Nombre _____ Número de lista

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

23 de noviembre de 2018

IIC2113 – Diseño Detallado de Software

Examen

Instrucciones:

- Sea preciso: no es necesario escribir extensamente pero sí ser preciso.
- En caso de ambigüedad, utilice su criterio y explicite los supuestos que considere convenientes.
- Responda y entregue cada pregunta en hojas separadas. Si no responde una pregunta debe entregar de todas formas la hoja correspondiente a la pregunta.
- Indique su nombre y número de lista en cada hoja de respuesta.
- Esta interrogación fue diseñada para durar 140 minutos.

1. (0.6 pts) Explique a qué se refieren los principios *Single Responsibility* y *Open-Closed*. Acompañe sus explicaciones junto con un ejemplo concreto en *pseudo*-código que no cumpla con el principio. Luego, realice una versión modificada de su ejemplo que sí cumpla con el principio.

Nombre _____ Número de lista

2. (1.0 pts) Relacione los siguientes patrones de diseño con la situación descrita que más se adecue para su utilización. Cada patrón debe ir asociado solamente con un escenario.

a) Builder

b) Command

c) Singleton

d) Memento

e) Façade

f) Prototype

g) Adapter

h) Interpreter

_g_ Un programa debe interactuar con diferentes librerías de correos, pero cada una tiene una interfaz distinta.

__ Un programa que simplifica la interacción de múltiples sensores *IoT* que colaboran entre ellos.

_e_ Enviar un mensaje *SMS* requiere código complejo que implica gestionar interacciones con múltiples servicios.

__ Un catalogo virtual de platos de cocina en que agrupaciones de ingredientes representan elementos de los platos.

_b_ Un editor de imágenes que al cerrarse inesperadamente permita recuperar los últimos cambios realizados una vez que se abra nuevamente.

_a_ Un programa debe permitir el envío de un mismo mensaje en distintos formatos (tales como audio o correo electrónico).

__ Un catalogo virtual de decoración que reutiliza una misma imagen en distintos ambientes visuales.

__ Una aplicación debe llevar un registro de todos los usuarios que han accedido a información restringida.

_h_ Una aplicación en que los usuarios sean capaces de crear y ejecutar *scripts* seguros dentro de esta.

_d_ Un programa debe ser capaz de replicarse en distintos procesos. Un nuevo proceso debe iniciar una nueva instancia con la misma información del programa.

_f_ Una aplicación para envío masivo de correos que presentan una base en común pero que se modifica según los destinatarios.

_c_ Una aplicación con múltiples hilos de ejecución (*multithreading*) necesita consistencia al interactuar con una base de datos.

__ Una aplicación de conciertos que notifica a sus usuarios cuando sus grupos favoritos irán a una ciudad.

Nombre _____ Número de lista

3. (0.8 pts) Considere el siguiente contexto:

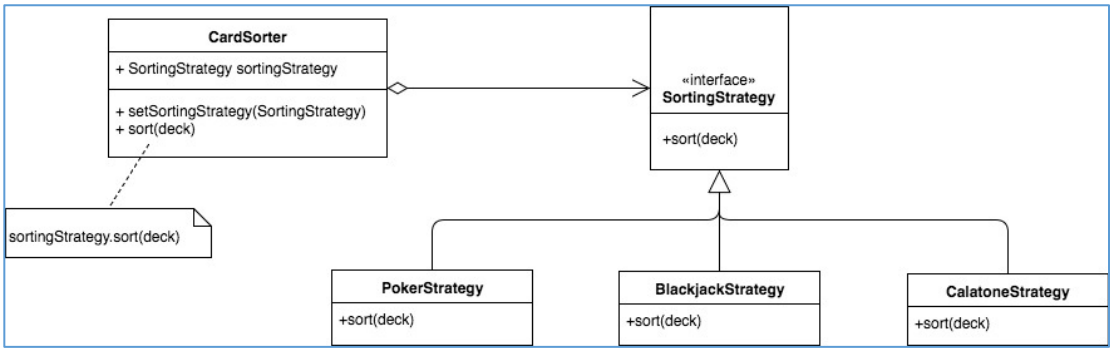
"Existen rumores de que los juegos de cartas en los casinos son adulterados para favorecer a *la casa*. Por esta razón la superintendencia de casinos quiere enviar a los distintos locales un fiscalizador para revisar las barajas de cartas utilizadas. Algunos de los controles que realizarían son contar las cartas de cada tipo o revisar si tienen alguna marca que las diferencie.

Para facilitar esta labor la superintendencia está evaluando la construcción de un programa para ordenar las cartas en base a su valor y pinta. La implementación debe considerar la eficiencia al ordenar las barajas, ya que el tiempo es fundamental para los funcionarios. En este sentido, se sabe que un algoritmo para ordenar cartas de las mesas de póquer tiene un rendimiento mucho menor al ordenar cartas utilizadas en mesas de *Blackjack*. Por esta razón la aplicación debe contar con distintos algoritmos de ordenación, en el que su uso se decidirá en base a los juegos en que se utilizaron las cartas. La implementación también debe permitir que se agreguen nuevos algoritmos con simples modificaciones al código fuente."

a) Proponga un patrón de diseño *GoF* que se podría aplicar al diseño del programa descrito. Justifique su elección en base al contexto del problema. Explicite todos los supuestos que realice.

Un posible patrón de diseño GoF es *Strategy*, ya que permite definir un conjunto de algoritmos intercambiables según la estrategia que se quiere utilizar. En el contexto del problema, la aplicación contará con distintos algoritmos de ordenación, y en base al origen de las cartas se deberá elegir una estrategia para ordenarlas.

b) Implemente un diagrama de clases *UML2* para representar el patrón de diseño anterior en la solución del problema. Explique cómo y en qué componentes se refleja el patrón de diseño. Su respuesta debe tener un nivel de detalle suficiente para que permita implementar el patrón en base al diagrama.



La interfaz *SortingStrategy* permite que la clase *CardSorter* intercambie estrategías de ordenamiento sin preocuparse de la implementación (a través de los métodos

setSortingStrategy y *sort*). Luego se pueden crear clases que implementen la interfaz *SortingStrategy*, tales como *PokerStrategy* o *BlackjackStrategy*, las cuales implementan su propia forma para ordenar cartas. De esta forma, si se quieren agregar más algoritmos para ordenar cartas se tendrían que agregar clases con la interfaz *SortingStrategy*, lo que haría que las modificaciones en el código principal fuesen mínimas porque se trabaja a nivel de interfaces.

4. (0.6 pts) Para cada uno de los siguientes *code smells* indique a qué familia pertenece, explique a qué se refiere y cuáles son los beneficios al evitar su presencia en el código:

a) *Temporary Field*:

b) *Middle Man*:

c) *Divergent Change*:

Nombre _____ Número de lista

5. (0.9 pts) Para cada uno de los siguientes 3 extractos de código identifique el *code smell* predominante, indique a qué familia pertenece y realice un *refactor* para eliminarlo:

```ruby
class CircularObject
  def initialize(radius)
    @radius = radius
  end

  def perimeter
    2 * Math::PI * @radius
  end

  def volume
    4/3 * Math::PI * @radius**3
  end
end

class Circle < CircularObject
  def volume
    raise NotImplementedError
  end
end

class Sphere < CircularObject
  def perimeter
    raise NotImplementedError
  end
end
```

Extracto 1: *Refused Bequest* – Mover los métodos *perimeter* y *volume* a las clases en concreto

```ruby
class Office
  def initialize(country, city, street, street_number)
    @country = country
    @city = city
    @street = street
    @street_number = street_number
    @employees = []
  end

  def add_employee(employee)
    @employees << employee
  end

  def distance_from(country, city, street, street_number)
    GPS.distance(
      from: [country, city, street, street_number],
      to: [@country, @city, @street, @street_number]
    )
  end

  def get_direction
    street + " " + street_number + ", " + city + ", " + country
  end
end
```

Nombre _____ Número de lista

Extracto 2: *Data Clumps* – Se debe agregar una clase *Address* para agrupar esos atributos, además de utilizarla en *get_direction*

```ruby
puts "What is your major?"

major = gets.chomp # user input

case major
when "Biology"
  puts "Mmm the study of life itself!"
when "Computer Science"
  puts "I'm a computer!"
when "English"
  puts "No way! What's your favorite book?"
when "Math"
  puts "Sweet! I'm great with numbers!"
else
  puts "That's a cool major!"
end
```

Extracto 3: *Primitive Obsession* – Se debería utilizar un diccionario directamente (*Switch Statements* no es válido)

Nombre _____ Número de lista

6. (0.9 pts)

a. Con respecto al diseño de *tests* unitarios: ¿qué diferencia implica realizar pruebas de *BlackBox* en comparación a pruebas de *WhiteBox*?

Las pruebas de *WhiteBox* tendrán mayor sentido y detalle ya que se tiene acceso a la implementación del código al momento de diseñar los *tests*. Por otra parte, las pruebas de *BlackBox* pueden ser redundantes e incompletas ya que no se tiene claridad de qué se está probando.

b. ¿A qué se refiere el concepto de *coverage* sobre una batería de pruebas? ¿Cómo se relaciona con la calidad de un *software*?

Coverage: proporción del código fuente ejecutado al momento de correr los *tests*. *Coverage* no tiene ninguna relación con la calidad del *software* ya que no garantiza nada.

c. Explique la diferencia entre los 4 tipos de *coverage* vistos en clases. Base su respuesta en un único extracto de código que sirva para ejemplificar los distintos niveles de exigencia entre estos.

Nombre _____ Número de lista

7. (0.6 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 acoplamiente 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 que acceden a 1 o más atributos en común de una clase.

Nombre _____ Número de lista

8. (0.6 pts)

a. ¿Con qué objetivos se puede aplicar ingeniería reversa en el contexto del *software*?
   - Descubrir el diseño de una aplicación (también cuenta como "crear copias/alternativas")
   - Realizar análisis de seguridad
   - Mejorar especificaciones de un *software legacy*
   - Facilitar la reutilización de productos

b. ¿Por qué en la ingeniería reversa la completitud generalmente es inversamente proporcional al nivel de extracción?

   Generalmente son inversamente proporcionales porque a mayor nivel de extracción menos acceso se tiene a las especificaciones, por lo que es más difícil extraer información lo que hace que el análisis sea menos completo.