sin tener que recargar la ventana completa.

- Si la *ZClass* define una o más relaciones de composición (por ejemplo "toda computadora tiene uno o más monitores"), el sistema genera un formulario anidado para que el usuario capture los datos de ese "sub artículo", así como botones para agregar más o quitar los que ya capturó.

10. El usuario llena el formulario y adicionalmente puede tomar n fotografías del artículo, para mostrar características particulares del mismo.

11. El usuario elige la opción "guardar", lo cual ocasiona que se envíen los datos capturados mediante AJAX.

12. El sistema recorre cada uno de los campos proporcionados por el usuario y los utiliza para crear un *ZObject* (con sus correspondientes *ZClassAttributesZObject* y *ZObjectsZRelationship*) mismo que almacena en la base de datos.

Nuevamente, si en alguno de los pasos el sistema detecta un dato inválido, impedirá la creación del artículo y reportará el error al usuario para que éste tome la acción adecuada.

Por otra parte, si el registro se almacenó correctamente, el sistema muestra un mensaje de éxito y habilita un botón llamado "duplicar registro", cuya función es crear un nuevo registro con todos sus datos iguales al anterior (excepto fotografías). Esta es una funcionalidad que el usuario solicitó para agilizar el levantamiento, ya que en un mismo espacio solían encontrarse numerosos equipos que sólo variaban en su número de serie.

1.4.3. Análisis y reporte

Esta etapa inicia cuando se ha realizado el inventario de todas las entidades solicitadas por el cliente y termina con la entrega de n diagnósticos en PDF (uno por cada entidad) con los resultados del análisis.

El flujo de datos es como sigue:

1. Un especialista entra al módulo de consultas e indica el espacio que desea consultar. La búsqueda puede ser a cualquier nivel de la jerarquía (por ejemplo toda una entidad, todo un edificio, todo un piso).

2. A continuación, elige la categoría de artículos de los que desea obtener información.

3. Por último, el especialista indica si desea ver todos los datos en forma tabular o bien si prefiere un resumen condensado con gráficas y totales.

4. El sistema toma los parámetros de la búsqueda, verifica que sean válidos y los utiliza para llamar a los métodos de consulta del núcleo. Por simplicidad, se listan sólo los pasos esenciales de este proceso:

   - Por cada espacio incluido en la selección del usuario, se obtiene su *ZObject* correspondiente.
   - La categoría que eligió el usuario se pasa como parámetro al método *relationship_objects()* del *ZObject* que representa al espacio, lo cual devuelve un arreglo de *ZObject* con los artículos solicitados.
   - Se ejecuta *property_value_hash()* y *relationship_objects()* sobre cada *ZObject* obtenido en el paso anterior para obtener los valores concretos de todas sus propiedades.
   - Si el usuario eligió la vista condensada, se realiza un procesamiento adicional para calcular promedios, porcentajes y demás.

5. El sistema toma los resultados del paso anterior, los serializa en formato JSON[4] y los entrega a la capa de interfaz para que sean presentados al usuario.

6. Si el usuario eligió la vista tabular, se habilita una opción para descargar los resultados en un archivo *csv* (proceso que se realiza en JavaScript sin tener que llamar al servidor).

Estos pasos se repiten tantas veces como el especialista requiera hasta concluir la elaboración del diagnóstico, mismo que podrá cargar al sistema en formato PDF para que pueda ser consultado por el cliente.

[4]JavaScript Object Notation, formato estándar para representar objetos en JavaScript como texto.

Capítulo 2

Sistema dinámico de generación y administración de bases de datos en PHP

2.1. Motivación

Bastaron sólo unos meses para que la solución en Ruby comenzara a mostrar una severa degradación de desempeño.

Gracias a su diseño, agregar nuevas características y dar mantenimiento a las instancias existentes era muy sencillo, por lo que atender solicitudes de cambios del cliente era cuestión de un par de días a lo más. Sin embargo, por más que se desarrollaron parches para reducir los tiempos de respuesta, ninguno de estos fue suficiente para mantener el paso de los volúmenes de datos cada vez más grandes que los usuarios pretendían manejar.

Por otra parte, los elevados costos de operación y la escasa mano de obra de los sistemas en Ruby contribuyeron también a la decisión de reconstruir el sistema e, incluso, fueron los factores principales en la elección de la nueva plataforma: PHP[1].

PHP fue concebido desde sus inicios para ser utilizado en desarrollo web y es considerado el lenguaje más popular para desarrollar este tipo de sistemas (casi 80 % de todos los sitios web existentes lo utilizan en alguna proporción), por lo que cuenta con numerosos recursos de aprendizaje, grandes comunidades de usuarios y muchos más proveedores de hospedaje web que Ruby, lo cual implica costos de operación dramáticamente inferiores a los que estábamos manejando con el sistema anterior.

[1] Acrónimo recursivo, PHP: Hypertext Preprocessor.

Ahora bien, para este entonces la definición del sistema ya estaba en un estado de madurez mucho mayor: El usuario ya tenía una idea más sólida de los datos que le interesaba manejar y los desarrolladores ya teníamos bien identificados los casos de uso y los flujos de datos que éstos implicaban. Esto nos permitió optar por un diseño simplificado, que resolviera bien lo que tenía que resolver, sin pretender ser una herramienta universal que al final perjudicaba a los casos particulares (que es lo que habíamos intentado en Ruby).

2.2. Descripción general

A pesar de todos los defectos del sistema en Ruby, los usuarios estuvieron de acuerdo en seguirlo utilizando mientras construíamos el sistema PHP, lo cual restó presión al nuevo desarrollo.

Como la nueva prioridad era el desempeño y ya no estábamos corriendo contra el reloj, votamos por no utilizar ningún *framework* puesto que ninguno de los que evaluamos parecía ofrecer beneficios contundentes.

Ahora bien, mientras que *Ruby on Rails* facilitaba la separación en módulos de un mismo proyecto (permitiendo a varios programadores trabajar sobre el mismo con nula interferencia) no sería así con el sistema en PHP y mucho menos con la ausencia de un *framework* que indicara claramente como separar y organizar el código fuente.

Para solventar este problema, decidimos separar el trabajo en subsistemas y hacer que estos se comunicaran entre sí mediante llamadas a procedimientos remotos (o RPC[2] por sus siglas en inglés). La idea era que cada subsistema fuera ligero, tuviese puntos de entrada bien definidos y pudiera instalarse en su propio servidor web.

Con este criterio fue posible separar a los subsistemas en aquellos que el usuario manipularía de manera directa (a los que llamaremos *front end*, se puede ver un ejemplo en la página siguiente) y aquellos que sólo servirían como almacenes de información "tras bambalinas" (los llamaremos *back end*). Estos últimos no requerían una interfaz web elaborada ya que sólo los desarrolladores tendríamos la facultad de utilizarlos.

[2]Remote Procedure Call, término general para referirse a la invocación remota de funciones, procedimientos o métodos.

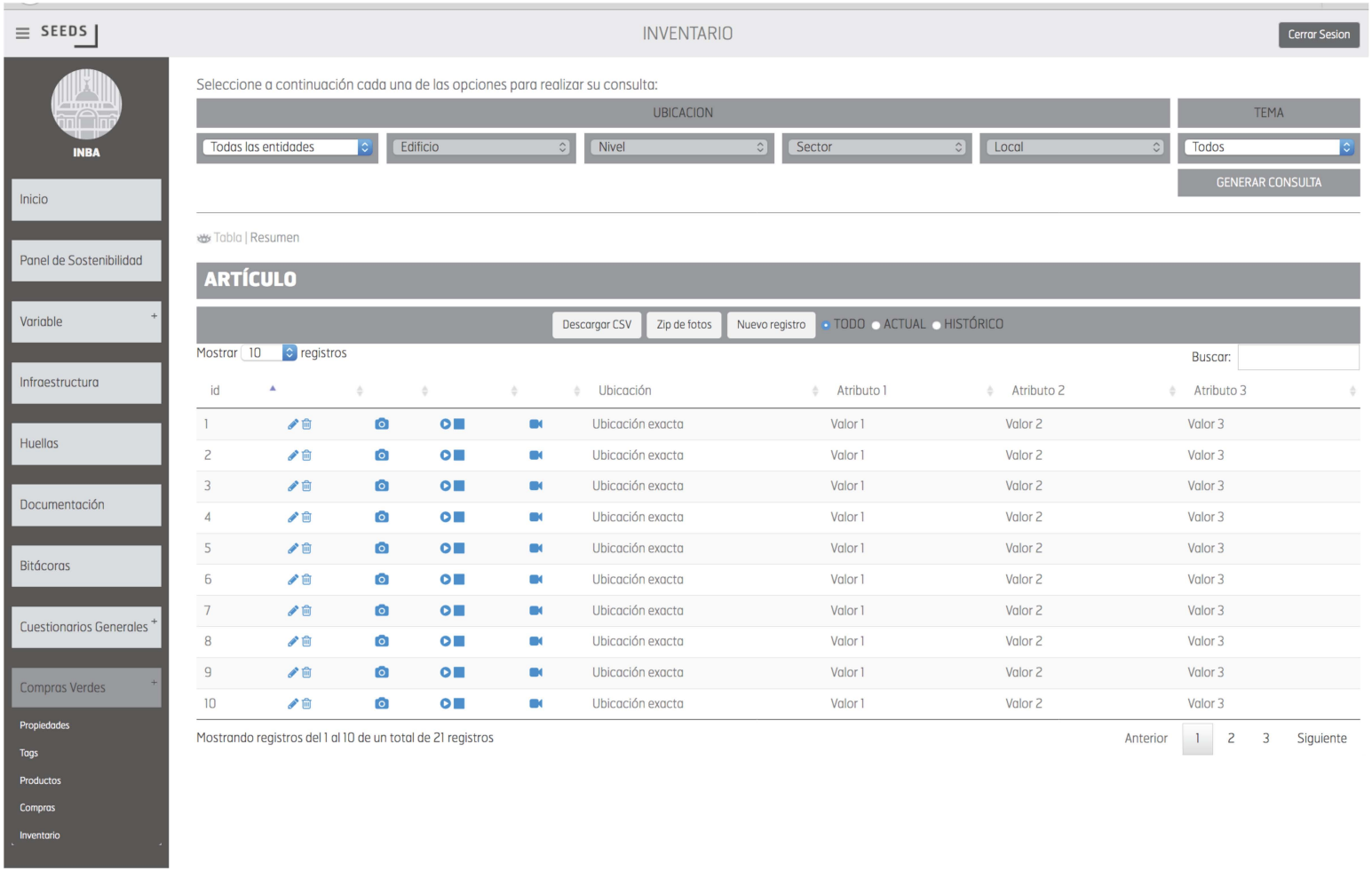

Figura 2.1: Captura de pantalla del sistema PHP.

A continuación se presenta la separación de los subsistemas en *front end* y *back end* (cada uno acompañado de una breve descripción) así como un diagrama que ilustra a grandes rasgos las interacciones entre ellos (en la página 23).

Los subsistemas del *front end*:

- Portada: Página que condensa los resultados más importantes del sistema. Se pretendía publicarla a nivel nacional.
- Inventario: Utilizado por los especialistas durante el levantamiento de datos y elaboración del diagnóstico ambiental.
- Documentos: Repositorio de todo tipo de archivos para uso interno del cliente.

Los subsistemas del *back end*:

- Espacios: Almacena, administra y provee acceso a toda la información referente a entidades, edificios, pisos y demás.
- Núcleo: Almacena, administra y provee acceso al inventario de cada espacio y al repositorio de documentos del cliente.

Nuevamente usamos el patrón de diseño MVC para estructurar el sistema. Sin embargo, debido que ahora se encontraba distribuido en varios servidores, cada subsistema jugaba solo algunos de los papeles dictados por dicho patrón de diseño. En concreto: los subsistemas del *front end* se encargaban de presentar la interfaz (Vista) y decidir cómo se comportaba ésta ante las acciones del usuario (Controlador), mientras que los subsistemas del *back end* se encargaban de la lógica y persistencia de los datos en sí (Modelo).

2.3. Módulos

Los subsistemas del *front end* no están compartimentados en módulos ni requieren estarlo. Únicamente se pueden identificar dos partes: la primera es el código HTML y CSS que dicta *cómo se ve* la interfaz que el usuario utiliza, mientras que la segunda consiste de código JavaScript más una ligera capa en PHP que juntas indican *cómo se comporta* dicha interfaz.

Respecto al *back end* yo sólo participé en el desarrollo del subsistema "núcleo", por lo que el resto de esta sección se dedica a describir los módulos que lo conforman.

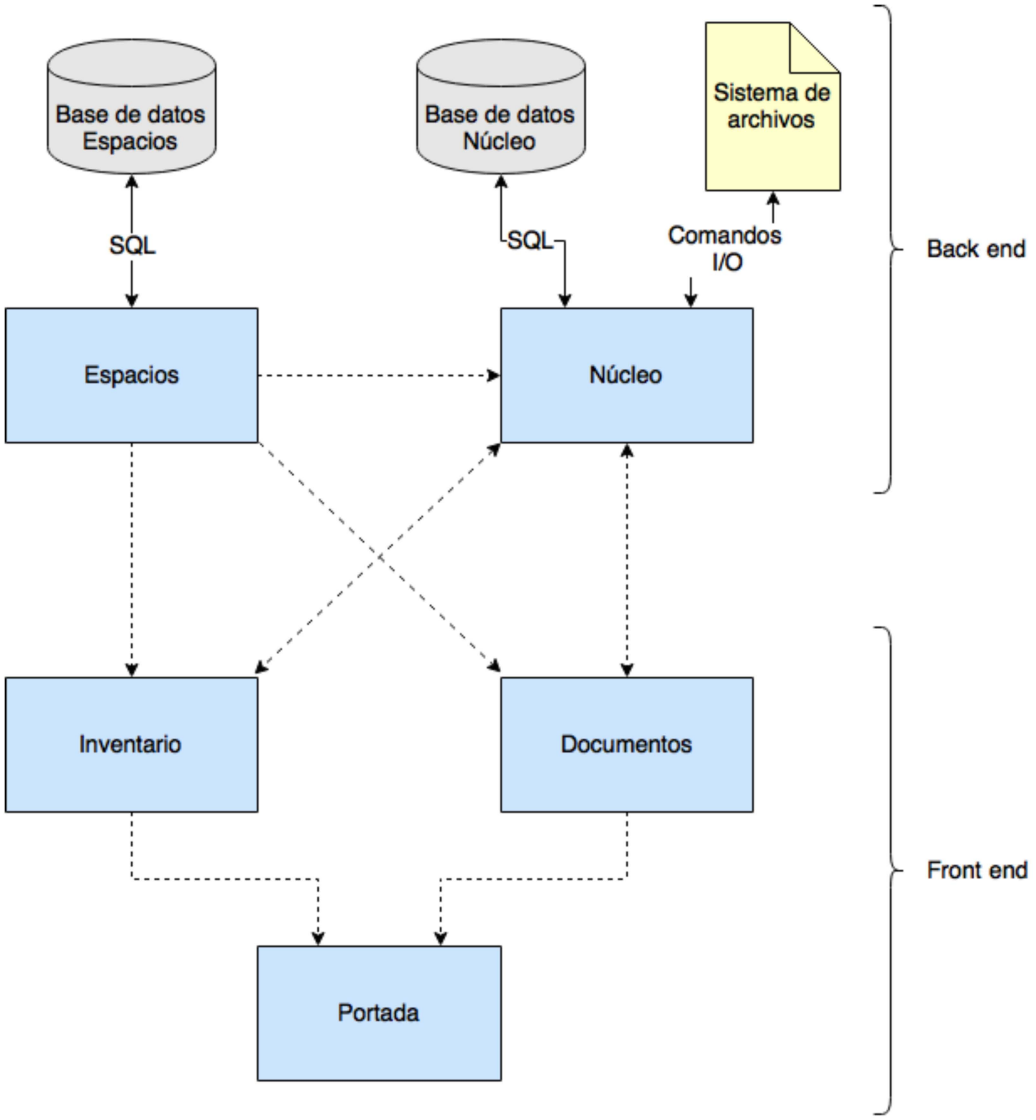

Figura 2.2: Subsistemas del sistema en PHP. Las líneas punteadas representan RPC mediante HTTP.

2.3.1. Base de datos

A nivel manejador de base de datos se tiene un conjunto pequeño de tablas de sistema más una tabla por cada tipo de artículo que el usuario desea inventariar. Las tablas de sistema almacenan metainformación y sirven como directorio del resto de la base de datos. Las tablas "de usuario" almacenan los registros individuales de cada tipo de artículo.

Estas son las tablas de sistema (la función de algunas será evidente hasta secciones posteriores):

- *spaces:* Sirve como puente con el subsistema "espacios".
- *attachments:* Almacena metadatos de documentos en el sistema de archivos local.
- *attachment_types:* Catálogo de "tipos de archivos adjuntos". Los más comunes incluyen PDF, imagen, archivo de audio y archivo de video.
- *tags:* Almacena *tags* (etiquetas) que los usuarios pueden definir de manera arbitraria.
- *attachment_tags:* Auxiliar para etiquetar archivos adjuntos.
- *spaces_attachments:* Auxiliar para manejar los archivos adjuntos de cada espacio.
- *forms:* Directorio de artículos que se pueden guardar en el inventario.

Como ejemplo de tablas de usuario supongamos que se desea inventariar computadoras, teléfonos e impresoras. En este caso deberán existir las siguientes tablas:

- *computadora:* Inventario de equipos de cómputo.
- *computadora_attachments:* Auxiliar para manejar los archivos adjuntos de cada equipo de cómputo.
- *computadora_tags:* Auxiliar para etiquetar cada equipo de cómputo.
- *telefono:* Inventario de aparatos telefónicos.
- *telefono_attachments:* Auxiliar para manejar los archivos adjuntos de cada aparato telefónico.
- *telefono_tags:* Auxiliar para etiquetar cada aparato telefónico.
- *impresora:* Inventario de impresoras.
- *impresora_attachments:* Auxiliar para manejar los archivos adjuntos de cada impresora.
- *impresora_tags:* Auxiliar para etiquetar cada impresora.

A nivel PHP se tienen dos clases: *DB* y *DBEntity*. *DB* se encarga de establecer la conexión con el manejador de base de datos, ejecutar enunciados SQL y traducirlas a estructuras de datos de PHP. *DBEntity* es una clase abstracta que define las operaciones esenciales que cualquier ente "almacenable" debe tener. Naturalmente, por cada tabla que defina el usuario deberá existir una clase que herede de *DBEntity* y que se encargue de implementar las operaciones específicas que le correspondan. Siguiendo el ejemplo de antes, se deberán crear 3 clases: *Computadora*, *Telefono* e *Impresora*.

Cabe mencionar que, a diferencia del sistema Ruby, esta vez no se incluyó soporte para herencia y por tanto pueden existir tablas que se asemejen demasiado o incluso sean idénticas salvo por el nombre. Estudiaremos las ventajas y desventajas de esto más adelante.

2.3.2. Adjuntos

Consta de las clases *Attachment* y *AttachmentType* que en conjunto contienen la lógica para leer y escribir sobre el sistema de archivos subyacente, así como registrar dichas operaciones en las tablas de la base de datos correspondientes.

Ahora bien, para que un programa externo pueda utilizar estas clases se definieron cuatro operaciones RPC: *create*, *update*, *delete* y *find_attachments*. Cada una de estas operaciones está contenida en su propio archivo con extensión *.php* y puede ser utilizada por cualquier programa capaz de hacer una petición HTTP.

Por ejemplo, suponiendo que el núcleo estuviese hospedado en un servidor cuya dirección web fuese *ejemplo.com* y que quisiéramos agregar una fotografía de un cierto equipo de cómputo, se tendría que hacer una petición HTTP a *ejemplo.com/attachments/create* enviando como parámetros el tipo de objeto (computadora), su identificador y el archivo con extensión *.png* o *.jpg* que se desea adjuntar.

2.3.3. Espacios

Consta de una única clase llamada apropiadamente *Space* que contiene la lógica para comunicarse con el subsistema "espacios".

Principalmente, contiene métodos para solicitar cierta información de todos los espacios mediante llamadas RPC y almacenarla en la base de datos local. Concretamente, se guarda el identificador único de cada espacio y del espacio que lo contiene.

De esta forma, toda la administración de los espacios (creación, modificación, reordenamiento, renombramiento y demás) se realiza en su propio subsistema, mientras que en éste lo único que interesa es saber que el artículo con identificador *x* está almacenado en el espacio con identificador *y*.

Este módulo define una única RPC llamada *find_spaces* que permite realizar consultas como "obtener todos los espacios donde hay computadoras" o bien "obtener el espacio donde está la computadora con identificador *x*".

2.3.4. Etiquetas

Consta de la clase *Tag* que define métodos para la creación, edición y eliminación de etiquetas, así como métodos para agregar y quitar etiquetas a artículos, espacios o adjuntos.

Las RPC de este módulo son *create*, *update*, *delete*, *add_tags*, *remove_tags* y por último *find_tags*, la cual permite realizar consultas como por ejemplo "obtener las etiquetas de los espacios cuyos identificadores son *x*, *y* y *z* ".

2.3.5. Consultas

Consta de la clase *Finder* que define métodos para realizar consultas al directorio de artículos o a las tablas de artículos en sí.

Las RPC que define este módulo son *find_forms* y *find_items*. La primera permite conocer los tipos de artículos que se pueden guardar en el sistema, mientras que la segunda responde a consultas como "obtener todas las computadoras con etiqueta *bajo consumo*".

2.3.6. Composición

Consta de la clase *RelationshipManager* que contiene la lógica para establecer una relación de composición entre dos tipos de artículos (por ejemplo, que una sola computadora pueda tener uno o más monitores).

La RPC que define este módulo es *relate_items* que permite realizar operaciones como "hacer que la computadora con identificador *x* tenga asociados los monitores con identificadores *y* y *z* ". Naturalmente, el sistema verifica que sea posible realizar la operación deseada y devuelve un mensaje de error en caso contrario.

2.3.7. Administrador de formularios

Es una pequeña interfaz que permite administrar el directorio de artículos que se pueden almacenar en el sistema.

Es la única parte de este subsistema que contiene vistas (en HTML, CSS y JavaScript) y controladores (en PHP) que si bien proveen un mecanismo claro y sencillo para manipular el núcleo, jamás serían utilizadas por el usuario final. En cambio, optamos por solicitar al cliente una descripción de las modificaciones que pretendía realizar y permitir únicamente que personal capacitado hiciera uso de este módulo (ya que siendo el corazón del sistema, podría tener graves consecuencias en caso de que se usara de manera incorrecta).

2.4. Flujo de datos

Se mantienen las mismas tres etapas para la generación del diagnóstico ambiental que en el sistema en Ruby. A continuación se describen los flujos de datos de cada fase en el nuevo sistema.

2.4.1. Preparación

Antes de poder utilizar el sistema se deben realizar las siguientes tareas:

1. Instalar y configurar el software de servidor web Apache en cada máquina que deba hospedar un subsistema.

2. Instalar y configurar el software para manejo de base de datos MySQL en las máquinas que hospedan subsistemas del *back end*.

3. Modificar los archivos de configuración de cada subsistema para que conozcan las URL de los demás y se puedan comunicar mediante RPC.

4. Copiar el código de cada subsistema al servidor web que le corresponda.

5. Los encargados del subsistema "espacios" deben cargar la jerarquía completa de los inmuebles del cliente (incluyendo los planos arquitectónicos).

Hecho esto era posible proceder con la configuración inicial del núcleo, misma que consistía en:

1. Ejecutar los métodos de la clase *Space* que realizan las RPC necesarias para obtener, procesar y almacenar de manera local los datos requeridos de los espacios.

2. Acceder al módulo "administrador de formularios" mediante un navegador web.

3. Por cada tipo de artículo que el cliente desee inventariar:

   a) Elegir la opción "Nueva forma" del menú principal.

   b) Utilizar los controles en pantalla para definir el nombre del tipo de artículo, sus atributos, los tipos de dato de cada atributo, cuáles atributos son obligatorios y opcionalmente las relaciones de composición.

   c) El sistema toma la información proporcionada para crear las tablas y clases necesarias e integrarlas a sí mismo en tiempo de ejecución.

Al término de este proceso, el núcleo ya está listo para recibir y proveer información a los subsistemas del *front end*, mismos que no requieren configuración adicional.

2.4.2. Levantamiento

Nuevamente se requirió el apoyo de la brigada de levantamiento para alimentar datos al sistema. Para ellos el proceso permaneció sin cambios salvo por la posibilidad de etiquetar cualquier artículo del inventario.

A continuación se listan los pasos que el personal de la brigada debía seguir para ingresar un artículo al sistema. Como se puede apreciar, son los mismos que para el sistema en Ruby (ya que buscamos que la transición entre sistemas fuera transparente) y por ello denotaremos con *énfasis* los detalles que cambiaron de un sistema a otro.

1. *Al ingresar a la dirección web del subsistema "inventario"* el usuario elige la opción "captura".

2. El sistema *hace un llamado RPC al subsistema "espacios"* y obtiene la lista con los de más alta jerarquía (por ejemplo, la lista de las facultades si se tratara de Cuidad Universitaria).

3. El sistema *recibe los datos en formato JSON, listos para usarse en la interfaz.*

4. *Utilizando la respuesta JSON*, el sistema dibuja en la interfaz un selector para que el usuario indique en dónde se encuentra.

5. El usuario elige alguno de los espacios. Su selección se comunica al *subsistema "espacios"* mediante AJAX.

6. El sistema obtiene la lista de los subespacios contenidos en el espacio que eligió el usuario.

7. Los pasos 3-6 se repiten tantas veces como sea necesario hasta que el usuario haya indicado correctamente su ubicación actual.

8. De manera similar a la selección del espacio, el usuario interactúa con el sistema para elegir el tipo de artículo que desea capturar (por ejemplo, primero elige "equipo eléctrico", después "equipo de cómputo" y por último "laptop") *la única diferencia es que estas consultas van al subsistema núcleo.*

9. *Utilizando directamente el JSON obtenido*, el sistema dibuja un formulario que permite capturar toda la información requerida para describir al artículo deseado, con las siguientes consideraciones:

    - Dado que cada atributo indica qué tipo de dato es, el sistema es capaz de dibujar un campo de captura adecuado. Por ejemplo, si es un atributo de tipo fecha, mostrará un selector de tipo calendario.
    - Dado que cada atributo indica si es obligatorio o no, el sistema genera código de validación en JavaScript para impedir que el usuario omita la captura de dicho atributo en caso de que sí lo sea.
    - Si la clase define una o más relaciones de composición (por ejemplo "toda computadora tiene uno o más monitores"), el sistema genera un formulario anidado para que el usuario capture los datos de ese "subartículo", así como botones para agregar más o quitar los que ya capturó.

10. El usuario llena el formulario y adicionalmente puede tomar n fotografías del artículo.

11. *El usuario puede agregar tantas etiquetas al artículo como desee. El sistema autocompletará el nombre de las etiquetas existentes y creará "al vuelo" aquellas que sean nuevas.*

12. El usuario elige la opción "guardar", lo cual ocasiona que se envíen los datos capturados mediante AJAX.

13. El sistema recorre cada uno de los campos proporcionados por el usuario y los utiliza para *realizar inserciones SQL directamente en las tablas correspondientes*.

Si en alguno de los pasos el sistema detecta un dato inválido, impedirá la creación del artículo y reportará el error al usuario para que éste tome la acción adecuada.

Al igual que el sistema en Ruby, este también provee la función de "duplicar registro", misma que es muy socorrida por los usuarios para reducir los tiempos de captura de registros similares.

2.4.3. Análisis y reporte

Con el mismo afán de no alterar la experiencia del usuario, procuramos calcar el proceso que ya seguían los usuarios en esta fase.

Una vez más, la lista de pasos es la misma y solamente se indican las diferencias por medio de *énfasis*.

1. Un especialista entra al *subsistema "inventario", elige la opción "consultas"* e indica el espacio que desea consultar. La búsqueda puede ser a cualquier nivel de la jerarquía (por ejemplo toda una entidad, todo un edificio, todo un piso).

2. A continuación, elige la categoría de artículos de los que desea obtener información.

3. Por último, el especialista indica si desea ver todos los datos en forma tabular o bien si prefiere un resumen condensado con gráficas y totales.

4. El sistema toma los parámetros de la búsqueda, verifica que sean válidos y los utiliza para llamar a los métodos de consulta del núcleo. Por simplicidad, se listan sólo los pasos esenciales de este proceso:

    - *La selección de espacio se traduce a una condición de búsqueda SQL.*
    - La categoría que eligió el usuario *se utiliza para localizar la tabla donde se encuentran almacenados los registros correspondientes.*

- Se ejecuta *una consulta directa a la tabla obtenida en el paso anterior*.
- Si el usuario eligió la vista condensada, se realiza un procesamiento adicional para calcular promedios, porcentajes y demás.

5. El sistema toma los resultados del paso anterior, los serializa en formato JSON y los entrega a la capa de interfaz para que sean presentados al usuario.

6. Si el usuario eligió la vista tabular, se habilita una opción para descargar los resultados en un archivo *csv* (proceso que se realiza en JavaScript sin tener que llamar al servidor).

Estos pasos se repiten tantas veces como el especialista requiera hasta concluir la elaboración del diagnóstico, mismo que podrá cargar al sistema en formato PDF para que pueda ser consultado por el cliente.

Capítulo 3

Comparativo de las implementaciones

El objetivo de este capítulo es colocar a los sistemas lado a lado y estudiarlos bajo la lupa de diversos criterios que suelen utilizarse al evaluar la calidad de un software.

Dicho de otro modo, mientras que capítulos anteriores se expusieron las razones, estructura y comportamiento de cada sistema, en éste se pretende dar una idea de las implicaciones que tenía adquirir y utilizar uno u otro.

Se incluyen métricas de interés tanto para el cliente (experiencia de usuario e inversión requerida) como para el equipo de desarrolladores (costos asociados a dar soporte y mantenimiento).

3.1. Rendimiento

Antes de comparar los datos recabados del desempeño observado en ambos sistemas, conviene conocer la infraestructura sobre la que operaba cada uno.

La figura en la página 34 contiene las características del hardware utilizado durante los periodos de mayor uso de cada sistema (mismos que tuvieron una separación de aproximadamente un año). Los datos que se presentan en secciones posteriores corresponden de igual manera a estos periodos de alta actividad.

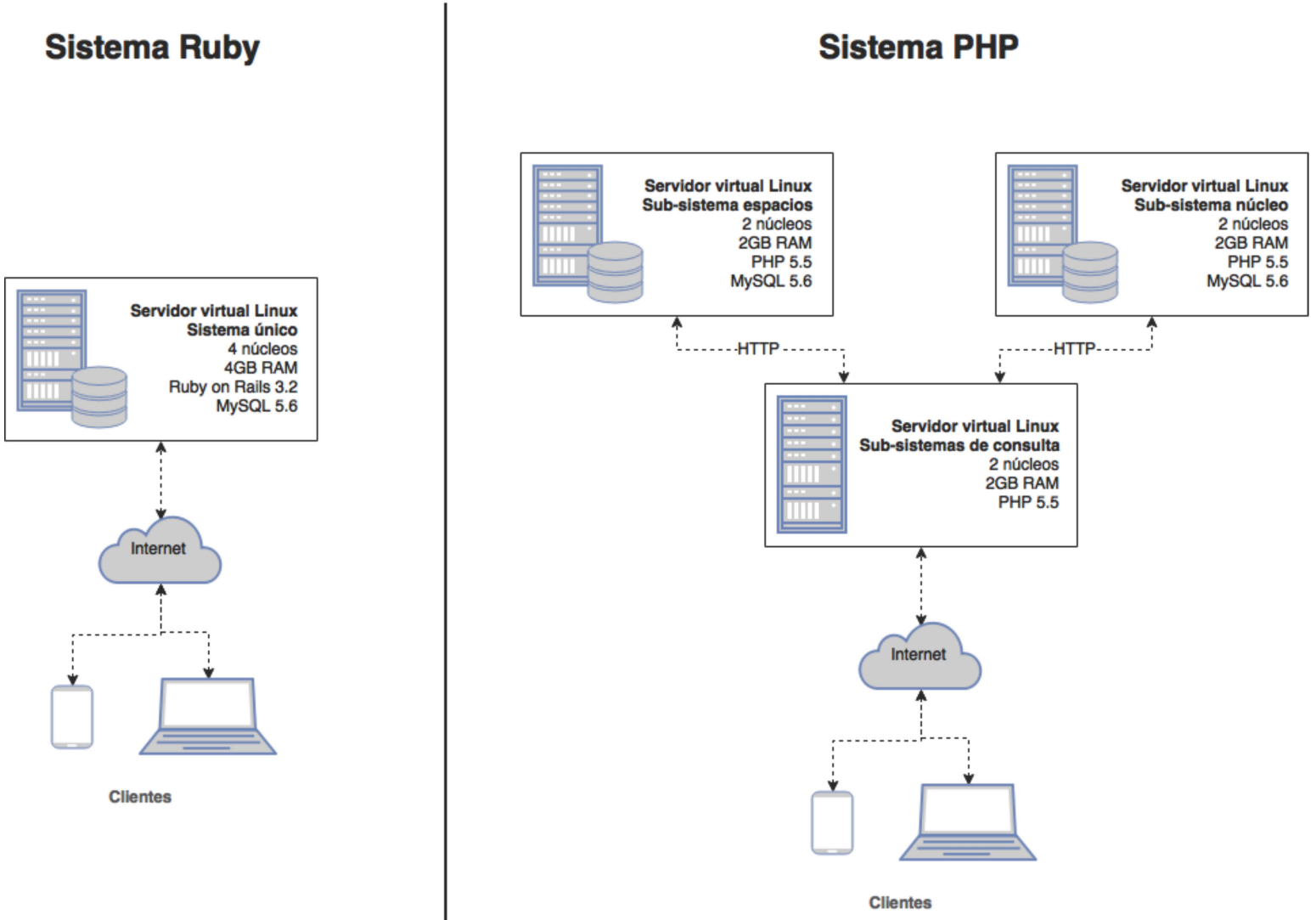

Figura 3.1: Diagrama de infraestructura de ambos sistemas.

3.1.1. Tiempo de respuesta

La tabla 3.1 presenta un conjunto de operaciones de ejemplo y el tiempo que tomaba cada sistema en procesar cada una, tomado desde que el usuario presionaba el botón correspondiente hasta que los resultados se desplegaban en pantalla y desapareciera cualquier elemento gráfico que indicara que el sistema se encontraba ocupado.

Operación	**Ruby**	**PHP**
1. Carga asíncrona de espacios	2 seg	<1 seg
2. Guardar registro	3 seg	<1 seg
3. Consultar luminarias en un cuarto	10 seg	2 seg
4. Consultar luminarias en un edificio	50 seg	3 seg
5. Consultar todas las luminarias capturadas	5 min	6 seg
6. Consultar refrigeradores en un cuarto	5 seg	<1 seg
7. Consultar refrigeradores en un edificio	15 seg	1 seg
8. Consultar todos los refrigeradores capturados	1 min	2 seg

Tabla 3.1: Tiempos de respuesta para diversas operaciones.

La operación 1 ocurría cuando un usuario se disponía a capturar un registro, para lo cual debía especificar su ubicación mediante una serie de "*combo boxes*" que le permitían navegar la jerarquía espacial del recinto donde se encontrara. Al especificar un elemento de la jerarquía, el sistema realizaba una llamada asíncrona para obtener los elementos del siguiente nivel. Por ejemplo, al elegir "edificio A" transcurría cierto tiempo antes de que aparecieran las opciónes para "planta baja", "piso 1" y "piso 2".

La operación "Guardar registro" indica el tiempo transcurrido desde que un usuario presionaba el botón "Guardar" hasta que el sistema contestaba con el mensaje apropiado de "éxito" o "error".

Las operaciones 3 a 5 se refieren a una misma consulta ("obtener luminarias") pero con distintos alcances espaciales (del más particular al más general). Naturalmente, entre más específico es el espacio, la consulta devolverá un número menor de registros.

Con las operaciones 6 a 8 se pretende evidenciar el impacto que tiene elegir un tipo de artículo mucho menos común que en el grupo anterior de operaciones (tan solo en un cuarto suelen encontrarse varias luminarias, mientras que refrigeradores habrá si acaso 2 o 3 en todo un edificio).

3.1.2. Concurrencia

Primero se describirá el soporte que provee cada lenguaje para lograr concurrencia, así como la opción que se eligió para cada sistema. Posteriormente se presentará el impacto que tuvieron estas decisiones en ambiente de producción.

Concurrencia en Ruby

Por sí solo, el lenguaje de programación Ruby define los mecanismos necesarios para desarrollar aplicaciones concurrentes (principalmente por medio de la clase *Thread* de su biblioteca estándar[2]). Sin embargo, es de vital importancia señalar que existen numerosas implementaciones de la máquina virtual de Ruby y que estas difieren en la forma en que proveen concurrencia a bajo nivel.

Dicho esto, es necesario agregar que la máquina virtual oficial de Ruby (mejor conocida como MRI[1]) no soportaba concurrencia real al momento de desarrollar el sistema, esto es, sí se puede utilizar la clase *Thread* para crear hilos, pero ya a bajo nivel estos resultan ser falsos puesto que existe un mecanismo llamado GIL[2] cuya función es asegurarse de que un solo hilo puede estarse ejecutando en todo momento [3].

Como consecuencia de esto, algunas de las implementaciones alternativas decidieron enfocarse en proveer concurrencia verdadera, dos de las más destacadas son "Rubinius" y "JRuby".

El objetivo principal de la implementación Rubinius es el alto desempeño (motivo por el cual fue considerada como primera opción); sin embargo, en esas fechas presentaba severas incompatibilidades con *Ruby on Rails* y por lo tanto fue descartada.

Por otra parte, JRuby es una implementación que provee compilación JIT[3] de Ruby a *bytecode* para la Máquina Virtual de Java. De esta forma, es posible beneficiarse de todas las optimizaciones que se han realizado a dicha máquina virtual con el paso de los años, entre ellas, ejecución de hilos a nivel sistema operativo.

[1] *Matz's Ruby Interpreter*: en honor al creador de Ruby, Yukihiro "Matz" Matsumoto.
[2] *Global Interpreter Lock*: "Candado Global del Intérprete".
[3] *Just In Time*: "Justo A Tiempo", entiéndase "al momento".

Decidida la implementación de Ruby que se utilizaría, bastaba sólo encontrar un servidor web para *Ruby on Rails* compatible con JRuby. De las diversas opciones que se probaron, la ganadora fue "Puma"[1], tanto porque tenía buenas evaluaciones dentro de la comunidad de *Rails*, como porque permitía ajustar cómodamente los números máximo y mínimo de hilos que se debían utilizar para cada ambiente (desarrollo, pruebas y producción).

Concurrencia en PHP

La implementación oficial de PHP (mejor conocida como *Zend Engine*) delega por completo la responsabilidad de concurrencia al servidor web sobre el que se ejecuta, siendo las opciones más comunes Apache y NGINX.

Siguiendo un razonamiento similar al que usamos al elegir PHP como lenguaje de programación para la versión mejorada, elegimos Apache por ser la opción más popular, con mayor soporte y costos mínimos de hospedaje web.

Ahora bien, Apache ofrece dos alternativas para concurrencia: hilos y procesos. Desafortunadamente, gran número de módulos adicionales de PHP no son *thread-safe*[4] así que no quedó otra opción más que usar la variante multiprocesos[4].

Comportamiento

Si bien ambos sistemas permitían atender varias peticiones web al mismo tiempo, se hará evidente por qué en un caso resultó ventajoso mientras que en el otro no.

Recordemos que en el caso de Ruby se utiliza un total de 9 tablas para almacenar *todo* el inventario *más* todos los espacios. Por lo tanto, si un usuario hace una consulta pesada esto impactará directamente a todos los demás (independientemente de si éstos están consultando el mismo tipo de artículos u otros completamente diferentes), ya que por cada consulta es necesario revisar varias tablas (atiborradas de registros de todos los demás tipos de artículo) y así poder ensamblar los *ZObject* correspondientes a la búsqueda.

En el caso de PHP no será así dado que cada tipo de artículo cuenta con su propia tabla, por lo que se realiza *una sola* consulta a la misma y el resultado se devuelve casi íntegro a la capa de interfaz, razón por la cual es muy poco frecuente que se den colisiones a nivel de la base de datos.

[4]Un programa se considera *thread-safe* si su lógica obliga a los hilos a utilizar recursos compartidos de manera segura.

3.1.3. Manejo de grandes volúmenes de datos

Un problema usual de cualquier sistema de almacenamiento y consulta de datos es la degradación de desempeño inherente que se observa conforme la base de datos va creciendo con el paso del tiempo.

Esto es inevitable, puesto que entre mayor es el espacio de búsqueda que hay que recorrer, más tiempo le llevará al manejador ejecutar las instrucciones necesarias para obtener sólo aquellos registros que el usuario desea.

En consecuencia, es común que los usuarios perciban la diferencia entre los tiempos de respuesta de una instalación "fresca" de un sistema contra los de una instalación que ya lleva meses (o años) en operación. Por supuesto, una característica deseable de un sistema es que esta diferencia sea soportable, inocua o incluso (en el mejor de los casos) completamente imperceptble.

Si bien tanto el sistema Ruby como el sistema PHP exhibieron este comportamiento, en el caso de Ruby este efecto es multiplicado varias veces debido a la naturaleza de su diseño.

Esto se puede ilustrar de la siguiente forma: Supongamos que se tienen dos instancias, una con el sistema Ruby y otra con el sistema PHP, y que ambas tienen exactamente los mismos registros para todos los tipos de artículos, en particular, supondremos que 500 MB son de luminarias y 400 MB son de equipo de cómputo. Adicionalmente, supongamos que se presentan los siguientes incrementos en ambos sistemas:

	Luminarias	**Equipo de Cómputo**	**Suma**
Mes 1	500 MB	40 MB	540 MB
Mes 2	400 MB	0 MB	400 MB
Mes 3	600 MB	1 GB	1.6 GB
Total	1.5 GB	1.04 GB	2.54 GB

Tabla 3.2: Crecimiento de algunos artículos durante tres meses.

Observe que al término del primer mes la cantidad de datos de luminarias se ha duplicado, mientras que para equipo de cómputo aumentó sólo en un 10 %. En un sistema donde se almacenan los datos de manera tradicional (como es el caso de nuestro sistema PHP) uno esperaría que, tras estos incrementos, se pudieran ver afectadas moderadamante la operaciones sobre la tabla Luminarias, y sólo imperceptiblemente en la tabla Equipo de cómputo.

Más aún, al concluir el segundo mes, se espera un impacto en desempeño similar en Luminarias mientras que Equipo de cómputo debería permanecer sin cambios (puesto que en ese mes no se capturó ningún registro allí).

En otras palabras, se espera que el tiempo de respuesta se incremente en un factor que dependa solamente de la tabla afectada.

Efectivamente, el sistema PHP mostró un comportamiento acorde a las predicciones. Sin embargo, en el sistema Ruby se observó una adaptación al crecimiento mucho muy deficiente: siguiendo con el ejemplo, al término del primer mes no sólo se vería afectada la operación sobre Luminarias, sino también sobre Equipo de cómputo y lo que es peor, en un factor dependiente de ambos incrementos en tamaño. Lo mismo sucede en el segundo mes, Equipo de cómputo se ve afectada cuando ni siquiera se agregaron registros de ese tipo. Para concluir, al cierre del tercer mes se observa un efecto contrario, es decir, Luminarias se ve más afectada de lo que debería.

Note que ilustramos sólo el caso de dos tipos de artículos cuando en realidad los clientes estaban manejando al menos 20 tipos distintos y con incrementos variables en almacenamiento, ya que de ciertos tipos de artículos se ingresaban varias fotos o incluso videos cortos a la base de datos, mientras que en otros los campos de texto eran suficientes para describir al artículo.

3.2. Robustez y estabilidad

En general ambos sistemas presentaron pocos errores en tiempo de ejecución, pues en ambos se agregó suficiente código para manejar correctamente situaciones imprevistas tales como desconexión súbita (tanto entre cliente y servidor web como entre servidor web y base de datos), consultas que regresan cero registros, datos mal capturados, entre otros.

Sin embargo, el servidor PHP reaccionaba de manera menos elegante puesto que, al estar programado enteramente por nosotros, cuando llegaba a ocurrir un error éste ocasionaba que Apache mostrara una pantalla completamente blanca y texto plano con una descripción muy breve del mismo, lo cual llegó a dificultar en algunas ocasiones descubrir la causa raíz del problema.

En cambio, el sistema Ruby contaba con todo el soporte de un *framework* desarrollado modularmente por numerosos contribuidores, quienes están obligados a seguir y respetar ciertos lineamientos de programación antes de que su código reciba el visto bueno y sea integrado al repositorio central. Por este motivo, en las raras ocasiones en las que ocurría un error, éste era desplegado en texto rojo en un área designada especialmente para ello en la interfaz, al

mismo tiempo que se escribía en un archivo de log con mayor grado de detalle para que fuera revisada más tarde por el equipo de desarrollo. Toda esta funcionalidad venía incluída de fábrica y requería configuración mínima para funcionar correctamente.

Naturalmente podríamos haber programado un módulo (o serie de módulos) que permitieran al sistema PHP manejar los errores de manera similar a *Ruby on Rails*, mas ésto habría consumido valioso tiempo de desarrollo en algo que casi nunca se utilizaría.

En cuanto a estabilidad, ningún sistema mostró comportamiento errático o impredecible, y las pocas ocasiones que se presentó corrupción en los datos fue en etapas tempranas de desarrollo de cada uno.

3.3. Reutilización y mantenimiento

Dado que el núcleo del sistema Ruby fue concebido para ser reutilizado en diversos proyectos de naturaleza variada, no es de sorprenderse que en este aspecto resultara superior al sistema PHP mismo que, en contraste, fue creado específicamente para el proyecto en cuestión.

Así, en el sistema Ruby fue posible agregar nueva funcionalidad haciendo uso de métodos existentes (principalmente de la capa "núcleo") sin modificación alguna en numerosas ocasiones, mientras que en PHP implicaba diseñar y programar gran cantidad de código nuevo, además de que era más tedioso integrarlo con el código existente.

Ilustraremos estas declaraciones mediante un ejemplo concreto:

Un caso de uso de prioridad secundaria, implementado en ambos sistemas una vez que se tuvo cubierta la funcionalidad base, fue el de agregar un botón para desplegar la galería de fotos de un artículo. La figura 3.2 en la página siguiente muestra el código en Ruby que se ejecuta del lado del servidor al momento de recibir esta petición.

En la primera línea del método *generate_and_view_gallery* se hace una búsqueda del *ZObject* correspondiente, usando el método de fábrica *find* de *Ruby on Rails* para obtener un objeto usando su llave primaria de la base de datos.

```ruby
class DataTableController < LoggedInController

  ...

  def generate_and_view_gallery
    zobj = (ZObject.find params[:id])
    @images_with_ids = Thing.gallery_pictures_for zobj
    render :partial => "image_gallery", locals: {:images_with_ids => @images_with_ids}
  end
```

Figura 3.2: Código Ruby que maneja la solicitud de galería de imagenes.

A continuación, se hace una llamada al método *gallery_pictures_for* (cuyo código se muestra en la figura 3.3), mismo que se encarga de obtener las URL de cada imagen así como un identificador para cada una.

Por último, se ejecuta la función *render* (también propia de *Ruby on Rails*) para indicar que se debe devolver a la interfaz el *partial*[5] llamado *image_gallery*, con la información que se extrajo anteriormente (nótese que la decoración "@" sirve para ampliar el alcance de una variable, y en este caso indica que *images_with_ids* debe persistir entre controlador y vista).

```ruby
class Thing

  ...

  def self.gallery_pictures_for zobject

    my_gallery = zobject.relationship_objects("Gallery", nil, false).first
    my_images = my_gallery.relationship_objects("Images", nil, false)

    actual_images = Array.new

    my_images.each_with_index do |gal_img, index|

      img_url = gal_img.attribute('picture', nil, true).url
      actual_images[index] = [img_url, gal_img.id]

    end

    return actual_images
  end

  ...
```

Figura 3.3: Código Ruby para extracción de galería de imágenes.

[5]Nombre técnico que le da *Ruby on Rails* a un fragmento reutilizable de código HTML.

En *gallery_pictures_for* se obtiene primero la galería asociada al artículo usando el método *relationship_objects* del núcleo, y después se extraen las imágenes de la galería usando el mismo método (observe que nos ahorramos por completo cualquier tipo de referencia a la base de datos).

Posteriormente, se recorren los *ZObject* obtenidos en un ciclo para extraer sólo la información requerida (URL e identificador) y devolverla al controlador.

```
<!-- partial: image_gallery -->
<!-- file name: _image_gallery.html.erb -->

<div class="image-set">
  <% images_with_ids.each do |img| %>
    <a href="<%= img[0] %>" data-lightbox="item_gallery" data-img-id="<%= img[1]%>" ></a>
  <% end %>
</div>
```

Figura 3.4: *partial* para formar la galería.

Finalmente, dentro del *partial* (figura 3.4) se hace un ciclo para generar una etiqueta *a* por cada imagen de la galería, colocando la URL de la imagen en el atributo *href*. Observe que la etiqueta tiene atributos no estándar de HTML (aquellos que inician con "data") mismos que son utilizados por un *plug-in* de JavaScript para generar una galería de fotos con funcionalidades adicionales tales como *zoom*, botones de "anterior" y "siguiente", entre otras.

Veamos ahora el mismo ejemplo en PHP. La figura 3.5 (en la página siguiente) muestra el contenido del archivo *attachments.php*, cuya función es atender solicitudes de archivos adjuntos asociados a artículos, procesarlas y devolver el resultado en forma de un diccionario JSON.

Tras importar los archivos *.php* correspondientes (sentencias *require_once*), fue necesario programar manualmente la detección de un error común en HTTP, a saber, que el servidor reciba una solicitud con el método incorrecto (por ejemplo, POST en vez de GET). Esto inmediatamente salta a la vista como algo que no concierne a la lógica de esta operación, algo que debería resolverse de otra forma en otra parte. Muchos *frameworks* (incluyendo *Ruby on Rails*) utilizan un canalizador para definir las parejas válidas de direcciones y métodos HTTP, y si bien pudimos haber programado uno para este proyecto, eso habría atentado contra los principios que definimos como básicos para el mismo, a saber, simplicidad y ligereza en el código.

Posterior al manejo del posible error HTTP, se hace una llamada de verificación de parámetros (para algunos se revisa que existan y para otros que no vengan vacíos). Cabe señalar que esta verificación se hacía de forma automática en *Ruby on Rails*.

```php
<?php
require_once (dirname(__FILE__) . '/../src/core/controller.php');
require_once (dirname(__FILE__) . '/../src/core/Attachment.class.php');

if($_SERVER['REQUEST_METHOD'] != "GET")
{
  http_response_code(405);
  exit();
}

$param_result = check_params(["existence" => ["structured"],
  "emptiness" => ["forms_objects", "space_ids", "tag_ids", "space_option",
  "tag_option", "attachment_type_ids", "attachment_type_option", "structured"]]);

if (!empty($param_result["errors"]))
{
  echo json_encode($param_result, JSON_UNESCAPED_SLASHES);
  exit();
}

$forms_objects = (!isset($_GET["forms_objects"])) ? [] : $_GET["forms_objects"];
$space_ids = (!isset($_GET["space_ids"])) ? [] : $_GET["space_ids"];
$tag_ids = (!isset($_GET["tag_ids"])) ? [] : $_GET["tag_ids"];
$space_option = (!isset($_GET["space_option"])) ? 1 : $_GET["space_option"];
$tag_option = (!isset($_GET["tag_option"])) ? 1 : $_GET["tag_option"];
$attachment_type_ids = (!isset($_GET["attachment_type_ids"])) ? [] : $_GET["attachment_type_ids"];
$attachment_type_option = (!isset($_GET["attachment_type_option"])) ? NULL : $_GET["attachment_type_option"];
$structured = (!isset($_GET["structured"])) ? NULL : $_GET["structured"];

echo json_encode(Attachment::hunt($forms_objects, $space_ids, $tag_ids, $space_option, $tag_option,
  $attachment_type_ids, $attachment_type_option, $structured), JSON_UNESCAPED_SLASHES);
?>
```

Figura 3.5: Parte del código para obtener la galería de imágenes en PHP.

Si se detectó algún error en parámetros, se informa a la interfaz y termina la ejecución mediante una función estándar de PHP llamada *exit*. De lo contrario, se extrae el valor de cada parámetro (observe que los nombres están "codificados en duro"[6]) y finalmente se pasan a un método de la clase *Attachment* llamado *hunt*, del cual no colocaremos su código fuente pues este consta de más de 200 líneas. Baste mencionar que dentro de su lógica se construye manualmente un enunciado *SELECT* de SQL (lo cual implica numerosas líneas adicionales de código sólo para asegurar la correcta generación de la cadena que la contiene).

La figura 3.6 (en la siguiente página) muestra la parte del archivo *forma.php* que se ocupa de generar la galería de imágenes usando la información que devolvió el controlador. Aquí sólo agregaremos que, si bien el código es similar al fragmento en Ruby que realizaba la misma tarea, en este caso se encuentra contenido dentro de un archivo mucho más grande, por lo que no se presta a ser reutilizado de forma tan inmediata como el *partial* expuesto anteriormente.

[6] *Hard-coded*: valores escritos de forma rígida, estática.

Esta falta de abstracción ocurre a lo largo de todo el proyecto, es decir, muchos archivos contienen código muy similar ya que cuando se requería agregar nueva funcionalidad la única opción práctica era copiar, pegar y adecuar manualmente a las diferencias.

```
...

<div id="gallery">

  <div class="form-group">
    <label class="col-sm-2 control-label">Galería</label>
    <div class="col-sm-10">
    <button id="delete_selected" class="btn btn-danger" disabled="disabled">Eliminar seleccionadas</button>
    <button id="add_photo" class="btn">Agregar foto</button>
    <br/>
    <br/>

    <?php
    foreach ($obj["gallery"] as $att_id => $att_hash)
    {
      ?>
      <div class="gallery_item" style="background-image: url('<?php echo $base_addr . $att_hash["path"] ?>')">
        <input type="checkbox" class="gallery_checkbox" data-id="<?php echo $att_id ?>">
      </div>
      <?php
    }

    ?>
    </div>
  </div>
</div>

...
```

Figura 3.6: Fragmento de código HTML para formar la galería en PHP.

Usando argumentos similares, es fácil ver que dar mantenimiento al sistema Ruby era mucho más sencillo que a su contraparte PHP por lo que, si bien ambos sistemas generaban una cantidad similar de incidentes al mes, los de Ruby eran identificados, corregidos, probados y liberados en mucho menor tiempo y con menor grado de dificultad.

3.4. Costo de operación

La tabla 3.3 (en la siguiente página) muestra los costos aproximados mensuales de cada plataforma (tanto el hospedaje como el salario promedio de un desarrollador) durante el periodo de mayor uso de cada sistema. Tome en cuenta que probablemente estos valores hayan cambiado desde entonces.

Al ser PHP la plataforma web más utilizada a nivel mundial durante más tiempo que ninguna otra, los proveedores de hospedaje han invertido la mayor cantidad de recursos en ella para agilizar, abaratar y simplificar su costo de implementación y mantenimiento.

Concepto	Ruby	PHP
Hospedaje	$1,000 MXN	$100 MXN
Desarrollador	$20,000 MXN	$10,000 MXN

Tabla 3.3: Costos operativos mensuales (aproximados) 2013 - 2014.

Por esto mismo, la modalidad más popular para contratar hospedaje PHP es mediante el uso compartido de servidores web Apache y manejadores MySQL junto con otros sitios y aplicaciones web, esto es, no se tiene acceso al sistema operativo sino que toda la administración y configuración se realiza mediante una consola web restrictiva, provista por el mismo servicio de hospedaje. Así, uno solo debe cargar todos los archivos que conforman la aplicación web, así como crear y administrar la base de datos, todo mediante distintos módulos de la consola web.

Por otra parte, dado que *Ruby on Rails* es mucho más joven en el mercado, sólo unos cuantos proveedores de hospedaje han intentado algo parecido al hospedaje compartido que hay para PHP. Desafortunadamente, estas plataformas son tanto restrictivas (en el sentido de que pemiten utilizar sólo ciertas versiones de Ruby, *Ruby on Rails* y sus paquetes) como poco confiables (si existe algún error, el personal de mantenimiento puede tardar semanas o meses en resolverlo, dependiendo de muchos factores fuera de nuestro control).

Por ello, la opción más segura para hospedar una aplicación hecha con *Ruby on Rails* es rentar un servidor virtual con acceso *root*[7] y realizar una configuración manual, a la medida. Debido a los recursos y riesgos que implica, este tipo de hospedaje ha sido (y seguirá siendo) una opción mucho más costosa.

El costo de los desarrolladores obedece similarmente a cuestiones de oferta y demanda. Conviene agregar que nadie respondió a nuestra oferta para el proyecto en Ruby durante los meses que estuvo publicada, mientras que para PHP pudimos contratar ayuda en cuestión de días.

La misma escasez de recursos en Ruby nos llevó a intentar la contratación y capacitación en sitio de personas no familiarizadas con el lenguaje, y si bien estas personas pudieron aprender los fundamentos de *Ruby on Rails* en un tiempo relativamente corto, la complejidad del resto de la solución (sobre todo los aspectos de metaprogramación) fue lo que finalmente los orilló a desertar.

[7]También llamado "súper usuario", es una cuenta con todos los permisos a nivel Sistema Operativo.

Capítulo 4

Conclusiones

En sí no fue una mala decisión haber elegido *Ruby on Rails* y diseñar el sistema de la forma que lo hicimos. El error recae en no haber hecho una distinción clara entre un prototipo funcional y la implementación final, es decir, desde un inicio debimos considerar que el sistema Ruby vería un retiro temprano y que a mediano plazo sería necesario reemplazarlo por algo más adecuado a la realidad.

De esta forma se habría sacado el máximo provecho a las ventajas que ofreció Ruby (desarrollo y mantenimiento ágiles) al mismo tiempo que se habrían evitado los contratiempos producidos por el cambio de plataforma.

Si bien los lenguajes modernos como Ruby y plataformas de desarrollo ágil como *Ruby on Rails* reducen mucho el esfuerzo requerido para crear sistemas computacionales complejos, esta experiencia nos dejó muy claro que no es prudente dejarse persuadir tan fácilmente por estas bondades y, mucho menos, pretender que esta facilidad en el desarrollo se traducirá en un producto eficiente y/o escalable en el mundo real.

De manera complementaria, tampoco es conveniente demeritar a las herramientas con más años en el mercado por el mero hecho de que desarrollar con ellas pueda resultar tedioso y el código que se genere no sea tan elegante ni estructurado como con sus contrapartes más jóvenes.

También nos llevamos una valiosa lección sobre los peligros que tiene "sobrediseñar"[1] un sistema: en el afán de crear una plataforma de desarrollo cómoda y ampliamente reutilizable, terminamos comprometiendo la experiencia de uso del sistema de manera irreversible.

[1] *Overdesign*: modelar, abstraer y estructurar el código de manera excesivamente elaborada, al grado que resulta contraproducente.

Por el contrario, crear software simple y llano (podría decirse "poco atractivo") tiene la gran ventaja de que, al no tener que "dar tantas vueltas", se ejecuta y devuelve sus resultados de manera pronta y directa (cualidades altamente deseables por los usuarios).

Una lección más general (pero igualmente importante) es la de haber errado con una primera versión del sistema, sólo para haber tenido más tarde la oportunidad de estudiar los acontecimientos y detectar fallos a lo largo de nuestro proceso de toma de decisiones.

Muchas veces uno ni siquiera concibe que sea capaz de tales descuidos hasta que los comete y es testigo de las consecuencias. Nótese que tampoco estamos condenando este comportamiento, después de todo, fue gracias a nuestro tropiezo inicial que crecimos como desarrolladores y que fabricamos una solución mucho más útil para nuestros clientes (experiencia que llevaremos con nosotros durante el resto de nuestra vida profesional).

Cabe aclarar que esto aplica a todos los niveles, desde el individual, pasando por la pequeña y mediana empresa hasta los grandes equipos de programación, es decir, ¿cuántas veces no se ha oído sobre plataformas comerciales de software que son desechadas total o parcialmente en favor de soluciones más adecuadas? Incluso podría argumentarse que esto es una propiedad inherente del software, estar siempre en constante evolución y mejora para acercarse mucho (mas nunca alcanzar) la perfección.

Para finalizar, se ofrece una breve lista de consejos y lineamientos cuyo propósito es orientar al desarrollador al momento de decidir sobre los aspectos explorados a lo largo de este trabajo.

- Hacer el análisis y diseño no sólo de las funcionalidades que debe ofrecer la solución (entradas, salidas, transformaciones de datos y demás), sino también dedicar tiempo a dimensionar las cargas de trabajo mínima y máxima que deberá soportar cuando se ponga en manos de los usuarios.

- Elegir las herramientas de desarrollo no sólo basándose en precios, tendencias o facilidad de uso, sino también estudiar cuidadosamente la forma en que cada una opera a bajo nivel (si es interpretado o compilado, si utiliza procesos o hilos, si soporta ejecución distribuida o alguna variante de éstas) y evaluar si estas características lograrán cumplir los objetivos de desempeño identificados anteriormente.

- No temer el desarrollo y puesta en marcha de soluciones simples, que resuelvan el problema de la manera más sencilla, y migrar hacia otra solución (posiblemente desarrollada con otra herramienta) sólo una vez que se tenga un panorama más claro sobre el entorno real de operación y que se hayan identificado las mejores oportunidades para aplicar patrones de diseño u otras técnicas de programación como reflexión, llamadas remotas de procesos, cachés y demás.

- Si a pesar de ésto uno termina con una plataforma de software con características indeseables, tener la madurez para admitir errores previos y comunicárselos al equipo y a los clientes para que, en conjunto, se haga un replanteamiento que acerque la solución a un estado óptimo para todos los involucrados.

En general los integrantes del equipo de desarrollo estuvimos de acuerdo que este proyecto representó una muy valiosa experiencia para nuestro desarrollo profesional.

Anexo 1:
Breve introducción al lenguaje Ruby

Se incluye este anexo para presentar de manera general la sintaxis y estructura del lenguaje de programación Ruby por medio de un ejemplo.

Siguiendo las convenciones de Ruby, el código que sigue debe estar contenido en un archivo llamado *ClaseHija.rb* y el ambiente de Ruby donde se utilice debe tener instaladas *gems* que definan *modulo1* y *modulo2*.

Tome en cuenta que las líneas que inician con # son comentarios y que, al ser un lenguaje tipificado de manera dinámica, ninguna declaración de variables indica de qué tipo son.

```
# Para importar todas las clases de "modulo1"
require 'modulo1'

# Para importar sólo la clase "AlgunaClase" de "modulo2"
require 'modulo2/AlgunaClase'

# Indica que el código que sigue forma parte de "UnModulo"
module UnModulo

  # Inicia la declaración de "ClaseHija", que hereda de
  # "ClasePadre"
  class ClaseHija < ClasePadre

    # Indica que "ClaseHija" tiene 3 atributos y genera
    # automáticamente getters y setters para ellos
    attr_accessor :attr1, :attr2, :attr3

    # Constructor que recibe 3 parámetros y los utiliza para
```

```
# crear un ejemplar de la clase. Observe que sólo se
# indican los nombres de los parámetros sin especificar
# su tipo
def initialize(param1, param2, param3)
  self.attr1 = param1
  self.attr2 = param2
  self.attr1 = param3
end

# Define un método estático que recibe dos parámetros y
# devuelve el resultado de aplicarles el operador +, cuyo
# significado se sobrecarga según el tipo de param1 y param2
def self.metodo_estatico(param1, param2)
  return param1 + param2
end

# Define un método de ejemplar que recibe un parámetro
# opcional (cuyo valor por defecto es 1 en caso de que el
# usuario decida omitirlo), lo compara contra un atributo
# del ejemplar y devolverá alguno de los otros atributos
# según el resultado de la comparación
def metodo_de_ejemplar(param1=1)
  if param1 > self.attr1
    return self.attr2
  else
    return self.attr3
  end
end

# Es costumbre en Ruby colocar un signo de admiración
# al final del nombre de los métodos mutadores, ya sea que
# modifiquen alguno de los parámetros que recibe o alguno
# de los atributos del ejemplar que ejecuta el método
def realiza_modificacion!(param)
  self.attr1 = param[0]
  param[0] = 'a'
end

# De manera similar, se acostumbra colocar un signo de
# interrogación cuando el método devuelve un booleano
```

```
    def es_igual?(param)
      if param == self.attr1
        return true
      else
        return false
      end
    end

    # El siguiente "if" declara el punto de entrada del
    # programa, similar al "main" de lenguajes como Java o C
    if __FILE__ == $0

      ej1 = ClaseHija.new('hola', 'a', 'todos')
      ej2 = ClaseHija.new('soy', 'un', 'ejemplo')

      # "puts" es una función para imprimir en la consola
      puts ej1.attr1()

      # uso del método estático
      puts ClaseHija.metodo_estatico(3, 6)

      # uso de los métodos de ejemplar
      if ej1.es_igual?('hola')
        ej1.realiza_modificacion!('taza')
        puts "modificación realizada"
      else
        puts ej2.metodo_de_ejemplar(3)
      end
    end
  end
end
```

Anexo 2:
Referencia del núcleo del sistema en Ruby

Como parte del proyecto desarrollamos 9 clases que nos permiten, usando "reflection", añadir clases, atributos y métodos dinámicamente para ajustar el sistema general a los requisitos particulares de un tipo de usuario.

A continuación se listan los componentes principales de cada clase.

ZClass

Modela una clase (en el contexto de orientación a objetos). Su tabla correspondiente en la base de datos es *z_classes*.

Atributos

- *name*

  Cadena que almacena el nombre de la *ZClass*. Es obligatorio proporcionarlo y debe ser único.

- *abstract*

  Es *true* si la *ZClass* es abstracta (y por lo tanto no se pueden crear *ZObject* que la implementen), *false* en otro caso.

- *system*

  Es *true* si la *ZClass* es de sistema (definida por los desarrolladores, no por el usuario), *false* en otro caso.

Métodos estáticos

- *self.find_z_class(search)*

  Devuelve una *ZClass* válida (o *nil*) dependiendo del parámetro de búsqueda *search*.

Métodos de ejemplar (relativos a herencia)

- *add_parent!(another_zclass)*

  Crea y devuelve una *ZInheritance* que define la relación de herencia directa entre *another_zclass* y esta *ZClass*, donde *another_zclass* jugará el papel de padre. Si las *ZClass* ya estaban relacionadas, este método no realiza ningún cambio y devuelve *nil*.

- *add_child!(another_zclass)*

  Crea y devuelve una *ZInheritance* que define la relación de herencia directa entre *another_zclass* y esta *ZClass*, donde *another_zclass* jugará el papel de hijo. Si las *ZClass* ya estaban relacionadas, este método no realiza ningún cambio y devuelve *nil*.

- *remove_parent!(another_zclass)*

  Elimina la *ZInheritance* que identifica a *another_zclass* como padre de esta *ZClass*.

- *remove_child!(another_zclass)*

  Elimina la *ZInheritance* que identifica a esta *ZClass* como el padre de *another_zclass*.

- *out_of_my_life!(another_zclass)*

  Elimina cualquier *ZInheritance* que defina una relación de herencia directa entre esta *ZClass* y *another_zclass*.

- *direct_child_inheritances()*

  Devuelve una lista con las *ZInheritance* donde esta *ZClass* juega el papel de hijo.

- *direct_parent_inheritances()*

  Devuelve una lista con las *ZInheritance* donde esta *ZClass* juega el papel de padre.

- *direct_descendants(simplified)*

  Devuelve una lista con las *ZClass* que heredan directamente de esta *ZClass*. Si *simplified* es *true* devuelve el resultado de forma serializada.

- *direct_ancestors(simplified)*

  Devuelve una lista con las *ZClass* de las cuales esta *ZClass* hereda directamente. Si *simplified* es *true* devuelve el resultado de forma serializada.

- *all_descendants(simplified)*

  Devuelve una lista con todas las *ZClass* que heredan directa o indirectamente de esta *ZClass*. Si *simplified* es *true* devuelve el resultado de forma serializada.

- *all_ancestors(simplified)*

  Devuelve una lista con todas las *ZClass* de las cuales esta *ZClass* hereda directa o indirectamente. Si *simplified* es *true* devuelve el resultado de forma serializada.

- *common_ancestors_with(another_zclass, simplified)*

  Devuelve la lista de ancestros comunes (directos o indirectos) entre esta *ZClass* y *another_zclass* (puede ser una lista vacía). Si *simplified* es *true* devuelve el resultado de forma serializada.

- *is_direct_ancestor_of?(another_zclass)*

  Devuelve *true* si *another_zclass* hereda directamente de esta *ZClass*, *false* en otro caso.

- *is_direct_descendant_of?(another_zclass)*

  Devuelve *true* si esta *ZClass* hereda directamente de *another_zclass*, *false* en otro caso.

- *is_direct_ancestor_or_descendant_of?(another_zclass)*

  Devuelve *true* si esta *ZClass* y *another_zclass* tienen una relación de herencia directa, ya sea que esta *ZClass* sea el padre o el hijo de *another_zclass*.

- *is_ancestor_of?(another_zclass)*

  Devuelve *true* si *another_zclass* hereda directa o indirectamente de esta *ZClass*, *false* en otro caso.

- *is_descendant_of?(another_zclass)*

  Devuelve *true* si esta *ZClass* hereda directa o indirectamente de *another_zclass*, *false* en otro caso.

- *is_ancestor_or_descendant_of?(another_zclass)*

  Devuelve *true* si esta *ZClass* y *another_zclass* tienen una relación de herencia directa o indirecta, ya sea que esta *ZClass* sea el ancestro o el descendiente de *another_zclass*.

- *is_cousin_of?(another_zclass)*

  Devuelve *true* si esta *ZClass* y *another_zclass* tienen algún ancestro (directo o indirecto) en común, *false* en otro caso.

- *is_relative_of?(another_zclass)*

  Devuelve *true* si esta *ZClass* y *another_zclass* tienen una relación de herencia o bien comparten algún ancestro (directa o indirectamente).

Métodos de ejemplar (relativos a atributos)

- *attribute!(params)*

  Crea un nuevo *ZClassAttribute* y lo asocia directamente a esta *ZClass*. *params* es un diccionario con la información necesaria para crear un *ZClassAttribute*. Si ya existe un *ZClassAttribute* con el mismo nombre, este método intentará modificar el ya existente.

- *remove_attribute!(attr)*

  Elimina al *ZClassAttribute* llamado *attr* de esta *ZClass*.

- *own_attributes(simplified)*

  Devuelve la lista de *ZClassAttribute* que están definidos directamente en esta *ZClass*. Si *simplified* es *true* devuelve el resultado de forma serializada.

- *own_attribute_named(search)*

  Devuelve el *ZClassAttribute* llamado *search* definido directamente en esta *ZClass* (*nil* si no hay tal).

- *all_attributes(simplified)*

  Devuelve la lista de todos los *ZClassAttribute* que esta *ZClass* hereda más los propios. Si *simplified* es *true* devuelve el resultado de forma serializada.

- *attribute_named(search)*

  Devuelve el *ZClassAttribute* llamado *search* definido en esta *ZClass* o en alguna de las *ZClass* de las que hereda (*nil* si no hay tal).

Métodos de ejemplar (relativos a composición)

- *relationship!(params)*

  Crea y devuelve una nueva *ZRelationship* en la cual esta *ZClass* juega el papel de propietario. *params* es un diccionario con la información necesaria para definir a la *ZRelationship*. Si el nombre proporcionado ya existe, este método intentará modificar la *ZRelationship* existente.

- *remove_relationship!(params)*

  Elimina las *ZRelationship* que coincidan con los parámetros de búsqueda *params*.

- *from_relationships(simplified)*

  Devuelve una lista con todos las *ZRelationship* donde esta *ZClass* juega el papel de propietario. Si *simplified* es *true* devuelve el resultado de forma serializada.

- *to_relationships(simplified)*

  Devuelve una lista con todos las *ZRelationship* donde esta *ZClass* juega el papel de propiedad. Si *simplified* es *true* devuelve el resultado de forma serializada.

- *all_relationships(simplified)*

  Devuelve una lista con todos las *ZRelationship* donde aparece esta *ZClass*, sin importar si es propietario o propiedad. Si *simplified* es *true* devuelve el resultado de forma serializada.

- *from_relationships_to(another_zclass, simplified)*

  Devuelve una lista con todas las *ZRelationship* donde esta *ZClass* juega el papel de propietario y *another_zclass* el de propiedad. Si *simplified* es *true* devuelve el resultado de forma serializada.

- *to_relationships_to(another_zclass, simplified)*

  Devuelve una lista con todas las *ZRelationship* donde *another_zclass* juega el papel de propietario y esta *ZClass* el de propiedad. Si *simplified* es *true* devuelve el resultado de forma serializada.

- *relationships_to(another_zclass, simplified)*

  Devuelve una lista con todas las *ZRelationship* que relacionan a esta *ZClass* y a *another_zclass*, sin importar quien sea el propietario. Cuando *simplified* es *true* devuelve el resultado de forma serializada.

- *from_relationship_named(search)*

  Devuelve la *ZRelationship* llamada *search* en donde esta *ZClass* juega el papel de propietario (*nil* si no hay tal).

- *to_relationship_named(search)*

  Devuelve la *ZRelationship* llamada *search* en donde esta *ZClass* juega el papel de propiedad (*nil* si no hay tal).

- *relationship_named(search)*

  Devuelve la *ZRelationship* llamada *search* en donde aparece esta *ZClass*, sin importar si es propietario o propiedad (*nil* si no hay tal).

Métodos de ejemplar (relativos a materialización)

- *spawn_object()*

  Crea y devuelve un nuevo *ZObject* que es ejemplar unicamente de esta *ZClass*.

- *all_direct_objects(simplified)*

  Devuelve una lista con todos los *ZObject* que son ejemplares directos de esta *ZClass*. Si *simplified* es *true* devuelve el resultado de forma serializada.

- *all_objects(simplified)*

  Devuelve una lista con todos los *ZObject* que son ejemplar de esta *ZClass* o de alguna de las *ZClass* que heredan de ella (directa o indirectamente). Si *simplified* es *true* devuelve el resultado de forma serializada.

- *object_implements_directly?(obj)*

  Devuelve *true* si *obj* es ejemplar directo de esta *ZClass*, *false* en otro caso.

- *object_implements?(obj)*

  Devuelve *true* si *obj* es ejemplar de esta *ZClass* o de alguna de las *ZClass* que heredan de ella, *false* en otro caso.

ZDataType

Modela un tipo de dato (ejemplos: entero, texto, fecha, archivo). Su tabla correspondiente en la base de datos es *z_data_types*.

Todo *ZClassAttribute* es de un único *ZDataType* en un momento dado.

Atributos

- *name*

  El nombre del tipo de dato. Es obligatorio proporcionarlo y debe ser único.

Métodos estáticos

- *self.find_z_data_type(search)*

  Devuelve un *ZDataType* válido (o *nil*) de acuerdo al parámetro de búsqueda *search*.

ZClassAttribute

Modela un atributo de una clase. Una *ZClass* puede tener n atributos (o ninguno). Su tabla correspondiente en la base de datos es *z_class_attributes*.

Atributos

- *z_class_id*

  La llave primaria de la *ZClass* a la que pertenece este *ZClassAttribute*. Es obligatorio proporcionarla.

- *z_data_type_id*

  La llave primaria del *ZDataType* que define el tipo de dato actual de este *ZClassAttribute*. Es obligatorio proporcionarla.

- *name*

  Cadena que indica el nombre del atributo. Es obligatorio proporcionarlo y debe ser único.

- *string_default*

  Valor por defecto del atributo (cuando es de tipo cadena).

- *numeric_default*

  Valor por defecto del atributo (cuando es de tipo numérico).

- *timestamp_default*

  Valor por defecto del atributo (cuando es de tipo *timestamp*).

- *attachment_default*

  Valor por defecto del atributo (cuando es de tipo archivo).

- *static*

  Es *true* si el *ZClassAttribute* es estático (es decir, los *ZObject* no pueden redefinirlo), *false* en otro caso.

Métodos de ejemplar

- *default_value()*

  Dependiendo del *ZDataType* de este *ZClassAttribute*, decide si devolver *numeric_default*, *timestamp_default*, *attachment_default* o *string_default*

- *not_duplicate()*

  Este método se ejecuta cada vez que se quiere crear un ejemplar de *ZClassAttribute*. Valida que la *ZClass* no contenga ya un *ZClassAttribute* con el mismo nombre que éste. Si la validación falla, el *ZClassAttribute* no se crea y se imprime un mensaje de error.

- *attachment_info()*

  Si este *ZClassAttribute* es de tipo archivo, devuelve una cadena con el nombre del archivo, su tamaño y su extensión. De lo contrario devuelve la cadena vacía.

ZInheritance

Modela una relación de herencia directa entre dos *ZClass*. Su tabla correspondiente en la base de datos es *z_inheritances*.

Atributos

- *parent_id*

  La llave primaria de la clase padre en la tabla *z_classes*. Es obligatorio proporcionarla.

- *child_id*

  La llave primaria de la clase hija en la tabla *z_classes*. Es obligatorio proporcionarla.

Métodos de ejemplar

- *not_duplicate()*

  Este método se ejecuta cada vez que se quiere crear un nuevo ejemplar de *ZInheritance*. Valida que no exista ya otra *ZInheritance* que defina la misma relación de herencia entre dos *ZClass*. Si la validación falla, la *ZInheritance* no se crea y se imprime un mensaje de error.

- *no_incest_please()*

  Este método se ejecuta cada vez que se quiere crear un nuevo ejemplar de *ZInheritance*. Valida que no exista ya una relación de herencia (directa o indirecta) entre las *ZClass* que está intentando relacionar esta *ZInheritance* (de lo contrario podrían definirse ciclos con las relaciones de herencia). Si la validación falla, la *ZInheritance* no se crea y se imprime un mensaje de error.

ZRelationship

Modela una relación de composición entre dos *ZClass*. Por ejemplo, supongamos que se tienen dos *ZClass*, "automóvil" y "llanta", sería adecuado definir dos *ZRelationship*: una que indique que un automóvil debe tener exactamente 4 "llantas activas" y otra que indique que un automóvil puede o no tener una "llanta de repuesto". Note que en cada *ZRelationship* se debe indicar qué hacer cuando el automóvil es modificado o eliminado, de lo contrario puede ser que la aplicación no se comporte como el usuario desee (para modelar este comportamiento existe la clase *ZRuleType*). Su tabla correspondiente en la base de datos es *z_relationships*.

Atributos

- *from_class_id*

  La llave primaria de la clase propietaria (en nuestro ejemplo, "automóvil") en la tabla *z_classes*. Es obligatorio proporcionarla.

- *to_class_id*

  La llave primaria de la clase propiedad (en nuestro ejemplo, "llanta") en la tabla *z_classes*. Es obligatorio proporcionarla.

- *from_name*

  El nombre de la relación desde el punto de vista de la clase propietaria (en nuestro ejemplo, "llantas activas"). Es obligatorio proporcionarlo.

- *to_name*

  El nombre de la relación desde el punto de vista de la clase propiedad (en nuestro ejemplo, "automóvil activo"). Es obligatorio proporcionarlo.

- *from_cardinality*

  El número máximo de objetos de la clase propietaria (en nuestro ejemplo, uno). Es obligatorio proporcionarlo.

- *to_cardinality*

  El número máximo de objetos de la clase propiedad (en nuestro ejemplo, cuatro). Es obligatorio proporcionarlo.

- *from_update_rule_id*

  Llave primaria de una regla de la tabla *z_rule_types* que indica la acción que debe tomarse cuando se modifique el objeto propietario. Es obligatorio proporcionarla.

- *to_update_rule_id*

  Llave primaria de una regla de la tabla *z_rule_types* que indica la acción que debe tomarse cuando se modifique el objeto propiedad. Es obligatorio proporcionarla.

- *from_delete_rule_id*

  Llave primaria de una regla de la tabla *z_rule_types* que indica la acción que debe tomarse cuando se elimine el objeto propietario. Es obligatorio proporcionarla.

- *to_delete_rule_id*

  Llave primaria de una regla de la tabla *z_rule_types* que indica la acción que debe tomarse cuando se elimine el objeto propiedad. Es obligatorio proporcionarla.

Métodos de ejemplar

- *readable_description()*

  Devuelve una cadena con una descripción fácil de entender de la relación.

- *not_duplicate()*

  Este método se ejecuta cada vez que se quiere crear un ejemplar de *ZRelationship*. Valida que no exista ya otra *ZRelationship* que relacione a las mismas clases con los mismos nombres de propietario y propiedad. Si la validación falla, la *ZRelationship* no se crea y se imprime un mensaje de error.

ZRuleType

Modela una regla de propagación de cambios para las *ZRelationship*. Su tabla correspondiente en la base de datos es *z_rule_types*.

Atributos

- *name*

  Cadena que indica el nombre de la regla. Es obligatorio proporcionarlo y debe ser único.

Métodos estáticos

- *self.get_z_rule_type(candidate)*

  Si *candidate* es un número, buscará una *ZRuleType* con esa llave primaria y si es una cadena, buscará una *ZRuleType* con ese nombre. Si la búsqueda fue exitosa, devuelve la *ZRuleType* encontrada, de lo contrario devuelve *nil*.

ZObject

Modela un ejemplar de una clase. Su tabla correspondiente en la base de datos es *z_objects*.

Atributos

- *z_class_ids*

  Lista con las llaves primarias de todas las *ZClass* de las que este *ZObject* es ejemplar. Es obligatorio proporcionarla.

Métodos estáticos

- *self.find_by_attribute_value(attr, value, z_class_list, deep_search)*

  Devuelve una lista de *ZObject* cuyo atributo *attr* tiene el valor *value*.

  z_class_list es una lista de *ZClass* y es opcional proporcionarla, mientras que *deep_search* es un booleano.

  Si se proporciona *z_class_list* y *deep_search* es *false*, el método devolverá *ZObject* que sean ejemplares unicamente de las *ZClass* listadas, pero si es *true* el método devolverá *ZObject* que sean ejemplar de las *ZClass* listadas o de cualquier *ZClass* que sea descendiente de ellas.

- *self.object_with_classes(zclass_list)*

  Devuelve un nuevo *ZObject* que es ejemplar de todas las *ZClass* listadas en *z_class_list*, siempre y cuando estas no sean abstractas o definan relaciones de herencia inválidas.

Métodos de ejemplar (utilidades)

- *to_s()*

  Devuelve una cadena que contiene el identificador único de este *ZObject* seguido de los nombres de todas las *ZClass* de las que es ejemplar.

- *z_clone(options)*

  Devuelve un nuevo *ZObject* que es copia del *ZObject* actual. *options* es un diccionario con diversas opciones para controlar cómo se realiza la copia.

Métodos de ejemplar (relativos a materialización)

- *implements_ancestors(simplified)*

  Devuelve una lista con todas las *ZClass* de las cuales este *ZObject* es ejemplar. Si *simplified* es *true* devuelve el resultado de forma serializada.

- *all_implements_ancestors(simplified)*

  Devuelve una lista con todas las *ZClass* (y todos sus ancestros) de las cuales este *ZObject* es ejemplar. Si *simplified* es *true* devuelve el resultado de forma serializada.

- *is_member_of_class?(some_zclass)*

  Devuelve *true* si este *ZObject* es ejemplar directo de *some_zclass*, *false* en otro caso.

- *is_kind_of_class?(some_zclass)*

  Devuelve *true* si este *ZObject* es ejemplar de *some_zclass* o de alguno de sus ancestros, *false* en otro caso.

- *add_direct_implementation!(some_zclass)*

  Agrega *some_zclass* a la lista de *ZClass* de las que este *ZObject* es ejemplar, siempre y cuando *some_zclass* no sea abstracta y el objeto no sea ya ejemplar de alguno de sus descendientes.

- *remove_direct_implementation!(some_zclass)*

  Quita *some_zclass* de la lista de *ZClass* de las que este *ZObject* es ejemplar, siempre y cuando esto no ocasione que la lista quede vacía.

Métodos de ejemplar (relativos a atributos)

- *attribute_list(simplified)*

  Devuelve una lista con todos los *ZClassAttribute* definidos por todas las *ZClass* de las que este *ZObject* es ejemplar. Si *simplified* es *true* devuelve el resultado de forma serializada.

- *attribute_value_hash(simplified)*

  Devuelve una lista de parejas *(ZClassAttribute, ZClassAttributesZObject)* que indica todos los atributos de este *ZObject* y sus respectivos valores. Si *simplified* es *true* devuelve el resultado de forma serializada.

- *has_attribute_named?(name)*

  Devuelve *true* si alguna de las *ZClass* de las que este *ZObject* es ejemplar tiene un *ZClassAttribute* llamado *name*, *false* en otro caso.

- *attribute(name, value)*

  Si se omite *value*, devuelve el valor que tiene actualmente este *ZObject* para el atributo llamado *name* (o *nil* si no tiene tal).

  En caso de que se proporcione *value*, el método creará un nuevo ejemplar de *ZClassAttributesZObject* cuya función es indicar que el atributo *name* de este *ZObject* debe tener el valor *value*. Si el tipo de dato es inválido o el objeto no tiene un atributo llamado *name*, el método devolverá *nil*.

Métodos de ejemplar (relativos a composición)

- *from_relationship_list(simplified)*

  Devuelve una lista con las *ZRelationship* definidas por las *ZClass* de las que este *ZObject* es ejemplar, siempre y cuando dichas *ZClass* jueguen el papel de propietario. Si *simplified* es *true* devuelve el resultado de forma serializada.

- *to_relationship_list(simplified)*

  Devuelve una lista con las *ZRelationship* definidas por las *ZClass* de las que este *ZObject* es ejemplar, siempre y cuando dichas *ZClass* jueguen el papel de propiedad. Si *simplified* es *true* devuelve el resultado de forma serializada.

- *relationship_list(simplified)*

  Devuelve una lista con todas las *ZRelationship* definidas por las *ZClass* de las que este *ZObject* es ejemplar. Si *simplified* es *true* devuelve el resultado de forma serializada.

- *from_relationship_value_hash(simplified)*

  Devuelve una lista de parejas *(ZRelationship, ZObjectsZRelationship)* que indica todas las relaciones donde este *ZObject* es propietario y sus respectivos conjuntos de objetos. Si *simplified* es *true* devuelve el resultado de forma serializada.

- *to_relationship_value_hash(simplified)*

  Devuelve una lista de parejas *(ZRelationship, ZObjectsZRelationship)* que indica todas las relaciones donde este *ZObject* es propiedad y sus respectivos conjuntos de objetos. Si *simplified* es *true* devuelve el resultado de forma serializada.

- *relationship_value_hash(simplified)*

  Devuelve una lista de parejas *(ZRelationship, ZObjectsZRelationship)* que indica todas las relaciones de este *ZObject* y sus respectivos conjuntos de objetos. Si *simplified* es *true* devuelve el resultado de forma serializada.

- *has_relationship_named?(name)*

  Devuelve *true* si alguna de las *ZClass* de las que este *ZObject* es ejemplar define una *ZRelationship* llamada *name*.

- *relationship_objects(name, obj_set)*

  Si se omite *obj_set*, devuelve el conjunto de objetos que tiene actualmente este *ZObject* para la relación llamada *name* (o el conjunto vacío si no tiene tal).

  En caso de que se proporcione *obj_set*, el método creará un nuevo ejemplar de *ZObjectsZRelationship* que indica que la relación *name* de este *ZObject* debe tener asociado el conjunto *obj_set*. Si por alguna razón este *ZObject* tiene dos relaciones que se llamen igual, pero en una es propietario y en otra es propiedad (cosa que semánticamente no tiene sentido), el método preferirá la *ZRelationship* donde el objeto es propietario. Si los objetos no son de la clase adecuada o este *ZObject* no tiene una relación llamada *name*, el método devolverá el conjunto vacío.

ZClassAttributesZObject

Modela el valor concreto de un atributo para un objeto en particular. Su tabla correspondiente en la base de datos es *z_class_attributes_z_objects*.

Atributos

- *z_class_attribute_id*

  Llave primaria del *ZClassAttribute* del que este *ZClassAttributesZObject* es representante. Es obligatorio proporcionarla.

- *z_object_id*

  La llave primaria del *ZObject* dueño de este *ZClassAttributesZObject* . Es obligatorio proporcionarla.

- *string_value*

  Valor del atributo cuando es de tipo cadena.

- *numeric_value*

  Valor del atributo cuando es de tipo numérico.

- *timestamp_value*

  Valor del atributo cuando es de tipo *timestamp*.

- *attachment_value*

  Valor del atributo cuando es de tipo archivo.

- *is_default*

  Es *true* si el valor actual es igual al valor por defecto tal como está definido en el *ZClassAttribute* correspondiente, *false* en otro caso.

Métodos de ejemplar

- *value(val)*

  Si se omite *val*, devuelve el valor actual de este *ZClassAttributesZObject*, tomando en cuenta el tipo de dato dictado por su *ZDataType*.

  En caso de que se proporcione *val*, el método actualizará el valor almacenado y después comprobará si es igual al valor por defecto especificado en el *ZClassAttribute* correspondiente (para actualizar la bandera *is_default* adecuadamente). Si el tipo de dato es inválido, el método devolverá *nil*.

ZObjectsZRelationship

Modela una relación entre dos *ZObject* concretos, de acuerdo a las reglas definidas por alguna *ZRelationship* en particular. Su tabla correspondiente en la base de datos es *z_objects_z_relationships*.

Atributos

- *from_z_object_id*

  La llave primaria del *ZObject* propietario. Es obligatorio proporcionarla.

- *to_z_object_id*

  La llave primaria del *ZObject* propiedad. Es obligatorio proporcionarla.

- *z_relationship_id*

  La llave primaria de la *ZRelationship* que este *ZObjectsZRelationship* representa. Es obligatorio proporcionarla.

Métodos de ejemplar

- *correct_from_class()*

  Este método se ejecuta cada vez que se quiere crear un ejemplar de *ZObjectsZRelationship*.

  Valida que el objeto propietario sea ejemplar de la *ZClass* (o de alguna de sus descendientes) designada como propietaria en la *ZRelationship* correspondiente. Si la validación falla, la *ZObjectsZRelationship* no se crea y se imprime un mensaje de error.

- *correct_from_cardinality()*

  Este método se ejecuta cada vez que se quiere crear un ejemplar de *ZObjectsZRelationship*.

  Valida que la creación de esta *ZObjectsZRelationship* no rebase la cardinalidad de objetos propietarios definida en la *ZRelationship* correspondiente. Si la validación falla, la *ZObjectsZRelationship* no se crea y se imprime un mensaje de error.

- *correct_to_class()*

  Este método se ejecuta cada vez que se quiere crear un ejemplar de *ZObjectsZRelationship*.

  Valida que el objeto propiedad sea ejemplar de la *ZClass* (o de alguna de sus descendientes) designada como propiedad en la *ZRelationship* correspondiente. Si la validación falla, la *ZObjectsZRelationship* no se crea y se imprime un mensaje de error.

- *correct_to_cardinality()*

  Este método se ejecuta cada vez que se quiere crear un ejemplar de *ZObjectsZRelationship*.

Valida que la creación de esta *ZObjectsZRelationship* no rebase la cardinalidad de objetos propiedad definida en la *ZRelationship* correspondiente. Si la validación falla, la *ZObjectsZRelationship* no se crea y se imprime un mensaje de error.

Bibliografía

[1] Evan Phoenix et al. *Puma: A modern, concurrent web server for Ruby.* URL: http://puma.io.

[2] Yukihiro Matsumoto. *Thread: Ruby Standard Library Documentation.* URL: http://ruby-doc.org/stdlib-1.9.3/libdoc/thread/rdoc/.

[3] Jesse Storimer. *Nobody understands the GIL.* URL: http://www.jstorimer.com/blogs/workingwithcode/8085491-nobody-understands-the-gil.

[4] Inc. Zerigo. *Apache: multi-threaded vs multi-process (pre-forked).* URL: https://www.zerigo.com/article/apache_multi-threaded_vs_multi-process_pre-forked.