Conceptos fundamentales

de los Paradigmas de Programación

Cátedra de Paradigmas de Programación
Facultad Regional Buenos Aires
Universidad Tecnológica Nacional

Ing. Lucas Spigariol

Buenos Aires – 2008

Presentación

¿Por qué "paradigmas"?

A lo largo de la historia de las ciencias de la computación han ido surgiendo herramientas, reglas, conceptos y otros elementos que permitieron la creación de los más variados lenguajes de programación. Algunos nacieron y al poco tiempo desaparecieron, mientras que otros hace años ya que se diseñaron y siguen vigentes. Muchos se fueron adaptando y renovándose para subsistir y se vuelven casi irreconocibles de sus orígenes y en cambio otros permanecen fieles a sus principios fundantes y casi inalterables, más allá de ciertos cambios cosméticos.

Si algo caracteriza al panorama actual del desarrollo de sistemas es su complejidad y heterogeneidad. No existe una sola forma de pensar y encarar las soluciones, no son uniformes los conceptos que fundamentan los lenguajes, no es única la manera de programar. Los modelos de datos, estructuras de control, mecanismos de evaluación, sentencias, enlaces, expresiones, declaraciones y tantos otros elementos que conforman los lenguajes de programación actuales son muy diferentes entre unos y otros, hasta opuestos, pero es posible detectar cuáles son los conceptos que marcan diferencias mayores o menores, muestran puntos de contacto o de inflexión, establecen criterios de clasificaciones, y así, aportan elementos teóricos para sistematizar el análisis de la programación.

De esta manera, tiene sentido hablar de la existencia de diferentes "paradigmas" de programación que aportan los fundamentos teóricos y conceptuales para desarrollar sistemas de una manera en particular, incluso podríamos decir con una "filosofía" especial, que los caracterizan, identifican y a la vez diferencian de los otros paradigmas.

La materia "Paradigmas de Programación" brinda, para la formación de un ingeniero en sistemas, un conjunto de conceptos fundamentales de la programación que, relacionados entre si, con sus diferencias, similitudes e influencias recíprocas, conforman paradigmas de programación. El objetivo es dar un marco teórico conceptual que permita analizar y construir soluciones informáticas, y la práctica de analizar una herramienta técnica o conceptual, ya sea un lenguaje de programación u otro producto, a la luz de estos conceptos.

Se estudian los principales paradigmas de programación, recuperando los conceptos principales que los asemejan y distinguen entre sí, el sentido y utilidad de cada uno y el impacto que generan en la configuración y uso de los lenguajes de programación. El conocimiento de la sintaxis y herramientas propias de cada lenguaje no es el foco, sino un medio para comprender el paradigma y un caso particular de aplicación que se pueden trasladar a la mayoría de los otros lenguajes de cada paradigma. Se analiza la pertinencia de situarse en el marco conceptual de un paradigma si se decide utilizarlo para la construcción de una pieza de software.

El presente material aborda una serie de conceptos que son transversales a los diferentes paradigmas y propone una organización de los paradigmas. Está realizado en base a otros textos anteriores de la cátedra en los que había participado como autor, y con el aporte de otros docentes de la Facultad Regional Buenos Aires de la Universidad Tecnológica Nacional, consultando bibliografía especializada y teniendo en cuenta mi experiencia de 15 años de llevar adelante materias troncales de programación en la carrera de Ingeniería en Sistemas de Información.

El objetivo principal de este trabajo, es que en sus manos se convierta en una herramienta de utilidad para su formación permanente y su ejercicio profesional.

Ing. Lucas Spigariol

Capítulo 1

Paradigmas de programación

- **Paradigmas**
- **Programación**

Paradigmas

Un **paradigma de programación** es un **modelo** básico de diseño e implementación de programas, que permite desarrollar software conforme a ciertos **principios** o **fundamentos específicos** que se aceptan como válidos, un marco conceptual que determina los bloques básicos de construcción de software y los criterios para su uso y combinación. En otras palabras, es una colección de modelos conceptuales que juntos modelan el proceso de diseño, orientan la forma de definir los problemas y, por lo tanto, determinan la estructura final de un programa.

Desde una mirada más amplia, los paradigmas son **la forma de pensar y entender un problema y su solución**, y por lo tanto, de enfocar la tarea de la programación.

En cada paradigma hay conceptos específicos que son diferentes, y a la vez, muchos elementos en común que son medulares de la programación. También existen conceptos que si bien reciben un mismo nombre para todos, tienen diferentes implicancias en cada paradigma.

Estos conceptos, según la forma en que se presentan y articulan, constituyen las características que permiten definir cada paradigma (y también los lenguajes) y a la vez poder relacionarlos, ya sea por similitud u oposición. Hay conceptos originarios de ciertos paradigmas que se pueden aplicar en otros o ser combinados de múltiples formas. Son herramientas conceptuales cuya importancia es de primera magnitud para poder encarar con posibilidades de éxito proyectos de desarrollo de software de mediana o gran escala.

Programación

La existencia de diferentes paradigmas de programación, implica que también haya diversos conceptos de "programa". Por lo tanto se los puede compararlos para descubrir su especificidad, su dominio de aplicación, sus ventajas y limitaciones, tanto para poder elegir la mejor solución como para combinarlos.

Sin embargo, es importante remarcar que también son muchos los puntos de contacto en la formas de construcción de soluciones y principalmente en los objetivos que persiguen y en los criterios de lo que se puede denominar una buena programación:

- Plantear **modelos cercanos a la realidad** que permitan abstraerse de las especificaciones computacionales y lograr una relación lo más fluida posibles entre el dominio de aplicación y el programa.
- Diseñar implementaciones de manera que puedan ser **extendidas** y **modificadas** con el mínimo impacto en el resto de su estructura, ante cambios en la realidad o nuevos requerimientos.
- Dar flexibilidad a las soluciones para que puedan ser **reutilizadas** en múltiples contextos, incluso diferentes a los que les dieron origen.
- Diseñar una **articulación funcional** adecuada de las diferentes entidades que conforman el sistema.
- Desarrollar un **código claro**, simple y compacto.
- Construir **soluciones genéricas** que permitan abstraerse de las particularidades propias de cada tipo de entidad de software y a la vez atender a la especificidad de cada una de ellas.
- **Focalización** de las funcionalidades y componentes del sistema para poder trabajar sobre eficiencia

Capítulo 2

Los diferentes paradigmas

- **Principales paradigmas**
- **Clasificación y evolución histórica**
  - Los orígenes imperativos
  - La ruptura declarativa
  - El mundo de objetos
- **Otras formas de desarrollar programas**
  - Heurísticas
  - Concurrencia

Principales paradigmas

En la actualidad, los principales paradigmas que tienen vigencia, tanto por su desarrollo conceptual y su lugar importante en las ciencias de la computación, como por su presencia significativa en el mercado, son los siguientes:

- Paradigma **Lógico**
- Paradigma **Funcional**
- Paradigma **Imperativo o procedural**
- Paradigma de **Objetos**

Clasificación y evolución histórica

A los paradigmas se los puede clasificar conceptualmente de diversas maneras según los criterios que se prioricen. Partiendo de los principios fundamentales de cada paradigma en cuanto a las orientaciones sobre la forma para construir las soluciones y teniendo en cuenta su evolución histórica, se pueden distinguir mayores o menores similitudes entre los paradigmas que permiten organizarlos esquemáticamente en subgrupos y relacionarlos entre sí.

Los orígenes imperativos

Los primeros lenguajes de programación se basaron sobre un conjunto de premisas de funcionamiento que luego, con el correr de los años, se fueron complejizando y ampliando, pero que comparten suficientes características en común para considerarlas parte de un mismo paradigma que se denomina **imperativo o procedural**, según el énfasis que se haga en un aspecto u otro**.** En un programa tienen un papel dominante los **procedimientos** compuestos por **sentencias imperativas**, es decir aquellas que indican realizar una determinada operación que modifica los datos guardados en memoria, mediante un algoritmo en el que se detalla la **secuencia de ejecución** mediante estructuras de control. Utiliza **variables y estructuras de datos**, vinculados a celdas de memoria**,** sobre las que se realizan **asignaciones destructivas**, provocando **efecto de lado**. Un programa realiza su tarea ejecutando una **secuencia de pasos** elementales regida por **instrucciones de control** expresamente indicadas en el programa que marcan el flujo de ejecución de operaciones para resolver un problema específico. Siendo muy variados y numerosos, sus lenguajes cuentan con herramientas que permiten organizar los programas en diversos módulos o procedimientos que se distribuyen las responsabilidades, generar unidades de software genéricas, utilizar tipos de datos simples y estructuras de datos, entre otras características. En otras palabras, un programa detalla una secuencia o algoritmo que indica "**cómo**" se va procesando paso a paso la información necesaria.

Los lenguajes de este paradigma que siguen vigentes en la actualidad, lo han logrado por su **simplicidad**, su **versatilidad** y **robustez**. Además, sus elementos característicos siguen siendo retomados por numerosos lenguajes de otros paradigmas, por lo que tiene una gran importancia dentro de las ciencias de la computación.

La ruptura declarativa

Posteriormente, surgieron diferentes corrientes que objetaron los fundamentos de la programación del momento y plantearon otras premisas desde las cuales desarrollar programas. En consecuencia, se crearon nuevos lenguajes de programación, que en la actualidad se los puede conceptualizar en el marco de los **paradigmas declarativos**, incluyendo al **paradigma lógico** y al **paradigma funcional**.

Lo que los diferenció de los otros lenguajes de la época, y que consiste hoy en su principal propiedad, en poder lograr **desentenderse de los detalles de implementación un programa** en cuanto al control de ejecución, las asignaciones de memoria y la secuencia de sentencias imperativas, o sea todo lo que implique "**cómo**" se resuelve el problema, y **concentrarse en la declaración o descripción de "qué" es la solución de un problema**, creando programas de

un nivel más alto. Por lo tanto, su principal característica es la **declaratividad**, por la que sus programas especifican un conjunto de declaraciones, que pueden ser proposiciones, condiciones, restricciones, afirmaciones, o ecuaciones, que caracterizan al problema y **describen su solución**. A partir de esta información el sistema utiliza **mecanismos internos de control**, comúnmente llamado "motores", que evalúan y relacionan adecuadamente dichas especificaciones, de manera de obtener la solución. En general, las variables son usadas en expresiones, funciones o procedimientos, en los que se unifican con diferentes valores mediante el "encaje de patrones" (*pattern matching*) manteniendo la **transparencia referencial**. En ellas no se actualizan los estados de información ni se realizan asignaciones destructivas.

Que se denomine a algunos paradigmas como declarativos, no implica que no exista declarativdad en lo demás paradigmas, como tampoco que no haya ningún rasgo procesual o imperativo en estos. Significa que dichos conceptos son característicos, identificatorios y que se presentan con mayor claridad en los respectivos lenguajes. Tampoco quiere decir que sea esa la única diferencia. Simplemente es un modelo teórico para relacionar los grupos de paradigmas que tienen mayor afinidad conceptual entre ellos. La noción de **declarativdad** en mayor o menor medida está presente en cualquier paradigma, por lo que se pueden construir soluciones muy diferentes que la apliquen a su manera.

El mundo de objetos

Más recientemente, surgieron nuevos lenguajes de programación que, sin dejar de lado las premisas del paradigma imperativo, incorporaron nuevos elementos que fueron adquiriendo una importancia creciente y un impacto tal en la forma de desarrollar soluciones, que pueden conceptualizarse como otro paradigma: "**objetos"**.

Un programa hecho en objetos incluye sentencias imperativas, requiere de la implementación de procedimientos en los que se indica la secuencia de pasos de un algoritmo y hay asignaciones destructivas con efecto de lado, pero son herramientas que se utilizan en un contexto caracterizado por la presencia de **objetos que interactúan entre sí** enviándose mensajes, que deja en un segundo plano a las otras propiedades. Es una perspectiva y una comprensión del sistema diferente, más integrales, más cercanas al dominio real del problema que a las características de la arquitectura de la computadora, que se traduce en un diseño y una programación con estilo propio.

Se fundamenta en concebir a un sistema como un conjunto de entidades que representan al mundo real, los **"objetos"**, que tienen **distribuida la funcionalidad e información** necesaria y que **cooperan entre sí** para el logro de un objetivo común.

Cuenta con una estructura de desarrollo modular basada en **objetos**, que son definidos a partir de **clases**, como implementación de tipos abstractos de datos. Utiliza el **encapsulamiento** como forma de abstracción que separa las interfaces de las implementaciones de la funcionalidad del sistema (**métodos**) y oculta la información (**variables**) y un mecanismo de envío de **mensajes**, que posibilita la interacción entre los objetos y permite la **delegación** de responsabilidades de unos objetos a otros. Para realizar código genérico y extensible tiene **polimorfismo**, basado en el **enlace dinámico**, que permite que a entidades del programa interactuar indistintamente con otras y la **Herencia**, que permite que los objetos sean definidos como extensión o modificación de otros.

Otras formas de desarrollar programas

Existen también otros modelos de programación, cuya conceptualización como paradigmas es discutida, como el **heurístico** y el **concurrente**.

El proceso de generación de lenguajes y demás herramientas de programación es dinámico, cambiante y con velocidades diferentes en el mercado que en las ciencias de la computación, por lo que surgen constantemente otras formas de programación, que tienen puntos en común con alguno de los paradigmas mencionados y a la vez introducen otros elementos o acentúan algunas características, pero que es difícil de sostenerse conceptualmente como paradigmas, como por ejemplo, la programación orientada a eventos, a aspectos, etc.

Heurísticas

La **programación heurística** se basa en una forma de modelar el problema, en lo que respecta a la representación de su estructura, estrategias de búsqueda y métodos de resolución, mediante **reglas de "buena lógica"** o de "sentido común", relacionadas con el razonamiento humano, que son denominadas heurísticas. Estas reglas proporcionan, entre varios cursos de acción, uno que presenta visos de ser el **más "prometedor"**, pero **no garantiza** necesariamente el **curso de acción más efectivo**.

El entorno de programación heurístico es un entorno basado en el conocimiento humano (la experiencia), adaptativo, incremental y simbólico, y aplicable a dominios específicos en los que una buena heurística guía un proceso algorítmico o proporciona resultados superiores a éste.

Las especificaciones más relevantes del tratamiento heurístico deben tener en cuenta las características de la heurística, de la información y de la definición del problema:

- Una buena heurística debe ser simple, con **velocidad de búsqueda que no se incremente exponencialmente**, precisa y robusta.
- **La información a tratar es fundamentalmente inexacta**, simbólica o limitada, como también los resultados obtenidos, en los que no se puede garantizar un 100% de exactitud.
- La información utilizada como criterio para decidir entre los distintos cursos de acción está basada en el conocimiento previo sobre el dominio del problema, y tiene un crecimiento "incremental" a medida que se avanza en la ejecución, **incorporando el conocimiento obtenido durante la búsqueda**.
- Las especificaciones del problema deben ser claras y pueden ser de **optimización de soluciones previas** o de **satisfacción de nuevos problemas**, y por otro lado, pueden producir una o múltiples soluciones.

No se ha desarrollado un lenguaje de programación que sea en sí "heuristico", sino que en la práctica se implementan soluciones o se construyen herramientas particulares con las características mencionadas en diferentes lenguajes de programación de diferentes paradigmas.

Concurrencia

La concurrencia se ha convertido en una herramienta compartida por la mayoría de los lenguajes actuales de programación de los diferentes paradigmas. Si bien en sus orígenes surgió como un concepto distintivo con características propias que justificaron su catalogación como paradigma, en la actualidad está totalmente integrado dentro de la dinámica de funcionamiento de cualquier lenguaje de programación de alto nivel.

El concepto fundamental de la **programación concurrente** es la noción de **proceso**, entendido como una unidad de software con **su propia secuencia y su seguimiento de control**. La **ejecución simultánea** de más de un proceso permite que cooperen para resolver un mismo problema y, a la vez, requiere necesariamente que compartan los recursos del sistema y compitan por acceder a ellos. Para lograrlo, se implementan distintas estrategias de interacción entre los procesos, tales como mecanismos de comunicación y sincronización.

La **concurrencia en un lenguaje de programación y el paralelismo en el hardware son dos conceptos independientes**. Las operaciones de hardware ocurren en paralelo si ocurren al mismo tiempo. Las operaciones en el software son concurrentes si pueden ejecutarse en paralelo, aunque no necesariamente deben ejecutarse así. Podemos tener concurrencia en un lenguaje sin hardware paralelo, así como ejecución en paralelo sin concurrencia en el lenguaje.

Capítulo 3

Conceptos transversales

- **Abstracción**
- **Mudularización, encapsulamiento y delegación**
- **Declaratividad**
- **Tipos de datos**
- **Estructuras de datos**
- **Polimorfismo y software genérico**
- **Transparencia referencial y efecto de lado**
- **Asignación y unificación**
- **Modo de evaluación**
- **Orden superior**
- **Recursividad**

Abstracción

Entendiendo un sistema como una **abstracción de la realidad**, los **datos** son las entidades que representan cada una de los aspectos de la realidad que son significativos para el funcionamiento del sistema. Para que tenga sentido como abstracción de la realidad, cada dato implica un determinado **valor** y requiere de una convención que permita representarlo sin ambigüedad y procesarlo de una manera confiable.

En la misma línea, la **lógica del procesamiento** de los datos es también una abstracción de los procesos que suceden en la realidad que conforma el dominio de la aplicación. El proceso de abstraerse progresivamente de los detalles y así manejar **niveles de abstracción**, es el que permite construir sistemas complejos. Las unidades de software que realizan la funcionalidad del sistema requieren también de convenciones y criterios de ordenamiento y articulación interna tanto para un funcionamiento confiable y eficiente del sistema, como para que el proceso de construcción y mantenimiento del software sea de la forma lo más simple posible.

Mudularización, encapsulamiento y delegación

Una estrategia central de la programación es buscar la manera de organizar y distribuir la funcionalidad de un sistema complejo en **unidades más pequeñas de software con se responsabilizan de tareas específicas y que interactúan entre ellas**. Estas unidades reciben nombres diferentes según cada lenguaje de programación, como rutinas, funciones, procedimientos, métodos, predicados, subprogramas, bloques, entidades, siendo "módulos" una de las más frecuentes y que dan origen al término.

La clave de **cada módulo no conoce el funcionamiento interno de los demás módulos** con los que interactúa **sino sólo su interfaz**, es decir, la forma en que debe enviarle información adicional en forma de parámetros y cómo va a recibir las respuestas. Esta propiedad recibe diversos nombres, como el de **encapsulamiento**, ocultación de información o "caja negra". Ante la modificación de una funcionalidad en particular del sistema, en la medida que su implementación esté encapsulada en un módulo, el impacto que produce su cambio no afectará a los otros módulos que interactúan con él.

En concordancia con la distribución de responsabilidades entre las diferentes unidades de software, la **delegación** consiste en la invocación que desde un módulo se efectúa a otro módulo, de manera que el que invoca explicita qué es lo que pretende y el que es invocado se ocupa de todo lo necesario para realizarlo. Puede realizarse de numerosas maneras, variando el criterio de distribución de responsabilidades, el modo de evaluación, la forma de paso de parámetros, los tipos de datos que utiliza, de acuerdo a las posibilidades y restricciones de cada lenguaje y paradigma.

Declaratividad

La declaratividad, en términos generales, se basa en la separación del conocimiento sobre la definición del problema con la forma de buscar su solución, una **separación entre la lógica y el control**.

En un programa declarativo se especifican un **conjunto de declaraciones**, que pueden ser proposiciones, condiciones, restricciones, afirmaciones, o ecuaciones, que caracterizan al problema y **describen su solución**. A partir de esta información el sistema utiliza **mecanismos internos de control,** comúnmente llamado **"motores"**, que evalúan y relacionan adecuadamente dichas especificaciones, de manera de obtener la solución. De esta manera, en vez de ser una secuencia de órdenes, un programa es un conjunto de definiciones sobre el dominio del problema.

Basándose en la noción de **delegación**, la declaratividad plantea como criterio para distribuir las responsabilidades, separar las relacionadas con modelar o definir el conocimiento del problema de aquellas de manipular ese conocimiento para alcanzar un objetivo concreto. En otras palabras, distinguir el "qué" del "cómo".

La declaratividad brinda la posibilidad de usar una misma descripción en múltiples contextos en forma independiente de los motores que se utilicen. Permite focalizar por un lado en las cuestiones algorítmicas del motor, por ejemplo para trabajar en forma unificada sobre eficiencia, y por otro en la definición del dominio del problema y la funcionalidad de la aplicación en sí.

La noción opuesta, aunque en cierta medida complementaria, de la declaratividad, puede denominarse "**proceduralidad**". Los programas procedurales se construyen indicando explícitamente la secuencia de ejecución en la que se procesan los datos y obtienen los resultados. Para ello se detalla un **conjunto de sentencias**, ordenadas mediante estructuras de control como decisiones, iteraciones y secuencias, que conforman "algoritmos".

En un sistema complejo, no se puede hablar de declaratividad o proceduralidad como conceptos excluyentes o totalizantes, sino que coexisten y se relacionan en una permanente tensión. Dependiendo de los lenguajes y de las herramientas que se utilicen, y en particular del diseño del sistema, habrá partes del sistema que por su sentido o ubicación dentro del sistema global serán más declarativas o procedurales que otras, de manera de aprovechar las ventajas respectivas.

Tipos de datos

Un **tipo de dato**, o como también es llamado, un tipo abstracto de dato, es un **conjunto de valores** y de **operaciones asociadas** a ellos.

La utilización de diversos tipos de datos permite la agrupación o clasificación del gran volumen y variedad de valores que es necesario representar y operar en un sistema, según sus semejanzas y diferencias.

Tomando como criterio las similitudes en cuanto al contenido de lo que representan, una primera condición es la existencia de un conjunto de valores o entidades **homogéneos en su representación**. Otra condición es que los elementos del mencionado conjunto se **comporten en forma uniforme** respecto a una serie de operaciones.

Cada paradigma de programación, y en particular cada lenguaje, tiene su forma de determinar tanto la conformación de cada tipo de dato, con sus valores y operaciones, como la forma en que se relacionan entre sí conformando un **sistema de tipo de datos**.

En un lenguaje **fuertemente tipado**, toda variable y parámetro deben ser definidos de un tipo de dato en particular que se mantiene sin cambios durante la ejecución del programa, mientras que en uno **débilmente tipado** no, sino que pueden asumir valores y tipos de datos diferentes durante la ejecución del programa.

El tipo de dato al que pertenece una entidad determina la operatoria que se puede realizar con ello. Una tarea que realizan muchos de los lenguajes de programación como forma de su mecanismo interno es el **chequeo del tipo de dato** de las entidades del programa. Esta acción se realiza de diferentes

maneras, con mayor o menor flexibilidad, y en diferentes momentos, como la compilación o la ejecución del programa, y en otros no se realiza.

De todas maneras, el tipo de datos permite entender qué entidades tienen sentido en un contexto, independientemente de la forma de chequeo o si el tipado es débil o fuerte. Por ejemplo, a un bloque de software que recibe como argumento una variable, conocer de qué tipo de dato es le permite saber qué puede hacer con ella.

Estructuras de datos

Los valores atómicos, es decir, aquellos que no pueden ser descompuestos en otros valores, se representan mediante **tipos de datos simples**.

En contrapartida, en un **tipo de dato compuesto** los valores están compuestos a su vez por otros valores, de manera que conforma una **estructura de datos**. Cada uno de los valores que forman la estructura de datos corresponde a algún tipo de dato que puede ser tanto simple como compuesto. Su utilidad consiste en que se pueden procesar en su conjunto como una unidad o se pueden descomponer en sus partes y tratarlas en forma independiente. En otras palabras, las estructuras de datos son conjuntos de valores.

Polimorfismo y software genérico

El polimorfismo es un concepto para el que se pueden encontrar numerosos y diferentes definiciones. En un sentido amplio, más allá de las especificidades de cada paradigma y del alcance que se le de a la definición del concepto desde la perspectiva teórica desde la que se lo aborde, el objetivo general del **polimorfismo** es construir piezas de software genéricas que trabajen indistintamente con diferentes tipos de entidades, para otra entidad que requiere interactuar con ellas; en otras palabras, que dichas entidades puedan ser intercambiables. Mirando con mayor detenimiento los mecanismos que se activan y sus consecuencias para el desarrollo de sistemas, se pueden distinguir dos grandes situaciones.

Hay polimorfismo cuando, ante la existencia de dos o más bloques de software con una misma interfaz, otro bloque de software cualquiera puede trabajar indistintamente con ellos. Esta noción rescata la existencia de tantas implementaciones como diferentes tipos de entidades se interactúe.

Una entidad emisora puede interactuar con cualquiera de las otras entidades de acuerdo a las características de la interfaz común, y la entidad receptora realizará la tarea solicitada de acuerdo a la propia implementación que tenga definida, independientemente de las otras implementaciones que tengan las otras entidades. Consistentemente con la noción de delegación, la entidad emisora se desentiende de la forma en que las otras entidades implementaron sus respuestas, ya sea fue igual, parecida o totalmente diferente.

Analizando el concepto desde el punto de vista de las entidades que responden a la invocación, el proceso de desarrollo debe contemplar la variedad y especificidad de cada una para responder adecuadamente a lo que se les solicita. Desde el punto de vista de la entidad que invoca, el proceso es transparente, y es en definitiva ésta, la que se ve beneficiada por el uso del concepto.

Otra forma de obtener un bloque de software genérico es ante el caso en que las entidades con las que el bloque quiere comunicarse, en vez de tener diferentes implementaciones, aún siendo de diferente tipo, sean lo suficientemente similares para compartir una misma y única implementación. Desde el punto de vista de la entidad que invoca, el proceso continúa siendo transparente y sigue aprovechando los beneficios de haber delegado la tarea y haberse despreocupado de qué tipo de entidad es la receptora. En este caso, el concepto de polimorfismo se relaciona o se basa en la coexistencia de herencia, de variables de tipo de dato o en las formas débiles de declaración y chequeo de tipos de datos, dependiendo de las diferentes herramientas de cada paradigma.

Transparencia referencial y efecto de lado

La **transparencia referencial** consiste en que el valor de una expresión depende únicamente del valor de sus componentes, de manera que siempre que se evalúa el mismo bloque de software con los mismos parámetros, se obtiene el mismo resultado, sin importar el comportamiento interno de dicho bloque. Su evaluación no produce un cambio en el estado de información del sistema que pueda afectar una posterior evaluación del mismo u otro bloque.

Entre las ventajas, la transparencia referencial da robustez, ya que se garantiza que el agregado de nuevas unidades de software no interfieren con las existentes general algún efecto colateral. Los tests son más confiables, ya que toda consulta o evaluación, responde siempre de la misma forma. El conjunto de variables o condiciones iniciales a tener en cuenta está acotado a la misma consulta. Los errores tienden a ser más locales y a no propagarse por todo el sistema. Aporta independencia entre las unidades de software, ya que trabajan sobre valores diferenciados.

El **efecto de lado** (*side effect*), también traducido como **efecto colateral**, se produce cuando el resultado de la evaluación de un bloque de software depende de otros valores o condiciones del ambiente más allá de sus parámetros. Su evaluación incluye acciones que afectan un estado de información que sobrevive a la evaluación del bloque, por lo que influye en una posterior evaluación del mismo u otro bloque.

La utilización del efecto de lado representa mejor la dinámica del sistema, ya que cuando hay un **estado** de información que mantener actualizado, es un efecto deseado que el funcionamiento del sistema provoque cambios en la información. Refleja la interacción de diferentes partes del sistema, ya que una

unidad de software puede realizar cambios en la información que incidan en el funcionamiento de otra unidad.

Asignación y unificación

La **asignación destructiva** es una operación que consiste en cambiar la información representada por una **variable**, de forma tal que si se consulta su valor antes y después de dicha operación, se obtiene un resultado distinto. Estas asignaciones se realizan repetitivamente sobre la misma celda de memoria, remplazando los valores anteriores. La asignación determina el **estado** de una variable, que consiste en el valor que contiene en un momento en particular. La asignación destructiva es la forma más usual de provocar efecto de lado, pero no la única, ya que, dependiendo de los lenguajes, hay otro tipo de instrucciones que también permiten modificar el estado de información de un sistema.

La **unificación** es un mecanismo por el cual una variable que no tiene valor, asume un valor. Una vez unificada, o "ligada", como también se le dice, una variable no cambia su valor, por lo que no existe la noción de estado. La duración de la unificación está condicionado por el alcance que tienen las variables en cada lenguaje en particular, pudiendo una variable unificarse con varios valores alternativos en diferentes momentos de la ejecución del bloque de software en el que se encuentran. Se suele denominar como "indeterminada" a una variable sin ligar o no unificada. La unificación se encuentra íntimamente relacionado con el mecanismo de "**encaje de patrones**" (*pattern matching*)**.**

Modo de evaluación

Los parámetros que se utilizan en la invocación de un bloque de software cualquiera pueden ser evaluados en diferentes momentos de acuerdo al modo de evaluación que utilice el lenguaje de programación.

La **evaluación ansiosa** consiste en que los argumentos son evaluados antes de invocar al bloque de software y es responsabilidad de la entidad que invoca.

La **evaluación diferida** plantea que la evaluación de los argumentos es responsabilidad del bloque de software invocado, quien decide el momento en que lo hará. Provoca que se difiera la evaluación de una expresión, permitiendo que en algunos casos, de acuerdo a cómo sea la implementación, no sea necesario evaluarla nunca, con el consiguiente beneficio en términos de eficiencia.

La evaluación diferida es utilizada en cierto tipo de situaciones que serían consideradas erróneas o imposibles de resolver con la evaluación ansiosa, como por ejemplo los bucles o las listas infinitas.

Orden superior

Asumiendo un esquema de un único orden, en un programa existen por un lado datos y por otro los procedimientos –y todo bloque de software- que trabajan con ellos, de manera tal que los procedimientos reciben datos como parámetros y devuelven datos como resultados. La noción de **orden superior** plantea que **los procedimientos son tratados como datos** y en consecuencia pueden ser utilizados como parámetros, representados en variables, devueltas como resultados u operados en cálculos más complejos con otros datos. Se denomina de orden superior a los procedimientos que **reciben como argumentos** a otros procedimientos.

Recursividad

La recursividad, entendida como iteración con asignación no destructiva, está relacionada con el principio de **inducción**. En general, un bloque de software recursivo se define con al menos un término recursivo, en el que se vuelve a invocar el bloque que se está definiendo, y algún término no recursivo como caso base para detener la recursividad.

Bibliografía

- **ALONSO AMO, F y SEGOVIA PEREZ, F**. *Entornos y Metodologías de Programación*. Paraninfo.
- **WATT, David**. *Programming Languajes Concepts and Paradigms*. Prentice Hall.
- **GHEZZI, Carlo y JAZAYERI, Mehdi**. *Conceptos de Lenguajes de Programación*. Díaz de los Santos.
- **RAVI y SETHI.** *Lenguajes de programación - Conceptos y constructores.* Addison Wesley.

Índice