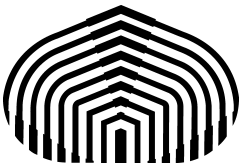

UNIVERSIDAD SIMÓN BOLÍVAR
DEPARTAMENTO DE COMPOUTACION Y TECNOLOGIAS DE LA INFORMACIÓN

CI3361 — Laboratorio de Lenguajes de Programación

Abril - Julio 2016

Asignación completa de la tercera y última etapa del laboratorio de lenguajes de progamación

---

Paradigma de Estudio: Orientación a Objetos

Esta asignación tiene como objetivo repasar e incorporar conceptos y aspectos que son relevantes en la ciencia de la computación y en particular en la programación orientada a objetos (POO). Para ello, el estudiante se estará apoyando en el lenguaje de programación *Ruby* y se le pedirá que lo use para resolver distintos problemas. En principio habrán 9 aseveraciones (teóricas) que deben argumentarse y serán imprescindibles de responder para validar la segunda parte de la asignación; los ejercicios prácticos. La evaluación práctica consta de problemas sobre el manejo y recorrido de arboles (tomen un respiro, no es tan grave), así como extensión de clases, herencia, despacho dinámico, despacho doble y el uso de *mixins*.

La asignación completa representa 25 puntos sobre el curso

1 Teoría

En esta primera sección se expondrán 9 aseveraciones que pueden ser ciertas, falsas o como han visto a lo largo del curso teórico, contraparte de este, *pueden depender*. Los paradigmas de programación, sistemas de tipos, alcances, asociaciones, así como otros aspectos, califican a un lenguaje pero esto podría, o no, limitar las capacidades del mismo. En ciertas ocasiones puede llegar a ser difuso pero la idea de esta primera parte de la asignación es que explore los posibles casos de la aseveraciones, donde podrían estar en lo correcto, donde podrían no estarlo y hacia donde se inclinaría la balanza. La idea es que el estudiante exponga su punto de vista como programador que a trabajado con varios lenguajes, ha explorado distintos paradigmas y concluya con un si, un no o *depende*.

1.1 Aseveraciones

Lea detenidamente cada una de las siguientes afirmaciones, analice, dude y responda en el archivo `teoria.md` siguiendo las especificaciones en el mismo.

1. Los lenguajes de programación orientados a objetos que poseen herencia simple están limitados a incorporar comportamientos de un solo ancestro al momento de definir una clase.

2. Lenguajes de POO con un sistemas de tipos estático (C++, Java, C#) no tienen la posibilidades de elegir la implementación de un método a tiempo de ejecución (despacho dinámico).

3. La introspección y reflexibidad son conceptos que se manejan en la POO pero no guardar ninguna relación entre sí.

4. En un lenguaje con un sistema de tipos dinámico la sobrecarga de métodos es innata y representa una comodidad dado que permite implementar un mismo método para distintos tipos.

5. En los lenguajes POO existen los términos interfaz, módulo, clase abstracta, rol, etc; definidos como objetos que pueden encapsular definiciones de clases o implementaciones concretas de métodos.

6. Los métodos virtuales permiten asociar, al momento de compilar, una implementación de un método sobrecargado con una llamada del mismo; eliminando el *overhead* del despacho dinámico.

7. Cuando un lenguaje de POO tiene herencia simple no ocurre el problema del diamante pero de igual forma pueden existir llamadas ambiguas de métodos, dado que incorporar interfaces, módulos, protocolos, etc, no evita colisión de nombres.

8. El paso de mensaje es un término que se maneja en modelos concurrentes, también de POO y es equivalente a la llamada de una función.

9. Sin importar la herencia del lenguaje de POO, una clase podría tener más de un ancestro.

1.2 Condiciones

Responder esta sección es obligatoria y necesaria para que la segunda parte, la sección práctica, sea válida (el *quiz* no tiene este requisito). Llenando la plantilla en el archivo `teoria.md` y siguiendo estas indicaciones adicionales:

- Cada argumentación, en promedio, debería caber en 2 *tweets* y no debería tener más de 420 caracteres, aunque si lo considera necesario puede excederse.
- Puede realizarse de forma individual o máximo en grupos de dos personas.
- La entrega de esta sección será el mismo día que la entrega del proyecto.
- El estudiante o grupo debe entregar las repuestas generando un *PDF* a partir de la plantilla, llamarlo `rb_XX_XXXXX_XX_XXXXX.pdf` y si el grupo es de un estudiante, el *PDF* se llamará `rb_XX_XXXXX.pdf`. En cualquier caso, `XX_XXXXX` sera sustituido por un carné.
- Esta sección tiene un valor de *5 puntos*.

2 Práctica

El objetivo de esta segunda etapa es usar conceptos que forman parte del paradigma de orientación a objetos, como *duck typing*, despacho dinámico, despacho doble y *mixins*; a consideración del estudiante. Es importante resaltar que el uso de introspección, reflexibidad, variables globales, variables de clase y variables de instancia de clase (particular de *Ruby*) no es necesario y está prohibido.

Esta evaluación consiste en 3 fases, la primera involucra definición y recorrido sobre arboles (Binario y Rosa) y para ello se deberá definir un *mixin* el cuál encapsula el recorrido BFS; suponiendo que cualquier estructura que lo incluya responderá al método `each`. En segundo lugar, se usaran las estructuras de arboles de la fase anterior junto a unas clases, llamadas **mutadores**, que alteran el valor de los nodos dentro del árbol. Por último, se pide simular un comportamiento de los lenguajes funcionales sobre las estructuras de árboles previamente definida.

2.1 Árboles

Antes que nada se deben definir las clases que representan a las estructuras abstractas que se piden y se describen a continuación:

- **Árbol binario**. Posee un atributo (variable de instancia) que guarda el valor del nodo actual (un objeto) y este debe poder solicitarse o modificarse (*get* o *set*). Además, por definición de árbol binario, la clase posee una referencia al hijo izquierdo y al hijo derecho; ambas referencias sólo pueden ser `nil` o una instancia de esta misma clase. Dichas referencias no deben poder ser modificadas posterior a la instanciación de la clase.
- **Árbol rosa**. Similar a la clase anterior, y cumpliendo las mismas condiciones, pero la única diferencia radica en la cantidad de hijos que puede poseer cada nodo de un árbol rosa; dicha cantidad no tiene límite. En lugar de tener dos referencias, existirá un arreglo de referencias para cada hijo, es decir, un arreglo de árboles rosa.

Cada clase debe definir el método `each`, el cuál tiene la finalidad de **iterar** sobre los hijos del **nodo actual** cuando los mismos estén definidos. Este método será usado

por el módulo *BFS* que definirá y se describe más adelante.

2.2 Módulo BFS

La idea de implementar un módulo es que pueda ser utilizado por varias clases, en este caso por las clases *ArbolBinario* y *ArbolRosa*, suponiendo que cada clase responde a ciertos métodos, por ahora el módulo sólo requiere del método `each` y el *getter* del valor dentro del nodo, para proveer un conjunto de clases o métodos; esta técnica se conoce como *mixins*.

Suponiendo que la clase que incluya dicho módulo posee ciertos métodos, la misma brinda métodos adicionales partiendo de esa asunción. En nuestro caso particular el módulo debe ofrecernos estos métodos:

- `bfs`. Este método recibe un bloque de manera **implícita**, itera desde *self* haciendo un recorrido *BFS* y en el camino va llamando al bloque con cada nodo de la iteración. Terminada la iteración, el árbol y toda su estructura debe permanecer igual, previo al recorrido.
- `recoger`. Este método recibe un predicado (bloque) de manera **explícita**, itera desde *self* haciendo un recorrido *BFS* y en el camino va recogiendo los nodos que cumplan con el predicado.

Recuerde, sólo deberá suponer que la clase responde al método `each` y al método para solicitar lo que contiene el nodo (*getter*). Los métodos del *mixin*, no deberán crear variables de instancia o clases adicionales.

2.3 Tipos de nodos

Dentro de cada nodo, independientemente de las clases mencionadas previamente, el objeto que puede almacenarse sólo será instancia de las clases *Fixnum*, *String* o *Array*. Estas clases serán re-definidas y extendidas a discreción del programador.

El caso particular de la clase *Array*, sus elementos serán solamente instancia de las clases *Fixnum*, *String* o *Array*.

2.4 Mutadores

Los mutadores son clases que deberán ser definidas por el programador y su función es mutar un objeto. Estas clases tienen una súper clase común llamada `Mutador` que no podrá ser instanciada, dado que por especificación de este enunciado se mantendrá abstracta.

La idea es que estos mutadores sean recibidos por una instancia de las clases *Fixnum*, *String* o *Array* y alteren la misma. Es notable que al momento de recibir un mutador, la instancia (de *Fixnum*, *String* o *Array*) no sabrá a quien recibe y no podrá ser preguntado, dado que es obvio uso de introspección.

Los posibles mutadores son *Singular, Uniforme y Oscuro*, dejando 9 posibles escenarios:

- Una instancia de *Fixnum*
  - Con un mutador **Singular** quedaría la multiplicación del ultimo dígito con la suma de los dígitos restantes.
  - Con un mutador **Uniforme** quedaría el promedio de sus dígitos.
  - Con un mutador **Oscuro** quedaría una nueva instancia sin los dígitos en posición impar.

  Si el número tiene un dígito, no hay mutación. Los decimales se redondean
- Una instancia de *String*
  - Con un mutador **Singular** quedarían los caracteres pertenecientes a ``singular'' en mayúscula. Ej: ``pluma'' → ``pLUmA''.
  - Con un mutador **Uniforme** quedarían los caracteres en mayúscula y minúscula de forma intercalada. Ej: ``dRaGON'' → ``DrAgOn''.
  - Con un mutador **Oscuro** quedaran los caracteres en posición impar concatenados del lado izquierdo y los otros concatenados del lado derecho. Ej: ``bRyu'' → ``Ruby''

- Una instancia de *Array*
  - Con un mutador **Singular** quedarían todos los elementos interpolados en un *string* separados por un espacio. Ej: `[‘‘Me’’, [‘‘he comido’’, 4], [[‘‘mangos’’]]]` → `‘‘Me he comido 4 mangos’’`
  - Con un mutador **Uniforme** mutaría los elementos del arreglo de manera uniforme.
  - Con un mutador **Oscuro** selecciona 50% de los elementos de forma aleatoria y los muta de manera oscura.

2.5 Un poco de simulación

En esta última parte se pide implementar un módulo, siguiendo las mismas especificaciones de la sección 2.2, pero que realice un recorrido en *DFS* y de igual forma ofrecer un método para realizar tal recorrido (el método se llamará `dfs`). Además, dicho modulo debe proveer un segundo método:

- `fold`. Este método recibe un valor base y un bloque de manera **implícita**, itera desde *self* haciendo un recorrido *DFS* y en el camino va llamando al bloque con cada nodo de la iteración junto al acumulador. El bloque sólo recibirá dos argumentos.

2.6 Condiciones

Esta etapa deber ser realizada por grupos de dos personas máximo (ciertas condiciones aplican). Debe entregarse junto a la evaluación teórica para que sea válida y sigue estas condiciones:

- Debe ser entregado el día 30-06-2016 antes de las 11:59 pm VET.
- Debe entregarse un único archivo .zip, .tar.gz o .tar.bz2 con el nombre `p3_XX_XXXXX_XX_XXXXX.tar.gz` o `p3_XX_XXXXX.tar.gz` si se realizo de forma individual, sustituyendo `XX_XXXXX` por un número de carné. Dentro del archivo comprimido estarán los archivos:

  - `trees.rb`. Definición de árboles.
  - `nodos.rb`. Definición de los nodos y mutadores.
  - `mod_bfs.rb`. Definición del módulo bfs.
  - `mod_fold.rb`. Definición del módulo fold.

- Se evaluará el estilo y buenas practicas de programación. Debe emplear una indentación adecuada y consistente en cualquier editor.
- Los archivos deben estar debidamente documentado.
- Su proyecto será corregido usando Ruby 2.2, usando las librerías estándar. No está permitido utilizar librerías externas ni gemas.
- Esta sección tiene un valor de *20 puntos*.

3 Temas y Tópicos de Referencia

Los temas a evaluar están presentes en toda la asignación pero en resumen se consideran estos conceptos:

- Herencia de clases
- Extensión de clases
- Introspección y Reflexibidad
- *Duck typing*
- Sobre-escritura de métodos
- Despacho dinámico y despacho doble.
- Métodos virtuales
- Sistema de tipos estático y dinámico
- Sistema de tipos explícito e implícito
- Protocolos/Interfaces/Módulos/Roles