PROGRAMACIÓN ORIENTADA A OBJETOS

José Sánchez Juárez

Agosto, 2019

II

Índice general

Índice de figuras

Índice de cuadros

Introducción a la programación orientada a objetos

> ARTICULO 1.- El Instituto Politécnico Nacional es la institución educativa del Estado creada para consolidar, a través de la educación, la Independencia Económica, Científica, Tecnológica, Cultural y Política para alcanzar el progreso social de la Nación, de acuerdo con los objetivos Históricos de la Revolución Mexicana, contenidos en la Constitución Política de los Estados Unidos Mexicanos.
>
> Ley orgánica.
> Instituto Politécnico Nacional

El paradigma orientado a objetos es la forma más común de analizar, diseñar,y desarrollar sistemas de aplicación, especialmente muy grandes. Para entender este paradigma, podríamos preguntarnos: ¿que exactamente significa la frase .ºrientado a objetos¿ Buscando en la literatura, etiquetar algo como orientado a objetos implica que el objeto juega un rol central, y elaboramos esta característica como una perspectiva que ve el elemento de una situación dada por descomposición de objetos y relación de objetos. En un sentido amplio, esta idea podría aplicarse a cualquier entorno y los ejemplos de su aplicación pueden aplicarse en negocios, química, ingeniería e incluso filosofía. Nuestro negocio es con la creación de software y, por lo tanto, este libro se concentra en el análisis orientado a objetos, diseño e implementación de sistemas de software. Nuestras situaciones son, por tanto, problemas que son susceptibles a las soluciones de software y los sistemas de software que se crean en respuesta a estos problemas.

El diseño es una actividad complicada en cualquier contexto simple porque hay competencia de intereses, donde se tienen que hacer una selección de criterios en cada paso con información incompleta. Como un resultado, las decisiones son frecuentemente hechas de una combinación de reglas derivadas de experiencias pasadas. El diseño de software no es una excepción de esto, y en el proceso de diseñar un sistema, hay muchos puntos donde tales decisiones tienen que ser hechas. Haciendo selecciones informadas en cualquier campo de actividad se requiere un entendimiento de la filosofía y de las fuerzas que se han formado. Es

por ello apropiado comenzar nuestro estudio de análisis de software orientado a objetos y diseño aplicando la filosofía y el desarrollo en este campo hasta la actualidad. A través del caso de estudio usado en este texto, el lector encontrará ejemplos de como esta guía filosófica nos auxilia para hacer selecciones en todos los pasos.

Este capítulo, por lo tanto, pretende dar al lector una amplia introducción al complejo tema de desarrollo de software orientado a objetos. Comenzamos con una descripción general de la circunstancia que motivaron su desarrollo y por qué se convirtió en el enfoque deseado para desarrollo de software. En el curso de esta discusión, presentamos los conceptos centrales que caracterizan la metodología, cómo este desarrollo ha influido en nuestra visión del software, y algunos de sus pros y contras. Concluimos presentando una breve historia de la evolución del enfoque orientado a objetos.

¿Que es el desarrollo orientado a objetos?

La visión tradicional de un programa de computadora es la de un proceso que ha sido codificado en un formulario que se puede ejecutar en una computadora. Esta visión se originó del hecho de que la primera Las computadoras se desarrollaron principalmente para automatizar un proceso bien definido (es decir, un algoritmo) para el cálculo numérico, y se remonta a las primeras computadoras del programa almacenado. Acuerdo ingly, el proceso de creación de software fue visto como una traducción de una descripción en algunos Lenguaje "natural"para una secuencia de operaciones que podrían ejecutarse en una computadora. Como Muchos argumentarían que este paradigma sigue siendo la mejor manera de introducir la noción de programa. ming a un principiante, pero a medida que los sistemas se vuelven más complejos, su eficacia en el desarrollo Las soluciones se volvieron sospechosas. Este cambio de perspectiva por parte de los desarrolladores de software sucedió durante un período de tiempo y fue alimentado por varios factores, incluido el alto costo de desarrollo y los constantes esfuerzos para encontrar usos para el software en nuevos dominios. Uno podria argumenta con seguridad que las aplicaciones de software desarrolladas en años posteriores tuvieron dos diferencias características:

1. Comportamiento que fue difícil de caracterizar como un proceso.
2. Requisitos de confiabilidad, rendimiento y costo que los desarrolladores originales no enfrentaron.

El enfoque çentrado en el proceso"para el desarrollo de software utilizaba lo que se denomina funciones de arriba hacia abajo descomposición nacional. El primer paso en tal diseño fue reconocer lo que el proceso tenía para entregar (en términos de entrada y salida del programa), que fue seguido por descomposición del proceso en módulos funcionales. Se definieron estructuras para almacenar datos y el cómputo se realizó invocando los módulos, que realizaron algunos cálculos tanto en los elementos de datos almacenados. La vida de un diseño centrado en el proceso fue corta porque cambios en la especificación del proceso (algo relativamente poco común con algo numérico los ritmos en comparación con las aplicaciones comerciales) requirieron un cambio en todo el programa.

Esto a su vez resultó en una incapacidad para reutilizar el código existente sin una sobrecarga considerable. Como resultado, los diseñadores de software comenzaron a analizar sus propios enfoques y también a estudiar firmar procesos y principios que estaban siendo empleados por ingenieros en otras disciplinas. La polinización cruzada de ideas de otras disciplinas de ingeniería comenzó poco después, y surgieron disciplinas de "diseño de software.eingeniería de software". A este respecto, es interesante observar el proceso utilizado para diseñar elementos eléctricos simples. sistemas electromecánicos. Durante varias décadas, ha sido bastante fácil para las personas con conocimiento limitado de los principios de ingeniería diseñar y armar sistemas simples en sus patios y garajes. Tanto es así, que se ha convertido en un pasatiempo que incluso un niño de diez años podría perseguir. Las razones de este éxito son fáciles de ver: diseños fácilmente comprensibles, soluciones similares (estándar) para una serie de problemas, de fácil acceso y bien definidos "Biblioteca"de "bloques de construcción", intercambiabilidad de componentes entre sistemas, etc. Algunos de los pioneros en el campo del diseño de software comenzaron a preguntarse si no podían también diseñe software que utilice dichos componentes "listos para usar". El paradigma orientado a objetos, se podría argumentar que realmente ha evolucionado en respuesta a esta perspectiva. Hay, por supuesto, varias diferencias con el proceso de diseño del hardware (inevitable, porque la naturaleza del software es fundamentalmente diferente del hardware), pero se pueden establecer paralelos entre muchas de las definiciones de las características del diseño de hardware y lo que hoy defiende el buen software: Diseño recomendado. Esta metodología, como veremos en los capítulos siguientes, nos proporciona con un proceso paso a paso para el diseño de software, un lenguaje para especificar el resultado de cada paso del proceso para que podamos pasar sin problemas de una etapa a la siguiente, la capacidad para reutilizar diseños anteriores, soluciones estándar que se adhieren a principios de diseño bien razonados e, incluso, la capacidad de corregir gradualmente un diseño deficiente sin romper el sistema. La filosofía general aquí es definir un sistema de software como una colección de objetos de varios tipos que interactúan entre sí a través de interfaces bien definidas. A diferencia de un duro componente de software, un objeto de software puede diseñarse para manejar múltiples funciones y puede por lo tanto, participe en varios procesos. Un componente de software también es capaz de almacenar datos, que agrega otra dimensión de complejidad al proceso. La manera en que todos de esto se ha alejado de la visión tradicional orientada al proceso es que en lugar de implementar al realizar un proceso completo de principio a fin y definir las estructuras de datos necesarias en el camino, primero analice todo el conjunto de procesos y, a partir de este, identifique el software necesario con ponentes de cada componente que representa una abstracción de datos y está diseñado para almacenar información junto con procedimientos para manipular lo mismo. La ejecución de los procesos originales es luego desglosado en varios pasos, cada uno de los cuales puede asignarse lógicamente a uno de los componentes de software, los componentes también pueden comunicarse entre sí según sea necesario para completar el proceso.

Conceptos clave para el diseño orientado a objetos

Durante el desarrollo de este paradigma, como cabría esperar, varias ideas y los ensayos fueron juzgados y descartados. Con los años, el campo se ha estabilizado para que tengamos presente con seguridad las ideas clave cuya solidez ha resistido la prueba del tiempo.

0.0.1. El rol central de los objetos

La orientación a objetos, como su nombre lo indica, convierte a los objetos en la pieza central del diseño de software. El diseño de los sistemas anteriores se centró en los procesos, que eran susceptibles de cambio, y cuando se produjo este cambio, muy poco del viejo sistema era reutilizable". La noción de un objeto se centra en una pieza de datos y las operaciones (o métodos) podrían usarse para modificarlo. Esto hace posible la creación de una abstracción que es muy estable ya que no depende de los requisitos cambiantes de la aplicación. La ejecución de cada proceso depende en gran medida de los objetos para almacenar los datos y proporcionar operaciones necesarias; con algo de trabajo adicional, todo el sistema se .^{en}sambla.^{a} partir de objetos.

0.0.2. La noción de clase

Las clases permiten que un diseñador de software vea los objetos como diferentes tipos de entidades. Visitar los objetos de esta manera nos permiten utilizar los mecanismos de clasificación para clasificar estos tipos, definir jerarquías y comprometerse con las ideas de especialización y generalización de objetos.

0.0.3. Especificación abstracta de funcionalidad

En el curso del proceso de diseño, el ingeniero de software especifica las propiedades de los objetos. (y, por implicación, las clases) que necesita un sistema. Esta especificación es abstracta ya que no impone restricciones sobre cómo se logra la funcionalidad. Esta especificación, llamada interfaz o clase abstracta, es como un contrato para el implementador lo que también facilita la verificación formal de todo el sistema.

0.0.4. Un lenguaje para definir el sistema

El lenguaje de modelado unificado (UML) ha sido elegido por consenso como herramienta estándar para describir los productos finales de las actividades de diseño. Los documentos generados en este lenguaje se puede entender universalmente y, por lo tanto, es análogo a los "planosütilizados en otras disciplinas de ingeniería.

0.0.5. Soluciones estándar

La existencia de una estructura de objetos facilita la documentación de soluciones estándar, llamadas patrones de diseño. Las soluciones estándares se encuentran en todas las etapas del desarrollo de software, pero los patrones de diseño son quizás la forma más común de reutilización de soluciones.

0.0.6. Un proceso de análisis para modelar un sistema

La orientación a objetos nos proporciona una forma sistemática de traducir una especificación funcional a un diseño conceptual. Este diseño describe el sistema en términos de clases conceptuales de que los pasos posteriores del proceso de desarrollo generan las clases de implementación que constituyen el software terminado.

0.0.7. La notación de extandabilidad y adaptabilidad

El software tiene una flexibilidad que normalmente no se encuentra en el hardware, y esto nos permite modificar las entidades existentes en pequeñas formas de crear otras nuevas. la herencia, que crea un nuevo clase descendiente que modifica las características de una clase existente (ancestro) y su composición, que utiliza objetos que pertenecen a clases existentes como elementos para constituir una nueva clase, son mecanismos que permiten tales modificaciones con clases y objetos.

Otros conceptos relacionados

A medida que se desarrolló la metodología orientada a objetos, la ciencia del diseño de software progresó también, y se identificaron varias propiedades de software deseables. No es lo suficientemente central como para ser llamados conceptos orientados a objetos, estas ideas están estrechamente vinculadas a ellos y son quizás mejor entendidas debido a estos desarrollos.

0.0.8. Diseño modular y encapsulamiento

La modularidad se refiere a la idea de armar un sistema grande desarrollando un número de componentes distintos de forma independiente y luego integrarlos para proporcionar la requerida funcionalidad, este enfoque, cuando se usa correctamente, generalmente hace que los módulos individuales relativamente simples y, por lo tanto, el sistema sea más fácil de entender que uno diseñado como estructura monolítica. En otras palabras, dicho diseño debe ser modular. La función del sistema debe ser proporcionada por una serie de módulos bien diseñados y cooperantes. Cada modulo obviamente, debe proporcionar cierta funcionalidad claramente especificada por una interfaz. La interfaz también define cómo otros componentes pueden interactuar o comunicarse con el módulo. Nos gustaría que un módulo especifique claramente lo que hace, pero no exponga su implementación. Esta separación de preocupaciones da lugar a la noción de encapsulación, lo que significa que el módulo oculta detalles de su implementación de agentes externos. Los tipos de datos abstractos (ADT),

la generalización de tipos de datos primitivos como enteros y caracteres, es un ejemplo de aplicación de encapsulación. El programador especifica la colección de operaciones en el tipo de datos y las estructuras de datos que se necesitan para el almacenamiento de datos. Los usuarios del ADT realizan las operaciones sin preocuparse por la implementación.

0.0.9. Cohesión y acoplamiento

Cada módulo proporciona cierta funcionalidad; la cohesión de un módulo nos dice qué tan bien están las entidades dentro de un módulo para trabajar juntas para proporcionar esta funcionalidad. La cohesión es una medida de cuán enfocadas están las responsabilidades de un módulo. Si las responsabilidades de un módulo son no relacionadas o variadas para utilizar diferentes conjuntos de datos, se reduce la cohesión. Los módulos altamente cohesivos tienden a ser más confiables, reutilizables y comprensibles que los menos cohesivos. Al aumentar la cohesión, nos gustaría que todos los componentes contribuyan a algunas bien definidas responsabilidades del módulo. Esta puede ser una tarea bastante desafiante. En contraste, el peor enfoque sería asignar arbitrariamente entidades a los módulos, lo que resulta en un módulo cuyos constituyentes no tienen una relación obvia.

El acoplamiento se refiere a la dependencia de los módulos entre sí. El hecho mismo de que nosotros al dividir un programa en múltiples módulos se introduce algún acoplamiento en el sistema. Podría producirse pling debido a varios factores: un módulo puede referirse a variables definidas en otro módulo o un módulo puede llamar a métodos de otro módulo y usar el valor de retorno. La cantidad de acoplamiento entre módulos puede variar. En general, si los módulos no dependen de la implementación de los demás, es decir, los módulos dependen solo de la información publicada en las superficies de otros módulos y no en sus componentes internos, decimos que el acoplamiento es bajo. En En tales casos, los cambios en un módulo no requerirán cambios en otros módulos siempre y cuando ya que las interfaces en sí no cambian. El acoplamiento bajo nos permite modificar un modulo sin preocuparse por las ramificaciones de los cambios en el resto del sistema. Por el contrario, un alto acoplamiento significa que los cambios en un módulo requerirían cambios en otros módulos, que pueden tener un efecto dominó y también dificultar su comprensión del código.

0.0.10. Modificabilidad y comprobabilidad

Un componente de software, a diferencia de su contraparte de hardware, puede modificarse fácilmente en pequeños formas. Esta modificación se puede hacer para cambiar tanto la funcionalidad como el diseño. La habilidad de cambiar la funcionalidad de un componente permite que los sistemas sean más adaptables; los avances en la orientación a objetos han establecido estándares más altos para la adaptabilidad. Mejorando el el diseño a través del cambio incremental se logra refactorizando, nuevamente un concepto que debe su origen al desarrollo del enfoque orientado a objetos. Hay algún riesgo asociado con actividades de ambos tipos; y en ambos casos, la organización del sistema en términos de objetos y clases ha ayudado a desarrollar procedimientos sistemáticos que mitigan el riesgo.

La capacidad de prueba de un concepto, en general, se refiere tanto a la falsabilidad como a la facilidad con la que pueden encontrar contra ejemplos y la factibilidad práctica de reproducir dicho contra ejemplo. En el contexto de los sistemas de software, simplemente se puede expresar como la facilidad con la que se puede encontrar errores en un software y en qué medida la estructura del sistema facilita la detección de errores. Varios conceptos en las pruebas de software (por ejemplo, la idea de las pruebas unitarias) deben su importancia a los conceptos que surgieron del desarrollo del paradigma orientado a objetos.

Beneficios y desventajas del paradigma

Desde un punto de vista práctico, es útil examinar cómo la metodología orientada a objetos tiene modificado el panorama del desarrollo de software. Como con cualquier desarrollo, tenemos pros y contras. Las ventajas que se enumeran a continuación son en gran medida consecuencia de las ideas presentadas en las secciones anteriores.

1. Los objetos a menudo reflejan entidades en los sistemas de aplicación. Esto hace que sea más fácil para que un firmante para llegar a clases en el diseño. En un diseño orientado a procesos, es mucho más difícil encontrar una conexión que pueda simplificar el diseño inicial.

2. La orientación a objetos ayuda a aumentar la productividad mediante la reutilización del software existente. La herencia hace que sea relativamente fácil ampliar y modificar la funcionalidad proporcionada por un clase. Los diseñadores de idiomas a menudo suministran bibliotecas extensas que los usuarios pueden ampliar.

3. Es más fácil acomodar los cambios. Una de las dificultades con el desarrollo de aplicaciones está cambiando los requisitos. Con cierto cuidado durante el diseño, es posible para aislar las diferentes partes de un sistema en clases.

4. La capacidad de aislar cambios, encapsular datos y emplear modularidad reduce los riesgos involucrados en el desarrollo del sistema.

Las ventajas anteriores no vienen sin una etiqueta de precio. Quizás la víctima número uno del paradigma es la eficiencia. El proceso de desarrollo orientado a objetos introduce muchos capas de software, y esto ciertamente aumenta los gastos generales. Además, la creación de objetos y la destrucción es cara. Las aplicaciones modernas tienden a presentar una gran cantidad de objetos que interactúan entre sí de manera compleja y al mismo tiempo admiten un usuario con interfaz visual. Esto es cierto si se trata de una aplicación bancaria con numerosos objetos de cuenta o un videojuego que a menudo tiene una gran cantidad de objetos. Los objetos tienden a tener complejos asociaciones, que pueden dar como resultado una no localidad, lo que lleva a tiempos de acceso de memoria deficientes. Programadores y diseñadores educados en otros paradigmas, generalmente en el paradigma imperativo, les resulta difícil aprender y utilizar principios orientados a objetos. En llegar a clases, los diseñadores inexpertos pueden confiar demasiado en las entidades de la aplicación del sistema,

terminando con sistemas que no son adecuados para su reutilización. Los programadores también necesitan acclimatización de algunas personas que estiman que un programador tarda hasta un año en comenzar sentirse cómodo con estos conceptos. Algunos investigadores opinan que los entornos de gramática tampoco se han mantenido al día con la investigación en capacidades lingüísticas. Ellos consideran que muchos de los editores y las instalaciones de prueba y depuración siguen siendo fundamentalmente orientado al paradigma imperativo y no respalda directamente muchos de los avances tales como patrones de diseño.

Historia del paradigma

La historia del enfoque de programación orientada a objetos se remonta a la idea de ADT y el concepto de objetos en el lenguaje de programación Simula 67, desarrollado en la década de 1960 para realizar simulaciones. El primer verdadero lenguaje de programación orientado a objetos. que apareció antes de que la comunidad de desarrollo de software más grande fuera Smalltalk en 1980, desarrollado en Xerox PARC. Smalltalk utilizaba objetos y mensajes como base para la comunicación. Las clases pueden ser creadas y modificadas dinámicamente. La mayor parte del vocabulario en el paradigma orientado a objetos se originó a partir de este lenguaje. A fines de la década de 1970, Bjarne Stroustrup, que realizaba un doctorado en Inglaterra. Necesitaba un lenguaje para hacer simulación de sistemas distribuidos. Desarrolló un lenguaje indicador basado en el concepto de clase en Simula, pero este lenguaje no fue particularmente eficaz. Sin embargo, él persiguió su intento y desarrolló un lenguaje orientado a objetos en los laboratorios Bell como un derivado de C, que se convertiría en uno de los más exitosos lenguajes de programación, C ++. El lenguaje fue estandarizado en 1997 por el Instituto Nacional de Normas (ANSI) estadounidense. La década de 1980 vio el desarrollo de varios otros idiomas como ObjectLisp, CommonLisp, Common Lisp Object System (CLOS) y Eiffel. La creciente popularidad de la modelación orientado a objetos también impulsó cambios en el lenguaje Ada, originalmente patrocinado por el Departamento de Defensa de los Estados Unidos en 1983. Esto resultó en Ada 9x, una extensión de Ada 83, con conceptos orientados a objetos que incluyen herencia, polimorfismo y enlace dinámico. La década de 1990 vio dos grandes eventos. Uno fue el desarrollo de la programación Java, lenguaje de 1996. Java parecía ser un derivado de C ++, pero muchos de los controvertidos y los conceptos problemáticos en C ++ se eliminaron en él. Aunque era relativamente un lenguaje simple cuando se propuso originalmente, Java ha sufrido adiciones sustanciales, en el futuro las versiones que lo convertiran en un lenguaje moderadamente difícil. Java también viene con un impresionante colección de bibliotecas (llamadas paquetes) para soportar el desarrollo de aplicaciones. Un segundo evento principalmente fue la publicación del libro Design Patterns de Gamma et al. en 1994. El libro consideró preguntas de diseño específicas (23 de ellas) y proporcionó enfoques generales para resolver usando construcciones orientadas a objetos. El libro (como también el enfoque que defendió) fue un gran éxito ya que tanto los profesionales como los académicos pronto reconocieron su importancia. Los últimos años vieron la aceptación de algunos lenguajes dinámicos orientados a objetos que fueron desarrollados en la década de 1990. Los lenguajes dinámicos permiten a los usuarios una mayor

flexibilidad, por ejemplo la capacidad de agregar dinámicamente un método a un objeto en tiempo de ejecución. Uno de esos idiomas es Python, que se puede utilizar para resolver una variedad de aplicaciones, incluido el programa web ming, bases de datos, cómputos científicos y numéricos y redes. Otra lenguaje dinámico es el lenguaje, Ruby, que está aún más orientado a objetos, ya que todo en el lenguaje, incluso los números y los tipos primitivos, son un objeto.

Discusión y lectura adicional

En este capítulo, hemos dado una introducción al paradigma orientado a objetos. Los conceptos centrales orientados a objetos como clases, objetos e interfaces serán abordados en los próximos tres capítulos. La cohesión y el acoplamiento, que son los principales problemas de diseño de software, ser temas recurrentes para la mayor parte del texto. Se aconseja al lector que aprenda o se actualice los conceptos no orientados a objetos. Del lenguaje Java lea el Apéndice A antes de pasar al siguiente capítulo. Vale la pena y es divertido leer una breve historial de los lenguajes de programación desde un texto estándar sobre el tema como Sebesta [33]. El lector también puede encontrar útil obtener las perspectivas de los diseñadores de los lenguajes orientados a objetos (como el dado en C ++ por Stroustrup [37]).

0.0.11. Ejercicios

1. Identifique a los jugadores que tendrían interés en el proceso de desarrollo de software. ¿Cuales son las preocupaciones de cada uno? ¿Cómo se beneficiarían del modelo orientado a objetos?

2. Piense en algunas empresas comunes y en las actividades en las que participan los desarrolladores de software. ¿Cuáles son los conjuntos de procesos que les gustaría automatizar? ¿Hay alguno que necesite software solo para un proceso?

3. ¿Cómo apoya el modelo orientado a objetos la noción de ADT y encapsulación?

4. Considere una aplicación con la que esté familiarizado, como un sistema universitario. Dividir las entidades de esta aplicación en grupos, identificando así las clases.

5. En la pregunta 4, supongamos que ponemos todo el código (correspondiente a todas las clases) en una sola clase. ¿Qué pasa con la cohesión y el acoplamiento?

6. ¿Cuáles son los beneficios de aprender patrones de diseño?

Bases de la programación orienta a objetos

En el último capítulo, vimos que la estructura del programa fundamental en un programa orientado a objetos es el objeto. También describimos el concepto de una clase, que es similar a los ADT ya que puede usarse para crear objetos de tipos que el lenguaje no admite directamente. En el último capítulo, vimos que la estructura fundamental del programa en un objeto orientado. En este capítulo, describimos en detalle cómo construir una clase. Utilizaremos la gramática del lenguaje Java (como haremos a lo largo del libro). Vamos a presentar el Unified Modeling Language (UML), que es una notación para describir el diseño de sistemas orientados a objetos. También discutimos las interfaces, un concepto que nos ayuda a especificar los requisitos del programa y demostrar sus usos.

0.0.12. Las bases

Para entender la noción de objetos y clases, comenzamos con una analogía. Con respecto a un coche el fabricante decide construir un auto nuevo, se dedica un esfuerzo considerable en una variedad de actividades antes de que el primer automóvil salga de las líneas de montaje. Éstos incluyen:

1. Identificación de la comunidad de usuarios para el automóvil y evaluación de las necesidades del usuario. Para esto, el fabricante puede formar un equipo.

2. Después de evaluar los requisitos, el equipo puede expandirse para incluir ingenieros automotrices y otros especialistas que elaboran un diseño preliminar.

3. Se puede utilizar una variedad de métodos para evaluar y refinar el diseño inicial (el equipo puede tener experiencia en la construcción de un vehículo similar): se pueden construir prototipos, simulaciones y se puede realizar un análisis matemático.

Quizás después de meses de tal actividad, se complete el proceso de diseño. Otro paso que es necesario realizar es la construcción de la planta donde se producirá el automóvil. Se debe establecer una línea de montaje y contratar personas. Después de tales pasos, la compañía está lista para producir los automóviles. El diseño ahora se reutiliza para muchos tiempos de fabricación. Por supuesto, el diseño puede tener que ser ajustado durante el proceso según las observaciones de la empresa y los comentarios de los usuarios. El desarrollo de

sistemas de software a menudo siguen un patrón similar. Las necesidades del usuario tienen que ser evaluadas, se debe hacer un diseño y luego se debe construir el producto. Desde el punto de vista de los sistemas orientados a objetos, un aspecto diferente de la fabricación de automóviles El proceso de turing es importante. El diseño de un determinado tipo de automóvil requerirá tipos específicos del motor, la transmisión, el sistema de frenos, etc., y cada una de estas partes en sí misma tiene su diseño propio (diseño original), plantas de producción, etc. En otras palabras, la compañía sigue la misma filosofía en la fabricación de las piezas individuales que en la producción de el coche. Por supuesto, algunas partes se pueden comprar a los fabricantes, pero a su vez siguen el mismo enfoque. Dado que la actividad de diseño es costosa, un fabricante reutiliza el diseño para fabricar las partes o los autos. El enfoque anterior se puede comparar con el diseño de sistemas orientados a objetos que están compuestos de muchos objetos que interactúan entre sí. A menudo, estos objetos representan los jugadores de la vida real y sus interacciones representan interacciones de la vida real. Como el diseño de un auto es una colección de los diseños individuales de sus partes y un diseño de la interacción de estas partes, el diseño de un sistema orientado a objetos consiste en diseños de sus partes constituyentes y sus interacciones. Por ejemplo, un sistema bancario podría tener un conjunto de objetos que representan a los clientes, otro conjunto de objetos que representan cuentas y un tercer conjunto de objetos que corresponden a préstamos. Cuando un cliente realmente hace un depósito en su cuenta en la vida real, el sistema actúa sobre el objeto de la cuenta correspondiente para imitar el depósito en el software. Cuando un cliente solicita un préstamo, se crea un nuevo objeto de préstamo y se conecta al objeto del cliente; cuando un pago se realiza sobre el préstamo, el sistema actúa sobre el objeto del préstamo correspondiente. Obviamente, estos objetos tienen que ser creados de alguna manera. Cuando un nuevo cliente ingresa al sistema, deberíamos poder crear un nuevo objeto de cliente en software. Esta entidad de software, el objeto del cliente debe tener todas las características relevantes del cliente de la vida real. Por ejemplo, debería ser posible asociar el nombre y la dirección del cliente con este objeto sin embargo, los atributos del cliente que no son relevantes para el banco no serán representado en software. Como ejemplo, es difícil imaginar que un banco esté interesado en si un cliente es diestro; por lo tanto, el sistema de software no tendrá este atributo.

DEFINICIÓN 1 (Atributo.) *Un atributo es una propiedad que asociamos con un objeto; sirve para describir el objeto que contiene algún valor que se requiere para el procesamiento. [5] .*

El mecanismo de clase en lenguajes orientados a objetos proporcionan una forma de crear tales objetos. Una clase es un diseño que se puede reutilizar cualquier cantidad de veces para crear objetos. Por ejemplo, considerar un sistema orientado a objetos para una universidad. Hay objetos de estudiante, objetos de instructor, objetos de miembros del personal, etc. Antes de crear tales objetos, creamos las clases que sirven como planos para estudiantes, instructores, miembros del personal y cursos de la siguiente manera:

Listing 1: Clases.

```
1  public class Student {
2  // codigo pera implementar un solo estudiante
3  }
4  public class Instructor {
5  // codigo pera implementar un solo instructor
6  }
7  public class StaffMember {
8  // codigo pera implementar un solo miembro del staff
9  }
10 public class Course {
11 // codigo pera implementar un solo curso
12 }
```

Las definiciones anteriores muestran cómo crear cuatro clases, sin dar ningún detalle. (Nosotros deberíamos poner los detalles donde hemos dado comentarios.) La clase de token es una palabra clave que dice que estamos creando una clase y que el siguiente token es el nombre de la clase. De este modo, hemos creado cuatro clases: Estudiante, Instructor, StaffMember y Curso. El corchete izquierdo () significa el comienzo de la definición de la clase y el corchete derecho correspondiente () finaliza la definición. El token público es otra palabra clave que hace que la clase correspondiente esté disponible en todo el sistema de archivos. Antes de ver cómo poner los detalles de la clase, veamos cómo crear objetos usando estas clases. El proceso de crear un objeto también se llama instanciación. Cada clase introduce un nuevo nombre de tipo. Así estudiante, instructor, miembro del personal y Curso son los tipos que hemos introducido. El siguiente código crea una instancia de un nuevo objeto de tipo Estudiante.

Listing 2: Instancia de un nuevo objeto de tipo Estudiante.

```
1  new Student();
```

El nuevo operador hace que el sistema asigne un objeto de tipo Estudiante con suficiente almacenamiento para guardar información sobre un alumno. El operador devuelve la dirección de la ubicación que contiene este objeto. Esta dirección se denomina referencia. La declaración anterior se puede ejecutar cuando tenemos un nuevo estudiante admitido en la Universidad. Una vez que instanciamos un nuevo objeto, debemos almacenar su referencia en algún lugar, por lo que que podemos usar más tarde de alguna manera apropiada. Para esto, creamos una variable de tipo Estudiante.

Listing 3: Crear la variable estudiante.

```
1  Student harry;
```

Tenga en cuenta que la definición anterior simplemente dice que Harry es una variable que puede almacenar referencias a objetos de tipo Estudiante. Por lo tanto, podemos escribir

Listing 4: La variable harry.

```
1  harry = new Student();
```

No podemos escribir

Listing 5: No se use.

```
1 harry = new Instructor();
```

porque Harry es del tipo Estudiante, que no tiene ninguna relación (en la medida en que la clase decida a la de Instructor, que es del tipo de objeto creado del lado derecho de la tarea. Cada vez que instanciamos un nuevo objeto, debemos recordar la referencia a ese objeto en algún lado. Sin embargo, no es necesario que por cada objeto que instanciamos, declaremos una variable diferente para almacenar su referencia. Si ese fuera el caso, la programación sería tediosa. Vamos a ilustrar dando una analogía. Cuando un estudiante conduce a la escuela para tomar una clase, ella se ocupa solo de un número relativamente pequeño de objetos: los controles del automóvil, la carretera, los vehículos cercanos (y a veces sus ocupantes, aunque no siempre educadamente) y el tránsito, señales y signos. (Algunos también pueden tratar con un teléfono celular, ¡lo cual no es una buena idea!) Hay muchos otros objetos que el conductor (estudiante) conoce, pero que no está tratando con ellos. Del mismo modo, mantenemos las referencias a un número relativamente pequeño de objetos en nuestros programas. Cuando surge la necesidad de acceder a otros objetos, utilizamos las referencias que ya tenemos para cubrirlos. Por ejemplo, supongamos que tenemos una referencia a un objeto Student. Ese objeto puede tener un atributo que recuerda al asesor del alumno, un objeto Instructor. Si es necesario averiguar el asesor de un alumno determinado, podemos consultar el correspondiente objeto del alumno para obtener el objeto Instructor. Un solo objeto Instructor puede tener atributos que recuerdan todos los consejos del instructor correspondiente.

Implementando clases

En esta sección damos algunos de los conceptos básicos para crear clases. Centrémonos en la clase estudiante que inicialmente codificamos como

Listing 6: Clase estudiante.

```
1 public class Student {
2 // codigo pera implementar un solo student
3 }
```

Ciertamente nos gustaría la capacidad de dar un nombre a un estudiante: dado un objeto de estudiante, nosotros debemos poder especificar que el nombre del alumno es "Tom.º "Jane.º, en general, alguna cuerda, esto a veces se conoce como un comportamiento del objeto. Podemos pensar en los objetos de los estudiantes que tienen el comportamiento que responden al asignar un nombre. Para este propósito, modificamos el código de la siguiente manera.

Listing 7: Dar nombre a un objeto.

```
1 public class Student {
2 // codigo para hacer otras cosas
3 public void setName(String studentName) {
4 // codigo para recordar el nombre
5 }
```

```
6 }
```

El código que agregamos se llama método. El nombre del método es setName. Un método es como un procedimiento o función en programación imperativa en que es una unidad de código que no se activa hasta que se invoca. Nuevamente, como en el caso de procedimientos y funciones, los métodos aceptan parámetros (separados por comas en Java). Cada parámetro establece el tipo de parámetro esperado. Un método puede no devolver nada (como es el caso aquí) o devolver un objeto o un valor de un tipo primitivo. Aquí hemos puesto vacío delante del significado del nombre del método que el método no devuelve nada. Los corchetes izquierdo y derecho comienzan y finalizan el código eso define el método. A diferencia de las funciones y procedimientos, los métodos generalmente se invocan a través de objetos. El método setName se define dentro de la clase Student e invoca un objetos de tipo Estudiante.

Listing 8: Crear otra clase.

```
1 Student aStudent = new Student();
2 aStudent.setName("Ron");
```

El método setName () se invoca en ese objeto al que hace referencia un estudiante. Intuitivamente El código dentro de ese método debe almacenar el nombre en alguna parte. Recuerda que cada objeto Se le asigna su propio almacenamiento. Esta pieza de almacenamiento debe incluir espacio para recordar el nombre del alumno Embellecemos el código de la siguiente manera.

Listing 9: Crear otra clase.

```
1 public class Student {
2 private String name;
3 public void setName(String studentName) {
4 name = studentName;
5 }
6 public String getName() {
7 return name;
8 }
9 }
```

Dentro de la clase hemos definido el nombre de la variable de tipo String. Se llama un campo.

DEFINICIÓN 2 (Campo.) *Un campo es una variable definida directamente dentro de una clase y corresponde a un atributo Cada instancia del objeto tendrá almacenamiento para el campo.*

Examinemos el código dentro del método setName. Toma en un parámetro, studentName, y asigna el valor en ese objeto String al nombre del campo.

Es importante comprender cómo Java usa el campo de nombre. Cada objeto de tipo El estudiante tiene un campo llamado nombre. Invocamos el método setName () en el objeto re- fervido por un estudiante. Como aStudent tiene el nombre del campo e invocamos el método en un estudiante, la referencia al

nombre dentro del método actuará en el campo de nombre de un estudiante. El método getName () recupera el contenido del campo de nombre y lo devuelve. Para ilustrar esto más a fondo, considere dos objetos de tipo Estudiante.

Listing 10: Crear otra clase.

```
Student student1 = new Student();
Student student2 = new Student();
student1.setName("John");
student2.setName("Mary");
System.out.println(student1.getName());
System.out.println(student2.getName());
```

Se puede acceder a los miembros (campos y métodos por ahora) de una clase escribiendo

Listing 11: Crear otra clase.

```
<object-reference>.<member-name>
```

El objeto al que se refiere student1 tiene su campo de nombre establecido en "John", mientras que el objeto referido por student2 tiene su campo de nombre establecido en "Mary". El nombre del campo en el código

Listing 12: Crear otra clase.

```
name = studentName;
```

se refiere a diferentes objetos en diferentes instancias y, por lo tanto, diferentes instancias de campos. Escribamos un programa completo usando el código anterior.

Listing 13: Crear otra clase.

```
public class Student {
// code
private String name;
public void setName(String studentName) {
name = studentName;
}
public String getName() {
return name;
}
public static void main(String[] s) {
Student student1 = new Student();
Student student2 = new Student();
student1.setName("John");
student2.setName("Mary");
System.out.println(student1.getName());
System.out.println(student2.getName());
}
}
```

La palabra clave public delante del método setName () hace que el método esté disponible donde esté disponible el objeto. Pero, ¿qué pasa con la palabra

clave privada frente a ¿nombre del campo? Significa que solo se puede acceder a esta variable desde el código dentro de la clase Estudiante. Desde la linea

Listing 14: Crear otra clase.

```
1 name = studentName;
```

está dentro de la clase, el compilador lo permite. Sin embargo, si escribimos

Listing 15: Crear otra clase.

```
1 Student someStudent = new Student();
2 someStudent.name = "Mary";
```

fuera de la clase, el compilador generará un error de sintaxis. Como regla general, los campos a menudo se definen con el especificador de acceso privado y el método Los anuncios suelen hacerse públicos. La idea general es que los campos denotan el estado del objeto. y que el estado solo se puede cambiar interactuando a través de métodos predefinidos que denotar el comportamiento del objeto. Por lo general, esto ayuda a preservar la integridad de los datos. Sin embargo, en el ejemplo actual, es difícil argumentar que la consideración de integridad de datos juega un rol en hacer que el nombre sea privado porque todo lo que hace el método setName es cambiar el campo de nombre Sin embargo, si quisiéramos hacer algunas comprobaciones antes de cambiar el estudiante nombre (que no debería suceder tan a menudo), esto nos da una forma de hacerlo. Si hubiéramos guardado nombre público y otros codificados para acceder directamente al campo, haciendo que el campo sea privado más tarde rompería su código. Para un uso más justificado del privado, considere el promedio de calificaciones (GPA) de un estudiante. Claramente, necesitamos hacer un seguimiento del GPA y necesitamos un campo para ello. GPA no es algo eso se cambia arbitrariamente: cambia cuando un estudiante obtiene una calificación para un curso. Entonces haciendo su público podría conducir a problemas de integridad porque el campo puede ser cambiado inadvertidamente por código incorrecto escrito afuera. Por lo tanto, codificamos de la siguiente manera.

Listing 16: Crear otra clase.

```
1  public class Student {
2  // fields to store the classes the student has registered for.
3  private String name;
4  private double gpa;
5  public void setName(String studentName) {
6  name = studentName;
7  }
8  public void addCourse(Course newCourse) {
9  // code to store a ref to newCourse in the Student object.
10 }
11 private void computeGPA() {
12 // code to access the stored courses, compute and set the gpa
13 }
14 public double getGPA() {
15 return gpa;
16 }
```

```
17 public void assignGrade(Course aCourse, char newGrade) {
18 // code to assign newGrade to aCourse
19 computeGPA();
20 }
21 }
```

Ahora escribimos código para utilizar la idea anterior.

Listing 17: Crear otra clase.

```
1 Student aStudent = new Student();
2 Course aCourse = new Course();
3 aStudent.addCourse(aCourse);
4 aStudent.assignGrade(aCourse, 'B');
5 System.out.println(aStudent.getGPA());
```

El código anterior crea un objeto Estudiante y un objeto Curso. Llama al addCourse método en el alumno, para agregar el curso a la colección de cursos tomados por el alumno, y luego llama a asignarGrade. Tenga en cuenta los dos parámetros: aCourse y 'B'. Lo implícito lo que significa es que el estudiante ha completado el curso (aCourse) con una calificación de "B". los El código en el método debe calcular el nuevo GPA para el estudiante usando la información presumiblemente en el curso (como el número de créditos) y el número de puntos para una calificación de B'.

Constructores

La clase Estudiante tiene un método para establecer el nombre de un estudiante. Aquí ponemos el nombre del alumno después de crear el objeto. Esto es algo antinatural. Ya que cada estudiante tiene un nombre, cuando creamos un objeto de estudiante, probablemente también sepamos el nombre del estudiante. Sería conveniente almacenar el nombre del alumno en el objeto a medida que lo creamos. objeto. Para ver hacia dónde nos dirigimos, considere las siguientes declaraciones de variables de primitivas tipos de datos.

Listing 18: Crear otra clase.

```
1 int counter = 0;
2 double final PI = 3.14;
```

Ambas declaraciones almacenan valores en las variables a medida que se crean las variables. En el otro mano, el objeto Estudiante, cuando se crea, tiene un cero en cada bit de cada campo. Java y otros lenguajes orientados a objetos permiten la inicialización de campos utilizando qué se llaman constructores.

DEFINICIÓN 3 (Constructor.) *Un constructor es como un método en el que puede tener un especificador de acceso (como público o privado), un nombre, parámetros y código ejecutable. Sin embargo, los constructores. tienen las siguientes diferencias o características especiales.*

figura **??** [5] .

Algorithm 0.1: Reconocedor de palabras

```
1  c = SigCar();
2  if c == 'n' then
3  |  c = SigCar();
4  |  if c == 'e' then
5  |  |  c = SigCar();
6  |  |  if c == 'w' then
7  |  |  |  EsEnPan(Palabra Aceptada);
8  |  |  else
9  |  |  |  EsEnPan(Palabra No Aceptada);
10 |  else
11 |  |  EsEnPan(Falla);
12 else
13 |  EsEnPan(Falla);
```

Bibliografía

[1] Thomas W. Parsons, Introduction to compiler construction, Computer Science Press, 1992.

[2] Ralph Grishman, Computational Linguistics An Introduction, Cambridge University Press, 1986.

[3] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, Compiladores Principios, técnicas y herramientas, Addison Wesley, 1990.

[4] Torben Aegidius Mogensen, Basics of compiler design, DEPARTMENT OF COMPUTER SCIENCE UNIVERSITY OF COPENHAGEN, 2009.

[5] Linda Torczon and Keith D. Cooper, Engineering a Compiler, Second Edition, Elsevier 2012.

[6] Bernard Teufel, Stephanie Schmidt, Thomas Teufel, Compiladores, conceptos fundamentales, Addison-Wesley Iberoamericana, 1995.

[7] Jean Paul Tremblay, Paul G. Sorenson, The theory and practice of compiler writing, McGraw Hill Book Company, 1985.

[8] Mark P. Jones, jacc: just another compiler compiler for Java A Reference Manual and User Guide, Department of Computer Science Engineering OGI School of Science Engineering at OHSU 20000 NW Walker Road, Beaverton, OR 97006, USA February 16, 2004.

[9] www.lcc.uma.es/ galvez/ftp/tci/TutorialYacc.pdf