Resumen Papers para el Primer Parcial

Tomás Felipe Melli

May 2, 2025

Índice

1 Programming as Theory Building

El paper, escrito por Peter Naur en 1985 propone una reinterpretación de la función del programador en la industria. Desarticula la noción de programador como engranaje reemplazable en la construcción del software (que en la visión tradicional es un "text writer"). Naur critica la postura tradicional y propone pensar al programador como un punto de partida en la construcción de una teoría.

1.1 Program Life - Death - Revival

Naur enfatiza la importancia de que el equipo de programadores esté involucrado en el ciclo de vida del programa, donde "Life", "Death" y "Revival" representan diferentes etapas en la existencia de un programa:

1.1.1 Life

El programa sigue "vivo" mientras el equipo original de programadores, que posee la teoría del programa, esté en control de él. Este equipo tiene la capacidad de modificar el programa de manera inteligente, basándose en la teoría que desarrollaron.

1.1.2 Death

Un programa se considera "muerto" cuando el equipo original, que comprendía la teoría detrás del programa, se disuelve. El programa puede seguir funcionando, pero se vuelve cada vez más difícil de modificar o mejorar, ya que no hay un equipo capaz de responder inteligentemente a las demandas de cambios.

1.1.3 Revival

El "revival" ocurre cuando un nuevo equipo intenta reconstruir la teoría original para retomar el control del programa. Sin embargo, el texto sostiene que el revival es un proceso costoso, difícil y, en muchos casos, puede resultar en una teoría renovada que difiera de la original, lo que podría generar discrepancias con el texto del programa.

2 The Psycopathology of Everyday Things

En el primer capítulo del libro **The Design of Everyday Things**, Donald Norman introduce la problemática diaria y cotidiana de no entender los objetos con los que lidiamos. Estos son como el lavarropas, el microondas, teléfono, auto,... entre otros aparatos con marcada complejidad en términos de la cantidad de utilidades que presentan. Esta problemática, él argumenta que está íntimamente vinculada al diseño de los mismos. Menciona y desarrolla ciertos principios que serán sus argumentos principales para sostener que *The result is a world filled with frustration, with objects that cannot be understood, with devices that lead to error.*

2.1 Visibility

Norman presenta la **visibilidad** como el principio fundamental del diseño de las cosas. La define como *The correct parts must be visible, and they must convey the correct message.* Postula sobre los *natural signs* que es la manera natural de interpretar la forma de utilizar a los objetos en la que no es necesario ser consciente de ellos mismos. Concluyendo que se trata del *natural design.*
Donald introduce a modo de ejemplo las problemáticas que conlleva la falta de visibilidad en objetos específicos y hace uso de un concepto importante que es el *mapping.* Este concepto se refiere a *what you want to do and what appears to be possible.* En esta sección habla del *dishwasher* y cómo su mujer sólo se memorizó una sola configuración y trataba de ignorar el resto. Como crítica a los diseñadores dice: *The user needs help. Just the right things have to be visible; to indicate what parts operate and how, to indicate how the user is to interact with the device. Visibility indicates the mapping between intended actions and actual operations.* [...] *It is the lack of visibility that makes so many computer-controlled devices so difficult to operate. And it is the excess of visibility that makes the gadget-ridden, feature-laden modern audio set or video cassette recorder so intimidating.*

2.2 Affordances

El término **affordance** lo describe como lo percibido y las propiedades reales del objeto. (*the perceived and actual properties of the thing*). Norman dice que principalmente aquellas propiedades fundamentales que determinan cómo podría usarse. Él dice que las *affordances provide strong clues to the operations of things.* Insiste en que, cuando estas se toman bien en serio en el diseño, el usuario ya sabe qué tiene que hacer sin necesidad de fotos ni instructivos, y en caso de que el objeto sea simple y tenga dibujos, es porque fallaron en el diseño.

2.3 Causality

Norman propone tener en cuenta la psicología de la causalidad. Es decir, *something happens right after an action appears to be caused by that action.* Habla sobre que cuando la acción no tiene causa aparente el usuario concluye que la acción es inefectiva.

2.4 Conceptual Models / Mental Models

Es un modelo mental básicamente. La idea es simular la operación del objeto. Menciona que los conceptos de *affordances, constraints y mapping* entran en juego para poder predecir los efectos de nuestras acciones. Con esto en mente menciona que no es necesario entender absolutamente todo lo que compone a un objeto (capa de abstracción).
The mental model of a device is formed largely by interpreting its perceived actions and its visible structure. I call this visible part of the device the system image. When the system image is incoherent or innappropriate ..., then the user cannot easily use the device.

2.5 Mapping

Mapping is a technical term meaning the relationship between two things, in this case between the controls and their movements and the result in the world. Define el tipo *natural mapping* que ya hablamos algo, se repite a sí mismo. Es *taking advantage of physical analogies and cultural standards, leads to immediate understanding.* Esto introduce *the problem of determining the "naturalness" of mappings is difficult, but crucial.*

2.6 Immediate Feedback

Concepto clave. Norman dice *The lack of immediate feedback for the actions does not help.* Su definición es: *sending back to the user information about what action has actually been done, what result has been accomplished.*

2.7 The Paradox of Technology

Norman argumenta que, a medida que la tecnología avanza, los dispositivos tienden a volverse más complejos, lo que puede llevar a una curva de complejidad en forma de U: comienzan siendo complejos y difíciles de usar, luego se simplifican a medida que la industria madura, pero eventualmente, al agregar más funciones y capacidades, vuelven a volverse más complejos . Para abordar esta paradoja, Norman enfatiza la importancia de un buen diseño que haga la complejidad manejable. Esto implica crear dispositivos que sean intuitivos, con controles visibles y retroalimentación clara, para que los usuarios puedan comprender y utilizar la tecnología de manera efectiva sin sentirse abrumados por su complejidad inherente.

3 No Silver Bullet —Essence and Accident in Software Engineering

Frederick P. Brooks, Jr. fue ingeniero de software y científico de la computación yankee y escribió este ensayo en 1986.

3.1 Abstract

Para mejorar la productividad en el desarrollo de software, ya no basta con optimizar tareas técnicas (accidentales). Es hora de enfocarse en las tareas esenciales: diseñar estructuras conceptuales complejas. Se sugiere reutilizar soluciones existentes, usar prototipos rápidos, desarrollar el software de forma progresiva y formar a diseñadores conceptuales destacados.

3.2 Introduction

El desarrollo de software se compara con los hombres lobo de las pesadillas: algo que parece simple y familiar puede transformarse en un monstruo de retrasos, sobrecostos y errores. Por eso, muchos buscan una *silver bullet* mágica que solucione todos los problemas del desarrollo de software, como ha sucedido con los avances en hardware.

Sin embargo, no existe una solución única —ni tecnológica ni de gestión— que prometa mejorar radicalmente la productividad o la calidad del software. Esta visión no es pesimista, sino realista: aunque no habrá avances milagrosos, sí hay innovaciones prometedoras que, con esfuerzo y disciplina, pueden generar mejoras importantes.

3.3 Does It Have To Be Hard? – Essential Difficulties

El desarrollo de software es difícil por su naturaleza esencial, no por limitaciones técnicas actuales. La dificultad principal está en especificar, diseñar y probar estructuras conceptuales complejas, no en codificarlas.

3.3.1 Complexity

La **complejidad** es una propiedad esencial del software, no accidental. A diferencia de otros sistemas creados por el ser humano, el software:

- No tiene partes repetidas; cada componente suele ser único.
- Sus elementos interactúan de manera no lineal, incrementando la dificultad al crecer el sistema.

Esto genera múltiples problemas:

- Dificultad para comunicarse dentro del equipo.
- Fallos, retrasos y sobrecostos.
- Imposibilidad de prever todos los estados del sistema.
- Dificultades para mantener o extender el software.
- Problemas de seguridad debido a estados no visualizados.

3.3.2 Conformity

La **conformidad** es otra fuente esencial de dificultad. El software debe adaptarse a múltiples sistemas y normas humanas ya existentes, que:

- No siguen una lógica común.
- Son arbitrarios y cambiantes.

3.3.3 Changeability

El software cambia constantemente, porque:

- Define la función del sistema, que está sujeta a evolución.
- Es maleable y fácil de modificar.

Todo software exitoso termina siendo modificado, ya que:

- Se usa en contextos diferentes a los previstos.
- El entorno cambia (hardware, leyes, etc.).

3.3.4 Invisibility

El software es invisible: no tiene una representación geométrica clara. Las estructuras superpuestas (flujo de datos, control, relaciones) no son fácilmente visualizables, lo que:

- Dificulta el diseño mental.
- Complica la comunicación entre personas.

3.4 Past Breakthroughs Solved Accidental Difficulties

3.4.1 High-level languages

Lenguajes de alto nivel mejoraron hasta cinco veces la productividad. Permitieron trabajar con abstracciones en lugar de detalles de bajo nivel.

3.4.2 Time-sharing

Reduciendo el tiempo de espera entre escritura y prueba de código, el *time-sharing* mejoró la concentración del programador.

3.4.3 Unified programming environments

Entornos como Unix o Interlisp ofrecieron:

- Librerías integradas.
- Formatos de archivo unificados.
- Herramientas comunes.

3.5 Hopes for the Silver

3.5.1 Ada and other high-level language advances

Ada representa una mejora evolutiva en lenguajes, pero no es una *silver bullet*.

3.5.2 Object-oriented programming

La orientación a objetos mejora la claridad del diseño, pero no reduce la complejidad esencial.

3.5.3 Artificial intelligence

Brooks es escéptico. El principal reto del software es decidir qué hacer, no cómo expresarlo.

3.5.4 Expert systems

Ventajas:

- Separación entre lógica e información.
- Motor de inferencia reutilizable.
- Reglas modificables.

Problemas:

- Difícil extraer conocimiento de expertos.
- Las reglas deben seguir la estructura del software.

3.5.5 Automatic programming

Brooks cree que la programación automática rara vez especifica el problema, sino la solución. Funciona solo en problemas bien parametrizados.

3.5.6 Graphical Programming

Critica:

- Diagramas de flujo como mala abstracción.
- Pantallas con espacio limitado.
- Software difícil de representar visualmente.

3.5.7 Program verification

Verificar programas es útil, pero requiere mucho trabajo y no evita todos los errores. Lo más difícil es definir especificaciones correctas.

3.5.8 Environments and tools

Las herramientas ya han mejorado bastante. Lo más prometedor sería integrar bases de datos con información del sistema y el proyecto.

3.5.9 Workstations

Aunque más potentes, las estaciones de trabajo no reducen el tiempo de pensamiento del programador.

3.5.10 Buy versus build

La mejor estrategia es comprar software en lugar de construirlo. Hoy hay herramientas baratas y poderosas que permiten a los usuarios resolver problemas sin programar.

3.5.11 Requirements refinement and rapid prototyping

Lo más difícil es definir qué debe hacer el software. El *prototipado rápido* permite refinar requerimientos de forma iterativa y efectiva.

3.5.12 Incremental developmentgrow, not build, software

Brooks propone hacer crecer el software como un organismo:

- Empezar con una estructura que funcione (aunque haga poco).
- Agregar capacidades de forma incremental.
- Fomentar el diseño *top-down* y la moral del equipo.

3.5.13 Great designers

La clave está en los grandes diseñadores:

- Pueden hacer mejores soluciones con menos esfuerzo.
- Marcan una diferencia de hasta un orden de magnitud.

Recomendaciones:

- Valorar a los diseñadores tanto como a los gerentes.
- Identificar talento técnico temprano.
- Dar mentorías y liderazgo técnico.

4 Self: The Power of Simplicity

David Ungar y **Richard V. Heng** fueron los creadores de *Self* en *Sun Microsystems*, y este paper fue escrito por uno de ellos (*David*) junto con **Randall B. Smith** en 1987.

Self es un lenguaje orientado a objetos diseñado para fomentar el **exploratory programming** —el proceso de explorar el problema mientras se desarrolla el código, experimentando y aprendiendo de forma incremental. Se basa en un número simple y concreto de ideas: **Prototipos, Slots** y **Comportamiento**. Los *prototipos* combinan herencia e *instantiation* para ofrecer un espacio de trabajo simple y flexible. Los *slots* unifican variables y procedimientos en una única construcción. La particularidad de *Self* es que no diferencia entre estado y comportamiento.

4.1 Introducción

Al igual que *Smalltalk-80*, *Self* está pensado para impulsar el **exploratory programming**, por eso incluye *runtime typing* y *Automatic Storage Reclamation* (gestión automática de memoria). Se distingue de *Smalltalk* en que no utiliza ni clases ni variables, sino **prototipos** para la creación de objetos. Los objetos acceden a su información mediante el envío de mensajes a `self`, de ahí el nombre del lenguaje.

4.2 Principios de Diseño

4.2.1 Messages-at-the-Bottom

La operación fundamental es el *pasaje de mensajes*. No hay variables, sólo *slots* que contienen objetos que retornan a sí mismos.

4.2.2 Occam's Razor

Economía conceptual. El diseño de *Self* omite clases y variables. Cualquier objeto puede cumplir el rol de instancia o de espacio de memoria compartida. No hay distinción entre acceder a una variable y pasar un mensaje.

Self, al igual que *Smalltalk*, no incluye estructuras de control en el *language kernel*. Utiliza **polimorfismo y closures** para manejar el control dentro del lenguaje.

A diferencia de *Smalltalk*, en *Self* los objetos, closures y procedimientos se representan como **prototipos de registros de activación** (activation records). Estos registros almacenan la información relacionada con la ejecución de un procedimiento y son usualmente gestionados en la pila. También contienen sus variables y el entorno.

4.2.3 Concreteness

En un lenguaje basado en clases, los objetos se crean instanciando una clase. En *Self*, los objetos se crean clonando un prototipo. Cualquier objeto puede ser clonado.

4.3 Prototipos: Combinando Clases e Instancias

En *Smalltalk*, cada objeto contiene un puntero a su clase, que define su formato y comportamiento. En *Self*, los objetos contienen *slots* con nombres que almacenan tanto estado como comportamiento. Si un objeto recibe un mensaje y no tiene un *slot* coincidente, la búsqueda continúa a través de su puntero a su *parent*, implementando herencia.

Por ejemplo, un objeto `point` puede tener *slots* para `x` e `y`, mientras que su *parent* define operaciones comunes como suma o resta.

4.4 Comparación entre Prototipos y Clases

4.4.1 Relaciones más simples

En lenguajes con clases existen dos relaciones:

- **Es un**: instancia de una clase.
- **Tipo de**: subclase de otra clase.

En *Self*, solo existe una relación: **hereda de**, que describe cómo los objetos comparten comportamiento y estado.

4.4.2 Creación por Copia

La creación de nuevos objetos a partir de prototipos se logra mediante una operación simple: la *copia*, usando la metáfora biológica del clonaje. En cambio, la creación de objetos a partir de clases se realiza mediante *instanciación*, lo que implica la interpretación de la información de formato de una clase.

4.4.3 Ejemplos de Módulos Preexistentes

Los prototipos son más concretos que las clases porque son ejemplos de objetos, en lugar de descripciones de formato e inicialización. Estos ejemplos pueden ayudar a los usuarios a reutilizar módulos, ya que los hacen más fáciles de entender. En un sistema basado en prototipos, el usuario puede examinar un representante típico, en lugar de tener que interpretar su descripción.

4.4.4 Soporte para Objetos Únicos

Self ofrece un marco que facilita la inclusión de objetos únicos con su propio comportamiento. Dado que cada objeto tiene *slots* con nombres, y estas pueden almacenar tanto estado como comportamiento, cualquier objeto puede tener *slots* o comportamientos únicos.

4.4.5 Eliminación de la Meta-Regresión

En los sistemas basados en clases, un objeto no es autosuficiente; necesita otro objeto (su clase) para expresar su estructura y comportamiento, lo que lleva a una regresión infinita de metaclases. En los sistemas basados en prototipos, un objeto puede incluir su propio comportamiento, eliminando esta **meta-regresión**.

4.4.6 Combinando Prototipos e Herencia

Self resuelve el problema de la meta-regresión combinando prototipos e herencia: coloca el comportamiento compartido en un objeto *padre* que es común para todos, incluidos los prototipos.

4.5 Combinando Estado y Comportamiento

En *Self*, no se accede directamente a las variables; en su lugar, los objetos envían mensajes para acceder a los datos almacenados en *slots* nombrados. Para modificar el valor de "x", en lugar de usar una asignación, el objeto envía un mensaje como "x: 17" para actualizar el valor de la *slot* correspondiente.

4.6 Closures

La comunidad de Scheme ha obtenido excelentes resultados utilizando **closures** (o expresiones lambda) como base para las estructuras de control. Esta capacidad es crucial para cualquier lenguaje que soporte tipos de datos abstractos definidos por el usuario.

4.6.1 Variables Locales

En *Self*, los *closures* y procedimientos usan los *slots* de los objetos para almacenar variables locales.

4.6.2 Enlace al Entorno

En *Self*, los *closures* usan un enlace al "padre" para resolver referencias a variables fuera de su alcance.

5 Polymorphic Hierarchy

Bobby Woolf es un ingeniero de software especializado en arquitectura de software empresarial, integración de sistemas y cloud. Escribió este paper en mayo del 1996. Él comenta que muchas de las funciones que escribe no son muy originales, ya que incluyen muchos getters, setters, métodos de inicialización y otros que simplemente implementan comportamientos ya definidos en una superclase. Sin embargo, destaca que estos métodos repetitivos, especialmente los que subimplementan métodos de una superclase, son fundamentales para el uso efectivo del polimorfismo. Cuando se usan de forma consistente y deliberada, los métodos polimórficos permiten crear clases polimórficas, lo que a su vez da lugar a jerarquías polimórficas. El autor introduce el concepto de "Template Class".

5.1 Reusing Method Descriptions

Acá hace uso del método `printOn:` como ejemplo para ilustrar cómo subimplementa métodos de superclases. En VisualWorks, `printString` se implementa en la clase `Object` usando `printOn:`, el cual es un método genérico. Él lo subimplementa en sus propias clases para personalizar la salida, añadiendo detalles sobre la clase y la instancia.

En lugar de volver a escribir comentarios extensos, simplemente documenta su implementación con algo como "See superimplementor", porque su método básicamente hace lo mismo que el de la superclase.

Además, señala que su implementación está en el mismo protocolo de métodos (`printing`) que el original. Esto refuerza la idea de que su método tiene el mismo propósito, y que las subimplementaciones deben mantenerse coherentes con el uso y la documentación del método padre.

5.2 One defining implementor

Cuando el autor escribe todos los métodos que responden a un mismo mensaje en una jerarquía de clases, solo comenta la implementación principal, que usualmente está en la superclase. Aunque esta implementación puede ser muy simple o genérica (como devolver self, lanzar un error `subclassResponsibility`, o aplicar un valor por defecto), es la que define el propósito del mensaje para toda la jerarquía.

Por eso, las subimplementaciones no necesitan una nueva descripción detallada, ya que deben seguir el mismo propósito. El autor simplemente escribe: "See superimplementor".

Este enfoque también se aplica a métodos encadenados con nombres similares, como:

- `Object >> changed` llama a `changed:`
- `changed:` llama a `changed:with:`

Una vez que entiendes `changed:with:`, los otros métodos son variantes que usan valores por defecto. Por eso, solo se comenta el más completo, y los otros tienen descripciones como "See changed:" o "See changed:with:".

5.3 Anatomy of a Method Description

¿Qué tipo de comentarios vale la pena incluir en la descripción de un método?

- Evitar repetir el nombre del método en la descripción. Por ejemplo, si el método se llama `productCode`, no tiene sentido escribir "Devuelve el código del producto", porque es obvio. Si el método es simplemente un getter o setter, prefiere usar descripciones como: "Getter" o "Setter", ya que no hay mucho más que decir si se entiende lo que hace la variable.
- Describir el propósito general del método en lugar de comentar línea por línea. Si hay código raro o difícil de entender, no lo explica con un comentario críptico. En su lugar, lo extrae a un nuevo método con un nombre descriptivo, y el comentario que habría escrito se convierte en la descripción de ese nuevo método.

5.4 Purpose and Implementation Details

Cómo estructura sus descripciones de métodos dividiéndolas en dos partes:

- **Propósito (Purpose)**
  - Describe qué hace el método.
  - Se redacta como: "Si envías este mensaje a este objeto, esto es lo que pasará..." Ejemplos:
    - "Ordena los elementos del receptor."
    - "Lee el siguiente ítem y lo devuelve."
    - "Devuelve si el receptor contiene errores."
  - Es reutilizable: todos los métodos que implementan el mismo mensaje en una jerarquía deben tener el mismo propósito.
  - Por eso, solo se documenta el propósito en la superclase, y las subclases escriben: "See superimplementor."
- **Detalles de Implementación (Implementation Details)**
  - Explican cómo se hace lo que el método hace.
  - Es opcional, y solo se incluye si el código tiene algo raro, poco intuitivo o específico que vale la pena explicar.
  - No es reutilizable: cada implementación puede (y debe) tener sus propios detalles.

5.5 Description Reuse for Polymorphism

Al principio, Woolf veía las clases como unidades aisladas y elegía superclases solo para heredar funcionalidad. Con el tiempo, entendió que las clases deben pensarse en jerarquías, donde cada clase depende del contexto que aportan sus superclases. Ahora diseña subclases considerando cómo se diferencian de sus superclases. Estas últimas definen qué debe hacerse, mientras que las subclases implementan cómo hacerlo. Por ejemplo, en la jerarquía Collection, la clase base define las operaciones generales, pero cada subclase las implementa según su estructura interna (lista, hash, etc.). Para mantener coherencia, toda subimplementación debe conservar el mismo propósito que el método original de la superclase, aunque su implementación sea distinta.

5.6 Purpose is Polymorphic

Cuando todos los métodos de una jerarquía tienen el mismo propósito, son polimórficos, y así también lo es la jerarquía entera. Esto permite que otros objetos usen cualquier instancia de esa jerarquía indistintamente, ya que todas se comportan igual. Por ejemplo, si un objeto `Employee` tiene una lista de tareas (`toDoList`), no importa si es una `OrderedCollection` o una `SortedCollection`; mientras sigan cumpliendo los mismos métodos (`add:`, `remove:`, `size`, `first`), pueden intercambiarse sin problemas. La implementación concreta puede definirse después, sin afectar la lógica general del sistema.

5.7 Defining Polymorphism

El autor amplía la definición tradicional de polimorfismo. No basta con que dos métodos tengan el mismo nombre; deben también comportarse igual: aceptar los mismos parámetros, provocar los mismos efectos y devolver el mismo tipo de resultado. Por ejemplo, `value` y `value:` pueden tener muchos implementadores, pero no siempre son polimórficos, ya que su comportamiento cambia según la clase (en un `ValueModel` son accesores; en un `Block`, son evaluadores).

Del mismo modo, dos clases son polimórficas si entienden los mismos mensajes y sus métodos correspondientes son también polimórficos. Aunque no compartan toda la interfaz, pueden compartir una interfaz central (core interface), que permite que se usen indistintamente en ciertos contextos mientras esa interfaz común sea respetada.

5.8 Making a Hierarchy Polymorphic

En esta parte explica cómo el hábito de comentar los métodos con "See superimplementor" lo llevó a pensar y programar de forma más polimórfica, haciendo sus jerarquías más reutilizables, flexibles y fáciles de entender. Sin embargo, surgen problemas cuando no hay un superimplementor: si dos clases hermanas tienen métodos con el mismo propósito pero no comparten una jerarquía, eso indica que falta una clase abstracta común. En estos casos, la solución es crear una superclase abstracta que defina el comportamiento común (el método con su propósito y una implementación por defecto), y hacer que las clases concretas hereden de ella. Así se formaliza el polimorfismo y se evita la duplicación.

5.9 The Template Class Pattern

El autor introduce el concepto de Template Class, una clase abstracta que define la interfaz común para una jerarquía de clases polimórficas, dejando los detalles de implementación a las subclases. Esta idea es similar al patrón Template Method, pero a nivel de clase. Mientras el Template Method define la estructura de un método, el Template Class define el comportamiento general de una familia de clases. El resultado es una jerarquía coherente y completamente polimórfica.

5.10 The ValueModel hierarchy

La jerarquía ValueModel en VisualWorks es un ejemplo claro de una jerarquía polimórfica. La clase ValueModel define una interfaz común con mensajes como `value`, `value:` y `onChangeSend:to:`. Las subclases (como ValueHolder, AspectAdaptor, y TypeConverter) implementan estos mensajes según sus propias necesidades, pero respetan la misma interfaz.

Esto permite que el código colaborador (como los widgets de interfaz gráfica) use cualquier instancia de la jerarquía sin importar la subclase específica, gracias al polimorfismo a nivel de clase.

Otros ejemplos clásicos de jerarquías polimórficas incluyen `Collection`, `Magnitude`, `Number`, `Boolean`, y `String`, cuya popularidad se debe en gran parte a lo fácil que resulta usarlas justamente por ser polimórficas.

6 A Simple Technique for Handling Multiple Polymorphism

Daniel H. H. Ingalls fue una figura clave en la creación del lenguaje Smalltalk.

Este paper fue presentado en la conferencia OOPSLA (Object-Oriented Programming, Systems, Languages, and Applications) en septiembre de 1986. Ingalls, en ese momento, trabajaba en Apple Computer, Inc.

Abstract

El paper trata sobre situaciones de múltiple polimorfismo, donde más de un término en una expresión es polimórfico (por ejemplo, tanto el objeto receptor como un argumento pueden tener distintos tipos posibles en tiempo de ejecución). Estas situaciones no están bien resueltas por los lenguajes orientados a objetos convencionales, que solo hacen *simple dispatch* (basado en el tipo del receptor del mensaje).

6.1 Problem

En estos casos, los programadores suelen recurrir a verificaciones explícitas de tipo (ej. *isMemberOf:*) dentro de métodos para distinguir qué hacer según el tipo del argumento. Esto:

- Rompe la modularidad.
- Aumenta la complejidad del código.
- Lleva a un crecimiento combinatorio del código a medida que aumentan los tipos posibles.
- Va contra los principios que la programación orientada a objetos buscaba resolver.

Ejemplo citado en el paper: Mostrar objetos gráficos (Rectangle, Oval, etc.) sobre diferentes tipos de puertos (DisplayPort, PrinterPort, etc.) lleva a código como:

```
Rectangle displayOn: aPort
aPort isMemberOf: DisplayPort ifTrue: [ ... ]
aPort isMemberOf: PrinterPort ifTrue: [ ... ]
```

Este tipo de código crece rápidamente y es difícil de mantener.

6.2 Solution: Double Dispatch

El autor propone una técnica que preserva la modularidad y permite manejar polimorfismo múltiple usando mecanismos ya disponibles en cualquier lenguaje orientado a objetos: el envío *encadenado de mensajes* (*double dispatch*). ¿Cómo funciona?

1. Cada objeto gráfico implementa un método displayOn: que reenvía el mensaje al puerto, pasando a sí mismo:

   Rectangle displayOn: aPort
   aPort displayRectangle: self

2. Cada puerto implementa un conjunto de métodos específicos para cada tipo de objeto gráfico:

   DisplayPort displayRectangle: aRect
   "código para mostrar un rectángulo en un DisplayPort"

   PrinterPort displayRectangle: aRect
   "código para mostrar un rectángulo en una impresora"

De este modo, cada uno de los dos objetos polimórficos contribuye a la resolución del comportamiento final mediante un doble despacho de mensajes.

6.3 Ventajas

- Se mantiene la modularidad: no se necesita modificar código existente al agregar nuevos tipos.
- Facilidad de extensión: agregar una nueva clase gráfica implica solo un método displayOn: y nuevas implementaciones en las clases de puerto.
- El código es local a cada clase y no se rompe si se expande el sistema.
- Es aplicable a cualquier lenguaje orientado a objetos.

7 The Null Object Pattern - Bobby Woolf

Bobby Woolf escribe en 1996 este paper con el intento de: *"Provide a surrogate for another object that shares the same interface but does nothing. The Null Object encapsulates the implementation decisions of how to "do nothing" and hides those details from its collaborators."*

Motivation

A veces una clase necesita un colaborador, aunque no requiera que este haga algo. Aun así, quiere tratar del mismo modo a colaboradores activos e inactivos. Por ejemplo, en el paradigma *Model-View-Controller (MVC)* de Smalltalk-80, una View usa un Controller para obtener entrada del usuario. Esto corresponde al *Strategy Pattern*. Sin embargo, una vista puede ser de solo lectura y no necesitar un controlador, aunque su implementación sí lo espera. - El *Model-View-Controller (MVC)* es un patrón que promovía una separación de responsabilidades de forma de lograr un código más modular. El *controlador* interpreta la entrada del usuario y la envía al modelo. El *modelo* actualiza sus datos. La *vista*, que está observando al modelo, se actualiza para reflejar los nuevos datos.

Si algunos objetos de la clase necesitan controlador y otros no, la clase igualmente debe ser una subclase de *View* y, por lo tanto, tener un controlador. Usar *nil* como controlador no es una buena solución, ya que implicaría agregar código condicional constantemente para evitar errores de mensajes no entendidos por *nil*.

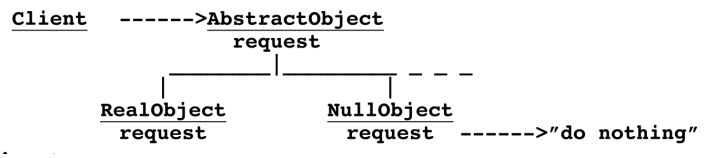
Una alternativa es un controlador en modo solo lectura, pero eso implica lógica innecesaria si siempre será de solo lectura. La mejor opción es un controlador que no haga nada: una subclase especial llamada **NoController**, que *implementa toda la interfaz de Controller pero sin realizar acciones*. Por ejemplo, responde false a *isControlWanted* y retorna self en *startUp*.

Esto ilustra el **Null Object Pattern**, que define *cómo crear una clase que encapsula el comportamiento de "no hacer nada", ocultando esa complejidad y haciéndola reutilizable*. La clave del patrón es una clase abstracta con la interfaz común, de la cual el *Null Object* es una subclase.

7.1 Applicability - Cuándo usar el Null Object Pattern

1. *Cuando un objeto necesita un colaborador* El patrón no crea esta colaboración, sino que aprovecha una ya existente.
2. *Cuando algunos colaboradores deben no hacer nada* Es útil cuando ciertas instancias del colaborador no deben realizar ninguna acción.
3. *Cuando se quiere que el cliente no distinga entre un colaborador real y uno que no hace nada* Así, el cliente no tiene que verificar si es *nil* u otro valor especial.
4. *Cuando se desea reutilizar el comportamiento de "no hacer nada"* Esto garantiza consistencia entre diferentes clientes que necesitan ese comportamiento.
5. *Cuando todo el comportamiento que podría ser "no hacer nada" está encapsulado en la clase colaboradora* Si parte del comportamiento es "no hacer nada", lo más probable es que la mayoría o todo lo sea.

7.2 Structure



7.3 Participants

1. *Client (View)* Requiere un colaborador.
2. *AbstractObject (Controller)*
   - Declara la interfaz del colaborador del Client.
   - Implementa el comportamiento por defecto común a todas las clases, cuando es apropiado.
3. *RealObject (TextController)*
   - Define una subclase concreta de AbstractObject cuyas instancias proveen el comportamiento que el Client espera.
4. *NullObject (NoController)*
   - Proporciona una interfaz idéntica a la de AbstractObject, permitiendo que pueda sustituirse por un objeto real.
   - Implementa esa interfaz sin realizar acciones. El significado de "no hacer nada" depende del comportamiento esperado por el Client.
   - Si existen múltiples formas de "no hacer nada", pueden requerirse varias clases NullObject.

7.4 Collaborations

Los clientes usan la interfaz de la clase AbstractObject para interactuar con sus colaboradores. Si el receptor es un *RealObject*, la solicitud se maneja proporcionando un comportamiento real. Si el receptor es un *NullObject*, la solicitud se maneja no haciendo nada o devolviendo un resultado nulo.

7.5 Consequences

1. *Define jerarquías de clases que incluyen objetos reales y null objects* Se pueden usar null objects en lugar de objetos reales cuando se espera que no hagan nada. El código del cliente puede trabajar con ambos indistintamente.
2. *Simplifica el código del cliente* Los clientes tratan por igual a colaboradores reales y null, sin necesidad de comprobar si son *null*, evitando código condicional extra.
3. *Encapsula el código de "no hacer nada" dentro del null object* Este código es fácil de ubicar, claro en su diferencia con otras clases, y puede evitar variables *null*, usando constantes o evitando su uso por completo.
4. *Facilita la reutilización del comportamiento de "no hacer nada"* Varios clientes que necesitan este comportamiento lo implementan de forma consistente. Si se necesita modificar, se cambia en un solo lugar.

5. *Dificulta distribuir o mezclar el comportamiento de "no hacer nada" en objetos colaboradores reales* Para reutilizarlo, las clases deben delegar dicho comportamiento en una clase que pueda ser un null object.

6. *Puede requerir crear una clase NullObject para cada nueva clase AbstractObject.*

7. *Puede ser difícil de implementar si los distintos clientes no están de acuerdo en qué significa "no hacer nada".*

8. *Siempre actúa como un objeto que no hace nada* Un Null Object no se convierte en un objeto real.

7.6 Implementation

1. *Null Object como Singleton* Como no tiene estado, puede reutilizarse una sola instancia en lugar de crear varias iguales. El *Singleton Pattern* asegura que una clase tenga una única instancia y proporciona un punto de acceso global a ella.

2. *Instancia especial de un Real Object* Para evitar la explosión de clases, el null object puede ser una instancia especial de RealObject, con variables en valores nulos que provoquen un comportamiento nulo.

3. *Clientes no coinciden en el null behaviour* Si distintos clientes esperan distintos "no hacer nada", se necesitan múltiples clases NullObject o variables configurables. También puede usarse el *Flyweight Pattern* si se comparte comportamiento común y se parametriza lo variable.

4. *Transformación a Real Object* Un Null Object no se convierte en un objeto real. Si se requiere esa transformación, se debe usar el *Proxy Pattern.*

5. *Null Object no es Proxy* Aunque similares, el Proxy accede y puede convertirse en el objeto real. El Null Object directamente reemplaza al objeto real y nunca cambia su comportamiento.

6. *Null Object como special Strategy* Puede ser una implementación del *Strategy Pattern* que simplemente no hace nada. Por ejemplo, *NoController* como estrategia que ignora la entrada del usuario.

7. *Null Object como special State* Puede ser una implementación de un estado en el *State Pattern* que hace poco o nada. Por ejemplo, el estado de usuario no logueado solo permite iniciar sesión.

8. *Null Object no es un mixin* Es una clase concreta, no diseñada para mezclar comportamiento, sino para ser un colaborador completo que puede ser sustituido cuando se desea un comportamiento nulo.

7.7 Sample Code

El ejemplo muestra la implementación del patrón **Null Object** en la clase *NullScope* dentro de la jerarquía *NameScope* en VisualWorks Smalltalk. En esta jerarquía, *NameScope* define las operaciones para buscar variables, manejar variables no declaradas e iterar sobre ellas

7.8 Known Uses

7.8.1 NoController

NoController, la clase de Null Object en el ejemplo motivador, es una clase en la jerarquía de Controladores de VisualWorks Smalltalk.

7.8.2 NullDragMode

NullDragMode es una clase en la jerarquía de DragMode de VisualWorks Smalltalk. Un DragMode se utiliza para implementar *placement and dragging* de visuales en el pintor de ventanas. Las subclases representan diferentes formas de *dragging.* Un ejemplo de una subclase es **CornerDragMode**, que representa el arrastre de un control de tamaño de una visual. Un **NullDragMode** es el contrapunto a **CornerDragMode** y representa un intento de redimensionar una visual que no puede redimensionarse (como una etiqueta de texto). El método `dragObject:startingAt:inController:` de **NullDragMode** utiliza un bloque vacío que no hace nada.

7.8.3 NullInputManager

NullInputManager es una clase en la jerarquía de InputManager de VisualWorks Smalltalk. Un InputManager proporciona una interfaz estándar para eventos de la plataforma que pueden afectar el manejo de *internationalized input.* **NullInputManager** representa plataformas que no soportan internacionalización. Sus métodos no hacen nada o muy poco, mientras que los métodos de su contraparte **X11InputManager** realizan trabajo real.

- Esto es un ejemplo del *Adapter Pattern.* Este actúa como un intermediario que convierte la interfaz de una clase en otra que el cliente espera, permitiendo que interactúen sin que el cliente necesite modificar las clases originales.

7.8.4 NullScope

NullScope es una clase en la jerarquía NameScope de VisualWorks Smalltalk. Un **NameScope** representa el alcance de un conjunto particular de variables. **NullScope** representa el alcance más externo, que nunca contiene ninguna variable. Cuando se busca una declaración de variable, cada alcance sigue buscando en su alcance externo hasta que encuentra la declaración o llega a **NullScope**. Este detiene la búsqueda y devuelve que la variable aparentemente no ha sido declarada. **NullScope** se implementa como un Singleton, ya que el sistema nunca necesita más de una instancia.

7.8.5 NullLayoutManager

El **LayoutManager** en el Java AWT Toolkit podría beneficiarse de un objeto nulo como **NullLayoutManager**. Si un contenedor no requiere un **LayoutManager**, su variable puede ser establecida en `nil`. Sin embargo, esto obliga a verificar constantemente si la variable es `nil`. Usar un **NullLayoutManager** evitaría estas verificaciones y simplificaría el código.

7.8.6 Null_Mutex

Null_Mutex es una clase de mecanismo de exclusión mutua en el marco ASX (ADAPTIVE Service eXecutive) implementado en C++. **Null_Mutex** no realiza ninguna acción de bloqueo, ya que su clase está destinada a servicios que siempre se ejecutan en un solo hilo y no necesitan concurrencia. Sus métodos `acquire` y `release` no hacen nada, eliminando la sobrecarga de obtener bloqueos innecesarios.

7.8.7 Null Lock

Null Lock es un tipo de modo de bloqueo en el sistema de gestión de bases de datos VERSANT. A diferencia de otros tipos de bloqueo (como el bloqueo de escritura o lectura), el **Null Lock** no bloquea otros locks y no puede ser bloqueado por otros, garantizando acceso inmediato al objeto. Aunque no realiza bloqueo real, actúa como si lo hiciera para operaciones que requieren algún tipo de bloqueo.

7.8.8 NullIterator

El **NullIterator** es una especialización del patrón **Iterator**. Cada nodo en un árbol puede tener su propio iterador para sus hijos. Los nodos hoja devuelven una instancia de **NullIterator**, que siempre indica que la iteración ha finalizado, ayudando a evitar pruebas especiales al iterar sobre estructuras vacías.

- El *Iterator Pattern* proporciona una forma de acceder secuencialmente a los elementos de una colección sin exponer su estructura interna.

7.8.9 Z-Node

En lenguajes procedurales, los **z-nodes** son nodos ficticios utilizados como el último nodo en una lista enlazada. Un **z-node** se usa como sustituto cuando faltan nodos hijos en un árbol. Los algoritmos de búsqueda pueden omitir los **z-nodes**, ya que cuando se encuentran con un **z-node**, saben que no se encontró el elemento buscado.

7.8.10 NULL Handler

El **Decoupled Reference Pattern** muestra cómo acceder a objetos a través de Handlers para ocultar su ubicación real del cliente. Cuando un cliente solicita un objeto que ya no está disponible, en lugar de hacer que el programa falle, el sistema devuelve un **NULL Handler**. Este Handler actúa como otros Handlers, pero cumple las solicitudes generando excepciones o causando condiciones de error.

- El *Decoupled Reference Pattern* desacopla a un cliente del objeto real al que quiere acceder, usando una referencia indirecta o manejador (*handler*). Esto permite que el cliente no conozca la ubicación, el ciclo de vida, o incluso la existencia concreta del objeto real.

7.9 Related Patterns

- **Singleton**: El `NullObject` suele implementarse como un *Singleton*, ya que no tiene estado y múltiples instancias se comportarían igual.
- **Flyweight**: Cuando varios objetos nulos comparten una implementación común, pueden usar el patrón *Flyweight* para reducir el uso de memoria.
- **Strategy**: El `NullObject` puede ser una estrategia que simplemente no hace nada.

- **State**: También puede representar un estado en el que el objeto no debe realizar ninguna acción.
- **Iterator**: Puede actuar como un iterador especial que no recorre ningún elemento.
- **Adapter**: Un *null adapter* simula adaptar otro objeto, pero en realidad no adapta nada.
- **Exceptional Value**: El `NullObject` es un caso especial de *Exceptional Value*, que representa circunstancias excepcionales de forma pasiva y segura.
  - **Whole Value** define el objeto.
  - **Exceptional Value** define el comportamiento excepcional del objeto.
  - **NullObject** es una implementación concreta de un Exceptional Value que *hace nada* en vez de fallar.
  - **CHECKS Pattern Language** es el marco que agrupa estos conceptos.

8 The Object Recursion Pattern - Bobby Woolf

8.1 Intent

Distribuir el procesamiento de un pedido sobre una estructura mediante la delegación polimórfica. La *Object Recursion* permite que este pedido, repetidas veces, se convierta en partes más sencillas de manejar.

8.2 Also Known As

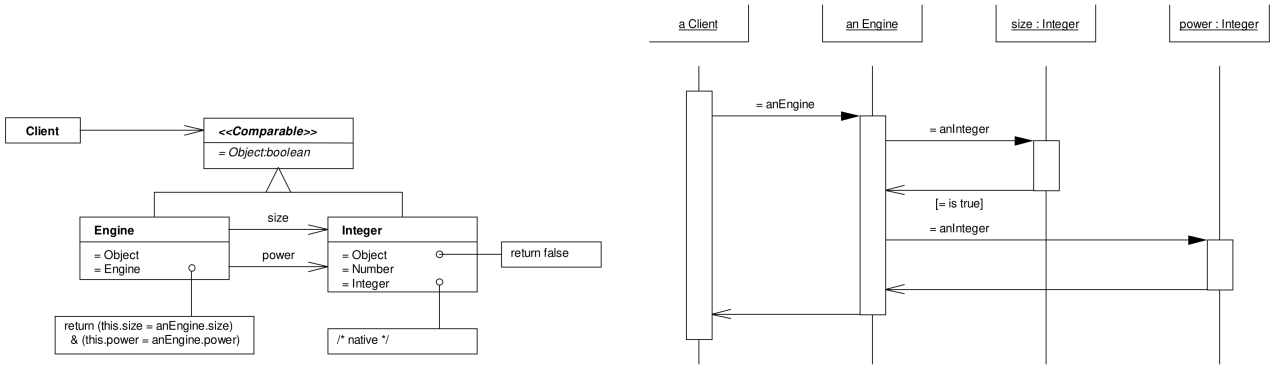
Recursive Delegation

8.3 Motivation

Comparar objetos simples es directo mediante operaciones nativas, pero comparar objetos complejos requiere una estrategia más sofisticada. Un enfoque tradicional consiste en usar un objeto **Comparer** que descompone ambos objetos en partes simples para compararlas. Sin embargo, este método implica alto acoplamiento, mantenimiento complejo y una exposición innecesaria de la estructura interna de los objetos.

La alternativa orientada a objetos es *delegar la comparación a los propios objetos*. En este enfoque, cada objeto sabe cómo compararse con otro de su tipo, verificando la equivalencia de sus partes relevantes. Si esas partes son complejas, se comparan recursivamente, hasta llegar a componentes simples comparables. Este patrón se conoce como **Object Recursion**, y permite distribuir la lógica de comparación de forma modular y extensible, respetando el encapsulamiento.

Por ejemplo, un objeto *Engine* puede definirse como equivalente a otro si coinciden su tamaño y potencia, los cuales son enteros comparables nativamente. Si los atributos fueran más complejos, ellos mismos aplicarían el mismo principio. Así, se construye una comparación profunda y recursiva sin necesidad de un comparador centralizado.



8.4 Keys

Un sistema que utiliza el patrón *Object Recursion* presenta las siguientes características esenciales:

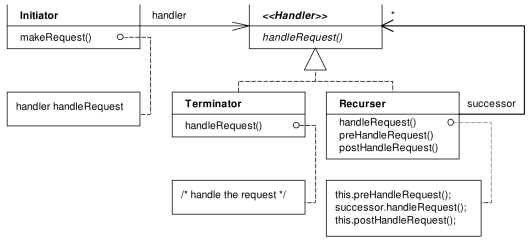
- Dos clases polimórficas: una maneja el mensaje de forma recursiva y la otra lo procesa directamente sin recursión.
- Un mensaje iniciador, generalmente enviado desde una tercera clase no polimórfica con respecto a las anteriores, que da comienzo al proceso recursivo.

8.5 Applicability

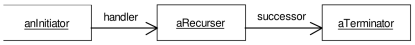
Aplica el patrón Object Recursion en sistemas con estructuras enlazadas cuando:

- Se debe pasar un mensaje a través de la estructura sin conocer de antemano su destino final.
- Es necesario difundir un mensaje a todos los nodos de una parte de la estructura.
- Se desea distribuir la responsabilidad de un comportamiento entre múltiples objetos dentro de dicha estructura.

8.6 Structure



A typical object structure might look like this:



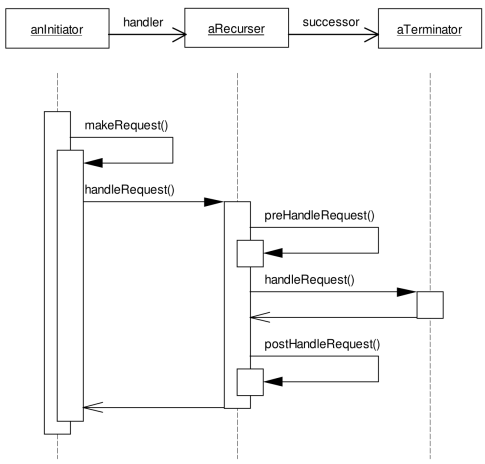
8.7 Participants

En el patrón Object Recursion, intervienen los siguientes participantes:

- **Initiator** (*Cliente*): Es quien inicia la solicitud mediante un mensaje como makeRequest(). Generalmente no es una subclase de Handler, y su mensaje es distinto del que los manejadores usan (handleRequest()).
- **Handler** (*Comparable*): Define el tipo de objetos que pueden manejar las solicitudes iniciadas por el cliente. Actúa como interfaz común para los objetos que participan en la recursión.
- **Recurser** (*Engine*): Es un tipo específico de Handler que mantiene una referencia a uno o más sucesores. Maneja la solicitud delegándola a sus sucesores, pudiendo ejecutar lógica adicional antes o después de dicha delegación. Un Recurser puede, en otros contextos, actuar como terminador.
- **Terminator** (*Integer*): Concluye el procesamiento de la solicitud implementándola completamente sin delegarla. También puede funcionar como Recurser en solicitudes distintas.

8.8 Collaboration

En el patrón Object Recursion, la colaboración entre los participantes sigue un flujo claro: el Initiator inicia una solicitud pidiéndole a un Handler que la procese. Si el Handler es un Recurser, este realiza el trabajo necesario y luego delega la solicitud a uno de sus sucesores (también Handler). Tras recibir el resultado del sucesor, puede complementarlo con lógica adicional, ejecutada antes o después de la delegación. Si tiene múltiples sucesores, puede delegar a cada uno de ellos de forma secuencial o incluso asincrónica. En cambio, si el Handler es un Terminator, procesa la solicitud completamente por sí mismo, sin delegarla, y devuelve el resultado correspondiente.



8.9 Consequences

El uso del patrón Object Recursion ofrece varias ventajas significativas. En primer lugar, permite *distributed processing*, ya que la solicitud se reparte entre múltiples objetos Handler organizados de la forma más adecuada para cumplir con la tarea. Además, brinda *responsibility flexibility*, puesto que el Initiator no necesita conocer la cantidad, organización o lógica interna de los Handlers; simplemente realiza la solicitud y deja que el sistema la resuelva. Esta organización puede incluso modificarse dinámicamente en tiempo de ejecución.

Otra ventaja es la *role flexibility*: un mismo Handler puede comportarse como Recurser en una solicitud y como Terminator en otra, según el contexto. Por último, mejora la *encapsulación*, ya que cada objeto maneja internamente cómo debe procesarse la solicitud.

No obstante, este patrón también presenta una desventaja: aumento de la *programming complexity*. La recursión, tanto procedural como orientada a objetos, puede ser difícil de entender y mantener, y su uso excesivo puede complicar el diseño del sistema.

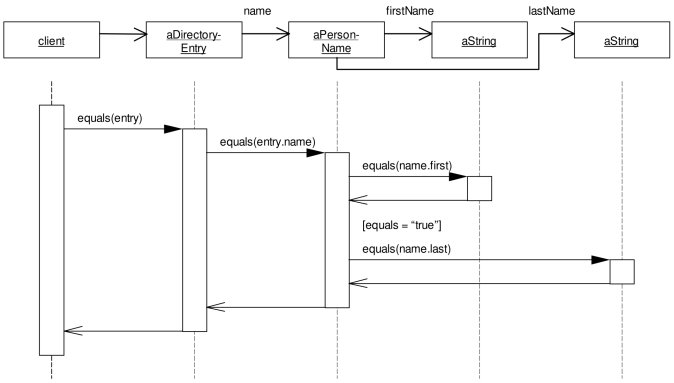
8.10 Implementation

Al implementar el patrón de Recursión de Objetos, se deben considerar dos aspectos importantes:

- *Separated initiator type*: El método Initiator.makeRequest() no debe ser polimórfico con Recursor.handleRequest(). Si todos los remitentes de un método son implementadores del mismo mensaje, y no existe un método externo no polimórfico que inicie la recursión, esos métodos pueden eliminarse sin afectar el programa.
- *Defining the successor*: El Recurser necesita uno o más sucesores, pero el Terminator no. Aunque el Terminator herede el enlace al sucesor, lo ignora en sus métodos. Si todos sus métodos ignoran el sucesor, el valor puede ser nulo y el enlace no es necesario.

8.11 Sample Code

Todos los objetos, por más complejo que sea, se componen de objetos simples (primitivas). Las comparaciones entre primitivas son directas y por tanto, actuarán como caso base de la recursión. El ejemplo plantea el caso de un objeto que representa la guía telefónica donde se almacena su nombre y número. En este escenario la recursión tiene dos niveles porque `DirectoryEntry.equals()` llama a `PersonName.equals()`, que llama a `String.equals()`. Como se ve a continuación:



8.12 Known Uses

El patrón de Recursión de Objetos se usa en muchos contextos de programación:

- *Equality*: utiliza *one-step recursion*. Cada implementador de equals también necesita un hash recursivo correspondiente.
- *Copy o clone*: usa *two-step recursion*. `copy()` llama a `simpleCopy()` (copia la raíz) y luego a `postCopy()` (copia las partes). `postCopy()` continúa la recursión.
- *Serialization algorithms*: recorre recursivamente el objeto, serializando primero la raíz y luego sus partes. Termina al llegar a objetos simples.
- *Object as String (toString, printString)*: convierte el objeto en texto comenzando por la raíz y luego pidiendo a cada parte que se represente a sí misma.
- *Tree structures*: pueden usar recursión para enviar mensajes desde hojas al nodo raíz o desde la raíz a todas las hojas (ej., en árboles gráficos para actualizar o redistribuir visualizaciones).

8.13 Related Patterns

Este patrón se relaciona con otros, pero se distingue principalmente por ser un patrón de comportamiento, no estructural:

- Vs. *Composite* y *Decorator*: aunque ambos pueden parecer recursivos al delegar a *Compsites* o hijos, su recursión es estructural y de solo un nivel. En cambio, la Recursión de Objetos es algorítmica y de profundidad ilimitada.
- Vs. *Chain of Responsibility*: usa directamente Recursión de Objetos para pasar solicitudes a través de una estructura jerárquica o en cadena, buscando el *handler* adecuado.
- Vs. *Adapter*: una cadena de *Adapters* puede parecer recursiva, pero no lo es realmente porque la delegación no es polimórfica, lo cual contradice el principio del *Object Recursion*.
- Vs. *Interpreter*: el mensaje *interpret()* se aplica recursivamente a través del árbol de expresiones. Aquí se identifican claramente los roles del patrón de Recursión de Objetos: el *Client* (Initiator), la *AbstractExpression* (Handler), la *NonTerminalExpression* (Recurser) y la *TerminalExpression* (Terminator).
- Vs. *Iterator*: los iteradores internos sobre listas enlazadas o árboles pueden implementar recursión de objetos si el final está marcado por objetos del mismo tipo. Si usan null, la recursión es solo procedural.
- Vs. *Delegation* (como en Proxy): delegar un mensaje al mismo tipo de objeto es una forma simple de recursión de objetos, pero solo de un nivel.

9 Method Object Pattern

9.1 Smalltalk Best Practices Patterns - Kent Beck

En ocasiones, dentro del desarrollo de sistemas complejos, surgen métodos cuyo comportamiento central es complicado y difícil de simplificar mediante el patrón Composed Method. Estos métodos, que inicialmente pueden parecer simples, crecen progresivamente en líneas de código, parámetros y variables temporales hasta convertirse en estructuras difíciles de entender y mantener. El intento de aplicar Composed Method a estos métodos resulta ineficaz, ya que las distintas secciones del código suelen compartir una gran cantidad de variables. Cualquier fragmento extraído necesita múltiples parámetros —a veces seis u ocho— lo que termina por entorpecer la legibilidad y claridad del código, en lugar de mejorarla.

Ante esta situación, se recomienda emplear el patrón denominado Method Object. Este patrón consiste en crear una clase específica para representar una ejecución del método original. Esta clase encapsula tanto al receptor del mensaje como a sus argumentos y variables temporales, ahora convertidas en variables de instancia. Esta estructura permite que todas las partes del comportamiento complejo compartan un espacio de nombres común, facilitando así la posterior aplicación de Composed Method sin necesidad de pasar múltiples argumentos.

La implementación de este patrón sigue una secuencia clara. En primer lugar, se crea una clase con un nombre derivado del selector del método original. A esta clase se le asignan como variables de instancia: el receptor original del método, todos sus argumentos y sus variables temporales. Posteriormente, se define un método constructor que reciba el receptor y los argumentos del método original. Luego, se traslada el cuerpo del método a un nuevo método llamado `compute`, sustituyendo las referencias a los argumentos por las variables de instancia correspondientes y eliminando las declaraciones de variables temporales, ya que ahora forman parte del estado del objeto. Finalmente, el método original se reescribe para que cree una instancia de la nueva clase y le envíe el mensaje `compute`.

Este patrón, aunque no se usa frecuentemente, puede resultar crucial en casos específicos. El autor menciona que inicialmente no pensaba incluirlo en su recopilación de patrones, hasta que logró convencer a un cliente importante gracias a su aplicación. En la práctica, el uso del Method Object permitió reorganizar un método de 150 líneas, que no se dejaba dividir fácilmente, en una estructura más clara y mantenible. Luego de aplicar el patrón, no solo fue posible emplear Composed Method con eficacia, sino que se eliminaron variables innecesarias, el código se redujo a la mitad y se identificó y corrigió un error que pasaba desapercibido en la versión original.

En conclusión, el patrón Method Object ofrece una solución efectiva para simplificar métodos complejos que comparten múltiples variables, permitiendo mejorar la legibilidad y modularidad del código, y facilitando su mantenimiento, especialmente cuando otras técnicas de refactorización resultan inadecuadas.