Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

25 de octubre de 2016

IIC2113 - Diseño Detallado de Software

Interrogación 2

Instrucciones: Sea preciso: no es necesario escribir mucho pero sí ser preciso. En caso de ambigüedad, utilice su criterio y explicite los supuestos que considere convenientes. Esta interrogación fue diseñada para durar 80 minutos.

Responda cada pregunta en hojas separadas y recuerde poner su nombre en cada una.

1. (1.0 pt) Nombre 3 características de una buena métrica. Explique el objetivo principal para utilizar métricas en un proyecto de *software*.

   - Simple y computable
   - Intuitiva
   - Consistente y objetiva
   - Unidades de medición expresivas
   - Independiente del lenguaje de programación
   - Reflejar recomendaciones para mejorar

   El objetivo principal para utilizar métricas en un proyecto de *software* es obtener evidencia concreta, objetiva y empírica del estado del proyecto. De esta forma, se pueden cuantificar diferentes características del proyecto (tales como calidad, efectividad o productividad), para luego diseñar planes de acción y verificar que estos realmente mejoran los indicadores.

2. (0.9 pts) Nombre y explique 3 métricas propuestas por Chidamber & Kemerer (*CK metrics suite*).

   - **Wheighted Methods per Class (WMC):** Suma de la complejidad de cada método de una clase. Indicador predictivo del esfuerzo necesario para mantener y extender una clase.
   - **Depth of Inheritance Tree (DIT):** Distancia máxima de una clase base a una 'hoja' de la jerarquía. Refleja la complejidad del diseño.
   - **Number Of Children (NOC):** Cantidad de subclases directas de una clase. Refleja el nivel de abstracción en del diseño.
   - **Coupling Between Objects classes (CBO):** Cantidad de clases con las que colabora una clase. Refleja el nivel de acoplamiento entre objetos.
   - **Response For a Class (RFC):** Cantidad de métodos únicos invocados desde una clase. Refleja la complejidad del código.

- **Lack of Cohesion Of Methods (LCOM):** Cantidad de grupos de métodos relacionados que acceden a 1 o más atributos en común de una clase. Refleja el nivel de cohesión dentro de una clase.

3. (0.6 pts) Desarrolle 3 razones por las cuales se podría explicar que "buenos desarrolladores" generen "mal código" (*code smells*).

- Cumplir con presupuesto o plazos muy exigentes
- Se realizan cambios en el código sin revisar/actualizar el diseño
- Se extiende código *legacy* sin mejorarlo
- No se consulta opinión de pares sobre el diseño o implementación
- No se considera el código en su totalidad, solamente partes aisladas

4. (1.8 pts) Identifique *code smells* presentes en el siguiente extracto de código. Explique qué problemas generan cada uno de estos *code smells*.

```ruby
class Pokedex
  def initialize
    # Lista para guardar pokemones
    @pl = []
  end

  # Agrega nombre, ataque y defensa de un pokémon a la lista
  def add_pokemon(n, a, d)
    @pl << [n, a, d]
  end

  # Para aumentar el ataque de un pokémon
  def increase_attack(n)
    # Guarda si el pokémon fue encontrado
    f = false
    for pk in @pl
      if pk[0] == n
        pk[1] += 1
        f = true
      end
    end
    if !f
      puts n + " no encontrado"
    end
  end

```

```ruby
  # Para aumentar la defensa de un pokémon
  def increase_defense(name)
    # Guarda si el pokémon fue encontrado
    f = false
    for pk in @pl
      if pk[0] == name
        pk[2] += 1
        f = true
      end
    end
    if !f
      puts name + " no encontrado"
    end
  end

  # Para obtener los atributos de un pokémon
  def get_stats(name)
    for pk in @pl
      if pk[0] == name
        puts pk[1].to_s + "/" + pk[2].to_s
      end
    end
  end

```

```ruby
    # Se llama al invocar puts sobre una instancia
    def to_s
      i = 0
      temp = ""
      for pk in @pl
        i += 1
        temp += i.to_s + ". " + pk[0] + "\n"
      end
      temp
    end
  end

  pokedex = Pokedex.new
  pokedex.add_pokemon('Pikachu', 12, 10)
  pokedex.add_pokemon('Cubone', 8, 12)
  pokedex.get_stats('Pikachu')
  puts pokedex
```

- **Primitive Obsession**: Este *code smell* se hace presente en la forma de trabajar con las instancias de pokémones. Se puede observar principalmente en la línea 9, donde se utiliza un arreglo en vez de una clase Pokemon. Esto genera que el código sea menos extensible, menos organizado y más complejo de entender.
- **Divergent Change**: Si se cambia la estructura del arreglo que guarda los pokémones, todos los métodos de la clase Pokedex tienen que cambiar. Esto impide la extensibilidad del código, ya que es menos flexible y organizado.
- **Comments**: La mayoría de los comentarios no agregan información relevante, y podrían ser reemplazados por nombres de variables más representativos. Los comentarios dificultan la comprensión del código, haciéndolo menos intuitivo y obligando a leer más.
- **Duplicate Code**: Hay varios fragmentos de código duplicados, en especial los que iteran sobre la lista de pokémones. Esto genera que el código sea más extenso, difícil de entender y reduce la extensibilidad al dificultar cambios.
- **Inappropiate Intimacy:** La clase Pokedex no debería modificar los valores de ataque y defensa, ni imprimir los atributos de un pokemón. Este *code smell* aumenta el acoplamiento y mezcla responsabilidades.

5. (0.5 pts) Con respecto a los atributos de calidad *Maintainability*, *Flexibility* y *Testability* de una aplicación: ¿cuál es el *trade-off* que se debe considerar al momento de desarrollar un proyecto?

El principal *trade-off* que presentan estos atributos sobre el proyecto es su costo. Mantener altos estándares de mantenibilidad, flexibilidad y testeabilidad puede ser muy costoso, y puede ser que no sea necesario para el contexto del proyecto.

6. (1.2 pts) Diseñe *tests* unitarios para el siguiente código. No es necesario que implemente código, aunque puede utilizar pseudo-código si lo desea. Se pide como mínimo que describa las pruebas con una breve descripción del escenario, un contexto (configuración inicial) y un resultado esperado. Considere que se descontará puntaje por *tests* redundantes.

```ruby
class Item
  attr_reader :name, :size
  def initialize(name, size)
    @size = size
    @name = name
  end
end
```

```ruby
class Fridge
  def initialize(max_space = 10)
    @max_space = max_space
    @current_space = 0
    @items = {}
  end

  def store(item)
    item_size = item.size
    item_name = item.name
    item_amount = @items[item_name]
    if @current_space + item_size <= @max_space
      @current_space += item_size
      if item_amount
        item_amount += 1
      else
        item_amount = 1
      end
      @items[item_name] = item_amount
      puts "Se guardo un #{item_name}"
    else
      puts "No hay espacio para guardar #{item_name}"
    end
  end

```

```ruby
    def take(item)
      item_name = item.name
      item_amount = @items[item_name]
      if item_amount
        item_amount -= 1
        @current_space += item.size
        @items[item_name] = item_amount
        puts "Quedan #{item_amount} unidades de #{item_name}"
      else
        puts "No hay unidades de #{item_name}"
      end
    end
  end
```

En esta pregunta se espera como mínimo pruebas para los métodos *store* y *take*. Las pruebas debían cubrir todos los caminos independientes de estos y probar condiciones de borde. Es importante que las pruebas sean unitarias: no mezclar responsabilidad de las pruebas y aislar la dependencia entre métodos.