Active Variables in SmallTalk 80

Tomás Felipe Melli

July 16, 2025

Índice

1 Motivación

En sistemas como *LOOPS* y *Magpie* ya existía la noción de *active values*: valores asociados a procedimientos auxiliares ("daemons") que se ejecutan automáticamente al acceder o modificar el valor. Smalltalk-80 no tenía esta capacidad de forma nativa.

Este trabajo propone una extensión al lenguaje Smalltalk-80 para incorporar **variables activas**, minimizando el impacto sobre el programador y evitando técnicas ad-hoc.

2 ¿Qué son las variables activas?

Una **variable activa** invoca automáticamente un *daemon* cuando se accede ("get") o se modifica ("set"). Esto permite:

- Visualización automática del estado (por ejemplo en consola o UI)
- Validación de valores
- Coerción de tipos
- Registro de eventos o debugging

El enfoque subyacente se alinea con la llamada **programación orientada a valores** (*value-oriented programming*) —propuesta originalmente en el contexto de LOOPS— donde los valores no son entidades pasivas, sino que *poseen comportamiento adjunto*. Es decir, la lógica del sistema puede residir en la forma en que los valores responden al ser accedidos o modificados, sin necesidad de control explícito en el flujo del programa.

Este paradigma permite encapsular comportamientos reactivos directamente en las variables, lo cual resulta útil para desarrollar sistemas interactivos, visuales o sensibles al contexto, como interfaces gráficas o herramientas de inspección.

Daemons son los procedimientos auxiliares que se asocian a las variables activas. Existen dos tipos:

- **Get daemons**: se ejecutan automáticamente al leer la variable.
- **Set daemons**: se ejecutan automáticamente al escribir la variable.

Ambos pueden implementarse como:

- Selectores de método (ej., `randomGetDaemon:`).
- Bloques Smalltalk (ej., `[:self :av | av value]`).

Estas funciones encapsulan la lógica del acceso y permiten incluso que una variable retorne un valor computado dinámicamente (por ejemplo, un número aleatorio distinto en cada lectura).

Finalmente, una **active variable** es un objeto contenedor que gestiona un valor interno y que responde a mensajes de lectura/escritura invocando los daemons asociados. El compilador transforma cada acceso y asignación en mensajes que activan dicha lógica, haciendo transparente su uso al programador.

3 Clases definidas

- InactiveVariable: contenedor simple de un valor.
- ActiveVariable: subclase que agrega:
  - `name`: nombre del slot activo.
  - `getDaemon`, `setDaemon`: métodos invocados al acceder/modificar.
- SpecificActiveVariable: variante que usa `blocks` en lugar de métodos como daemons.

4 Declaración y uso

Clases con variables activas

Las clases definen variables activas con:

```
Object subclass: #Test
   instanceVariableNames: 'randomValue'
   activeVariables: '(randomValue randomGetDaemon: randomSetDaemon:)'
```

4.1 Temporales activas

Dentro de un método también se pueden definir:

```
| x y (av1 getFn1 putFn1) |
```

5 Ejemplo funcional

```
Test methodsFor: 'daemons'
randomGetDaemon: av
    ^ av value next

Test methodsFor: 'examples'
test1
    randomValue := Random new.
    ^ Array with: randomValue with: randomValue with: randomValue.
```

Resultado: tres números distintos, porque el daemon retorna un nuevo valor cada vez.

6 Daemons

6.1 Get Daemons

- Método: `getDaemon: av`
- Bloque: `[:parent :activeVar | activeVar value]`

6.2 Set Daemons

- Método: `setDaemon: av setTo: newVal`
- Bloque: `[:parent :activeVar :newVal | activeVar value: newVal]`

Activación y control

- Activar: `makeActiveVariable:`
- Desactivar: `makeInactiveVariable:`
- Cambiar daemons: `set:getDaemon:`, `set:setDaemon:`, etc.

Todo puede hacerse a nivel de clase o de instancia.

7 Visualización: Dials and Gauges

Inspirado en LOOPS, los autores implementan un gauge visual simple (`LCD`):

```
r := Test new.
lcd := LCD onVar: 'randomValue' of: r.
lcd open.

r randomValue: 10.
r randomValue: 50.
```

Se planean otros instrumentos: termómetros, escalas, velocímetros, etc.

8 Advised Methods

Permiten insertar código antes o después de un método sin recompilarlo. Inspirado en Interlisp.

- `advise`: abre editor para agregar código auxiliar.
- `unadvise`: elimina el código agregado.

9 Aspectos técnicos

- El **compilador** Smalltalk se modifica para insertar los mensajes correspondientes.
- Las clases `Object`, `Behavior`, `ClassDescription`, etc. se extienden para manejar activación.
- Las instancias deben crearse con `basicNew` para garantizar inicialización correcta.

10 Conclusión

Este trabajo aporta dos extensiones clave al entorno Smalltalk:

1. **Variables activas**, que habilitan una programación reactiva orientada al estado.
2. **Métodos aconsejados** (*advised*), que permiten agregar lógica auxiliar sin modificar el método original.

Ambas extensiones anticipan características modernas como:

- Programación orientada a aspectos
- Observabilidad y tracing
- Programación reactiva y declarativa

11 Recorte copado del texto

We distinguish between active variables and general active objects. Active objects are objects which effect an operation which itself is not a part of the obvious flow of control which is controlling the object. As an example, an active object may invoke a daemon whenever a message is received. We have been unsuccessful in defining a general technique to activate an arbitrary object, therefore this implementation does not include support for active objects. Active variables are variables which invoke a daemon whenever they are accessed or changed. Our implementation currently supports active instance variables for objects and active temporary variables for methods. We also include a technique for advising methods, similar to the *advise* package of Interlisp [Xerox83], by allowing the insertion of arbitrary code before a method executes, or afterwards, or both.

1.1 Creating active variables

Classes with active variables must have the ability to record which instance variables are active. For this reason a slightly different metaclass is created when a class with active instance variables is defined. The way to declare a class to have an active variable (AV) is to include a line in the class definition message following "instanceVariables:" that looks like:

activeVariables: '(av1 getFn1 putFn1) (av2 getFn2 putFn2)'

The first field in the parenthesized list is the name of the AV, the second field is the message to be sent self when the value of the AV is accessed, and the third field is the message to be sent when the AV is changed.

To declare a temporary active variable in a method the temporary variables declaration of the method definition should look like: