Pregunta 3

Un equipo de desarrollo de una empresa es capaz de entregar en promedio 20 sp (puntos de relato) por semana con desviación estándar de 5 sp. El proyecto a desarrollar contempla 320 sp en total y debe ser entregado en 14 semanas.

a) ¿Cual es la probabilidad de que el equipo logre completar todo en 14 semanas o menos ?
b) Si quisieramos que la probabilidad de completar todo lo prometido en ese plazo sea de un 40% ¿Que cantidad de sp deberíamos comprometernos a tener en ese plazo?

Solución

Usemos la variable x para denotar la cantidad de sp logrados en 14 semanas

La media de sp para una semana es 20
Sabemos que en las distribuciones gausianas las medias se suman por lo que la media de las 14 semanas será de 14 * 20 = 280 sp

La varianza para una semana es 25 (5^2)
La varianza será la suma de las varianzas por lo que la desviación estándar para 14 semanas será de 14*25 = 350 sp
Luego la desviación estándar será la raiz cuadrada
Sigma = √ 350 = 18.7sp

Tenemos entonces para las 14 semanas una distribución normal N(280, 18.7)

Normalizando la variable x

$z = (x - 280)/18.7$

a) Para poder cumplir con el plazo en las 14 semanas el equipo debe poder entregar 320 sp o más.
   Luego, necesitamos saber $p(x \geq 320)$
   $z = (x -280)/18.7 \Rightarrow (320 - 280)/18.7 = 2.14$
   O sea necesitamos $p(z > 2.14) = 0.0162$ (de la tabla)

   Hay solo un 1,6% de probabilidad de cumplir 320 sps en 14 semanas

b) Llamemos z1 al punto buscado de la curva normal
   Necesitamos ahora que $p(z \geq z1) = 0.4$

   $p(z > z1) = 0.40$ De la tabla obtenemos $z1 = 0.25$

   $z = 0.25 \Rightarrow x = 0.25 * 18.7 + 280 = 284.7$

   Es decir, hay un 40% de probabilidad de entregar 284.5 sp en las 14 semanas

   Podemos comprometernos a 284 sp para una probabilidad del 40%

   Nota: Observar que si quisieramos asegurar un 50% de probabilidad el número a comprometer sería de 280 sp que es exactamente la media

Nota de Pauta:
Es muy importante llegar a la distribución normal N(280, 18.7) que requiere sumar las medias y las varianzas. Si tienen malo esto el resto de la pregunta estará todo malo.
Si tienen malo esta parte pero el procedimiento en a) y b) están correctos asignar la mitad del puntaje en cada caso.

Pregunta 2

Esta pregunta está dedicada a conceptos de testing.

a) A continuación encontrará dos párrafos en los cuales se ha reemplazado el tipo de test involucrado por una letra de la A a la I. Lo que Ud. debe hacer es cambiar cada una de estas letras por uno de los tests que se entregan en la tabla. Su respuesta debe ser de la forma A - test xxx, B - test xyz, etc.

   El equipo esta trabajando con integración continua. Normalmente se hace un _A_ inmediatamente después de hacer el build del día. Estas pruebas pueden tomar un tiempo considerable así es que se evita lo más posible hacerlas mas veces durante el día aunque se hagan correcciones en la línea de base. En estos casos utilizamos un _B_ si se trata de corregir un problema o un _C_ en otros casos. El _D_ y el _E_ requieren que el producto de software esté completo. La diferencia es que el _D_ apunta a una verificación mientras que el _E tiene más que ver con una validación del producto. Un tipo de _D_ es el _F_ o el _G_, mientras que _H_ pertenece al tipo _E_.

test de regresión	beta test	test de aceptación
smoke test	test automatizado	test de big bang
test de esfuerzo (stress)	test de usabilidad	sanity test
test unitario	test de sistema	test de carga (load)

b) Hay mucha gente que confunde los conceptos de test de integración y test de regresión. Explique en no más de 5 líneas las semejanzas y las diferencias entre ellos.

c) Hay mucha gente que confunde los conceptos de test de carga (load test) con test de esfuerzo (stress test). Explique en no más de 5 líneas las semejanzas y las diferencias entre ellos.

Solución

a) A - test de regresión
   B - sanity test
   C - smoke test
   D - test de sistema
   E - test de aceptación
   F- test de esfuerzo (stress)
   G - test de carga (load)
   H - beta test

b) El test de regresión puede ser usado como un test de integración si se usa para asegurar que las componentes que individualmente estaban correctas funcionan bien en conjunto. Sin embargo, el test de regresión puede ser usado también para asegurar que cualquier modificación no haya dañado alguna parte sin darnos cuenta por lo que puede correrse no solamente al integrar componentes sino en muchos otros casos.

c) Ambos tipos de test consisten en producir una situación de carga del software con alguna herramienta automatizada que simule un cierto número de usuarios o un cierto número de transacciones por minuto. La diferencia fundamental es que en el test de carga esos números son los esperados en operación normal (o algo mayores) y se espera que el sistema pueda manejarlo, en cambio en el test de stress la prueba es con números mucho mayores y se espera que el sistema en algún momento no sea capaz de manejarlo

Pregunta 4

Una aplicación web de e-commerce define en su modelo tres entidades, cuyos nombres y atributos se definen a continuación:

- Item: producto a la venta en la página
  - :name – nombre del producto, string
  - :price – precio del producto, número
- Person: usuario registrado en la página
  - :first_name – nombre de pila del usuario, string
  - :last_name – apellidos del usuario, string
- Order: registro de una orden de venta efectuada por un usuario
  - :date – fecha en que se efectuó la order, DateTime
  - :delivered – flag que indica si acaso ya se entregó la orden o no, true o false

En distintas partes de la aplicación es necesario mostrar listados de todos los productos disponibles para la venta, todos los usuarios registrados y todas las órdenes pendientes. Sin embargo, es necesario ordenar los datos según algún criterio antes de mostrarlos. Para ello se definen las siguientes políticas:

- Items deben ordenarse de menor a mayor precio.
- Persons deben ordenarse en orden alfabético por apellido, y luego por nombre en caso de que hayan dos personas con el mismo apellido.
- Orders: primero se muestran las pendientes, y luego las ya entregadas. En ambos casos debe ser por orden de antigüedad, las más antiguas primero.

Un desarrollador propone utilizar el algoritmo QuickSort para ordenar listas de cualquiera de estos tres tipos de datos. Se busca poder realizar esta tarea con el mismo método. A continuación una implementación preliminar del método escrito por el desarrollador, pero el problema de esta implementación es que incurre en excesiva duplicación de código, por lo que está lejos de ser ideal desde una perspectiva de diseño.

Utilice el patrón Strategy para proveer una implementación alternativa que resuelva el problema aquí descrito.

```
class Quicksort

  # ary can be a list of Items, Persons or Orders, but the list cannot contain a mixture of
objects from different types.
  def sort(ary)
    return [] if ary.empty?
    pivot = ary.delete_at(rand(ary.size))
    left = []
    right = []
    ary.each do |item|
      if item.is_a? Item
        if item.price < pivot.price
          left << item
        else
          right << item
        end
      end
      if item.is_a? Person
        # Omitted for brevity
      end
      if item.is_a? Order
        # Omitted for brevity
      end
    end
    return sort(left) + [pivot] + sort(right)
  end

end
```

Hints: Las clases String y DateTime ofrecen el operador <=> para comparar el orden de dos objetos del mismo tipo.

Solución

```ruby
class Quicksort
  def initialize(comparator)
    @comparator = comparator
  end
  def sort(ary)
    return [] if ary.empty?

    pivot = ary.delete_at(rand(ary.size))

    left = []
    right = []
    ary.each do |item|
      if @comparator.compare(item, pivot) < 0
        left << item
      else
        right << item
      end
    end

    return sort(left) + [pivot] + sort(right)
  end
end


class ItemComparator
  def compare(item1, item2)
    item1.price - item2.price
  end
end

class PersonComparator
  def compare(person1, person2)
    if person1.last_name != person2.last_name
      person1.last_name <=> person2.last_name
    else
      person1.first_name <=> person2.first_name
    end
  end
end

class OrderComparator
  def compare(date1, date2)
    if date1.delivered && !date2.delivered
      1
    elsif !date1.delivered && date2.delivered
      -1
    else
      date1.date <=> date2.date
    end
  end
end
```

Pregunta 1

Se ha construido un trozo de código que utiliza un patrón observador para vigilar y actuar frente a cambios en el sujeto que en este caso es un objeto de la clase Employee. Los objetos de esta clase representan empleados y tienen solo 3 atributos: *name, job* y *salary*.

Los objetos observadores corresponden a instancias de 2 clases: JobsLog y SalariesLog y, como buenos observadores se activan cuando el sujeto sufre cambios. Para simplificar, supondremos que la reacción de cada uno de ellos ante cambios en el estado del sujeto es muy simple:
Un objeto JobsLog imprimirá el nuevo cargo del empleado de la siguiente forma:
*** Pepe es un analista *** (suponiendo que es Pepe quien cambió)
Un objeto SalariesLog imprimirá el nuevo sueldo del empleado de la siguiente forma:
*** el sueldo de Pepe es 1.000.000 *** (suponiendo que es Pepe quien cambió)

A continuación un trozo de código que prueba el funcionamiento:

```
jaime = Employee.new('Jaime', 'Programador Junior', 1000000)
jobslog = JobsLog.new
salarieslog = SalariesLog.new
jaime.add_observer( jobslog )
jaime.add_observer( salarieslog )
jaime.job = "Programador"
jaime.salary = 1500000
jaime.delete_observer( jobslog )
jaime.job = "Programador Senior"
jaime.salary = 2000000
```

El output que produce es el siguiente:

```
*** Jaime es un Programador ***
*** el sueldo de Jaime es  1000000 ***
*** Jaime es un Programador ***
*** el sueldo de Jaime es  1500000 ***
*** el sueldo de Jaime es  1500000 ***
*** el sueldo de Jaime es  2000000 ***
```

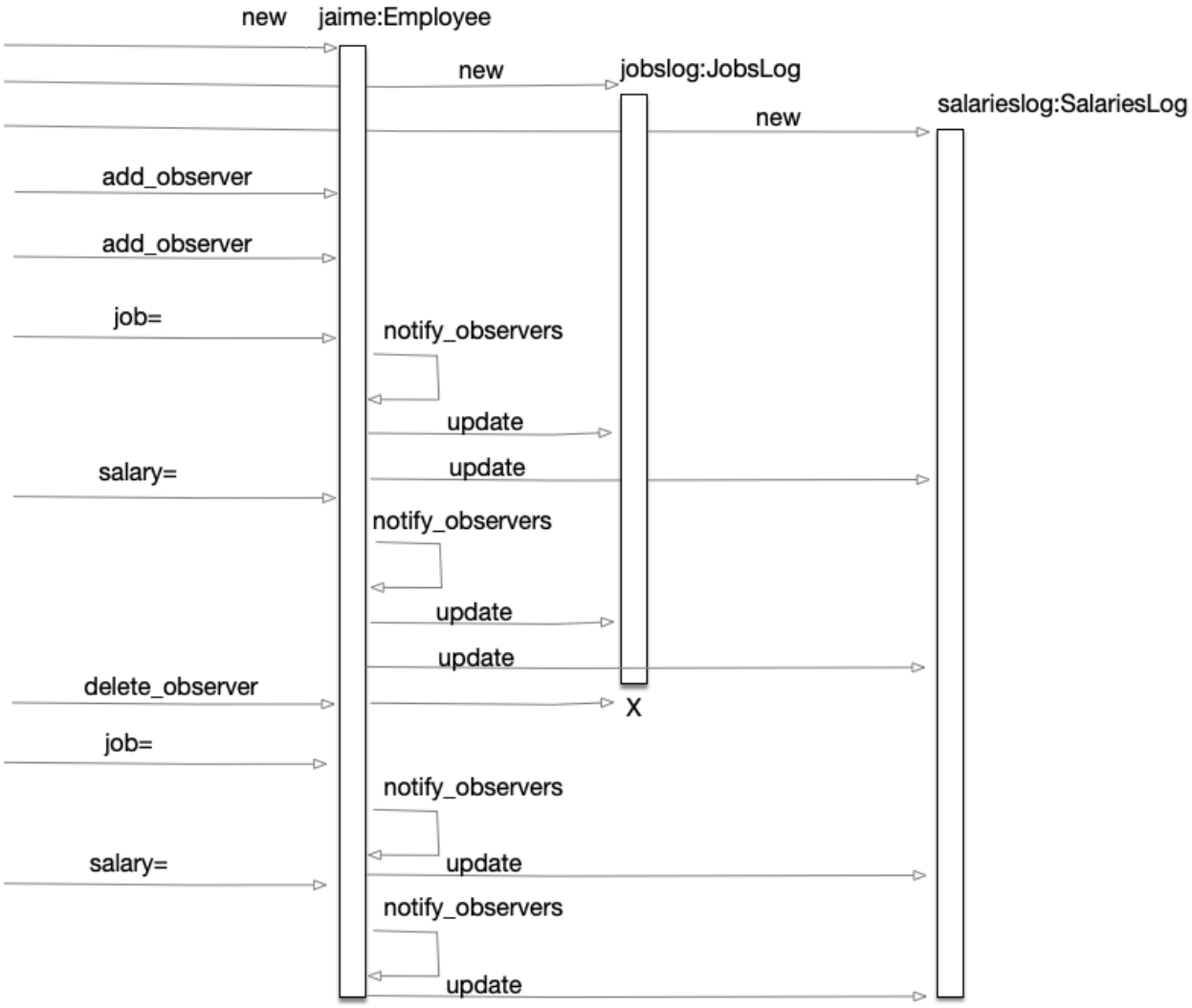
a) Escriba las clases Employee, SalariesLog y JobsLog y dibuje el diagrama de clases correspondiente

b) Dibuje un diagrama de secuencia que ilustre toda la secuencia de prueba y que permita explicar el output generado

Solución

```
class Employee
  attr_reader :name, :job
  attr_reader :salary
  def initialize( name, title, salary )
   @name = name
   @job = job
   @salary = salary
   @observers = []
  end

  def salary=(new_salary)
   @salary = new_salary
   notify_observers
  end

  def job=(new_job)
    @job = new_job
    notify_observers
  end

  def notify_observers
   @observers.each do |observer| observer.update(self)
   end
  end

  def add_observer(observer)
   @observers << observer
  end

  def delete_observer(observer)
   @observers.delete(observer)
 end
end

class JobsLog
  def update( changed_employee )
   puts("*** #{changed_employee.name} es un #{changed_employee.job} ***")
  end
end

class SalariesLog
  def update( changed_employee )
   puts("*** el sueldo de #{changed_employee.name} es  #{changed_employee.salary} ***")
   end
  end
```

b)



Observar que hay 6 veces que se invoca un update: dos para cada uno de los dos observadores primero:

```
*** Jaime es un Programador ***
*** el sueldo de Jaime es  1000000 ***
*** Jaime es un Programador ***
*** el sueldo de Jaime es  1500000 ***
```

y luego dos para el observador de sueldos que queda después de eliminar el de jobs.

```
*** el sueldo de Jaime es  1500000 ***
*** el sueldo de Jaime es  2000000 ***
```