ORM

TADP - 2019 - 2C - TP Metaprogramación

Descripción del dominio

Un ORM (o Object-Relational Mapping) es una herramienta que facilita la conversión entre el modelo de un programa en objetos y la estructura de tablas en la que se persiste. Si bien estas herramientas suelen lidiar con muchos problemas propios de la arquitectura y tecnologías específicas a las que apunta, nosotros vamos a dejar de lado muchas de estas complejidades para concentrarnos en desarrollar un prototipo básico de interfaz nativa que aproveche los conceptos de metaprogramación para configurar la persistencia de nuestras abstracciones y guardar el estado de un programa para recuperarlo en el futuro.

Entrega Grupal

En esta entrega tenemos como objetivo desarrollar la lógica necesaria para implementar la funcionalidad que se describe a continuación. Además de cumplir con los objetivos descritos, es necesario hacer el mejor uso posible de las herramientas vistas en clase sin descuidar el diseño. Esto incluye:

- Evitar repetir lógica.
- Evitar generar construcciones innecesarias (mantenerlo lo más simple posible).
- Buscar un diseño robusto que pueda adaptarse a nuevos requerimientos.
- Mantener las interfaces lo más limpias posibles.
- Elegir adecuadamente dónde poner la lógica y qué abstracciones modelar, cuidando de no contaminar el scope global.
- Aprovechar las abstracciones provistas por el metamodelo de Ruby.
- Realizar un testeo integral de la aplicación cuidando también el diseño de los mismos.
- Respetar la sintaxis pedida en los requerimientos.

Código Base

Con el propósito de evitar las complejidades asociadas al manejo de bases de datos el código del trabajo práctico no deberá interactuar con una herramienta de persistencia real sino contra un Mock provisto por la cátedra. Esta interfaz está intencionadamente limitada y no puede ser extendida para implementar la funcionalidad pedida. Pueden encontrar el código a utilizar junto con una explicación asociada al final del enunciado.

1. Persistencia de Objetos sencillos

Como primer objetivo buscamos poder persistir objetos con un estado interno sencillo (es decir, aquellos cuyos atributos apuntan exclusivamente a Strings, Números o Booleanos). Para eso vamos a extender la interfaz con la que describimos nuestras clases con las siguientes operaciones:

a. has_one(tipo, descripción)

Define un atributo persistible asociando a él un tipo básico. La descripción consiste (por ahora) en un Hash con la clave '*named*' cuyo valor debe ser el Symbol correspondiente al nombre del atributo.

```
class Person
  has_one String, named: :first_name
  has_one String, named: :last_name
  has_one Numeric, named: :age
  has_one Boolean, named: :admin

  attr_accessor :some_other_non_persistible_attribute
end
```

Nota: Ruby no tiene una clase ***Boolean***, así que es necesario encontrar una forma de hacer funcionar la sintaxis…

No debe ser posible definir dos atributos persistibles con el mismo nombre. Cualquier definición posterior a la inicial debe ser pisada con la última.

```
class Grade
  has_one String, named: :value    # Hasta acá :value es un String
  has_one Numeric, named: :value   # Pero ahora es Numeric
end
```

Los atributos persistibles deben poder leerse y setearse de forma normal; no es necesario (todavía) realizar ninguna validación sobre su tipo o contenido.

```
p = Person.new
p.first_name = "raul"   # Esto funciona
p.last_name = 8         # Esto también. Por ahora…
p.last_name             # Retorna 8
```

b. save!()

Los objetos persistibles deben entender un mensaje ***save!()*** que persista su estado a disco. Cada tipo persistible debe tener su propia "tabla" (para nosotros, un archivo en disco) llamada como la clase en donde se guarde una entrada por instancia. Al salvarse por primera vez, todos los objetos adquieren automáticamente un atributo persistible ***id*** que puede usarse como clave primaria para identificar inequívocamente a un objeto.

```
p = Person.new
p.first_name = "raul"
p.last_name = "porcheto"
p.save!
p.id                        # Retorna "0fa00-f1230-0660-0021"
```

c. refresh!()

Los objetos persistibles deben entender también un mensaje ***refresh!()*** que actualice el estado del objeto en base a lo que haya guardado en la base. Tratar de refrescar un objeto que no fue persistido resulta en un error.

```
p = Person.new
p.first_name = "jose"
p.save!

p.first_name = "pepe"
p.first_name                # Retorna "pepe"

p.refresh!
p.first_name                # Retorna "jose"

Person.new.refresh!         # Falla! Este objeto no tiene id!
```

d. forget!()

Así como podemos persistirlos, también debemos poder descartar los objetos de nuestra base utilizando el mensaje ***forget!()***. Una vez olvidado, el objeto debe desaparecer del registro en disco y ya no debe tener seteado el atributo **id**.

```
p = Person.new
p.first_name = "arturo"
p.last_name = "puig"
p.save
p.id                        # Retorna "0fa00-f1230-0660-0021"
p.forget
p.id                        # Retorna nil
```

Nota: No todos los objetos necesitan implementar estas operaciones, sólo aquellos que queramos persistir. La forma de identificar a estos tipos queda a criterio de cada grupo.

2. Recuperación y Búsqueda

Por supuesto, de nada serviría guardar objetos si no pudiéramos recuperarlos. Para esto vamos a hacer que las clases de los objetos persistibles tomen el rol de **Home**, y puedan ser usadas para recuperar sus instancias persistidas.

a. all_instances()

Retorna un Array con todas las instancias de la clase persistidas a disco. Obviamente las instancias retornadas tienen que ser REALMENTE instancias de la clase y entender todos los mensajes de la misma.

```
class Point
  has_one Number, named: :x
  has_one Number, named: :y
  def add(other)
    x = self.x + other.x
    y = self.y + other.y
  end
end

p1 = Point.new
p1.x = 2
p1.y = 5
p1.save!
p2 = Point.new
p2.x = 1
p2.y = 3
p2.save!

# Si no salvamos p3 entonces no va a aparecer en la lista
p3 = Point.new
p3.x = 9
p3.y = 7

Point.all_instances       # Retorna [Point(2,5), Point(1,3)]

p4 = Point.all_instances.first
p4.add(p2)
p4.save!

Point.all_instances       # Retorna [Point(3,8), Point(1,3)]

p2.forget!

Point.all_instances       # Retorna [Point(3,8)]
```

b. search_by_<what>(valor)

La mayoría de las veces no vamos a querer obtener todas las instancias, sino realizar una búsqueda específica. Agregamos entonces soporte para una *familia de mensajes* **find_by_<what>** (dónde <what> es el nombre de algún mensaje sin argumentos que las instancias entiendan) que retorna un Array con todas las instancias que, al recibir dicho mensaje, retornan el valor buscado.

```
class Student
  has_one String, named: :full_name
  has_one Number, named: :grade

  def promoted
    self.grade > 8
  end

  def has_last_name(last_name)
    self.full_name.split(' ')[1] === last_name
  end

end

# Retorna los estudiantes con id === "5"
Student.find_by_id("5")

# Retorna los estudiantes con nombre === "tito puente"
Student.find_by_full_name("tito puente")

# Retorna los estudiantes con nota === 2
Student.find_by_grade(2)

# Retorna los estudiantes que no promocionaron
Student.find_by_promoted(false)

# Falla! No existe el mensaje porque has_last_name recibe args.
Student.find_by_has_last_name("puente")
```

3. Relaciones entre objetos

No todo son strings en la vida. Vamos a hacer los cambios necesarios a la funcionalidad presentada en los puntos anteriores para soportar dos tipos relaciones importantes de nuestros modelos: **Composición** y **Herencia.**

a. Composición con un único objeto

Extender la definición de campos persistibles para soportar que el **has_one** admita como tipo para sus campos a otros tipos persistibles. Cuando un objeto persistible se salva se debe realizar el salvado en cascada de todos sus atributos persistibles (cada uno en su tabla correspondiente).

Para permitir que un objeto sea referenciado y actualizado desde varios lugares, los atributos persistibles de tipo complejo no deben guardarse en la tabla del objeto que los referencia, sino que este tiene que guardar su **id**.

Nota: No está en el scope de este TP soportar relaciones cruzadas.

```
class Student
  has_one String, named: :full_name
  has_one Grade, named: :grade
end

class Grade
  has_one Number, named: :value
end

s = Student.new
s.full_name = "leo sbaraglia"
s.grade = Grade.new
s.grade.value = 8
s.save!                         # Salva al estudiante Y su nota

g = s.grade                     # Retorna Grade(8)

g.value = 5
g.save!

s.refresh!.grade                # Retorna Grade(5)
```

b. Composición con múltiples objetos

Similar al **has_one**, queremos agregar un mensaje **has_many** que permita definir una relación de uno a muchos objetos persistibles.

```
class Student
  has_one String, named: :full_name
  has_many Grade, named: :grades
end

class Grade
  has_one Number, named: :value
end
```

```
s = Student.new
s.full_name = "leo sbaraglia"
s.grades                        # Retorna []
s.grades.push(Grade.new)
s.grades.last.value = 8
s.grades.push(Grade.new)
s.grades.last.value = 5
s.save!                         # Salva al estudiante Y sus notas

s.refresh!.grades               # Retorna [Grade(8), Grade(5)]

g = s.grades.last
g.value = 6
g.save!

s.refresh!.grades               # Retorna [Grade(8), Grade(6)]
```

Nota 1: Siendo que hay muchas similitudes entre el **has_many** y el **has_one**, es importante que recuerden tratar de evitar la repetición de lógica.

Nota 2: Se recomienda crear una tabla para cada relación de has_many, para facilitar las relaciones de muchos a muchos. Recuerden que no nos preocupa la eficiencia.

c. Herencia entre tipos

Finalmente, buscamos asegurarnos que nuestra persistencia funcione de modo consistente con la herencia de clases y linearización de mixines. Con este fin, debemos hacer los cambios necesarios para garantizar que:

- Los objetos, al salvarse, persisten de forma plana todos los atributos persistibles que heredan o incluyen.

```
# No existe una tabla para las Personas, porque es un módulo.
module Person
  has_one String, named: :full_name
end

# Hay una tabla para los Alumnos con los campos id, nombre y nota.
class Student
  include Person
  has_one Grade, named: :grade
end

# Hay una tabla para los Ayudantes con id, nombre, nota y tipo
class AssistantProfessor < Student
  has_one String, named: :type
end
```

- Los mensajes **all_instances** y **search_by**, al ser enviados a una superclase o mixin, traen también todas las instancias de sus descendientes.

```
module Person
  has_one String, named: :full_name
end

class Student
  include Person
  has_one Grade, named: :grade
end

class AssistantProfessor < Student
  has_one String, named: :type
end

Person.all_instances      #Trae todos los Estudiantes y Ayudantes
Student.search_by_id("5") #Trae Estudiantes y Ayudantes con id "5"
Student.search_by_type("a") # Falla! No todos entienden type!
```

4. Validaciones y Defaults

Hasta ahora todas las operaciones trabajaban sobre la premisa de que el usuario las use bien y respete los tipos de los atributos. Una implementación más seria podría controlar ciertos requisitos e incluso permitir refinar las restricciones de contenido más allá de los tipos. Vamos a implementar una serie de cambios para mejorar la consistencia de la herramienta y facilitar su uso.

a. Validaciones de tipo

El control más obvio que podemos realizar consiste en asegurarnos de que los objetos no puedan persistirse con atributos seteados a un tipo inesperado. Para controlar esto agregamos un mensaje **validate!()** a nuestros objetos persistibles que lance una excepción en caso de que alguno de sus atributos persistibles no esté seteado a una instancia del tipo declarado.

En el caso de los atributos de tipo complejo, se debe además cascadear el mensaje a las instancias asociadas.

Este mensaje debe enviarse automáticamente antes de llevar a cabo cualquier salvado.

```ruby
class Student
  has_one String, named: :full_name
  has_one Grade, named: :grade
end

class Grade
  has_one Number, named: :value
end

s = Student.new
s.full_name = 5
s.save!                         # Falla! El nombre no es un String!

s.full_name = "pepe botella"
s.save!                         # Pasa: grade es nil, pero eso vale.

s.grade = Grade.new

s.save!                         # Falla! grade.value no es un Number
```

b. Validaciones de contenido

A continuación, vamos a extender las opciones que reciben **has_one** y **has_many** para soportar nuevos tipos de validaciones:

- **no_blank:** Recibe un bool y, si es true, falla si el atributo es nil o “”.
- **from:** Recibe el valor mínimo que puede tomar un Number.
- **to:** Recibe el valor máximo que puede tomar un Number.
- **validate:** Recibe un bloque y lo ejecuta en el contexto del atributo (o cada elemento de un array). Si el bloque retorna un falsy, la validación falla.

```ruby
class Student
  has_one String, named: :full_name, no_blank: true
  has_one Number, named: :age, from: 18, to: 100
  has_many Grade, named: :grades, validate: proc{ value > 2 }
end

class Grade
  has_one Number, named: :value
end

s = Student.new
s.full_name = ""
s.save!                         # Falla! El nombre está vacío!
s.full_name = "emanuel ortega"
s.age = 15
s.save!                         # Falla! La edad es menor a 18!
s.age = 22
s.grades.push(Grade.new)
s.save!                         # Falla! grade.value no es > 2!
```

c. Valores por defecto

Por último, vamos a extender las opciones de **has_one** y **has_many** una vez más para soportar un campo **default** que defina un valor por defecto para el atributo. Este valor debe aplicarse al momento de instanciar el objeto y cada vez que se trate de salvar el estado y el campo esté seteado en nil.

```
class Student
  has_one String, named: :full_name, default: "natalia natalia"
  has_one Grade, named: :grade, default: Grade.new, no_blank: true
end

class Grade
  has_one Number, named: :value
end

s = Student.new
s.full_name                    # Retorna "natalia natalia"
s.name = nil
s.save!
s.refresh!
s.full_name                    # Retorna "natalia natalia"
```

Apéndice A: Interfaz DB

Se brindará una gema creada para simular la persistencia a una base de datos. Dentro del módulo TADB se encuentra el objeto DB. Este entiende un mensaje ***table(table_name)*** que retorna una interfaz para escribir en un archivo del nombre indicado en la carpeta **/db** en la raíz del proyecto. Esta carpeta probablemente deberá borrarse y volverse a crear a medida que se ejecuten los tests, para evitar que el resultado de un test impacte al siguiente, usando el mensaje ***clear_all*** de DB, o el mensaje ***clear*** de cada tabla***.***

Cada “tabla” permite insertar un **Hash** (mapas de clave valor utilizados por Ruby que pueden crearse con el literal: **{clave_uno: valor_uno, clave_2: valor_2, ...}**) que representa una fila de la tabla, utilizando el mensaje ***insert(hash)***. Siempre que un hash se inserta se genera un **id** para la nueva entrada y se lo persiste como una fila nueva. Los hashes solo pueden tener valores primitivos al persistirse: Strings, números o booleanos.

Las tablas también permiten listar todas las entradas con el mensaje ***entries***. Éste devuelve una lista de todos los hashes persistidos en el archivo al momento de ejecutar el método.

Finalmente, las tablas también entienden el mensaje ***delete(id)***, que borra de la tabla la entrada cuyo id es el que se pasó por parámetro, y el mensaje ***clear,*** que borra todas las entradas de una tabla.

Algo a tener en cuenta es que las tablas definidas en esta interfaz son ***schema-less.*** Esto implica que no se validará al momento de persistir que todos los elementos que se estén persistiendo tengan la misma estructura, ni tendrá ninguna restricción de constraints. Es responsabilidad de los alumnos asegurarse que la data persistida sea consistente.

```
require 'tadb'

televisores = TADB::DB.table("")
...
```