Índice

1. Apuntes de clases

1.1. De objetos, mensajes y variables

En el paradigma de objetos nos manejamos con las siguientes definiciones:

- Programa: Conjunto de Objetos que colaboran entre sí enviandose mensajes. No hay instrucción, procedimientos, statements, nada. Todo son mensajes. No hay nada más primitivo que un Objeto y un Mensaje.
- Objeto: representación de un ente del dominio del problema. Lo que hay que modelar es la esencia del ente. Y esta va a estar dada por los mensajes que recibe.
- Mensaje: no solo define el qué de un Objeto, sino el cómo.

Para que exista una colaboración tiene que haber un objeto emisor E y otro receptor R. E pone en un canal su mensaje y R lo recibe, y ejecuta el método, que generalmente tiene el mismo nombre que el mensaje, para poder dar la respuesta.
Una **comunicación** tiene las características de ser:

(1) Dirigida (no Broadcast)

(2) Sincrónica (hasta que no me contestan el mensaje no puedo hacer otra cosa)

(3) Siempre existe respuesta. Si no hay nada para devolver, se devuelve NIL. Un objeto que representa la nada.

(4) Receptor desconoce al Emisor.

Hay otros que hicieron paradigmas con estas características, pero cambiando el (2) por Actors. Es decir, un objeto va encolando mensajes y los va contestando a medida que puede. Pero no se frena en uno.
También cambiaron la (4), agregando la Subjetividad. Este cambio se basa en la idea psicológica que una persona no contesta un mensaje de la misma manera si el emisor es otro.
Hay **mensajes** de tres tipos:

- Unarios
- Binarios
- Keywords

Si tenemos esta colaboración:

```
aDict at: aNumber+3 put: anArray size
```

Se evalúa de izq a der, y primero unario, binario, keywords. Es decir:

1. `anArray size`
2. `aNumber+3`
3. `at: _ put: _`

Si la colaboración fuere:

```
aDict at: aNumber+3 put: otherNumber * 4
```

La evaluación sería

1. `aNumber+3`
2. `otherNumber*4`
3. `at: _ put: _`

En esta colaboración: `4+3*5`. Se evalúa entonces primero `4+3` y después el `*5`. Asique el resultado es 35.
El paradigma define como única relación, la **Relación de conocimiento**. Es decir, que un objeto conoce otros objetos. Y es lo que en otros lenguajes denominamos como **Variable**.
Acá no se permite decir que un objeto tiene otro. El objeto `1/2` conoce al 1 como numerador, y al 2 como denominador. Asique si nos metemos a la variable (que es un objeto también) denominador, este va a conocer a `value` que es 2, y a `name` que es 'denominador'
Ahora esta variable puede contestar algunos mensajes como value que va a devolver 2, también name que va a devolver 'denominador'. Así mismo sabe interpretar el mensaje binario `:= aValue`.
Ahora hay que diferenciar dos tipos de mensajes. Porque el mensaje `:=` manda un Objeto a la Variable. En cambio el mensaje `aVariable denominador`, es un mensaje que se dirije al `value` de `aVariable`.
Asique hay en principio tenemos dos tipos de mensajes:

1. `receptor mensaje`
2. `variable := objeto`

¿Porque la asignación no puede ser un mensaje? Porque los mensajes siempre se mandan a lo que referencia el objeto, y la asignacion va dirigida al objeto en sí (a la variable). Así el lenguaje define dos sintaxis:

```
r m // receptor mensaje
v := o // variable asignacion objet
```

Hay otro tipo de variables también: las llamads **pseudovariables**. Es como una variable, pero no hay que definirla, siempre existe. Además no se le puede asignar nada. Por ejemplo, `self`. Que lo único que hace es referenciar un objeto. Conceptualmente, esto es muy diferente de `this`. Este se refiere a una cosa, sin vida, solo estructura. Y `self` refiere a algo más vivo.
Otra pseudovariable es `thisContext`, que referencia al (objeto) contexto de ejecución, esa pila de stack que trackea el método que se está ejecutando.
Por ejemplo, dada la implementación del método

```
+ sumando
   thisContext method name // esto devuelve +
   | nuevoDenominador | // defino una variable local
   thisContext variableNamed: nuevoDenominador // esto devuelve el objeto
       referenciado por la variable local nuevoDenominador
```

Para resolver el primer problema de diseño vamos a usar diagramas. (Ver cuaderno)

1.2. Polimorfismo

En Smalltalk, el objeto 1 y true no es lo mismo. 1 sabe responder el mensaje "+". Asi como 0 no es false.
true y false son polimorficos respecto del mensaje not, and y or,
Hay que definir un objeto abstracto Boolean, con los metodos not, and, y or, que luego True y False van a implementar, y true y false van a ser instancias respectivas. Pero Boolean tienen

que tener una implementacion propia de los metodos, asique lo que se hace es poner self subclassResponsability. Que es una excepcion si se ejecuta.

Si true y false fueran instancia de una sola class Boolean, tendriamos q hacer una implementacion de not del estilo:

```
not:
  if( self ) ^ false
  ^ true
```

Pero `if(..)` tendria que o bien convertirlo en objeto mensaje, o agregarlo al grupo de sintaxis de Smalltalk que tengo hasta ahora:

1. `objeto mensaje`
2. `variable := objeto`
3. `[ ... ] mensaje`

Lo que hace al lenguaje mas cerrado. El programador no puede ver como esta hecho. Smalltalk quiere mantener su sintaxis lo mas chica posible, para que sea mas customizable al programador, asique lo implemento como objeto mensaje.

```
not:
  self ifTrue: [ ^ false ]
  ^ true
```

Pero `ifTrue` deberia ser un mensaje que Boolean pueda contestar, asique se implementa:

```
ifTrue: aClosure
  self ifTrue: ^ aClosure
  ^ nil
```

Regresion al infinitum! **Conculsion: No se puede implementar algebra booleana con un solo objeto!**
Asique volviendo al caso de un Booleano abstracto, `ifTrue` queda implementado asi
Y mantener una sintaxis minimalista.
———————- Como los if son problematicos (deja al programador la tarea de pensar el programa, cuando lo deberian hacer los objetos), queremos sacarlos y queremos implementarlo con polimorfismo

```
cond1: ifTrue: [^ aBlock1 value. ]
cond2: ifTrue: [^ aBlock2 value. ]
cond3: ifTrue: [^ aBlock3 value. ]
self error: '...'
```

¿Como sacamos un polimorfismo?

1. Opcional (porque ya puede existir la jerarquia): Crear jerarquia polimorfica con 1 abstraccion x cada if
2. Copiar closure de cada if a cada abstraccion usando mensajes polimorficos
3. Ponerle nombre a abstraccion
4. Poneler nombre a mensaje
5. Opcional (porque si ya existe la jerarquia, pueden existir los objetos): buscar objeto polimorfico
6. Reemplazar if x "objPolimorfico mensaje"

1.3. Lambdas vs. Closures vs. Full Closures

1.3.1. Closure

```
X >> m:
  | a |
  a:=1
  ^ [a := a + 1]

unX:= X new
aClosure := unX m
aClosure value. # 2
aClosure value. # 3
aClosure2 = unX m
aClosure value. # 2 # se crea un objeto a en un contexto de ejecucion diferente
```

Cuando se ejecuta un Closure, este bindea todas las variables que tiene al contexto de ejecucion que lo llamó. `[ .. ]` crea un nuevo objeto, y sabe que a `a` la busca en el contexto del que lo creo.

1.3.2. Lambda (Lisp)

```
X >> m:
  ^ [ a := a+1 ]
```

Esto compila aunque a no este definido.

```
aClosure value. # va a buscar en todos los contextos de ejecucion a a.
```

1.3.3. Full Closure (Smalltalk)

```
X >> m
  aClosure := [ ^ 10 ]
  ^ aClosure value + 5

aClosure value. # 10, porque nunca se llega a ejecutar \+ 5".
  cuando se llamo a [ ... ] salio del metodo.

X>> m
  ^ [ ^ 10 ]

aClosure value. # tira error. porque ya saliio del contexto de ejecucion cuando quiere vo
```

En los Full Closure, no solo bindea las variables al contexto de ejecución que lo llamó, sino que el return hace volver pero del contexto de ejecución padre.
Si no tuvieramos Full Closure, no podriamos implementar controles de flujo con mensajes. Si no se tienen Full Closure, la sintaxis tiene que implementar el If, exit (para salir del control de flujo).

1.3.4. While

¿Se puede hacer lo mismo con el While? Sí. Pero, ¿cómo?
`[...]` en Smalltalk esto crea un objeto.

```
a := 1
(a < 3) whileTrue: [ a:= a + 1 ]
```

Como cuando whileTrue termina, vuelve a ejecutar (a ¡ 3), y por lo tanto tiene que ser un Closure. Por lo tanto whileTrue se implementa en BlockClosure:

```
BlockClosure>>whileTrue: aBlock
  self value. # evaluo la condicion
    ifTrue: [
      aBlock value.
      self whileTrue: aBlock
    ]
```

Un lenguaje que elimina colas de la recursion se llaman tail recursives. Se puede hacer aca? Si, porque puedo acceder a los contextos de ejecucion, y sacar los closures que vaya apilando ahi. Asique como queda el whileTrue para que sea tail recursive,

```
BlockClosure>>whileTrue: aBlock
  self value. # evaluo la condicion
  ifTrue: [
    aBlock value.
    thisContext stack pop. # saco el contexto de ejecucion
    self whileTrue: aBlock
  ]
```

Aunque esto no termina andando en realidad.

2. Metamodelos

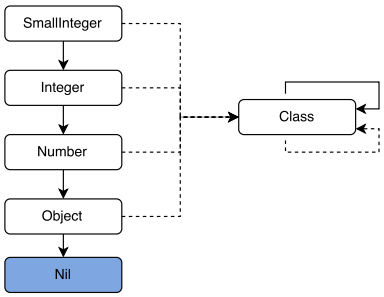

Figura 1: Metamodelo Básico

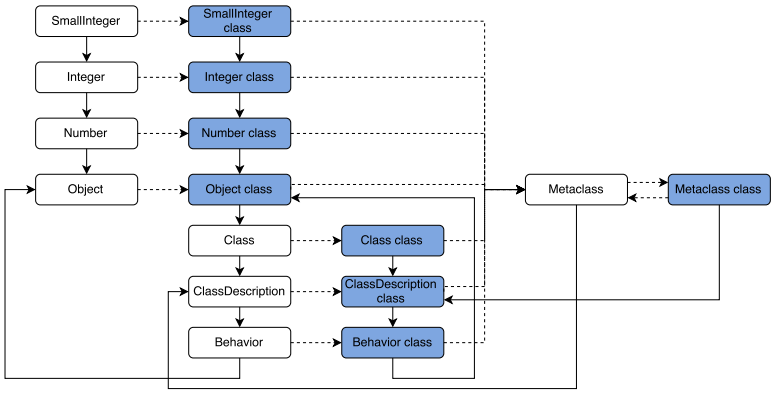

Figura 2: Metamodelo SmallTalk80

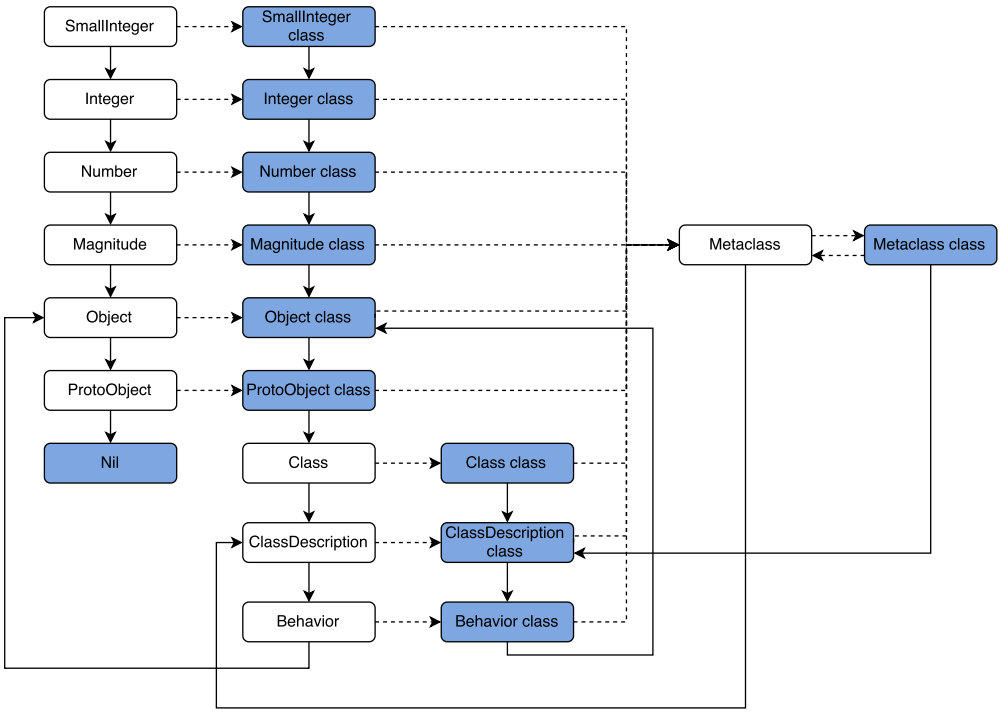

Figura 3: Metamodelo de Pharo4.0

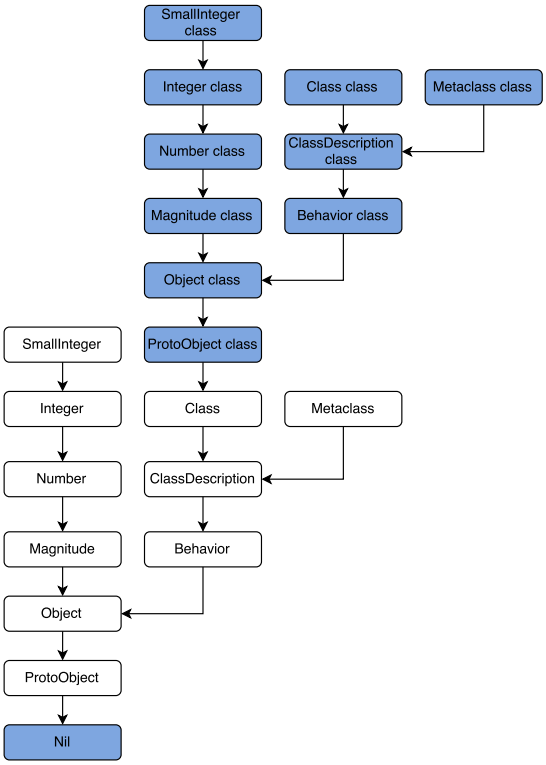

Figura 4: Grafo de Subclasificaciones

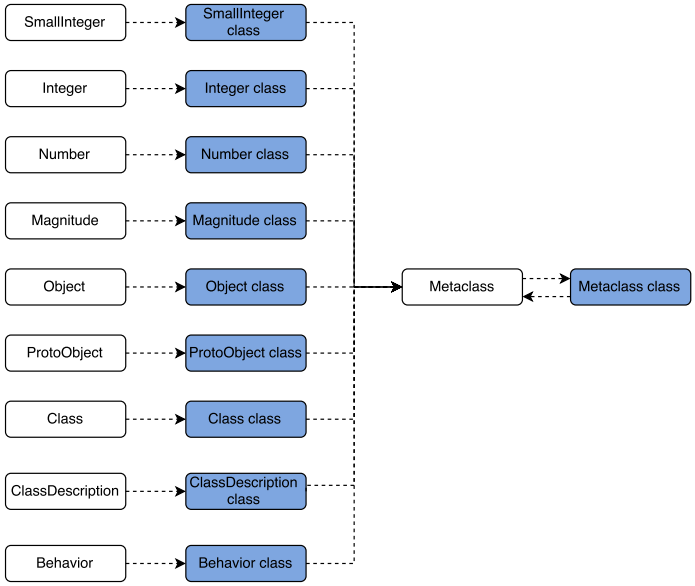

Figura 5: Grafo "Instancia de"

Figura 6: Metamodelo de Pharo4.0

Referencias: Linea punteada significa "Instancia de", y Linea lisa "Hereda de".

1. Los objetos en azul (¿las metaclases?), solo tienen una instancia. No contestan al mensaje `new`. ¿Quién hace la alocación por primera vez?

   Si a una metaclase se le agregan variables de instancia que inicializo en su método `initialize` pero luego las modifico, debo reinicializar la metaclase para que se apliquen los cambios a las variables. No es como el caso de las variables de instancia de una clase.

2. Cuando se envía `new` a cualquier objeto, el que aloca la memoria es `Behavior>>basicNew`.

   a) ¿Hace falta entonces dentro de cualquier implementación propia de `new` enviar la colaboración `super new` necesariamente?¿Si no lo hago alguien se encarga de esto?

   b) Si es `Behavior` quien implementa la alocación, ¿quién lo hace para `Object` y los que subclasifican de él? (El debugger no me deja meterme más adentro del `self new initialize` de `Object`)

   c) `ProtoObject new` tira el siguiente error y se rompe todo:

   ```
   *** System error handling failed ***
   Original error: MessageNotUnderstood: ProtoObject>>inspect.
   ```

   ¿Tiene algo que ver?

3. Enviarle la colaboración `superclass` a un objeto me devuelve lo que apunta la flecha lisa. Ej:

```
SmallInteger superclass
>> Integer
Metaclass superclass
>> ClassDescription
```

4. Enviarle la colaboración `class` a un objeto me devuelve lo que apunta la flecha punteada. Ej:

```
SmallInteger class
>> Smallinteger class
Integer class class
>> Metaclass
```

5. Enviar la colaboración `Metaclass new new` hace colgar Pharo. ¿Por qué? ¿Es el único objeto con el que pasa eso?

3. Resumen papers

3.1. Double Dispatch

Sacado del paper *Arithmetic and Double Dispatching in Smalltalk-80* de Kurt J. Hebel and Ralph E. Johnson.

3.2. Fail Fast

Sacado del paper *Fail Fast* de Jim Shore.

3.3. Null Pattern

Sacado del paper *The Null Pattern Object* de Bobby Woolf.

3.4. Object Recursion

Sacado del paper *The Object Recursion Pattern* de Bobby Woolf.

3.5. Programming as Theory Building

Sacado del paper *Programming as Theory Building* de Peter Naur.

3.6. Self

Sacado del paper *Self: The Power of Simplicity.*