repaso y entrada en calor para la materia

Inicio

Objetos 2 de la UNQ

Objetivos

- Refrescar conceptos básicos sobre programación orientada a objetos.
- Motivar la discusión sobre alternativas básicas de diseño de soluciones OO.

repaso y entrada en calor para la materia

- Revisaremos errores comunes antes de comenzar con los temas propios de Objetos 2. 1

Trabajo práctico de repaso y entrada en calor para la materia Objetos 2 de la UNQ

Programación Orientada a Objetos

1 - Evaluación de protocolos de una clase

Sea la clase Rectángulo con 4 variables de instancia (`esquinaSuperiorIzquierda`, `esquinaSuperiorDerecha`, `esquinaInferiorIzquierda` y `esquinaInferiorDerecha`). Elija uno de los protocolos presentados a continuación y justifique su elección. Recuerde que el protocolo es el conjunto de mensajes que entiende una clase o tipo.

< Smalltalk />

Opción 1)

```
Class Rectangulo>>new
Rectangulo>>esquinaSuperiorIzquierda: unPunto
Rectangulo>>esquinaSuperiorDerecha: unPunto
Rectangulo>>esquinaInferiorIzquierda: unPunto
Rectangulo>>esquinaInferiorDerecha: unPunto
```

Opción 2)

```
Class Rectangulo>>newEnOrigenEsquinaSuperiorIzquierda:unPunto alto:unNumero ancho:unNumero
Rectangulo>>reubicarEsquinaSuperiorIzquierdaEn: unPunto
Rectangulo>>ancho: unNumero
Rectangulo>>alto: unNumero
```

< Wollok />

Opción 1)

```
rectangulo = new Rectangulo()
rectangulo.esquinaSuperiorIzquierda(unPunto)
rectangulo.esquinaSuperiorDerecha(unPunto)
rectangulo.esquinaInferiorIzquierda(unPunto)
rectangulo.esquinaInferiorDerecha(unPunto)
```

Opción 2)

```
rectangulo = new Rectangulo(esquinaSuperiorIzquierda=unPunto, altura=unaAltura,
ancho=unAncho) rectangulo.reubicarEsquinaSuperiorIzquierdaEn(unPunto)
rectangulo.ancho(unAncho)
rectangulo.alto(unaAltura)
```

2 – Delegación

En una oficina hay un jefe que tiene un secretario el cual administra un fichero. Dadas las siguientes implementaciones seleccione la mejor alternativa y justifique.

< Smalltalk />

Opción 1)

```
Jefe>>trabajarConFicha: unaFicha
```

26/03/2020

Trabajo práctico de repaso y entrada en calor para la materia Objetos 2 de la UNQ

```
   self secretario fichero buscar: unaFicha.
```

Opción 2)

```
 Jefe>>trabajarConFicha: unaFicha
  self secretario buscharEnFichero: unaFicha

 Secretario>>buscarEnFichero: unaFicha
  self fichero buscar: unaFicha
```

< Wollok />

Opción 1)

```
class Jefe {
  method trabajarConFicha(unaFicha) {
    self.secretario().fichero().buscar(unaFicha)
  }
}
```

Opción 2)

```
class Jefe {
  method trabajarConFicha(unaFicha) {
    self.secretario().buscarEnFichero(unaFicha)
  }
}

class Secretario {
  method buscarEnFichero(unaFicha) {
    self.fichero().buscar(unaFicha)
  }
}
```

3 - Polimorfismo

Existen cuentas bancarias que pueden ser de tipo `CajaDeAhorro` y `CuentaCorriente`. Indique los defectos de cada una de las opciones.

< Smalltalk />

Opción 1)

```
CuentaBancaria>>extraer:unMonto
 |rojo|
 self class = CuentaCorriente ifTrue:[rojo:= self rojoPermitido.]
 ifFalse:[rojo := 0.]
 (self saldo + rojo >= unMonto)
 ifTrue: [self saldo: self saldo - unMonto].
```

Opción 2)

```
CuentaBancaria>>extraer:unMonto
```

```
 |rojo|
 self tipo = 'cuentacorriente'
 ifTrue:[rojo:= self rojoPermitido.]
 ifFalse:[rojo := 0.]
 (self saldo + rojo >= unMonto)
 ifTrue: [self saldo: self saldo - unMonto].
```

Opción 3)

```
CuentaBancaria>>extraer:unMonto
```

```
^self subclassResponsibility

CajaDeAhorro>>extraer:unMonto
 (self saldo >= unMonto)
 ifTrue: [self saldo: self saldo - unMonto].

CuentaCorriente>>extraer:unMonto
 (self saldo + self rojoPermitido >= unMonto )
 ifTrue: [self saldo: self saldo - unMonto].
```

Opción 4)

```
CuentaBancaria>>extraer:unMonto
 (self chequearSaldoParaExtraccion:unMonto)
 ifTrue: [self saldo: self saldo - unMonto].

CajaDeAhorro>>chequearSaldoParaExtraccion:unMonto
 ^self saldo >= unMonto

CuentaCorreinte>>chequearSaldoParaExtraccion:unMonto
 ^self saldo + self rojoPermitido >= unMonto
```

< Wollok />

Opción 1)

```
class CuentaBancaria
{
  method extraer(unMonto)
  {
    var rojo = 0
    if (self.class() == CuentaCorriente) rojo = self.rojoPermitido()
    if ((self.getSaldo() + rojo) >= unMonto) self.setSaldo(self.getSaldo() - unMonto)
  }
}
```

Opción 2)

```
class CuentaBancaria {
  method extraer(unMonto) {
    var rojo = 0
    if (self.getTipo() == 'cuentacorriente') rojo = self.rojoPermitido()
    if ((self.getSaldo() + rojo) >= unMonto) self.setSaldo(self.getSaldo() - unMonto)
  }
}
```

Opción 3)

```
class CuentaBancaria {
  method extraer(unMonto)
}

class CajaDeAhorro inherits CuentaBancaria {
  override method extraer(unMonto) {
```

```
      if (self.getSaldo() >= unMonto) self.setSaldo(self.getSaldo() - unMonto)
  }
}

class CuentaCorriente inherits CuentaBancaria {
  override method extraer(unMonto) {
    if ((self.getSaldo() + self.rojoPermitido()) >= unMonto)
      self.setSaldo(self.getSaldo() - unMonto)
  }
}
```

Opción 4)

```
class CuentaBancaria {
  method extraer(unMonto) {
    if(self.chequearSaldoParaExtraccion(unMonto)) self.setSaldo(self.getSaldo() - unMonto)
  }
}

class CajaDeAhorro inherits CuentaBancaria {
  override method chequearSaldoParaExtraccion(unMonto) = self.getSaldo() >= unMonto
}

class CuentaCorriente inherits CuentaBancaria {
  override method chequearSaldoParaExtraccion(unMonto) =
    (self.getSaldo() + self.rojoPermitido()) >= unMonto
}
```

Actividad de lectura #1

En el libro de Kent Beck Smalltalk Best Practices Patterns se discute la utilización de accessors. Lea las secciones Direct Variable Access e Indirect Variable Access que se encuentran al final de este Trabajo práctico.

Luego responda a las siguientes preguntas:
1. ¿Qué significa el acceso directo a las variables? De un ejemplo.
2. ¿Qué significa el acceso indirecto a las variables? De un ejemplo.
3. Qué ventajas y desventajas presenta cada estrategia referida a los getters y setters.

Fragmento del Libro: **Smalltalk Best Practice Patterns** de **Kent Beck**

Direct Variable Access

You need to access Common State as readably as possible.

How do you get and set the value of an instance variable?

Accessing state is a topic much like initialization. There are two good answers. One is more readable. One is more flexible. Also like initialization, you will find dogmatic adherents of both approaches.

The simple, readable way to get and set the values of instance variables is to use the variables directly in all the methods that need their values. The alternative requires that you send a message every time you need to use or change an instance variable's value.

When I first started programming in Smalltalk, at Tektronix in the mid-80s, the debate about accessing state was hot. There were factions on both sides making self-important statements (don't you just love research labs?)

The Indirect Variable Access crowd has won the hearts and mind of the Smalltalking public, mostly I think because most of the high volume training companies teach indirect access. The issue is nowhere near as simple as "direct access bad, indirect access good".

I tried to reopen the debate in my Smalltalk Report column a few years back. A little brush fire started that eventually fizzled out. I was wondering if I was making too big a deal of the merits of direct access. Maybe I should have left well enough alone.

Then I spent several months working for a client that insisted on indirect access. A professional programmer can take on any of a number of styles at will, so I was a good soldier and wrote my getters and setters.

After I was done with that client, I wrote some code for myself. I was amazed at how much smoother it read than what I had been working on previously. The only difference in style was direct versus indirect access.

What I noticed was that every time I read:
```
            x...
```

Smalltalk:
```
 ...self
```

Wollok:
```
 ...self.x...
```

I paused for a moment to remind myself that the message "x" was just fetching an instance variable. When I read:

```
 ...x...
```

I just kept reading.

I told Ward of my experience. He had another good explanation. When you write classes that only have a handful of methods, adding a getting and a setting method can easily double the number of methods in your class. Twice as many methods to buy you flexibility that you may never use.

On the other hand, it's awful frustrating to get a class from someone and see the opportunity to quickly subclass it, only to discover that they used Direct Variable Access so there is no way to make the changes you want without changing most of the code in the superclass.

Access and set variables directly.

Indirect Variable Access

You need to access Common State as flexibly as possible.

How do you get and set an instance variable's value?

Now I have to display my true schizophrenia. Having convinced you in Direct Variable Access that just using variables is good enough, I'm going to ask you to ignore that crazy bastard and listen to me talk some sense here.

When you use Direct Variable Access, many methods make the assumption that a variable's value is valid. For some code this is a reasonable assumption. However, if you want to introduce Lazy Initialization, to stop storing and start computing the value, or if you want to change assumptions in a new subclass, you will be disappointed that the assumption of validity is wide spread.

The solution is to always use Getting Methods and Setting Methods to access variables. Thus, instead of:

Smalltalk:

```
Point>>+ aPoint

 ^ x + aPoint x @ (y + aPoint y)
```

Wollok:

```
class Point {
  var x = 0
  var y = 0

  method +(aPoint) =
    new Point(x= (x + aPoint.x),
              y= (y + aPoint.y) )
}
```

you would see:

Smalltalk:

```
Point>>+ aPoint
 ^ self x + aPoint x @ (self y @ aPoint y)
    var x = 0
    var y = 0

    method +(aPoint) =
```

Wollok:

```
class Point {

    new Point(x= (self.x() + aPoint.x()),
              y= (self.y() + aPoint.y()) )
}
```

If you browse the instance variable references of a class that uses Indirect Variable Access you will only see two references to each variable, the Getting Method and the Setting Method.

What you give up with Indirect Variable Access is simplicity and readability. You have to define all those Getting Methods and Setting Methods. Even if you have your system do it for you, you have that many more methods to manage and document. As explained above, code using Direct Variable Access reads smoothly, because you are not forever reminding yourself "Oh yeah, that's just a Getting Method."

Access and change an instance variable's value only through a Getting Method and Setting Method.

If you need to code for inheritance, use Indirect Variable Access. Future generations will thank you.

You will need to define a Getting Method and a Setting Method for each variable. For variables holding collections, consider implementing Collection Accessor Methods and an Enumeration Method.

Getting Method

You are using Lazy Initialization or Indirect Variable Access.

How do you provide access to an instance variable?

Once you have decided to use Indirect Variable Access, you are committed to providing a message based protocol for getting and setting variable values. The only real questions are how you use it and what you call it.

Here's the real secret of writing good Getting Methods- make them private at first. I cannot stress this enough. You will be fine if the same object invokes and implements the Getting Method. Another way of saying this is you always send messages invoking Getting Methods to "self".

Some people try to ensure this by prefixing "my" to the name of the method. Thus, instead of:

Wollok:

```
method x() = x
```

Smalltalk:

```
x
  ^x
```

you have:

Wollok:

```
method myX() = x
```

Smalltalk:

```
myX
  ^x
```

This makes sure that code like:

Wollok:

```
self.bounds().origin().myX()
```

Smalltalk:

```
self bounds origin myX
```

looks stupid. I don't feel this is necessary. I'd rather give programmers (myself included) the benefit of the doubt. If some other object absolutely has to send my private Getting Method, then they should be able to do so in as readable a manner as possible.

Provide a method that returns the value of the variable. Give it the same name as the variable.

There are cases where you will publish the existence of Getting Methods for use in the outside world. You should make a conscious decision to do this after considering all the alternatives. It is much preferable to give an object more responsibility rather than have it act like a data structure.

Setting Method

You are using Indirect Variable Access.

How you change the value of an instance variable?

Everything I said once about Getting Methods I'd like to say twice about Setting Methods. Setting Methods should be even more private. It is one thing for another object to tear out your state, it is quite another for it to bash in new state. The possibilities for the code in the two objects to get out of sync and break in confusing ways are legion.

Revisiting naming, I don't feel it is necessary to prepend "my" to the names of Setting Methods. It might provide a little more protection from unauthorized use, but I don't think the extra difficulty reading is worth it.

Provide a method with the same name as the variable that takes a single parameter, the value to be set.

Even if I use Indirect Variable Access, if I have a variable that is only set at instance creation time, I would not provide a Setting Method. I'd use the Creation Parameter Method to set the all the values at once. Once you have a Setting Method, though, you should use it for all changes to the variable.

Set boolean properties with a Boolean Property Setting Method.

Actividad de lectura #2

En base al siguiente fragmento, responda: ¿en qué situación es conveniente utilizar el "Creation Parameter Method"?

_____

*Fragmento del Libro: **Smalltalk Best Practice Patterns** de **Kent Beck***

Creation Parameter Method

A Complete Creation Method needs to pass parameters on to the new instance. You need to initialize Common State.

How do you set instance variables from the parameters to a Complete Creation Method?

Once you have the parameters of a Complete Creation Method to the class, how do you get them to the newly created instance? The most flexible and consistent method is to use Setting Methods to set all the variables. Thus, a Point would be initialized with two messages:

Smalltalk:

```
Point class>>x: xNumber y: yNumber ^self
 new x: xNumber; y: yNumber; yourself
```

Wollok*:
```
class Point {

  method xy(xNumber, yNumber) {
    var newPoint = new Point()
    newPoint.x = xNumber
    newPoint.y = yNumber
    return newPoint
  }
}
```

(*Este código no es válido en Wollok ya que el lenguaje no necesita que se defina un método para crear una instancia de una clase con parámetros, a diferencia de otros lenguajes como Smalltalk y Java)

The problem I have run into with this approach is that Setting Methods can become complex. I have had to add special logic to the Setting Methods to check whether they are being sent during initialization, and if so just set the variable.

Remember the rule that says "say things once and only once"? Special casing a Setting Method for use during initialization is a violation of the first part of that rule. You have two circumstances -state initialization during instance creation and state change during computation- but only one method. You have two things to say and you've only said one thing.

The Setting Method solution also has the drawback that if you want to see the types of all the variables, you have to look at the Type Suggesting Parameter Names in several methods. You'd like the reader to be able to look at your code and quickly understand types. Create a single method that sets all the

variables. Preface its name with "set", then the names of the variables.

Trabajo práctico de repaso y entrada en calor para la materia Objetos 2 de la UNQ

Using this pattern, the code above becomes:

Smalltalk:

```
Point class>>x: xNumber y: yNumber ^self new
setX: xNumber y: yNumber


Point>>setX: xNumber y: yNumber x :=
xNumber.
y := yNumber.
^self
```

```
method setXY(xNumber, yNumber) {
    x = xNumber; y = yNumber
    return self
}
}
```

Wollok*:

```
class Point{
method xy(xNumber, yNumber)=
new Point().setXY(xNumber, yNumber)
```

(*Este código no es válido en Wollok ya que el lenguaje no necesita que se defina un método para crear una instancia de una clase con parámetros, a diferencia de otros lenguajes como Smalltalk y Java)

Note the Interesting Return Value in `setX:y:`. It is there because the return value of the method will be used as the return value of the caller.

If you are using Explicit Initialization, now is a good time to invoke it, to communicate that initialization is part of instance creation.

Actividad de lectura #3

En base al siguiente fragmento, responda: ¿cómo se debe proporcionar acceso a variables que referencian a una colección?

---

*Fragmento del Libro: **Smalltalk Best Practice Patterns** de **Kent Beck***

Collection Accessor Method

You are using Indirect Variable Access.

How do you provide access to an instance variable that holds a collection?

The simplest solution is just to publish a Getting Method for the variable. That way, any client that wants can add to, delete from, iterate over, or otherwise use the collection.

The problem with this approach is that it opens up too much of the implementation of an object to the outside world. If the object decides to change the implementation, say by using a different kind of collection, the client code may well break.

The other problem with just offering up your private collections for public viewing is that it is hard to keep related state current when someone else is changing the collection without notifying you. Here's a department that use a Caching Instance Variable to speed access to its total salary:

Smalltalk:

Department
  superclass: Object
  instance variables: employees totalSalary

  totalSalary
    totalSalary isNil ifTrue: [totalSalary := self computeTotalSalary]. ^totalSalary

  computeTotalSalary
    ^employees inject: 0 into: [:sum :each | sum + each salary]

  clearTotalSalary
    totalSalary := nil

Wollok:
```
 class Department {
  var employees = 0
  var totalSalary = null

  method totalSalary() {
    if (totalSalary.isNull()) totalSalary = self.computeTotalSalary()
    return totalSalary
  }

  method computeTotalSalary() = employees.sum({employee => employee.salary()})

  method clearTotalSalary() { totalSalary = null}
 }
```

What happens if client code deletes an employee without notifying the department?

Smalltalk: **Wollok:**

```
                                                          aDepartment.employees().remove(anEmployee)
 aDepartment employees remove: anEmployee
```

The `totalSalary` cache never gets cleared. It now contains a number inconsistent with the value returned by `computeTotalSalary`.

The solution to this is not to let other objects have your collections. If you use Indirect Variable Access, make sure Getting Methods for collections are private. Instead, give clients restricted access to operations on the collection through messages that you implement. This gives you a chance to do whatever other processing you need to.

The downside of this approach is that you have to actually implement all of these methods. Giving access to a collection takes one method. You might need four or five methods to provide all the necessary protected access to the collection. In the long run, it's worth it though, because your code will read better and be easier to change.

Implement methods that are implement with Delegation to the collection. To name the

methods, add the name of the collection (in singular form) to the collection messages.

If Department wanted to let others add and delete employees, it would implement Collection Accessor Methods:

Smalltalk:

```
addEmployee:     anEmployee
 self    clearTotalSalary.
 employees add: anEmployee
```

```
removeEmployee: anEmployee
 self clearTotalSalary.
 employees remove: anEmployee
```

Wollok:

```
method addEmployee (){
  self.clearTotalSalary()
  employees.add(anEmployee)
}

method removeEMployee(anEmployee){
  self.clearTotalSalary()
  employees.remove(anEmployee)
}
```

Don't just blindly name a Collection Accessor Method after the collection message it delegates. See if you can find a word from the domain that makes more sense. For example, I prefer:

to:

Smalltalk:

```
employs: anEmployee
 ^ employees includes: anEmployee
```

Wollok:

```
method employs(anEmployee) =
 employees.includes(anEmployee)
```

Implement an Enumeration Method for safe and efficient general collection

access. **Collection Accessor Method**

You are using Indirect Variable Access. You may have implemented a Collection Accessor Method.

How do you provide safe general access to collection elements?

Sometimes clients want lots of ways of accessing a private collection. You could implement twenty or thirty Collection Accessor Methods, but you haven't the time and you aren't even sure even that would be enough. At the same time, all the argument for not just making a Getting Method for the collection public still hold.

Implement a method that executes a Block for each element of the collection. Name the method by concatenating the name of the collection and "Do:".

The Enumeration Method for a Department's Employees looks like this:

Smalltalk:

```
Department>>employeesDo: aBlock
 employees do: aBlock
```
Wollok:

```
class Department {
  method employeesDo(aBlock) {
    employees.forEach(aBlock) }
}
```

Now client code that wants to get a collection of all of the Employees in a bunch of departments can use a Concatenating Stream:

Smalltalk:

```
 | writer |
 writer := WriteStream on: Array new.
 self departments do: [:eachDepartment | each employeesDo: [:eachEmployee |
    writer nextPut: eachEmployee]]
 ^ writer contents
```

Wollok:

```
method allEmployees() {
  var writer= new WriteStream()
  writer = self.departments.forEach({eachDepartment => eachDepartment.employeesDo
    ( {eachEmployee => writer.nextPut(eachEmployee)})
  })
  return writer.contents()
}
```

What if you want Departments to be able to contain other Departments, and not just Employees (this is an example of the modeling pattern Composite)? You can implement employeesDo: for both:

Smalltalk:

Department>>employeesDo: aBlock

```
employees do: [:each | each
employeesDo: aBlock]
```

Employee>>employeesDo: aBlock

```
aBlock value: self
```

Wollok:

```
class Department {
method employeesDo(aBlock) {
employee.forEach({ each =>
each.emploteesDo(aBlock)}) }
}
```

```
class Employee {
method employeesDo(aBlock) {
aBlock.apply(self)
}
}
```

Actividad de lectura #4

En base al siguiente fragmento, responda: ¿por qué son necesarios dos métodos para asignar el estado a una propiedad booleana?

---

*Fragmento del Libro: **Smalltalk Best Practice Patterns** de **Kent Beck***

Boolean Property Setting Method

You are using a Setting Method.

What protocol do you use to set a boolean property?

The simplest solution is to use a Setting Method. Let's say we have a Switch that stores a Boolean in an instance variable "isOn". The Setting Method is:

Smalltalk: Wollok:

```
Switch>>on: aBoolean
    isOn := aBoolean
```

```
var isOn = false
method on(aBoolean) {
    isOn = aBoolean
}
```

```
class Switch {
}
```

There are two problems with this approach. The first is that it exposes the representation of the status of the switch to clients. This has led me to situations where I make a representation change in one object that has to be reflected in many others. The second problem is that it is hard to answer simple questions like "who turns on the switch?"

Creating distinct methods for the two states causes you to create one more method than you would have otherwise. However, the cost of these methods is money well spent because of the improvement in communication of the resulting code.

Using the names of the two states as the names of the methods is tempting (Switch>>on and Switch>>off, in this example). As with Query Method, though, there is a potential confusion about whether you're interrogating the object or telling it what to do. Thus, even though adding another word to the selector results in a selector that is less natural to say, the added clarity is worth it.

Create two methods beginning with "make". One has property name, the other the negation. Add "toggle" if the client doesn't want to know about the current state.

Here are some examples from the image:

```
makeVisible/makeInvisible/toggleVisible makeDirty/makeClean
```