PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN

IIC2513 – Tecnologías y Aplicaciones Web (II/2016)

Profesor: Raúl Montes

Interrogación 1

6 de septiembre de 2016

Ejercicio 1 (35 %): Preguntas conceptuales

Responde en forma precisa y concisa las siguientes preguntas:

1. (1.2 pto) Explica qué es el *Duck Typing* y da un ejemplo en el contexto de Ruby.
2. (1.2 pto) En *Object Oriented Programming* un objeto puede tener atributos de instancia para representar su estado en un cierto momento; cuando éstos son públicos les puedes asignar directamente valores, como por ejemplo:

   ```
   initiative.title = 'Nuevo título'
   ```

   El código anterior es válido en Ruby pero... Ruby no tiene atributos de instancia públicos. Explica por qué esa línea es válida y qué exactamente está pasando cuando se ejecuta.
3. (1.2 pto) ¿Para qué sirve un *Mixin* en Ruby? Da un pequeño ejemplo con un trozo de código.
4. (1.2 pto) En una aplicación Web se utiliza una cantidad importante de tecnologías. Si comparas una aplicación Web con otra, podrás ver que a veces se usan diferentes tecnologías para un mismo propósito. Sin embargo, hay un cierto set de tecnologías de las cuales simplemente no tienes escapatoria: si quieres hacer una aplicación Web, tendrás que utilizarlas sí o sí. Menciona el nombre y función/rol que cumplen de 4 de ellas.
5. (1.2 pto) ¿Te acuerdas de la página Web de la UC? Cuando ingresas la URL en el *browser*, éste tiene una cantidad de trabajo no menor antes de que tú puedas disfrutar de su contenido, colores e imágenes. Explica, en orden, qué trabajo realiza el browser (máximo 7 líneas) (1 pto). ¿Se te ocurre alguna manera de que esa interacción pudiese ser más eficiente? Menciona una sola idea al respecto (0.2 ptos).

Introducción para ejercicios prácticos

Por esas cosas de la vida te topas con un aviso del Departamento de Caracoles de Carrera (DCC) en que buscan desarrolladores Web para continuar el desarrollo de una novedosa aplicación que simula carreras de caracoles. De esa manera se espera que en un futuro cercano se elimine por completo el maltrato de forzar a pobres caracoles, que sólo quieren estar tranquilamente al sol, a competir en largas y tediosas carreras. Por supuesto, esto te caló hondo y no pudiste dejar de contactar al DCC para ofrecer tu ayuda.

Te explican que la simulación que se realiza es discreta, considerando *ticks* en los cuales se calcula (con aleatoriedad) el avance de cada caracol en la carrera. Cuando un caracol recorre toda la distancia que corresponde a la carrera, se considera como el ganador. Si dentro de la simulación de un *tick* dos o más caracoles llegaron a la meta, se

considera un empate entre ellos. Además, se almacenan estadísticas de cada *tick* simulado, con el fin de poder realizar análisis en el futuro.

La aplicación cuenta con los siguientes modelos `ActiveRecord` y sus atributos:

- Race (carrera)
  - `name` (string): requerido, sólo puede contener letras mayúsculas y minúsculas.
  - `max_participants` (integer): número máximo de participantes. Requerido, entero mayor que 2.
  - `distance` (float): distancia en metros que tiene la carrera. Requerido, número mayor que 0.
  - `finished` (boolean): indica si la carrera ya terminó. Default false.
  - `race_participations` (asociación 1-N)
- Snail (caracol)
  - `name` (string): requerido, de largo al menos 5 caracteres.
  - `speed` (integer): requerido, entero entre 1 y 10.
  - `race_participation` (asociación 1-N)
- SnailRaceParticipation (participación de un caracol en una carrera)
  - `race` (asociación N-1)
  - `snail` (asociación N-1)
  - `sleeping` (boolean): indica si el caracol se quedó dormido en esta carrera. Default false.
  - `distance` (float): distancia que ha recorrido el caracol en esta carrera. Requerido, default 0.
- RaceTick (un tick simulado de una carrera, sólo con propósitos de Log)
  - `race` (asociación N-1)
  - `average_distance` (float): distancia promedio que avanzaron los caracoles en este tick. Requerido, float mayor o igual a 0.

Por supuesto, además de lo anterior, todos cuentas con los atributos "default" que agrega `ActiveRecord` (`id`, `created_at`, `updated_at`).

Ejercicio 2 (30 %): A little bit of Rails for Snails

Antes de que pudieras meter mano en la aplicación, el equipo al que te querías unir quizo asegurarse de que tenías un mínimo de conocimiento sobre Rails. Así que, respecto a los modelos listados anteriormente, te piden que:

1. (2.4 ptos) Escribas migraciones (completas o parciales) que creen 3 columnas de diferente tipo. Puede ser una sola migración que crea una tabla con 3 columnas diferentes, 3 migraciones cada una creando una columna diferente de las demás, u opciones intermedias.
2. (3.6 ptos) Escribas suficientes clases `ActiveRecord`, completas o incompletas (0.6 ptos) de manera que:
   a) (1.8 ptos) definas 3 validaciones diferentes (sobre el mismo o diferente atributo, en una misma o diferente clase).
   b) (1.2 ptos) definas 2 asociaciones diferentes.

No escribas más de lo pedido; si lo haces, sólo se corregirá lo primero que aparezca en tu respuesta hasta completar lo estrictamente solicitado.

Ejercicio 3 (35 %): Almacenando la grasa en un mejor lugar

Ahora que te dejaron meterte en el código, te diste cuenta de que los desarrolladores de esta aplicación no siguieron muy buenas prácticas a la hora de escribir sus controladores. En particular, te pones a mirar la acción que crea/simula un nuevo *Tick* para una carrera, y ves que tiene una enorme cantidad de código.

Entonces se te cruza como un flechazo por la mente la frase "Skiiinyy controoooooleeeeerrrsss" y te propones arreglar ese método. Una de las maneras de lograr esto es traspasando lógica de negocio desde controladores a tus modelos. Pero ojo, que no todo el código necesariamente será apropiado para un modelo *ActiveRecord*... pero no te preocupes: ¿sabes una de las maravillas de los lenguajes OO? ¡Puedes crear tus propias clases!

Lo que actualmente hace el controlador es:

- recibe el parámetro `race_id` con el id de la carrera a la que simulará un *tick*. Puedes suponer que se encuentra en una variable de instancia del controlador llamada `@race_id`.
- carga la instancia de `Race` con id `@race_id`.
- si la carrera terminó, almacena la lista de ganadores (`SnailRaceParticipation` cuya `distance` sea mayor o igual a `race.distance`) en la variable de instancia `@winners`.
- si la carrera no ha terminado, procede a simular un *tick*:
  - carga las `SnailRaceParticipation` de la carrera cargada, pero sólo los que tienen el atributo `sleeping` con valor false (sólo caracoles despiertos).
  - itera por cada `SnailRaceParticipation` y, para cada uno:
    - calcula la distancia recorrida como una multiplicación del atributo `speed` del caracol asociado a esta `SnailRaceParticipation` con un número aleatorio entre 0 y 1 (método `rand` genera justamente eso).
    - aumenta `distance` de este objeto en esa distancia recorrida.
    - guarda los cambios del objeto en la base de datos.
  - calcula el promedio de avance de todos los caracoles y crea una nueva tupla del modelo `RaceTick` con los datos correspondientes.
  - verifica si hay caracoles que pasaron la meta.
    - Si los hay, asigna en `@winners` un arreglo con los ganadores y guarda el objeto `Race` con el atributo `finished` con valor `true`.
    - Si no, asigna en `@leader` al caracol que va ganando la carrera (mayor distancia recorrida hasta el momento).

Concretamente, tu misión es migrar este código desde el controlador hacia los modelos *ActiveRecord* y hacia una nueva clase, `RaceManager`, según consideres apropiado. Parte de la lógica puede pertenecer a un modelo *ActiveRecord* (sobre todo lo relacioando a consultas a la base de datos) mientras que el flujo principal lo debes traspasar a `RaceManager`. No es necesario que ejecutes las instrucciones descritas arriba exactamente de la manera en que están expresadas, pero sí es importante que finalmente mantengas el mismo *input* (id de carrera) y el mismo *output* (variables de instancia `@winners` o `@leader` asignadas) en el controlador.

Escribe la clase `RaceManager`, las adiciones a las clases `ActiveRecord` que estimes necesarias y el código que finalmente queda en el controlador. Si logras tu cometido, el código del controlador te debiera quedar entre unas 3 y 7 líneas simples de Ruby.