

Índice

1. Apuntes de clases	2
1.1. Representación del mundo	2
1.1.1. Errores comunes de modelaje	2
1.1.2. Heurísticas para un buen modelaje	2
1.2. De objetos, mensajes y variables	3
1.3. Polimorfismo	4
1.3.1. ¿Como sacamos IFs, en pro de polimorfismo?	6
1.4. Lambdas vs. Closures vs. Full Closures	6
1.4.1. Closure	6
1.4.2. Lambda (Lisp)	6
1.4.3. Full Closure (Smalltalk)	7
1.4.4. While	7
1.5. Metamodelos	8
2. Resumen papers	11
2.1. Double Dispatch	11
2.2. Fail Fast	11
2.3. Null Pattern	11
2.4. Object Recursion	11
2.5. Programming as Theory Building	11
2.6. Self	11

1. Apuntes de clases

1.1. Representación del mundo

Platón definió al mundo en el que vivimos en dos: en ideas y conceptos, y elementos concretos. Para modelarlo correctamente, las clases tienen que representar justamente las ideas/conceptos, y la representación los elementos concretos tienen que ser instancia de la clase que representa su concepto.

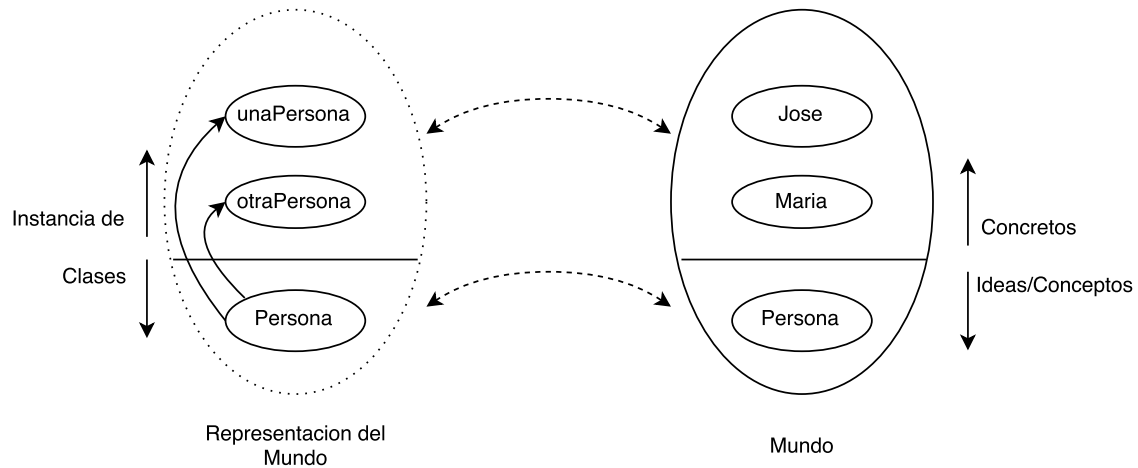


Figura 1: Representación del mundo

Luego para una mayor flexibilidad y escalabilidad las clases pueden subclasificarse.

1.1.1. Errores comunes de modelaje

1. Dar más responsabilidad (o conocimiento) a un objeto de lo que en la realidad tiene.
2. Permitir a los objetos modificarse, cuando en la realidad no lo hacen. Esto puede traer errores de estar trabajando con objetos incompletos.
3. La subclasificación debe hacerse a partir de propiedades esenciales y no accidentales. Las accidentales podrían cambiar en el tiempo, en cambio las esenciales no. Por ej, en el problema de modelar el facturador de una telefónica, una llamada no debería ni conocer su costo, ni subclasificarse en Llamada Local, Llamada Nacional y Llamada Internacional. Estas son propiedades de las llamadas que se eligieron en un momento para calcularle el costo, pero esto lo debería hacer un tercero que mira las características de la llamada y las subclasifique.

1.1.2. Heurísticas para un buen modelaje

1. Usar Objetos Inmutables.
2. Usar Objetos Completos.
3. Usar Objetos Válidos.
4. No usar Nil/Null ante la falta de parámetros. Esto lleva al mal uso de Ifs.
 - a) Usar NullObjects
 - b) Usar `ifDefinedDo: aClosure ifNone: []`

5. No usar setters (salvo que sea necesario). Usar `syncWith: anObject` (una operación atómica, usando un objeto que sabemos es válido por los puntos anteriores).
6. No usar getters que devuelvan objetos inmutables o copias, para no romper encapsulamiento.
7. Usar 1 solo `new/initialize`.

1.2. De objetos, mensajes y variables

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

Programa: conjunto de objetos que colaboran entre sí enviándose 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: sin parámetros. Ej, `objeto mensaje`
- Binarios: con un parámetro. Ej, `objeto mensaje objeto1`
- Keywords: con varios parámetros. Ej, `objeto mensajeConParam1: objeto1 param2: objeto2 ...`

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: _`

El resultado de esta la colaboración `4+3*5` es 35, porque se evalúa primero `4+3` y después el `*5`

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. Así que 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 dirige al `value` de `aVariable`.

Así que 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 asignación 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 llamadas **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
```

1.3. Polimorfismo

En Smalltalk, el objeto `1` y `true` no es lo mismo. `1` sabe responder el mensaje `+`. Así como `0` no es `false`. `true` y `false` son polimórficos respecto de los mensajes `not`, `and` y `or`.

	True	False
<code>not</code>	<code>^ false</code>	<code>^ true</code>
<code>and: aBoolean</code>	<code>^ aBoolean</code>	<code>^ self</code>
<code>or: aBoolean</code>	<code>^ self</code>	<code>^ aBoolean</code>

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 como Boolean tienen que tener una implementación propia de los métodos, lo que se hace es poner `self subclassResponsability`, que si se ejecuta tira una excepción.

¿Qué pasaría si `true` y `false` fueran instancias de una sola clase `Boolean`?

Entonces tendríamos q hacer una implementación de `not` del estilo:

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

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

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

Lo que hace al lenguaje más cerrado, y además el programador no puede ver cómo esta hecho. Como Smalltalk quiere mantener su sintaxis lo más chica posible, para que sea mas customizable al programador, se implementa como `objeto mensaje`.

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

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

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

Regresión al infinitum!

Conclusión: No se puede implementar álgebra booleana con un solo objeto!

Asique volviendo al caso de un Booleano abstracto, `ifTrue` queda implementado asi

	True	False
<code>not</code>	<code>^ false</code>	<code>^ true</code>
<code>and: aBoolean</code>	<code>^ aBoolean</code>	<code>^ self</code>
<code>or: aBoolean</code>	<code>^ self</code>	<code>^ aBoolean</code>
<code>ifTrue: aClosure</code>	<code>^ aClosure</code>	<code>^ nil</code>

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: '...'
```

1.3.1. ¿Como sacamos IFs, en pro de polimorfismo?

Hay una heurística y consiste en seguir los siguientes pasos:

1. Opcional (porque ya puede existir la jerarquía): Crear jerarquía polimórfica con una abstracción por cada if
2. Copiar closure de cada if a cada abstracción usando mensajes polimórficos
3. Ponerle nombre a la abstracción
4. Ponerle nombre al mensaje
5. Opcional (porque si ya existe la jerarquía, pueden existir los objetos): buscar objeto polimórfico
6. Reemplazar if por “objPolimorfico mensaje”

¡IMPORTANTE! Hay un límite para sacar IFs: no tiene sentido sacar el if cuando los colaboradores que participan en la condición no pertenecen al dominio del problema. Por ej,

```
CtaBancaria>>withdrawl: anAmount
(balance-anAmount) < 0 ifTrue: [ tirar excepcion ]
```

En este caso `balance-anAmount` es un número, y no pertenece al dominio del mundo bancario.

1.4. Lambdas vs. Closures vs. Full Closures

1.4.1. Closure

Cuando se ejecuta un Closure, este bindea todas las variables que tiene al contexto de ejecución que lo llamó.

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

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

[ .. ] crea un nuevo objeto, y sabe que a a la busca en el contexto del que lo creo.
```

1.4.2. Lambda (Lisp)

En los lamdas, las variables no se bindean al contexto de ejecución del que lo llama. Las definiciones de las variables se buscan en todos los contextos de ejecución.

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

Esto compila aunque `a` no este definido.

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

1.4.3. Full Closure (Smalltalk)

En los Full Closure, no solo bindea las variables al contexto de ejecución que lo llamó, sino que el return dentro del mismo hace retornar del contexto de ejecución padre.

```
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 primero sale del contexto de ejecucion
y despues quiere volver a salir desde dentro del [...].
```

Si no tuvieramos Full Closure, no podríamos 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).

1.4.4. 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 recursive. ¿Se puede hacer aca? Sí, porque puedo acceder a los contextos de ejecución, y sacar los closures que vaya apilando ahí.

Así que el whileTrue para que sea tail recursive hay que escribirlo así

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

1.5. Metamodelos

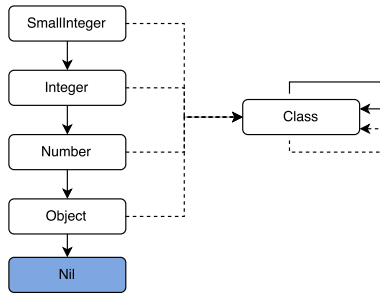


Figura 2: Metamodelo Básico

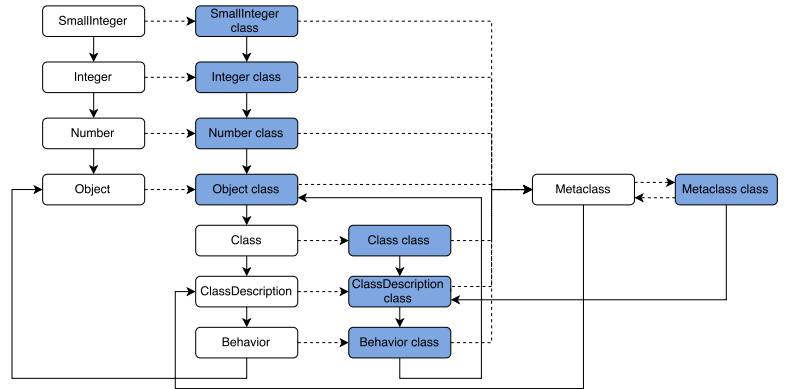


Figura 3: Metamodelo SmallTalk80

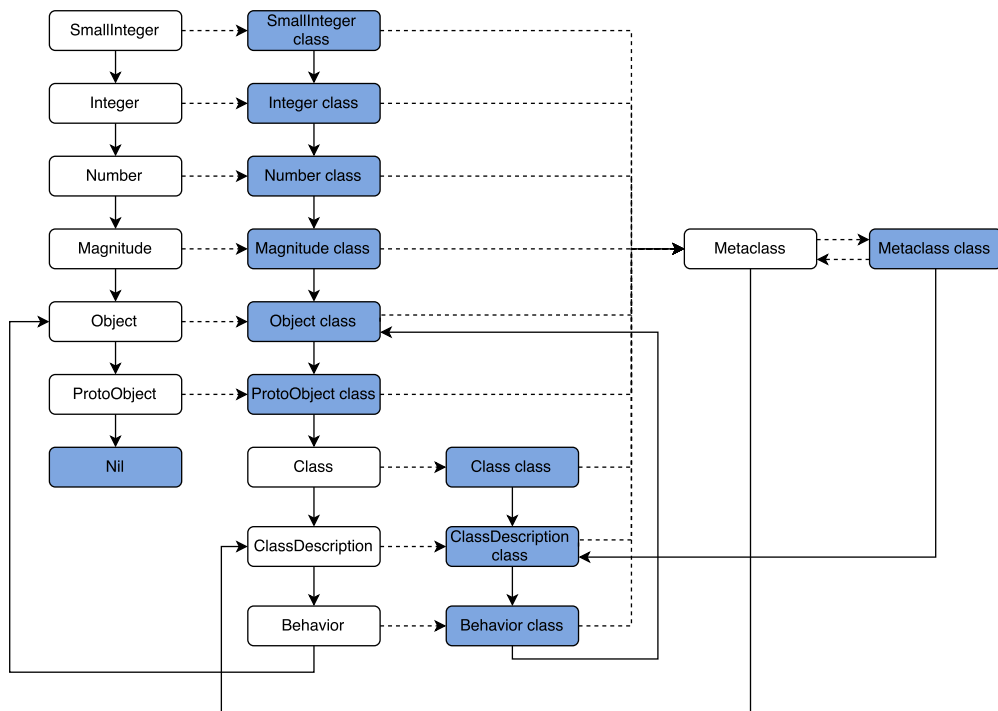


Figura 4: Metamodelo de Pharo4.0

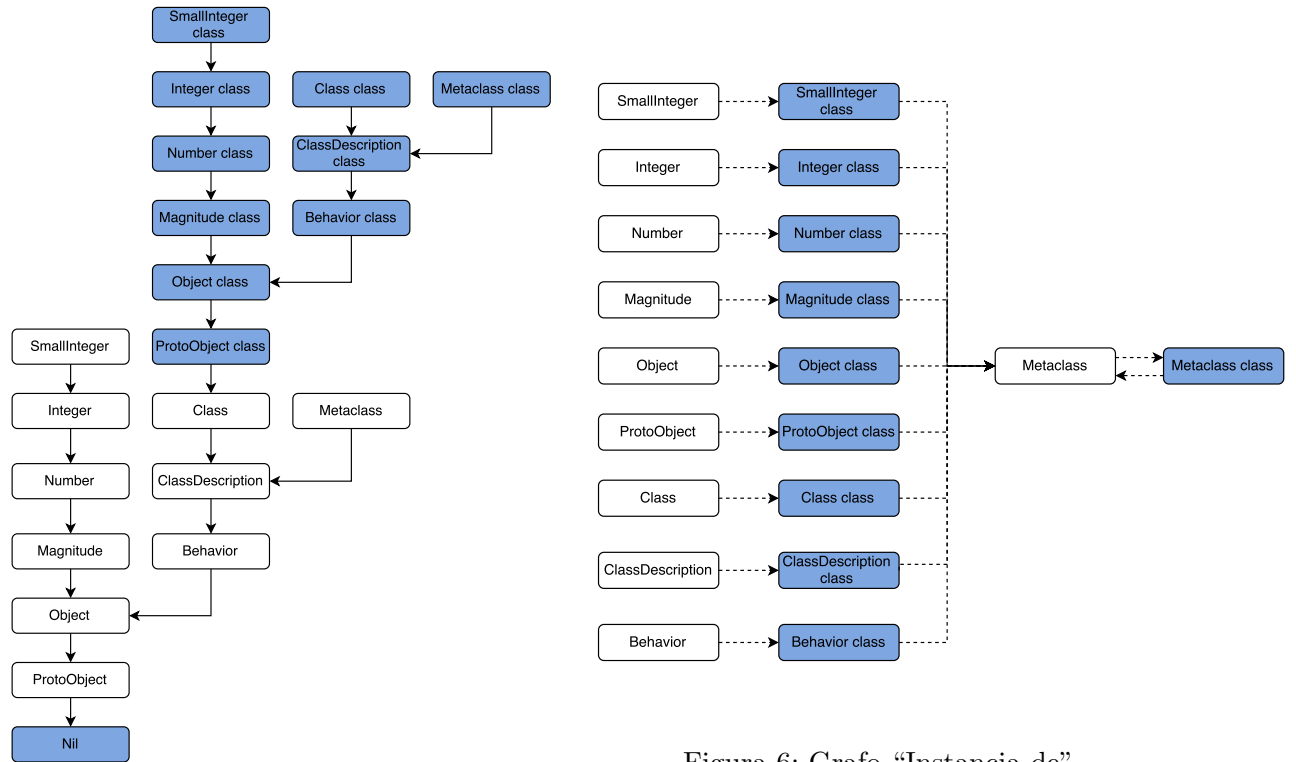


Figura 6: Grafo “Instancia de”

Figura 5: Grafo de Subclasificaciones

Figura 7: Metamodelo de Pharo4.0

Referencias: Línea punteada significa “Instancia de”, y Línea lisa “Hereda de”.

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

Si a una metaclass se le agregan variables de instancia que inicializo en su método **initialize** pero luego las modifico, debo reinicializar la metaclass 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) Siempre que se implementa **new** en una subclase, hay que llamar primero a **super new** para hacer la alocaación.
- b) Si es **Behavior** quien implementa la alocaación, ¿quién lo hace para **Object** y los que subclasifican de él?

Cuando un objeto recibe el mensaje **new**, busca su implementación en el protocolo de su clase, si este no la tiene lo busca en su padre, y así sucesivamente hasta eventualmente llegar a **Behavior** que sí lo tiene.

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

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

¿Tiene algo que ver? No, simplemente que **ProtoObject** no está preparado para ser debuggeado, por eso es que no se puede mensajear dentro de Pharo.

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?

Casos border que están mal implementados en Pharo.

2. Resumen papers

2.1. Double Dispatch

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

2.2. Fail Fast

Sacado del paper *Fail Fast* de Jim Shore.

2.3. Null Pattern

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

2.4. Object Recursion

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

2.5. Programming as Theory Building

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

2.6. Self

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