Resumen Metamodelo, Method Lookup y Excepciones

Tomás Felipe Melli

December 8, 2025

Índice

1 Metamodelo

El metamodelo de **Smalltalk** (*Smalltalk Object Model*) es el conjunto de directivas o reglas que permiten construir el sistema.

- **Rule 1.** Everything is an object.
- **Rule 2.** Every object is an instance of a class.
- **Rule 3.** Every class has a superclass.
- **Rule 4.** Everything happens by message sends.
- **Rule 5.** Method lookup follows the inheritance chain.
- **Rule 6.** Every class is an instance of a metaclass.
- **Rule 7.** The metaclass hierarchy parallels the class hierarchy : Una clase puede tener comportamiento propio (de clase) porque cada clase es instancia de su propia metaclase, y la jerarquía de metaclases copia 1-a-1 la jerarquía de clases, permitiendo herencia y redefinición de ese comportamiento.
- **Rule 8.** Every metaclass inherits from Class and Behavior.
- **Rule 9.** Every metaclass is an instance of Metaclass.
- **Rule 10.** The metaclass of Metaclass is an instance of Metaclass : El impacto de esta regla es que el metamodelo es circular. Para cerrar el modelo de objetos sin excepciones: evita una regresión infinita y mantiene que todo sea objeto y tenga clase.

Together, these 10 rules complete Smalltalk's object model.

2 Method Lookup, VTBL y Dispatch Dinámico

2.1 ¿Qué es Method Lookup?

Method lookup es el algoritmo que se utiliza para buscar un método a partir de un **receptor** y un **mensaje (selector)**. Es decir, dado un objeto (receptor) y un mensaje/método, el sistema debe decidir **qué implementación ejecutar**.

2.2 Lenguajes Dinámicamente Tipados

En los lenguajes dinámicamente tipados se utilizan estructuras más flexibles para resolver el envío de mensajes.

Dispatch Table Search (DTS)

La búsqueda del método se realiza en una **tabla o diccionario de métodos**. La clave típica es el par:

$$(\text{clase del receptor}, \text{nombre del método})$$

Global Lookup Cache (GLC)

Es una **cache global** de búsquedas de métodos.

- Guarda resultados de búsquedas previas.
- Si hay *hit*, se usa directamente.
- Si hay *miss*, se vuelve a realizar la búsqueda.
- La clave es: clase del receptor + nombre del método.

Inline Cache (IC)

Es una cache **por cada punto de envío del mensaje** (por cada call site).

- Guarda el tipo del receptor.
- Guarda el método resuelto.
- Si el tipo cambia, la cache se invalida y se vuelve a buscar.

Polymorphic Inline Cache (PIC)

Es una extensión del inline cache:

- Guarda **varios tipos posibles de receptores**.
- Cada tipo se asocia a su método correspondiente.
- Se usa cuando un mismo punto de envío recibe objetos de distintas clases.

2.3 Lenguajes Estáticamente Tipados: VTBL (Virtual Table)

En los lenguajes estáticamente tipados se utiliza la **Virtual Table (VTBL)**.

- Cada clase tiene su propia VTBL.
- La dimensión de la VTBL depende de la jerarquía de herencia.
- En la tabla quedan almacenadas las direcciones de los métodos.
- En tiempo de ejecución:
  - Se accede a la VTBL del objeto.
  - Se usa un **offset fijo** para obtener el método.

Ejemplo de Organización de VTBL

```
Clase A -> VTBL[0] -> m1

Clase B -> VTBL[0] -> m1
           VTBL[1] -> m2
           VTBL[2] -> m4

Clase C -> VTBL[0] -> m1
           VTBL[1] -> m2
           VTBL[2] -> m3
```

Ejemplo de Ejecución con Polimorfismo

```
A* a = new A();
a->m1();   // usa a->VTBL[0]

B* b = (B*) new C();
b->m1();   // usa b->VTBL[0] pero ejecuta la versión de C
```

El tipo estático de la variable es uno, pero el tipo dinámico real del objeto es el que decide qué método se ejecuta. Esto se resuelve mediante la VTBL.

Resumen

Lenguajes Estáticamente Tipados (C++, Java):

- Usan VTBL.
- Acceso por índice.
- Muy rápido.
- Estructura fija por clase.

Lenguajes Dinámicamente Tipados (Smalltalk, Python, Ruby):

- Usan Dispatch Table Search.
- Global Lookup Cache.
- Inline Cache.

- Polymorphic Inline Cache.
- Mayor flexibilidad.
- Mayor costo, optimizado mediante caches.

2.4 Funcionamiento del Method Lookup en Smalltalk

En Smalltalk, todo es envío de mensajes, y cada vez que se envía uno el sistema debe decidir qué método ejecutar. Para hacerlo rápido, usa varios niveles de búsqueda con caches.

El algoritmo de **Method Lookup** funciona de la siguiente manera:

1. Búsqueda en el Polymorphic Inline Cache (PIC)

El **PIC** está embebido en el código del punto de envío del mensaje (call site).

- Contiene varias clases posibles del receptor.
- Contiene el método correspondiente a cada clase.
- Internamente incluye al Inline Cache (IC).
- Está especializado para ese mensaje específico en ese lugar del programa.

Proceso:

- Se compara la clase real del objeto receptor con las clases cacheadas.
- Si hay coincidencia, se ejecuta el método directamente.
- Si no hay coincidencia, se continúa con la búsqueda en la GLC.

Este es el nivel más rápido del algoritmo, ya que evita búsquedas en estructuras más costosas.

2. Búsqueda en la Global Lookup Cache (GLC)

La **GLC** es una cache global para todo el sistema, no asociada a un único punto de envío.

La clave utilizada es la tupla:

$$(\text{selector}, \text{clase}, \text{método})$$

Además, utiliza el llamado **método de tres pruebas** para acelerar la comparación de dichas tuplas.

Proceso:

- Se consulta si el mensaje ya fue resuelto previamente para esa clase.
- Si hay *hit*, se obtiene el método y se ejecuta.
- Si hay *miss*, se continúa con la búsqueda en la DTS.

3. Búsqueda en la Dispatch Table Search (DTS)

La **DTS** es el mecanismo completo de búsqueda sin utilizar caches.

Proceso de búsqueda:

1. Se busca el mensaje en el **Method Dictionary** de la clase del receptor.
2. Si no se encuentra, se busca en el **Method Dictionary de la superclase**.
3. Este proceso se repite hasta recorrer toda la jerarquía de clases.

Si el método es encontrado:

- Se cachea el resultado en la GLC.
- Se ejecuta el método.

4. Envío de `doesNotUnderstand:`

Si el mensaje no se encuentra en ninguna clase de la jerarquía:

1. Se le envía automáticamente al objeto receptor el mensaje `doesNotUnderstand:`.
2. Este mensaje vuelve a ejecutar todo el algoritmo de Method Lookup:
    - PIC
    - GLC
    - DTS

3 Excepciones

3.1 Explicación Pragmática

Históricamente, el manejo de errores se realizaba mediante la **técnica de código de retorno**, donde las funciones devolvían códigos indicando éxito o error. Esta técnica presenta varios problemas:

- Genera código repetido.
- Es propensa a errores (olvido de validar el código).
- Mezcla la lógica del programa con la administración del error.
- No está estandarizada (excepto en lenguajes como Go).

Las **excepciones** surgen como una solución para eliminar ese código repetido, separando la lógica normal del flujo del manejo de errores.

3.2 Explicación Conceptual

Las excepciones están basadas en la técnica de **Design by Contract** (Bertrand Meyer). Un contrato es un acuerdo de obligaciones entre objetos que, si se cumple, garantiza un resultado definido.

En objetos, un contrato se compone de:

- **Pre-condiciones**
- **Post-condiciones**
- **Invariantes**

Las **excepciones se utilizan para indicar que un contrato no se cumple**.

3.3 Contrato

Un contrato define las condiciones que deben cumplirse entre los objetos que colaboran para que el sistema funcione correctamente.

Pre-condiciones

Son condiciones que deben cumplirse antes de ejecutar un método. Ejemplo: el monto a extraer de una cuenta debe ser ≥ 0.

Post-condiciones

Son condiciones que deben cumplirse luego de ejecutar un método. Ejemplo: luego de una extracción,

$$saldo = saldo_{previo} - monto$$

Invariantes

Son condiciones que siempre deben cumplirse en una clase. Ejemplo: el saldo de una cuenta bancaria nunca puede ser negativo.

3.4 Árbol y Stack de Ejecución

El **árbol de ejecución** representa la estructura de llamadas entre métodos. El **stack de ejecución** representa las llamadas activas en tiempo de ejecución.

Cuando se lanza una excepción:

- Se interrumpe la ejecución normal.
- Se recorren los métodos del stack buscando un handler adecuado.
- Si no se encuentra ningún handler hasta la raíz, la excepción queda **no handleada**.

3.5 ¿Cuándo levantar una Excepción?

- Cuando se rompe un contrato, especialmente una pre-condición.
- No deben usarse excepciones como control de flujo.
- Ejemplo válido: división por cero.
- Ejemplo inválido: salir de un bucle con una excepción.

3.6 ¿Quién Verifica los Contratos?

Escuela C:

- El emisor del mensaje verifica las pre-condiciones.
- Ventaja: performance.
- Desventaja: código repetido e inseguridad.

Escuela Lisp:

- El receptor del mensaje verifica las pre-condiciones.
- Ventaja: validaciones centralizadas y mayor seguridad.
- Desventaja: posible pérdida de rendimiento.

3.7 ¿Quién Informa y Quién Handlear?

Quién informa:

- Generalmente los métodos más bajos del árbol de ejecución.
- Son los que realmente detectan que el contrato se rompió.

Quién handlea:

- Los métodos más altos del árbol de ejecución.
- Tienen más contexto para decidir qué hacer.

3.8 Cómo se Pueden Handlear las Excepciones

Implementaciones cerradas:

- Se termina el bloque donde ocurrió la excepción.
- Se pasa al siguiente handler.

Implementaciones abiertas:

- Se termina el bloque donde ocurrió la excepción.
- Se pasa al siguiente handler.
- Se puede reintentar el bloque.
- Se puede continuar con otra colaboración.

3.9 Cómo se Deben Handlear

- **Solo se debe handlear una excepción si se puede resolver la ruptura del contrato.** Es decir, solo se debe capturar una excepción si realmente se puede restaurar un estado válido del sistema o tomar una decisión útil a partir del error.
- **No handlear excepciones si no se puede hacer nada útil.** Si no es posible recuperarse del error (por ejemplo, una división por cero o un acceso fuera de rango), la excepción debe propagarse y no ser capturada innecesariamente.
- **Nunca ocultar excepciones.** Capturar una excepción y no hacer nada con ella provoca pérdida de información, dificulta el debugging y puede dejar al sistema en un estado inconsistente.
- **Las excepciones deben handlearse en la raíz del árbol de ejecución.** Los niveles superiores del sistema tienen más contexto sobre la operación global y pueden decidir correctamente cómo actuar frente al error.

3.10 Qué Excepción Informar

Existen tres estrategias:

1. Un tipo de excepción por cada condición.
2. Usar siempre el mismo tipo de excepción.
3. Un enfoque mixto.

Solo deben crearse nuevas excepciones si estas van a ser handleadas.

3.11 Try–Catch vs `on:do:` en Smalltalk

En lenguajes como Java o C++ se usa:

```
try { ... } catch (E e) { ... }
```

En Smalltalk se usa:

```
[ bloque ] on: Excepcion do: [ handler ]
```

Ambos mecanismos representan el mismo concepto:

- Definen una condición de handleo.
- Capturan una excepción de un tipo dado.
- Ejecutan un bloque de recuperación.

3.12 Checked vs Unchecked Exceptions

Las excepciones pueden clasificarse en **checked** y **unchecked** según si el lenguaje obliga o no a declararlas y manejarlas.

Checked Exceptions:

- Son excepciones que el compilador obliga a declarar o capturar.
- Representan errores esperables del entorno (archivos, red, datos externos).
- Hacen explícito qué operaciones pueden fallar.
- Generan fuerte acoplamiento entre módulos.

Unchecked Exceptions:

- No obligan a ser declaradas ni capturadas.
- Representan principalmente errores de programación.
- Suelen indicar fallas de lógica (índices fuera de rango, referencias nulas).
- Generan menor acoplamiento.

El uso de checked exceptions es debatido porque:

- Fuerzan recompilación si se modifica una excepción.
- Ofuscan el código con declaraciones obligatorias.
- No siempre permiten una recuperación real del error.

3.13 Implementación de Excepciones

Según disponibilidad:

- Cerrada: Java, C#, Python, Ruby.
- Abierta: Smalltalk, Common Lisp, Self.

Según tratamiento del stack:

- Deshacer stack: Java, C#, Python.
- Mantener stack: Smalltalk, Lisp.

3.14 Código de Retorno vs Excepciones

Las excepciones:

- Separan el flujo normal del manejo de errores.
- Evitan código repetido.
- Son más expresivas que los códigos de retorno.

3.15 Conclusiones

- Las excepciones indican ruptura de contratos.
- No deben ocultarse.
- Se deben crear por demanda.
- Se deben handlear en niveles altos del árbol de ejecución.
- Usar objetos válidos incrementa la seguridad del sistema.