Paradigmas de programación

Un paradigma de programación provee y determina la visión y métodos de un programador en la construcción de un programa o subprograma. Diferentes paradigmas resultan en diferentes estilos de programación y en diferentes formas de pensar la solución de problemas (con la solución de múltiples "problemas" se construye una aplicación).
Existen múltiples paradigmas, difícilmente un lenguaje de programación pueda clasificarse solamente en un paradigma. Por ejemplo: Smalltalk y Java son lenguajes basados en el paradigma orientado a objeto. El lenguaje de programación Scheme y Lisp, soportan sólo programación funcional. En cambio Python, soporta múltiples paradigmas.
Hay muchos *tipos de paradigmas*, pero nos interesa distinguir dos de ellos:

Paradigmas Operacionales

Los lenguajes que siguen este tipo de paradigma especifican la programación como un conjunto de secuencias computacionales que se ejecutan paso a paso. Dentro del tipo operacional podemos encontrar el paradigma imperativo y el paradigma orientado a objetos. Los lenguajes que siguen el *paradigma imperativo* se centran en la acción, es decir, la computación se ve como una secuencia de acciones, especificadas paso a paso, que convierten los datos de entrada iniciales en los datos de salida finales. Algunos lenguajes que siguen este paradigma serian el Cobol, Fortran, BASIC, C, Ada, Pascal, etc.
Los lenguajes *orientados a objetos* son la evolución natural de los lenguajes que siguen el paradigma imperativo. Se podría decir, que el modulo o tipo abstracto evoluciona al concepto de objeto. Bajo la perspectiva del paradigma orientado a objetos, la programación consiste en definir cuáles son los objetos adecuados para resolver un problema determinado y resolver el problema mediante la interacción entre los distintos objetos a través del intercambio de mensajes. Algunos lenguajes que siguen este paradigma serían el Smalltalk, C++ o Java.

Paradigmas Declarativos

Un lenguaje declarativo se construye estableciendo hechos, reglas, restricciones, ecuaciones, transformaciones u otras propiedades que debe tener el conjunto de valores que constituyen la solución. A partir de esta información, el sistema debe de ser capaz de derivar un esquema de evaluación que nos permita computar una solución, es decir, no existe una descripción paso a paso de cómo llegar a la solución. Habitualmente se suelen incluir características operacionales para mejorar la eficiencia en la resolución de problemas (en este caso, a los lenguajes se les denomina pseudo declarativos).Dentro del paradigma declarativo podemos distinguir el *paradigma funcional* (lenguajes Haskell o Lisp), *lógico* (Prolog), *transformacional* (CLIPS, JESS) y *relacional* (SQL).

La problemática de la programación

Hasta el presente, la actividad de programar ha suscitado siempre reacciones diversas que llegan a la total contradicción. Para algunos, en efecto, no se trata más que de un juego de construcción infantil, en el que basta con encadenar instrucciones elementales (en reducido número) para llegar a resolver (prácticamente) cualquier problema. Para otros, por el contrario, se trata de producir (en el sentido industrial del término) aplicaciones con exigencias de calidad que se intenta evaluar de acuerdo con ciertos criterios; enumeremos algunos:

- la exactitud: aptitud de un programa para devolver los resultados deseados, en condiciones normales de utilización (por ejemplo, datos correspondientes a las "especificaciones");
- la robustez: capacidad de respuesta ante un alejamiento de las condiciones normales de utilización;
- la extensibilidad: facilidad con que un programa podrá ser adaptado para satisfacer una evolución de las especificaciones;

- la reutilización: posibilidad de utilizar ciertas partes (“módulos”) del programa para resolver otro problema;
- la portabilidad: facilidad con que se puede utilizar un mismo programa en diferentes implementaciones;
- la eficiencia: tiempo de ejecución, ocupación de memoria...

La contradicción a menudo no es más que aparente y relacionada esencialmente con la importancia de los proyectos en cuestión. Por ejemplo, es fácil escribir un programa exacto y robusto cuando comporta un centenar de instrucciones; ¡la situación cambia radicalmente cuando se trata de un proyecto de diez hombres-año! Asimismo, los aspectos de extensibilidad y reutilización no tendrán demasiada importancia en el primer caso, mientras que probablemente serán cruciales en el segundo, aunque sólo sea por razones económicas.

La programación estructurada

La programación estructurada ha hecho progresar de forma evidente la calidad de la producción de los programas. Pero, visto en perspectiva, hay que reconocer que sus propios fundamentos le imponían limitaciones “naturales”. En efecto, la programación estructurada se basaba en lo que se llama a menudo “la ecuación de Wirth”, es decir:

Algoritmos + Estructuras de datos = Programas

Ello, sin duda, ha permitido estructurar los programas y, por tanto, mejorar la exactitud y la robustez. Se esperaba que también permitiera mejorar la extensibilidad y la reutilización. Pero, en la práctica, se observó que la adaptación o la reutilización de código conducía a menudo a “romper” el “módulo” interesante, y ello porque era necesario rehacer la estructura de datos. Precisamente este tipo de dificultades se encuentran ecuación de Wirth, que separa totalmente los datos de los procedimientos que actúan sobre dichos datos.

Problemas de la programación estructurada

Tal como la hemos estudiado en la materia Algoritmos y Estructuras de Datos, la programación estructurada tiene las siguientes dificultades:

- Discrepancia con el modelo mental perceptivo:

Cuando nos enfrentamos al mundo que nos rodea, formamos un modelo de nuestras percepciones dentro del cerebro, que nos ayuda a comprenderlo. Dicho modelo se basa principalmente en las cosas que percibimos, a las que asignamos nombres sustantivos, como perro, casa, etc. El modelo consiste, principalmente, en un sistema de clasificación de dichas cosas, a las que posteriormente se asocian propiedades, representadas mediante adjetivos (como grande, verde, etc.), y comportamientos, representados con verbos (corre, cae, salta, etc.). En nuestra percepción, y no sólo sintácticamente, el nombre va antes que el verbo. Si un perro pasa corriendo por nuestro campo de visión, lo primero que pensamos no es “algo corre”, sino “un perro corre”.
La programación procedimental, por el contrario, se basa fundamentalmente en los verbos, que en nuestra estructura mental son auxiliares de los nombres. Para comprobar que esto es cierto, basta recordar cuáles son los nombres más corrientes de nuestros programas: abrir, cerrar, leer, escribir, guardar, etc. La jerarquía de una aplicación procedimental está formada por subprogramas, es decir, por verbos, mientras que los datos (que corresponderían a los nombres sustantivos) desempeñan un papel secundario.
La metodología de diseño más utilizada en las aplicaciones procedimentales es la descomposición funcional que, como su nombre indica, se basa en la especificación de los comportamientos sucesivos del programa: primero hacemos esto, después hacemos aquello, y así sucesivamente. Obsérvese que el término crucial (hacemos) es un verbo.
Como consecuencia de esta discrepancia, el diseño de las aplicaciones informáticas tiene una carga inicial: nuestro modelo mental sustantivo debe traducirse a uno basado en comportamientos, antes de que podamos comenzar a construir la aplicación. Sabemos que toda traducción exige un esfuerzo y lleva implícita cierta pérdida. Por tanto, parece que un modelo mental basado en nombres permitiría mejorar el diseño de las grandes aplicaciones del futuro.

- Es difícil modificar y extender los programas:

La introducción de componentes nuevos no suele poder hacerse de una forma simple y ordenada, sin que tenga que modificarse, en consecuencia, gran parte de la versión anterior de la aplicación. Esto se debe, sobre todo, a la existencia de datos globales o locales compartidos por varios subprogramas, que introducen interacciones ocultas entre ellos, por lo que cualquier cambio en uno de los subprogramas que afecte a estos datos compartidos puede provocar efectos secundarios imprevistos e indeseables que nos fuercen a depurar de nuevo secciones que, en principio, no deberían haberse visto afectadas por el cambio o la extensión.

- Es difícil mantener los programas:

Casi todos los sistemas informáticos muy extensos tienen errores ocultos, que no surgen a la luz hasta después de muchas horas de funcionamiento, y que, normalmente no serán detectados antes de que el producto en cuestión sea puesto en el mercado. Su detección y corrección puede ser muy difícil, debido precisamente a los efectos secundarios mencionados y a las interacciones entre los distintos módulos del sistema, cuyo efecto es tanto peor cuanto mayor sea la envergadura del mismo.

- Es difícil reutilizar los programas:

Todos los que tenemos experiencia en informática sabemos que es prácticamente imposible aprovechar en una aplicación nueva las subrutinas que se diseñaron para otra sin aportarles ningún cambio.
Evidentemente, no solemos programar partiendo de cero, sino que tomamos como base la versión anterior e incorporamos las modificaciones necesarias. Pero esto nos obliga a realizar una nueva depuración, pues todo cambio implica, casi necesariamente, y a menos que sea trivial, la introducción de algún error, lo que no habría sucedido si hubiésemos podido reutilizar la primera versión sin cambio alguno.
Todos estos problemas llevaron, en los años ochenta, a una profunda insatisfacción sobre el estado del desarrollo de software, y a la afirmación, tantas veces oída y repetida, de que esta disciplina se había quedado muy atrasada respecto al hardware, por lo que los costos de la informática se habían desplazado significativamente de una rama a la otra. Era preciso, se decía, cambiar nuestra forma de programar: aumentar la reutilización del código, disminuir el costo del diseño de las aplicaciones, reducir los ciclos de construcción de los productos, y facilitar su extensibilidad y mantenimiento.
La estrategia de la programación estructurada consiste en separar los datos del programa de las funciones que los manipulan. El programa o aplicación completa, consiste de múltiples datos y múltiples funciones. Esta forma de programar tiene sus orígenes en la arquitectura “Von Neumann” de las primeras computadoras (vista en la materia Arquitectura de Computadoras).
Esta manera de programar tiene dos problemas principales. El primer problema es obligar a un programador a pensar como la máquina, en lugar de lo opuesto. El segundo problema es que si se hiciera algún cambio en la estructura de alguno de los datos, potencialmente habría que modificar todas las funciones del programa para que éstas pudieran utilizar la nueva estructura.
¿Qué tan problemático pudiese ser esto? Pues que mejor ejemplo que el problema del año 2000 donde un dato tan insignificante como la fecha, que al cambiarse de dos a cuatro dígitos resultó en costos mundiales de cerca de 1 trillón de dólares. Lo que empeoró las cosas fue que todos estos programas tenían miles de funciones donde cada una de ellas requería de la fecha para funcionar correctamente, cómo en el caso de aplicaciones bancarias y nóminas de compañías.
¿Cómo puede ayudarnos la orientación a objetos a solucionar los dos problemas principales de la programación tradicional? La respuesta es que la orientación a objetos nos ayuda a mejorar radicalmente ambas situaciones gracias a que la unidad básica de programación es el objeto. A nivel organizacional el concepto del objeto nos acerca más a la manera de pensar de la gente al agregar un nivel de abstracción adicional, donde internamente la estructura del programa se ajusta a la arquitectura de la máquina. En relación al segundo problema, asignando a cada objeto sus propios datos y funciones locales, resulta un programa o aplicación definido exclusivamente en término de objetos y sus relaciones entre sí, de modo que un cambio en la estructura de los datos sólo afectará al objeto que lo contiene.

La Programación Orientada a Objetos

a) Objeto

Aquí es donde interviene la P.O.O (abreviatura de Programación Orientada a Objetos), que se fundamenta precisamente en el concepto de objeto, es decir, una asociación de datos y procedimientos (que son llamados métodos) que actúan sobre dichos datos. Por analogía con la ecuación de Wirth, se podría decir que la ecuación de la P.O.O. es:

Métodos + Datos = Objeto

b) Encapsulación

Pero esta asociación es más que una simple yuxtaposición. En efecto, en lo que se podría calificar de P.O.O. "pura", se observa lo que se denomina encapsulación de los datos. Ello significa que no es posible actuar directamente sobre los datos de un objeto; es necesario pasar por medio de sus métodos que cumplen así una función de interfaz obligatoria. Esto se traduce a veces diciendo que la llamada de un método es en realidad el envío de un "mensaje" al objeto.

El gran mérito de la encapsulación es que, visto desde el exterior, un objeto se caracteriza únicamente por las especificaciones de sus métodos, no revistiendo ninguna importancia la manera como estén realmente implementados los datos en su interior. Se describe a menudo esta situación diciendo que realiza una "abstracción de los datos" (lo cual expresa correctamente que los datos concretos de la implementación están escondidos). En relación con esto, se puede destacar que en programación estructurada, un procedimiento podría igualmente estar caracterizado (desde el exterior) por sus especificaciones, pero, sin encapsulación, la abstracción de los datos no tendría lugar.

La encapsulación de los datos presenta un interés manifiesto en cuanto a calidad de la aplicación. Facilita considerablemente el mantenimiento: una modificación eventual de la estructura de datos de un objeto no tiene incidencia más que sobre el propio objeto; los usuarios del objeto no se verán afectados por la modificación (cosa que no se daba, desde luego, en la programación estructurada). Del mismo modo, la encapsulación de los datos facilita en gran medida la reutilización de un objeto.

c) Clase

En P.O.O. aparece generalmente el concepto de clase. Dicho concepto corresponde simple-mente a la generalización de la noción de tipo que se encuentra en los lenguajes clásicos. Una clase, efectivamente," no es más que la descripción de un conjunto de objetos que tienen una estructura de datos común y disponen de los mismos métodos. Los objetos aparecen entonces como variables del tipo de la clase (en P.O.O. se dice también que un objeto es una "instancia" de su clase).

d) Herencia

Otro concepto importante en P.O.O. es el de la herencia. Permite definir una nueva clase a partir de una clase existente (¡que se reutiliza en bloque!), a la que se añaden nuevos datos y nuevos métodos. La concepción de la nueva clase, que "hereda" las propiedades y aptitudes de la anterior, puede así apoyarse en realizaciones anteriores perfectamente hasta el punto de "especializarlas" a voluntad. Como puede intuirse, la herencia facilita enormemente la reutilización de productos existentes, y ello tanto más cuanto que puede repetirse las veces que sea necesario (la clase C puede heredar de B, que puede a su vez heredar de A).

P.O.O. y lenguajes

Hemos enunciado Los grandes principios de la P.O.O. de una forma general, sin fijarnos en un lenguaje en particular. Ahora bien, evidentemente, ciertos lenguajes pueden ser concebidos radicalmente para aplicar al pie de la letra estos principios y realizar lo que denominamos la P.O.O. "pura". Es, por ejemplo, el caso de Simula, Smalltalk o, más recientemente, Eiffel. El misino fenómeno tuvo lugar, en su momento, con la programación estructurada y el lenguaje Pascal.

Por el contrario, también se puede intentar aplicar, con mayor o menor fortuna, lo que podríamos denominar "una filosofía orientada al objeto" de un lenguaje clásico (Pascal, C...). Encontramos aquí una idea comparable a la que consistía en aplicar los principios de la programación estructurada a lenguajes como Fortran o Basic.

El lenguaje C++ se sitúa a medio camino entre estos dos puntos de vista. En efecto, se ha obtenido añadiendo a un lenguaje clásico (C) las herramientas que permiten implementar todos los principios de la P.O.O. ¡Programar en C++ va por tanto más lejos que adoptar una filosofía P.O.O. en C. pero menos lejos que realizar P.O.O. pura con Eiffel!
La solución adoptada por B. Stroustrup tiene el mérito de preservar lo existente (compatibilidad entre C++ y programas ya escritos en C); permite también una "transición suave" de la programación estructurada a la P.O.O. En contrapartida, no impone en absoluto la aplicación estricta de los principios de la P.O.O. Como observará, en C++ nada impedirá (¡salvo el sentido común!) que cohabiten objetos (dignos de su nombre, porque realizan una encapsulación perfecta de sus datos) con funciones clásicas que tienen efectos colaterales sobre variables globales…