Liceo Compu-Market

Nombre:

Dilan David Castro Díaz

Grado:

5to Bachillerato en Computación

Sección:

“A”

Catedra:

Programación

Titulo:

Programación

Fecha de Entrega:

17/04/2017

Índice

Número Tema No. Página

1. [Introducción](#Introducción)……………………………………………………. 4
2. [Programación](#Programación)…………………………………………………… 5
3. [Historia](#Historia)………………………………………………………… 6
4. [Léxico y Programación](#Léxico_y_Programación) …………………………………………. 7
5. [Programación y Algoritmos](#Programas_y_algoritmos)……………………………………… 7
6. [Programación Declarativa](#Programación_declarativa)………………………………………... 8
7. [Programación Imperativa](#Programación_imperativa)………………………………………. 9-11
8. [Programación Estructurada](#Programación_estructurada)……………………………………... 11
9. [Paradigma de Programación](#Paradigma_de_Programación)…………………………………… 12-14
10. [Programación Modular](#Programación_modular)………………………………………… 14
11. [Programación Orientada a Objetos](#Programación_Orientada_a_Objetos)……………………………...15-23
12. [Compilación de la Programación](#Compilación_de_la_Programación)………………………………. 23
13. [Programación e Ingeniería del Software](#Prgramación_e_ingeniería_del_software)………………………… 24
14. [Objetivos de la Programación](#Objetivos_de_la_programación)…………………………………. 25
15. [Ciclo de Vida del Software](#Ciclo_de_vida_del_software)……………………………………. 26
16. [Código Espagueti](#Código_espagueti)……………………………………………... 27-28
17. [Error de Software](#Error_de_Software)…………………………………………….. 29-31
18. [Linea de Código Fuente](#Línea_de_código_fuente)………………………………………. 31
19. [Lenguaje de Programación](#Lenguaje_de_programación)……………………………………. 32-42
20. [Programación Automática](#Programación_automática) …………………………………… 42-44
21. [Programación Dirigida por Eventos](#Programación_dirigida_por_eventos)…………………………… 44-46
22. [Programación Extrema](#Programación_extrema)………………………………………... 46-51
23. [Programación en Pareja](#Programación_en_pareja)……………………………………….. 51-53
24. [Programación Dinámica](#Programación_dinámica)……………………………………… 54-56
25. [Filosofías del Desarrollo de Software](#Filosofía_del_desarrollo_de_software)………………………….. 56-57
26. [Desarrollo guiado por el diseño](#Desarrollo_guiado_por_el_diseño)……………………………….. 57
27. [Desarrollo guiado por pruebas](#Desarrollo_guiado_por_pruebas)………………………………… 58
28. [Ciclo de Desarrollo conducido por pruebas](#Ciclo_de_desarrollo_conducido_por_prueba)…………………….. 58-61
29. [Desarrollo iterativo y Creciente](#Desarrollo_iterativo_y_creciente)……………………………….... 61-66
30. [Ley de Brooks](#Ley_de_Brooks)………………………………………………… 66-69
31. [Scrum](#Scrum)……………………………………………………….... 69-75
32. [Conclusión](#Conclusión)……………………………………………………. 76

Introducción

En el presente trabajo podremos encontrar todo lo relacionado sobre Programación y algunos métodos de ayuda para errores de software y un concepto amplio sobre cada tema. También acerca de los métodos para poder trabajar mejor con un equipo de programadores y concejos para poder mejorar en estructuración de código que nos puede servir para problemas informáticos que nos puedan surgir.

Programación

La programación informática o programación algorítmica, acortada como programación, es el proceso de diseñar, codificar, depurar y mantener el código fuente de programas de computadora. El código fuente es escrito en un lenguaje de programación. El propósito de la programación es crear programas que exhiban un comportamiento deseado. El proceso de escribir código requiere frecuentemente conocimientos en varias áreas distintas, además del dominio del lenguaje a utilizar, algoritmos especializados y lógica formal. Programar no involucra necesariamente otras tareas tales como el análisis y diseño de la aplicación (pero sí el diseño del código), aunque sí suelen estar fusionadas en el desarrollo de pequeñas aplicaciones.

Programación es el proceso a través del cual un programa o aplicación informática es desarrollado.

Se conoce como programación en ciencias de la computación a los pasos que se abordan para crear el código fuente de un programa informático. De acuerdo con estos pasos, el código se escribe, se prueba y se perfecciona.

El software informático es aquel ejecutado por el hardware o dispositivos de una computadora, permitiendo que el usuario acceda a funciones y operaciones de todo tipo. Así, la programación es una de las actividades más determinantes en el desarrollo de sistemas eficientes, rápidos y amigables para todo tipo de usuarios.

Se conoce como programadores a aquello encargados de desarrollar este código con instrucciones para que el software se comporte de una u otra manera de acuerdo con las órdenes que reciba. El lenguaje de programación es, por otro lado, la serie de parámetros y códigos de los que el programador se vale para desarrollar software. Existen distintos tipos de lenguajes, como el C, BASIC o Ruby.

Además, existe la ingeniería el software, que se dedica a desarrollar modelos de software para programas de gran envergadura.

Típicamente, para programar un software o aplicación, el programador debe en principio reconocer el principal problema o tarea a la que se destinará el programa, definir los requisitos y tipo de funcionamiento, diseñar la arquitectura, implementar el programa, implantarlo o instalarlo y, luego, perfeccionarlo sobre la base de pruebas y errores.

Hoy en día existen todo tipo de lenguajes de programación, algunos más sencillos o que tienen el propósito de facilitar la tarea de desarrollar pequeñas aplicaciones. Entre ellos, Ruby es uno de los más popularizados en los últimos años, desarrollado por un programador japonés y que combina sintaxis de distintos lenguajes como Python o Perl.

Así, prácticamente cualquier usuario informático puede adquirir algunas nociones de programación y desarrollar aplicaciones a su medida.

Historia

Para crear un programa, y que la computadora lo interprete y ejecute las instrucciones escritas en él, debe escribirse en un lenguaje de programación. En sus inicios las computadoras interpretaban solo instrucciones en un lenguaje específico, del más bajo nivel, conocido como código máquina, siendo éste excesivamente complicado para programar. De hecho solo consiste en cadenas de números 1 y 0 (sistema binario). Para facilitar el trabajo de programación, los primeros científicos, que trabajaban en el área, decidieron reemplazar las instrucciones, secuencias de unos y ceros, por palabras o abreviaturas provenientes del inglés; las codificaron y crearon así un lenguaje de mayor nivel, que se conoce como Assembly o lenguaje ensamblador. Por ejemplo, para sumar se podría usar la letra A de la palabra inglesa add (sumar). En realidad escribir en lenguaje ensamblador es básicamente lo mismo que hacerlo en lenguaje máquina, pero las letras y palabras son bastante más fáciles de recordar y entender que secuencias de números binarios. A medida que la complejidad de las tareas que realizaban las computadoras aumentaba, se hizo necesario disponer de un método sencillo para programar. Entonces, se crearon los lenguajes de alto nivel. Mientras que una tarea tan trivial como multiplicar dos números puede necesitar un conjunto de instrucciones en lenguaje ensamblador, en un lenguaje de alto nivel bastará con solo una. Una vez que se termina de escribir un programa, sea en ensamblador o en algunos lenguajes de alto nivel, es necesario compilarlo, es decir, traducirlo completo a lenguaje máquina.1 Eventualmente será necesaria otra fase denominada comúnmente link o enlace, durante la cual se anexan al código, generado durante la compilación, los recursos necesarios de alguna biblioteca. En algunos lenguajes de programación, puede no ser requerido el proceso de compilación y enlace, ya que pueden trabajar en modo intérprete. Esta modalidad de trabajo es equivalente pero se realiza instrucción por instrucción, a medida que es ejecutado el programa.

Léxico y Programación

La programación se rige por reglas y un conjunto más o menos reducido de órdenes, expresiones, instrucciones y comandos que tienden a asemejarse a una lengua natural acotada (en inglés); y que además tienen la particularidad de una reducida ambigüedad. Cuanto menos ambiguo es un lenguaje de programación, se dice, es más potente. Bajo esta premisa, y en el extremo, el lenguaje más potente existente es el binario, con ambigüedad nula (lo cual lleva a pensar así del lenguaje ensamblador).

En los lenguajes de programación de alto nivel se distinguen diversos elementos entre los que se incluyen el léxico propio del lenguaje y las reglas semánticas y sintácticas.

Programas y algoritmos

Un algoritmo es una secuencia no ambigua, finita y ordenada de instrucciones que han de seguirse para resolver un problema. Un programa normalmente implementa (traduce a un lenguaje de programación concreto) uno o más algoritmos. Un algoritmo puede expresarse de distintas maneras: en forma gráfica, como un diagrama de flujo, en forma de código como en pseudocódigo o un lenguaje de programación, en forma explicativa.

Los programas suelen subdividirse en partes menores, llamadas módulos, de modo que la complejidad algorítmica de cada una de las partes sea menor que la del programa completo, lo cual ayuda al desarrollo del programa. Esta es una práctica muy utilizada y se conoce como "refino progresivo".

Según Niklaus Wirth, un programa está formado por los algoritmos y la estructura de datos.

La programación puede seguir muchos enfoques, o paradigmas, es decir, diversas maneras de formular la resolución de un problema dado. Algunos de los principales paradigmas de la programación son:

* [Programación declarativa](#Programación_declarativa)
* [Programación estructurada](#Programación_estructurada)
* [Programación modular](#Programación_modular)
* [Programación orientada a objetos](#Programación_Orientada_a_Objetos)

Programación declarativa

La programación declarativa, en contraposición a la programación imperativa, es un paradigma de programación que está basado en el desarrollo de programas especificando o "declarando" un conjunto de condiciones, proposiciones, afirmaciones, restricciones, ecuaciones o transformaciones que describen el problema y detallan su solución. La solución es obtenida mediante mecanismos internos de control, sin especificar exactamente cómo encontrarla (tan sólo se le indica a la computadora qué es lo que se desea obtener o qué es lo que se está buscando). No existen asignaciones destructivas, y las variables son utilizadas con transparencia referencial.

Diferencia entre imperativo y declarativo

En la [programación imperativa](#Programación_imperativa) se describe paso a paso un conjunto de instrucciones que deben ejecutarse para variar el estado del programa y hallar la solución, es decir, un algoritmo en el que se describen los pasos necesarios para solucionar el problema.

En la programación declarativa las sentencias que se utilizan lo que hacen es describir el problema que se quiere solucionar; se programa diciendo lo que se quiere resolver a nivel de usuario, pero no las instrucciones necesarias para solucionarlo. Esto último se realizará mediante mecanismos internos de inferencia de información a partir de la descripción realizada.

Tipos

Existen varios tipos de lenguajes declarativos:

* Los lenguajes lógicos, como Prolog.
* Los lenguajes algebraicos, como Maude y SQL.
* Los lenguajes funcionales, como Haskell y Erlang.

Ventajas

Se ha dicho que los lenguajes declarativos tienen la ventaja de ser razonados matemáticamente, lo que permite el uso de mecanismos matemáticos para optimizar el rendimiento de los programas.1

Son fiables, elegantes y expresivos.

Programación imperativa

La programación imperativa es uno de los paradigmas de programación de computadoras más generales, que describe la programación en términos del estado del programa y sentencias que cambian dicho estado. Los programas imperativos son un conjunto de instrucciones que le indican al computador cómo realizar una tarea, de la misma manera que el modo imperativo en los lenguajes naturales humanos le dice qué hacer al interlocutor. Esto existe en contraposición al paradigma de programación declarativa, que se basa en describir el problema o solución sin hacer referencia a un algoritmo. Las recetas de cocina y las listas de revisión de procesos, a pesar de no ser programas de computadora, son también conceptos familiares similares en estilo a la programación imperativa; donde cada paso es una instrucción.

Programación imperativa y hardware

La implementación de hardware de la mayoría de los computadores es imperativa; prácticamente todo el hardware de los computadores está diseñado para ejecutar código de máquina, que es nativo al computador, escrito en una forma imperativa. Esto se debe a que el hardware de los computadores puede modelarse como una máquina de Turing, o alguna otra forma de máquina abstracta que modifica su memoria o estado interno conforme sigue los pasos de un programa. Desde esta perspectiva de bajo nivel, la semántica del programa en cualquier momento está definida por los contenidos previos de la memoria, los datos de entrada y las instrucciones en el lenguaje de máquina nativo del computador.

Este paralelismo entre el paradigma imperativo y los objetos físicos que terminan realizando los cálculos significa que todo programa, así esté escrito en un lenguaje de alto nivel o incluso en uno completamente declarativo, eventualmente debe ser transformado a código máquina imperativo para tener efecto.

Lenguajes de programación

Los primeros lenguajes imperativos fueron los lenguajes de máquina de los computadores originales. En estos lenguajes, las instrucciones fueron muy simples, lo cual hizo la implementación de hardware fácil, pero obstruyendo la creación de programas complejos. Los lenguajes imperativos de alto nivel usan variables y sentencias más complejas, pero aún siguen el mismo paradigma cuando modifican los contenidos de las variables. Fortran, cuyo desarrollo fue iniciado en 1954 por John Backus en IBM, fue el primer gran lenguaje de programación en superar los obstáculos presentados por el código de máquina en la creación de programas complejos.

Algunos ejemplos de lenguajes de programación imperativos (o que son usados idiomáticamente en estilo imperativo a pesar de ofrecer otros estilos) son:

* BASIC
* C
* D1
* Fortran
* Pascal
* Pauscal en español
* Perl
* PHP
* Lua
* Java
* Python
* Go
* simple basic
* Javascript

Críticas

La programación imperativa goza de gran popularidad, pero se ha dicho que al permitir y depender de efectos colaterales como el uso de variables globales, dificulta notablemente el razonamiento formal, la depuración y la paralelización (la habilidad para dividir el trabajo en muchos hilos de ejecución simultánea).

Programación estructurada

La programación estructurada es un [paradigma de programación](#Paradigma_de_Programación) orientado a mejorar la claridad, calidad y tiempo de desarrollo de un programa de computadora, utilizando únicamente subrutinas y tres estructuras: secuencia, selección (if y switch) e iteración (bucles for y while), considerando innecesario y contraproducente el uso de la instrucción de transferencia incondicional (GOTO), que podría conducir a "código espagueti", que es mucho más difícil de seguir y de mantener, y era la causa de muchos errores de programación.

Surgió en la década de 1960, particularmente del trabajo de Böhm y Jacopini,1 y una famosa carta, «La sentencia goto, considerada perjudicial», de Edsger Dijkstra en 19682 — y fue reforzado teóricamente por el teorema del programa estructurado, y prácticamente por la aparición de lenguajes como ALGOL con adecuadas y ricas estructuras de control.

Orígenes de la programación estructurada

A finales de los años 1970 surgió una nueva forma de programar que no solamente daba lugar a programas fiables y eficientes, sino que además estaban escritos de manera que facilitaba su mejor comprensión, no sólo proveyendo ventajas durante la fase de desarrollo, sino también posibilitando una más sencilla modificación posterior.

El teorema del programa estructurado, propuesto por Böhm-Jacopini, demuestra que todo programa puede escribirse utilizando únicamente las tres instrucciones de control siguientes:

Secuencia.

Instrucción condicional.

Iteración (bucle de instrucciones) con condición al principio.

Solamente con estas tres estructuras se pueden escribir todos los programas y aplicaciones posibles. Si bien los lenguajes de programación tienen un mayor repertorio de estructuras de control, estas pueden ser construidas mediante las tres básicas citadas.

Paradigma de programación

Un paradigma de programación es una propuesta tecnológica adoptada por una comunidad de programadores y desarrolladores cuyo núcleo central es incuestionable en cuanto que únicamente trata de resolver uno o varios problemas claramente delimitados; la resolución de estos problemas debe suponer consecuentemente un avance significativo en al menos un parámetro que afecte a la ingeniería de software.

Un paradigma de programación representa un enfoque particular o filosofía para diseñar soluciones. Los paradigmas difieren unos de otros, en los conceptos y la forma de abstraer los elementos involucrados en un problema, así como en los pasos que integran su solución del problema, en otras palabras, el cómputo.

Tiene una estrecha relación con la formalización de determinados lenguajes en su momento de definición. Es un estilo de programación empleado.

Un paradigma de programación está delimitado en el tiempo en cuanto a aceptación y uso, porque nuevos paradigmas aportan nuevas o mejores soluciones que la sustituyen parcial o totalmente.

El paradigma de programación que actualmente es el más utilizado es la "orientación a objetos" (OO). El núcleo central de este paradigma es la unión de datos y procesamiento en una entidad llamada "objeto", relacionable a su vez con otras entidades "objeto".

Tradicionalmente, datos y procesamiento se han separado en áreas diferente del diseño y la implementación de software. Esto provocó que grandes desarrollos tuvieran problemas de fiabilidad, mantenimiento, adaptación a los cambios y escalabilidad. Con la OO y características como el encapsulado, polimorfismo o la herencia, se permitió un avance significativo en el desarrollo de software a cualquier escala de producción. La OO parece estar ligada en sus orígenes con lenguajes como Lisp y Simula, aunque el primero que acuñó el título de "programación orientada a objetos" fue Smalltalk.

Tipos más comunes de paradigmas de programación

En general la mayoría son variantes de los dos tipos principales, imperativa y declarativa:

* Programación imperativa o por procedimientos: es el más usado en general, se basa en dar instrucciones al ordenador de como hacer las cosas en forma de algoritmos. La programación imperativa es la más usada y la más antigua, el ejemplo principal es el lenguaje de máquina. Ejemplos de lenguajes puros de este paradigma serían el C, BASIC o Pascal.
* Programación orientada a objetos: está basada en el imperativo, pero encapsula elementos denominados objetos que incluyen tanto variables como funciones. Está representado por C++, C#, Java o Python entre otros, pero el más representativo sería el Smalltalk que está completamente orientado a objetos.
* Programación dinámica: está definida como el proceso de romper problemas en partes pequeñas para analizarlos y resolverlos de forma lo más cercana al óptimo, busca resolver problemas en O(n) sin usar por tanto métodos recursivos. Este paradigma está más basado en el modo de realizar los algoritmos, por lo que se puede usar con cualquier lenguaje imperativo.
* Programación dirigida por eventos: la programación dirigida por eventos es un paradigma de programación en el que tanto la estructura como la ejecución de los programas van determinados por los sucesos que ocurran en el sistema, definidos por el usuario o que ellos mismos provoquen.
* Programación declarativa: está basado en describir el problema declarando propiedades y reglas que deben cumplirse, en lugar de instrucciones. Hay lenguajes para la programación funcional, la programación lógica, o la combinación lógico-funcional. Unos de los primeros lenguajes funcionales fueron Lisp y Prolog.
* Programación funcional: basada en la definición los predicados y es de corte más matemático, está representado por Scheme (una variante de Lisp) o Haskell. Python también representa este paradigma. 1
* Programación lógica: basado en la definición de relaciones lógicas, está representado por Prolog.
* Programación con restricciones: similar a la lógica usando ecuaciones. Casi todos los lenguajes son variantes del Prolog.
* Programación multiparadigma: es el uso de dos o más paradigmas dentro de un programa. El lenguaje Lisp se considera multiparadigma. Al igual que Python, que es orientado a objetos, reflexivo, imperativo y funcional.1
* Lenguaje específico del dominio o DSL: se denomina así a los lenguajes desarrollados para resolver un problema específico, pudiendo entrar dentro de cualquier grupo anterior. El más representativo sería SQL para el manejo de las bases de datos, de tipo declarativo, pero los hay imperativos, como el Logo.
* Si bien puede seleccionarse la forma pura de estos paradigmas al momento de programar, en la práctica es habitual que se mezclen, dando lugar a la programación multiparadigma o lenguajes de programación multiparadigma.

Programación modular

La programación modular es un paradigma de programación que consiste en dividir un programa en módulos o subprogramas con el fin de hacerlo más legible y manejable.

Se presenta históricamente como una evolución de la programación estructurada para solucionar problemas de programación más grandes y complejos de lo que esta puede resolver.

Al aplicar la programación modular, un problema complejo debe ser dividido en varios subproblemas más simples, y estos a su vez en otros subproblemas más simples. Esto debe hacerse hasta obtener subproblemas lo suficientemente simples como para poder ser resueltos fácilmente con algún lenguaje de programación. Esta técnica se llama refinamiento sucesivo, divide y vencerás ó análisis descendente (Top-Down).

Un 'módulo' es cada una de las partes de un programa que resuelve uno de los subproblemas en que se divide el problema complejo original. Cada uno de estos módulos tiene una tarea bien definida y algunos necesitan de otros para poder operar. En caso de que un módulo necesite de otro, puede comunicarse con éste mediante una interfaz de comunicación que también debe estar bien definida.

Si bien un módulo puede entenderse como una parte de un programa en cualquiera de sus formas y variados contextos, en la práctica se los suele tomar como sinónimos de procedimientos y funciones. Pero no necesaria ni estrictamente un módulo es una función o un procedimiento, ya que el mismo puede contener muchos de ellos. No debe confundirse el término "módulo" (en el sentido de programación modular) con términos como "función" o "procedimiento", propios del lenguaje que lo soporte.

Programación orientada a objetos

La programación orientada a objetos (POO, u OOP según sus siglas en inglés) es un paradigma de programación que viene a innovar la forma de obtener resultados. Los objetos manipulan los datos de entrada para la obtención de datos de salida específicos, donde cada objeto ofrece una funcionalidad especial.

Muchos de los objetos pre-diseñados de los lenguajes de programación actuales permiten la agrupación en bibliotecas o librerías, sin embargo, muchos de estos lenguajes permiten al usuario la creación de sus propias bibliotecas.

Está basada en varias técnicas, incluyendo herencia, cohesión, abstracción, polimorfismo, acoplamiento y encapsulamiento.

Su uso se popularizó a principios de la década de 1990. En la actualidad, existe una gran variedad de lenguajes de programación que soportan la orientación a objetos.

Introducción

Los objetos son entidades que tienen un determinado "estado", "comportamiento (método)" e "identidad":

La identidad es una propiedad de un objeto que lo diferencia del resto; dicho con otras palabras, es su identificador (concepto análogo al de identificador de una variable o una constante).

Un objeto contiene toda la información que permite definirlo e identificarlo frente a otros objetos pertenecientes a otras clases e incluso frente a objetos de una misma clase, al poder tener valores bien diferenciados en sus atributos. A su vez, los objetos disponen de mecanismos de interacción llamados métodos, que favorecen la comunicación entre ellos. Esta comunicación favorece a su vez el cambio de estado en los propios objetos. Esta característica lleva a tratarlos como unidades indivisibles, en las que no se separa el estado y el comportamiento.

Los métodos (comportamiento) y atributos (estado) están estrechamente relacionados por la propiedad de conjunto. Esta propiedad destaca que una clase requiere de métodos para poder tratar los atributos con los que cuenta. El programador debe pensar indistintamente en ambos conceptos, sin separar ni darle mayor importancia a alguno de ellos. Hacerlo podría producir el hábito erróneo de crear clases contenedoras de información por un lado y clases con métodos que manejen a las primeras por el otro. De esta manera se estaría realizando una "programación estructurada camuflada" en un lenguaje de POO.

La programación orientada a objetos difiere de la programación estructurada tradicional, en la que los datos y los procedimientos están separados y sin relación, ya que lo único que se busca es el procesamiento de unos datos de entrada para obtener otros de salida. La programación estructurada anima al programador a pensar sobre todo en términos de procedimientos o funciones, y en segundo lugar en las estructuras de datos que esos procedimientos manejan. En la programación estructurada solo se escriben funciones que procesan datos. Los programadores que emplean POO, en cambio, primero definen objetos para luego enviarles mensajes solicitándoles que realicen sus métodos por sí mismos.

Origen

Los conceptos de la POO tienen origen en Simula 67, un lenguaje diseñado para hacer simulaciones, creado por Ole-Johan Dahl y Kristen Nygaard, del Centro de Cómputo Noruego en Oslo. En este centro se trabajaba en simulaciones de naves, que fueron confundidas por la explosión combinatoria de cómo las diversas cualidades de diferentes naves podían afectar unas a las otras. La idea surgió al agrupar los diversos tipos de naves en diversas clases de objetos, siendo responsable cada clase de objetos de definir sus "propios" datos y comportamientos. Fueron refinados más tarde en Smalltalk, desarrollado en Simula en Xerox PARC (cuya primera versión fue escrita sobre Basic) pero diseñado para ser un sistema completamente dinámico en el cual los objetos se podrían crear y modificar "sobre la marcha" (en tiempo de ejecución) en lugar de tener un sistema basado en programas estáticos.

La POO se fue convirtiendo en el estilo de programación dominante a mediados de los años 1980, en gran parte debido a la influencia de C++, una extensión del lenguaje de programación C. Su dominación fue consolidada gracias al auge de las interfaces gráficas de usuario, para las cuales la POO está particularmente bien adaptada. En este caso, se habla también de programación dirigida por eventos.

Las características de orientación a objetos fueron agregadas a muchos lenguajes existentes durante ese tiempo, incluyendo Ada, BASIC, Lisp más Pascal, entre otros. La adición de estas características a los lenguajes que no fueron diseñados inicialmente para ellas condujo a menudo a problemas de compatibilidad y en la capacidad de mantenimiento del código. Los lenguajes orientados a objetos "puros", por su parte, carecían de las características de las cuales muchos programadores habían venido a depender. Para saltar este obstáculo, se hicieron muchas tentativas para crear nuevos lenguajes basados en métodos orientados a objetos, pero permitiendo algunas características imperativas de maneras "seguras". El lenguaje de programación Eiffel de Bertrand Meyer fue un temprano y moderadamente acertado lenguaje con esos objetivos, pero ahora ha sido esencialmente reemplazado por Java, en gran parte debido a la aparición de Internet y a la implementación de la máquina virtual Java en la mayoría de navegadores web. PHP en su versión 5 se ha modificado; soporta una orientación completa a objetos, cumpliendo todas las características propias de la orientación a objetos.

Conceptos fundamentales

La POO es una forma de programar que trata de encontrar una solución a estos problemas. Introduce nuevos conceptos, que superan y amplían conceptos antiguos ya conocidos. Entre ellos destacan los siguientes:

**Clase**

Definiciones de las propiedades y comportamiento de un tipo de objeto concreto. La instanciación es la lectura de estas definiciones y la creación de un objeto a partir de ella.

**Herencia**

Por ejemplo, herencia de la clase C a la clase D, es la facilidad mediante la cual la clase D hereda en ella cada uno de los atributos y operaciones de C, como si esos atributos y operaciones hubiesen sido definidos por la misma D. Por lo tanto, puede usar los mismos métodos y variables públicas declaradas en C. Los componentes registrados como "privados" (private) también se heredan, pero como no pertenecen a la clase, se mantienen escondidos al programador y sólo pueden ser accedidos a través de otros métodos públicos. En el caso de los componentes registrados como "protegidos" (protected) también se heredan, pero solo para esa clase, no para futuras clases heredadas. Esto es así para mantener hegemónico el ideal de POO.

**Objeto**

Instancia de una clase. Entidad provista de un conjunto de propiedades o atributos (datos) y de comportamiento o funcionalidad (métodos), los mismos que consecuentemente reaccionan a eventos. Se corresponden con los objetos reales del mundo que nos rodea, o con objetos internos del sistema (del programa).

**Método**

Algoritmo asociado a un objeto (o a una clase de objetos), cuya ejecución se desencadena tras la recepción de un "mensaje". Desde el punto de vista del comportamiento, es lo que el objeto puede hacer. Un método puede producir un cambio en las propiedades del objeto, o la generación de un "evento" con un nuevo mensaje para otro objeto del sistema.

**Evento**

Es un suceso en el sistema (tal como una interacción del usuario con la máquina, o un mensaje enviado por un objeto). El sistema maneja el evento enviando el mensaje adecuado al objeto pertinente. También se puede definir como evento la reacción que puede desencadenar un objeto; es decir, la acción que genera.

**Atributos**

Características que tiene la clase.

**Mensaje**

Una comunicación dirigida a un objeto, que le ordena que ejecute uno de sus métodos con ciertos parámetros asociados al evento que lo generó.

**Propiedad o atributo**

Contenedor de un tipo de datos asociados a un objeto (o a una clase de objetos), que hace los datos visibles desde fuera del objeto y esto se define como sus características predeterminadas, y cuyo valor puede ser alterado por la ejecución de algún método.

**Estado interno**

Es una variable que se declara privada, que puede ser únicamente accedida y alterada por un método del objeto, y que se utiliza para indicar distintas situaciones posibles para el objeto (o clase de objetos). No es visible al programador que maneja una instancia de la clase.

**Componentes de un objeto**

Atributos, identidad, relaciones y métodos.

**Identificación de un objeto**

Un objeto se representa por medio de una tabla o entidad que esté compuesta por sus atributos y funciones correspondientes.

En comparación con un lenguaje imperativo, una "variable" no es más que un contenedor interno del atributo del objeto o de un estado interno, así como la "función" es un procedimiento interno del método del objeto.

Características de la POO

Existe un acuerdo acerca de qué características contempla la "orientación a objetos". Las características siguientes son las más importantes:

**Abstracción**

Denota las características esenciales de un objeto, donde se capturan sus comportamientos. Cada objeto en el sistema sirve como modelo de un "agente" abstracto que puede realizar trabajo, informar y cambiar su estado, y "comunicarse" con otros objetos en el sistema sin revelar "cómo" se implementan estas características. Los procesos, las funciones o los métodos pueden también ser abstraídos, y, cuando lo están, una variedad de técnicas son requeridas para ampliar una abstracción. El proceso de abstracción permite seleccionar las características relevantes dentro de un conjunto e identificar comportamientos comunes para definir nuevos tipos de entidades en el mundo real. La abstracción es clave en el proceso de análisis y diseño orientado a objetos, ya que mediante ella podemos llegar a armar un conjunto de clases que permitan modelar la realidad o el problema que se quiere atacar.

**Encapsulamiento**

Significa reunir todos los elementos que pueden considerarse pertenecientes a una misma entidad, al mismo nivel de abstracción. Esto permite aumentar la cohesión (diseño estructurado) de los componentes del sistema. Algunos autores confunden este concepto con el principio de ocultación, principalmente porque se suelen emplear conjuntamente.

**Polimorfismo**

Comportamientos diferentes, asociados a objetos distintos, pueden compartir el mismo nombre; al llamarlos por ese nombre se utilizará el comportamiento correspondiente al objeto que se esté usando. O, dicho de otro modo, las referencias y las colecciones de objetos pueden contener objetos de diferentes tipos, y la invocación de un comportamiento en una referencia producirá el comportamiento correcto para el tipo real del objeto referenciado. Cuando esto ocurre en "tiempo de ejecución", esta última característica se llama asignación tardía o asignación dinámica. Algunos lenguajes proporcionan medios más estáticos (en "tiempo de compilación") de polimorfismo, tales como las plantillas y la sobrecarga de operadores de C++.

**Herencia**

Las clases no se encuentran aisladas, sino que se relacionan entre sí, formando una jerarquía de clasificación. Los objetos heredan las propiedades y el comportamiento de todas las clases a las que pertenecen. La herencia organiza y facilita el polimorfismo y el encapsulamiento, permitiendo a los objetos ser definidos y creados como tipos especializados de objetos preexistentes. Estos pueden compartir (y extender) su comportamiento sin tener que volver a implementarlo. Esto suele hacerse habitualmente agrupando los objetos en clases y estas en árboles o enrejados que reflejan un comportamiento común. Cuando un objeto hereda de más de una clase se dice que hay herencia múltiple; siendo de alta complejidad técnica por lo cual suele recurrirse a la herencia virtual para evitar la duplicación de datos.

**Modularidad**

Se denomina "modularidad" a la propiedad que permite subdividir una aplicación en partes más pequeñas (llamadas módulos), cada una de las cuales debe ser tan independiente como sea posible de la aplicación en sí y de las restantes partes. Estos módulos se pueden compilar por separado, pero tienen conexiones con otros módulos. Al igual que la encapsulación, los lenguajes soportan la modularidad de diversas formas.

**Principio de ocultación**

Cada objeto está aislado del exterior, es un módulo natural, y cada tipo de objeto expone una "interfaz" a otros objetos que especifica cómo pueden interactuar con los objetos de la clase. El aislamiento protege a las propiedades de un objeto contra su modificación por quien no tenga derecho a acceder a ellas; solamente los propios métodos internos del objeto pueden acceder a su estado. Esto asegura que otros objetos no puedan cambiar el estado interno de un objeto de manera inesperada, eliminando efectos secundarios e interacciones inesperadas. Algunos lenguajes relajan esto, permitiendo un acceso directo a los datos internos del objeto de una manera controlada y limitando el grado de abstracción. La aplicación entera se reduce a un agregado o rompecabezas de objetos.

**Recolección de basura**

La recolección de basura (garbage collection) es la técnica por la cual el entorno de objetos se encarga de destruir automáticamente, y por tanto desvincular la memoria asociada, los objetos que hayan quedado sin ninguna referencia a ellos. Esto significa que el programador no debe preocuparse por la asignación o liberación de memoria, ya que el entorno la asignará al crear un nuevo objeto y la liberará cuando nadie lo esté usando. En la mayoría de los lenguajes híbridos que se extendieron para soportar el Paradigma de Programación Orientada a Objetos como C++ u Object Pascal, esta característica no existe y la memoria debe desasignarse expresamente.

Algunos lenguajes orientados a objetos

Simula (1967) es aceptado como el primer lenguaje que posee las características principales de un lenguaje orientado a objetos. Fue creado para hacer programas de simulación, en donde los "objetos" son la representación de la información más importante.

Smalltalk (1972 a 1980) es posiblemente el ejemplo canónico, y con el que gran parte de la teoría de la programación orientada a objetos se ha desarrollado.

Entre los lenguajes orientados a objetos se destacan los siguientes:

* ABAP
* ABL
* ActionScript
* ActionScript 3
* C Sharp (C#)
* Clarion
* Clipper
* D
* Object Pascal (Embarcadero Delphi)
* Gambas
* GObject
* Genie
* Harbour
* Eiffel
* Fortran 90/95
* Java
* JavaScript
* Lexico
* Objective-C
* Ocaml
* Oz
* R
* Pauscal (en español)
* Perl
* PHP
* PowerBuilder
* Processing.
* Python
* Ruby
* Self
* Smalltalk
* Magik
* Vala
* VB.NET
* Visual FoxPro
* Visual Basic 6.0
* Visual DataFlex
* Visual Objects
* XBase++
* DRP
* Scala

Muchos de estos lenguajes de programación no son puramente orientados a objetos, sino que son híbridos que combinan la POO con otros paradigmas.

Al igual que C++, otros lenguajes, como OOCOBOL, OOLisp, OOProlog y Object REXX, han sido creados añadiendo extensiones orientadas a objetos a un lenguaje de programación clásico.

Un nuevo paso en la abstracción de paradigmas de programación es la Programación Orientada a Aspectos (POA). Aunque es todavía una metodología en estado de maduración, cada vez atrae a más investigadores e incluso proyectos comerciales en todo el mundo.

Compilación de la Programación

El programa escrito en un lenguaje de programación de alto nivel (fácilmente comprensible por el programador) es llamado programa fuente y no se puede ejecutar directamente en una computadora. La opción más común es compilar el programa obteniendo un módulo objeto, aunque también puede ejecutarse en forma más directa a través de un intérprete informático.

El código fuente del programa se debe someter a un proceso de traducción para convertirlo a lenguaje máquina o bien a un código intermedio, generando así un módulo denominado "objeto". A este proceso se le llama compilación.

Habitualmente la creación de un programa ejecutable (un típico.exe para Microsoft Windows o DOS) conlleva dos pasos. El primer paso se llama compilación (propiamente dicho) y traduce el código fuente escrito en un lenguaje de programación almacenado en un archivo de texto a código en bajo nivel (normalmente en código objeto, no directamente a lenguaje máquina). El segundo paso se llama enlazado en el cual se enlaza el código de bajo nivel generado de todos los ficheros y subprogramas que se han mandado compilar y se añade el código de las funciones que hay en las bibliotecas del compilador para que el ejecutable pueda comunicarse directamente con el sistema operativo, traduciendo así finalmente el código objeto a código máquina, y generando un módulo ejecutable.

Estos dos pasos se pueden hacer por separado, almacenando el resultado de la fase de compilación en archivos objetos (un típico .o para Unix, .obj para MS-Windows, DOS); para enlazarlos en fases posteriores, o crear directamente el ejecutable; con lo que la fase de compilación puede almacenarse solo de forma temporal. Un programa podría tener partes escritas en varios lenguajes, por ejemplo, Java, C, C++ y ensamblador, que se podrían compilar de forma independiente y luego enlazar juntas para formar un único módulo ejecutable.

Programación e ingeniería del software

Existe una tendencia a identificar el proceso de creación de un programa informático con la programación, que es cierta cuando se trata de programas pequeños para uso personal, y que dista de la realidad cuando se trata de grandes proyectos.

El proceso de creación de software, desde el punto de vista de la ingeniería, incluye mínimamente los siguientes pasos:

Reconocer la necesidad de un programa para solucionar un problema o identificar la posibilidad de automatización de una tarea.

Recoger los requisitos del programa. Debe quedar claro qué es lo que debe hacer el programa y para qué se necesita.

Realizar el análisis de los requisitos del programa. Debe quedar claro qué tareas debe realizar el programa. Las pruebas que comprueben la validez del programa se pueden especificar en esta fase.

Diseñar la arquitectura del programa. Se debe descomponer el programa en partes de complejidad abordable.

Implementar el programa. Consiste en realizar un diseño detallado, especificando completamente todo el funcionamiento del programa, tras lo cual la codificación (programación propiamente dicha) debería resultar inmediata.

Probar el programa. Comprobar que pasan pruebas que se han definido en el análisis de requisitos.

Implantar (instalar) el programa. Consiste en poner el programa en funcionamiento junto con los componentes que pueda necesitar (bases de datos, redes de comunicaciones, etc.).

La ingeniería del software se centra en los pasos de planificación y diseño del programa, mientras que antiguamente (programación artesanal) la realización de un programa consistía casi únicamente en escribir el código, bajo solo el conocimiento de los requisitos y con una modesta fase de análisis y diseño.

Referencias históricas

El trabajo de Ada Lovelace, hija de Anabella Milbanke Byron y Lord Byron, que realizó para la máquina de Babbage le hizo ganarse el título de primera programadora de computadoras del mundo, aunque Babbage nunca completó la construcción de la máquina. El nombre del lenguaje de programación Ada fue escogido como homenaje a esta programadora.

Objetivos de la programación

La programación debe perseguir la obtención de programas de calidad. Para ello se establece una serie de factores que determinan la calidad de un programa. Algunos de los factores de calidad más importantes son los siguientes:

* Correctitud. Un programa es correcto si hace lo que debe hacer tal y como se estableció en las fases previas a su desarrollo. Para determinar si un programa hace lo que debe, es muy importante especificar claramente qué debe hacer el programa antes de su desarrollo y, una vez acabado, compararlo con lo que realmente hace.
* Claridad. Es muy importante que el programa sea lo más claro y legible posible, para facilitar tanto su desarrollo como su posterior mantenimiento. Al elaborar un programa se debe intentar que su estructura sea sencilla y coherente, así como cuidar el estilo de programación. De esta forma se ve facilitado el trabajo del programador, tanto en la fase de creación como en las fases posteriores de corrección de errores, ampliaciones, modificaciones, etc. Fases que pueden ser realizadas incluso por otro programador, con lo cual la claridad es aún más necesaria para que otros puedan continuar el trabajo fácilmente. Algunos programadores llegan incluso a utilizar Arte ASCII para delimitar secciones de código; una práctica común es realizar aclaraciones en el código fuente utilizando líneas de comentarios. Contrariamente, algunos por diversión o para impedirle un análisis cómodo a otros programadores, recurren al uso de código ofuscado.
* Eficiencia. Se trata de que el programa, además de realizar aquello para lo que fue creado (es decir, que sea correcto), lo haga gestionando de la mejor forma posible los recursos que utiliza. Normalmente, al hablar de eficiencia de un programa, se suele hacer referencia al tiempo que tarda en realizar la tarea para la que ha sido creado y a la cantidad de memoria que necesita, pero hay otros recursos que también pueden ser de consideración para mejorar la eficiencia de un programa, dependiendo de su naturaleza (espacio en disco que utiliza, tráfico en la red que genera, etc.).
* Portabilidad. Un programa es portable cuando tiene la capacidad de poder ejecutarse en una plataforma, ya sea hardware o software, diferente a aquella en la que se desarrolló. La portabilidad es una característica muy deseable para un programa, ya que permite, por ejemplo, a un programa que se ha elaborado para el sistema GNU/Linux ejecutarse también en la familia de sistemas operativos Windows. Esto permite que el programa pueda llegar a más usuarios más fácilmente.

Ciclo de vida del software

El término ciclo de vida del software describe el desarrollo de software, desde la fase inicial hasta la fase final, incluyendo su estado funcional. El propósito es definir las distintas fases intermedias que se requieren para validar el desarrollo de la aplicación, es decir, para garantizar que el software cumpla los requisitos para la aplicación y verificación de los procedimientos de desarrollo: se asegura que los métodos utilizados son apropiados. Estos métodos se originan en el hecho de que es muy costoso rectificar los errores que se detectan tarde dentro de la fase de implementación (programación propiamente dicha), o peor aun, durante la fase funcional. El modelo de ciclo de vida permite que los errores se detecten lo antes posible y por lo tanto, permite a los desarrolladores concentrarse en la calidad del software, en los plazos de implementación y en los costos asociados. El ciclo de vida básico de un software consta de, al menos, los siguientes procedimientos:

* Definición de objetivos: definir el resultado del proyecto y su papel en la estrategia global.
* Análisis de los requisitos y su viabilidad: recopilar, examinar y formular los requisitos del cliente y examinar cualquier restricción que se pueda aplicar.
* Diseño general: requisitos generales de la arquitectura de la aplicación.
* Diseño en detalle: definición precisa de cada subconjunto de la aplicación.
* Programación (programación e implementación): es la implementación en un lenguaje de programación para crear las funciones definidas durante la etapa de diseño.
* Prueba de unidad: prueba individual de cada subconjunto de la aplicación para garantizar que se implementaron de acuerdo con las especificaciones.
* Integración: para garantizar que los diferentes módulos y subprogramas se integren con la aplicación. Este es el propósito de la prueba de integración que debe estar cuidadosamente documentada.
* Prueba beta (o validación), para garantizar que el software cumple con las especificaciones originales.
* Documentación: se documenta con toda la información necesaria, sea funcional final para los usuarios del software (manual del usuario), y de desarrollo para futuras adaptaciones, ampliaciones y correcciones.
* Mantenimiento: para todos los procedimientos correctivos (mantenimiento correctivo) y las actualizaciones secundarias del software (mantenimiento continuo).

El orden y la presencia de cada uno de estos procedimientos en el ciclo de vida de una aplicación dependen del tipo de modelo de ciclo de vida acordado entre el cliente y el equipo de desarrolladores. En el caso del software libre se tiene un ciclo de vida mucho más dinámico, puesto que muchos programadores trabajan en simultáneo desarrollando sus aportaciones.

Código espagueti

El código espagueti es un término peyorativo para los programas de computación que tienen una estructura de control de flujo compleja e incomprensible. Su nombre deriva del hecho que este tipo de código parece asemejarse a un plato de espaguetis, es decir, un montón de hilos intrincados y anudados.

Tradicionalmente suele asociarse este estilo de programación con lenguajes básicos y antiguos, donde el flujo se controlaba mediante sentencias de control muy primitivas como goto y utilizando números de línea.

Evolución de los lenguajes de programación

En los años 60 solo se podía definir el flujo de un programa a base de poner if's anidados y variables booleanas para decidir la lógica. Más adelante ya se podían escribir funciones, de lo más general a lo más específico y en la actualidad con la programación orientada objetos hay conceptos como la modularidad, abstracción, encapsulamiento, desacoplamiento, herencia, etc... Hay herramientas y conceptos más que suficientes como para evitar escribir este tipo de código.

Refactorización

No hay nada malo en escribir código spaghetti inicialmente si esto facilita la comprensión del problema, lo incorrecto es considerar el código como acabado dejándolo de esta forma. Lo correcto es iterar sobre varios repasos del código usando lo que se conoce como refactorización. Las modernos entornos de programación como Eclipse o Visual Studio incluso incorporan funciones automáticas que facilitan esta tarea.

De código spaghetti a programación funcional

Se puede dar una comparación con las reglas básicas del periodismo de como escribir una noticia. Estas indican que en el titular de la noticia tiene que mostrar la información más importante, a continuación entrar en detalle en el primer párrafo y después desglosar en los apartados subsecuentes. Así pues en el main del programa debe haber las funciones más generales y cada una de ellas tiene que llamar a continuación a funciones más específicas.

Con "refactorizar - extraer método" se puede realizar automáticamente cada rama de los if's del código spagettis se convierta en una función, mejorando mucho la visibilidad.

De programación funcional a programación orientada a objetos

Una vez se tienen las funciones claramente visibles es posible identificar cuales conforman una responsabilidad común y no entrelazada con el resto del código y por tanto es posible separarlas en objetos diferentes que ya tendrán los métodos y propiedades (que pueden ser privados, no visibles, desde el resto del código).

Error de software

Un error de software, error o simplemente fallo (también conocido por el inglés bug, «parásito, como las garrapatas y las pulgas»), problema en un programa de computador o sistema de software que desencadena un resultado indeseado. Los programas que ayudan a la detección y eliminación de errores de programación de software son denominados depuradores (debuggers).

Orígenes del término

En 1967, los creadores de Mark III informaron del primer caso de error en un ordenador causado por un bug. El Mark III, ordenador sucesor de ASCC Mark II, construido en 1944, sufrió un fallo en un relé electromagnético. Cuando se investigó ese relé, se encontró una polilla (bug) que provocó que el relé quedase abierto. Grace Murray Hopper, licenciada en física y destacada matemática que trabajó como programadora en el Mark II, pegó el insecto con cinta adhesiva en la bitácora.4

Este incidente es erróneamente referido como el origen de la utilización del término inglés bug («bicho») para indicar un problema en un aparato o sistema.5 6 En realidad, el término bug ya formaba parte del idioma inglés, al menos desde que Thomas Alva Edison lo utilizó en 1889 refiriéndose a interferencias y mal funcionamiento. Es posible que Hopper lo haya asociado por primera vez a la informática, en este caso, relacionado a un insecto real. Por otra parte, aunque durante los años 50 del siglo XX, Hopper también empleó en inglés el término debug al hablar de la depuración de errores en los códigos de programación, el primer uso registrado del término se encuentra en la Journal of the Royal Aeronautical Society de 1945.7

Defectos de diseño de programas

* Diseños con colores inapropiados para las personas que padecen daltonismo
* Diseños que usan textos con tipografías de difícil lectura por su tamaño o diseño
* Diseños que fuerzan el uso del ratón sin dejar alternativas de teclado para personas con disfunciones motrices
* Diseños con implicaciones culturales, por ejemplo usando partes del cuerpo que en una determinada cultura sean objeto de vergüenza o burla o símbolos con características de identidad cultural o religiosa
* Estimar que el equipo donde se instalará tiene determinadas características (como la resolución de la pantalla, la velocidad del procesador, la cantidad de memoria o conectividad a Internet) propias de un equipo de gama alta, en vez de diseñar el software para su ejecución en equipos normales

Errores de programación comunes

* División por cero
* Ciclo infinito
* Problemas aritméticos como desbordamientos (overflow) o subdesbordamientos (underflow).
* Exceder el tamaño del array
* Utilizar una variable no inicializada
* Acceder a memoria no permitida (Violación de acceso)
* Pérdida de memoria (memory leak)
* Desbordamiento o subdesbordamiento de la pila (estructura de datos)
* Desbordamiento de búfer (buffer overflow)
* Bloqueo mutuo (deadlock)
* Indizado inadecuado de tablas en bases de datos.
* Desbordamiento de la pila de recursión, cuando se dejan demasiadas llamadas en espera.

Defectos de instalación o programación

* Eliminación o sustitución de bibliotecas comunes a más de un programa o del sistema (DLL Hell).
* Reiniciar arbitrariamente la sesión de un usuario para que la instalación tenga efecto.
* Suponer que el usuario tiene una conexión permanente a internet.
* Utilizar como fuente enlaces simbólicos a ficheros que pueden cambiar de ubicación.

Línea de código fuente

La definición de línea de código fuente es esencialmente ambigua para la mayor parte del software. Su significado varía de un lenguaje de programación a otro, pero también dentro de un mismo lenguaje de programación.

Una línea de código fuente es cada una de las líneas de un archivo de código fuente de un programa informático. Habitualmente en cada línea se ejecuta una instrucción que tiene que ejecutar el programa. También es habitual tabular las estructuras de control del programa en cuestión para una lectura más fácil. Viene a ser como la oración en libros y textos escritos en general.

En ocasiones los programadores hablan del número de “líneas de código” que tiene cierto programa para hablar de la magnitud o complejidad de este.

En computación, el número de línea de una instrucción es un punto bastante útil a la hora de compilar el programa porque habitualmente los compiladores detectan errores de programación mostrando el número de línea donde se ha encontrado el error que el programador deberá corregir para una compilación satisfactoria.

Como curiosidad, algunos programadores se divierten complicando la forma de programar, bien por diversión, como reto entre programadores, o para que sea imposible de entender para un programador poco experimentado. A este pasatiempo se le denomina programación ofuscada y uno de los puntos más habituales para programar ofuscadamente es no escribir una instrucción por línea y no hacer tabulaciones, en ocasiones se escriben varias instrucciones por línea o a veces se corta una instrucción en varias líneas. Los más experimentados en este tipo de pasatiempos, se atreven incluso a realizar obras de Ascii art con las líneas de su código fuente.

En el lenguaje de programación C, por ejemplo, una línea de código puede ser:

1. una instrucción acabada en un salto de línea,
2. una instrucción acabada en un punto y coma,
3. cualquier línea del programa que acabe en un salto de línea (comentarios incluidos).

Por ejemplo:

**for** (i=0; i<100; ++i) {printf("hola");} */\* ¿Cuántas líneas tiene este programa? \*/*

Lenguaje de programación

Un lenguaje de programación es un lenguaje formal diseñado para realizar procesos que pueden ser llevados a cabo por máquinas como las computadoras.

Pueden usarse para crear programas que controlen el comportamiento físico y lógico de una máquina, para expresar algoritmos con precisión, o como modo de comunicación humana.1

Está formado por un conjunto de símbolos y reglas sintácticas y semánticas que definen su estructura y el significado de sus elementos y expresiones. Al proceso por el cual se escribe, se prueba, se depura, se compila (de ser necesario) y se mantiene el código fuente de un programa informático se le llama programación.

También la palabra programación se define como el proceso de creación de un programa de computadora, mediante la aplicación de procedimientos lógicos, a través de los siguientes pasos:

* El desarrollo lógico del programa para resolver un problema en particular.
* Escritura de la lógica del programa empleando un lenguaje de programación específico (codificación del programa).
* Ensamblaje o compilación del programa hasta convertirlo en lenguaje de máquina.
* Prueba y depuración del programa.
* Desarrollo de la documentación.

Existe un error común que trata por sinónimos los términos 'lenguaje de programación' y 'lenguaje informático'. Los lenguajes informáticos engloban a los lenguajes de programación y a otros más, como por ejemplo HTML (lenguaje para el marcado de páginas web que no es propiamente un lenguaje de programación, sino un conjunto de instrucciones que permiten estructurar el contenido de los documentos).

Permite especificar de manera precisa sobre qué datos debe operar una computadora, cómo deben ser almacenados o transmitidos y qué acciones debe tomar bajo una variada gama de circunstancias. Todo esto, a través de un lenguaje que intenta estar relativamente próximo al lenguaje humano o natural. Una característica relevante de los lenguajes de programación es precisamente que más de un programador pueda usar un conjunto común de instrucciones que sean comprendidas entre ellos para realizar la construcción de un programa de forma colaborativa.

Historia

Para que la computadora entienda nuestras instrucciones debe usarse un lenguaje específico conocido como código máquina, el cual la máquina comprende fácilmente, pero que lo hace excesivamente complicado para las personas. De hecho sólo consiste en cadenas extensas de números 0 y 1.

Para facilitar el trabajo, los primeros operadores de computadoras decidieron hacer un traductor para reemplazar los 0 y 1 por palabras o abstracción de palabras y letras provenientes del inglés; éste se conoce como lenguaje ensamblador. Por ejemplo, para sumar se usa la letra A de la palabra inglesa add (sumar). El lenguaje ensamblador sigue la misma estructura del lenguaje máquina, pero las letras y palabras son más fáciles de recordar y entender que los números.

La necesidad de recordar secuencias de programación para las acciones usuales llevó a denominarlas con nombres fáciles de memorizar y asociar: ADD (sumar), SUB (restar), MUL (multiplicar), CALL (ejecutar subrutina), etc. A esta secuencia de posiciones se le denominó "instrucciones", y a este conjunto de instrucciones se le llamó lenguaje ensamblador. Posteriormente aparecieron diferentes lenguajes de programación, los cuales reciben su denominación porque tienen una estructura sintáctica semejante a la de los lenguajes escritos por los humanos, denominados también lenguajes de alto nivel.

El primer programador de computadora que se haya conocido fue una mujer: Ada Lovelace, hija de Anabella Milbanke Byron y Lord Byron. Anabella inició en las matemáticas a Ada quien, después de conocer a Charles Babbage, tradujo y amplió una descripción de su máquina analítica. Incluso aunque Babbage nunca completó la construcción de cualquiera de sus máquinas, el trabajo que Ada realizó con éstas le hizo ganarse el título de primera programadora de computadoras del mundo. El nombre del lenguaje de programación Ada fue escogido como homenaje a esta programadora.

A finales de 1953, John Backus sometió una propuesta a sus superiores en IBM para desarrollar una alternativa más práctica al lenguaje ensamblador para programar la computadora central IBM 704. El histórico equipo Fortran de Backus consistió en los programadores Richard Goldberg, Sheldon F. Best, Harlan Herrick, Peter Sheridan, Roy Nutt, Robert Nelson, Irving Ziller, Lois Haibt y David Sayre.2

El primer manual para el lenguaje Fortran apareció en octubre de 1956, con el primer compilador Fortran entregado en abril de 1957. Esto era un compilador optimizado, porque los clientes eran reacios a usar un lenguaje de alto nivel a menos que su compilador pudiera generar código cuyo desempeño fuera comparable al de un código hecho a mano en lenguaje ensamblador.

En 1960, se creó COBOL, uno de los lenguajes usados aún en la actualidad, en informática de gestión.

A medida que la complejidad de las tareas que realizaban las computadoras aumentaba, se hizo necesario disponer de un método más eficiente para programarlas. Entonces, se crearon los lenguajes de alto nivel, como lo fue BASIC en las versiones introducidas en los microordenadores de la década de 1980. Mientras que una tarea tan sencilla como sumar dos números puede necesitar varias instrucciones en lenguaje ensamblador, en un lenguaje de alto nivel bastará una sola sentencia.

Elementos

Variables y vectores

Las variables son títulos asignados a espacios en memoria para almacenar datos específicos. Son contenedores de datos y por ello se diferencian según el tipo de dato que son capaces de almacenar. En la mayoría de lenguajes de programación se requiere especificar un tipo de variable concreto para guardar un dato específico. Por ejemplo, en Java, si deseamos guardar una cadena de texto debemos especificar que la variable es del tipo String. Por otra parte, en lenguajes como PHP este tipo de especificación de variables no es necesario. Además, existen variables compuestas llamadas vectores. Un vector no es más que un conjunto de bytes consecutivas en memoria y del mismo tipo guardadas dentro de una variable contenedor. A continuación, un listado con los tipos de variables y vectores más comunes:

|  |  |
| --- | --- |
| **Tipo de dato** | **Breve descripción** |
| Char | Estas variables contienen un único carácter, es decir, una letra, un signo o un número. |
| Int | Contienen un número entero. |
| Float | Contienen un número decimal. |
| String | Contienen cadenas de texto, o lo que es lo mismo, es un vector con varias variables del tipo Char. |
| Boolean | Solo pueden contener un cero o un uno. |

En el caso de variables booleanas, el cero es considerado para muchos lenguajes como el literal falso ("False"), mientras que el uno se considera verdadero ("True").

Condicionales

Las sentencias condicionales son estructuras de código que indican que, para que cierta parte del programa se ejecute, deben cumplirse ciertas premisas; por ejemplo: que dos valores sean iguales, que un valor exista, que un valor sea mayor que otro... Estos condicionantes por lo general solo se ejecutan una vez a lo largo del programa. Los condicionantes más conocidos y empleados en programación son:

* If: Indica una condición para que se ejecute una parte del programa.
* Else if: Siempre va precedido de un "If" e indica una condición para que se ejecute una parte del programa siempre que no cumpla la condición del if previo y si se cumpla con la que el "else if" especifique.
* Else: Siempre precedido de "If" y en ocasiones de "Else If". Indica que debe ejecutarse cuando no se cumplan las condiciones previas.

Bucles

Los bucles son parientes cercanos de los condicionantes, pero ejecutan constantemente un código mientras se cumpla una determinada condición. Los más frecuentes son:

* For: Ejecuta un código mientras una variable se encuentre entre 2 determinados parámetros.
* While: Ejecuta un código mientras que se cumpla la condición que solicita.

Hay que decir que a pesar de que existan distintos tipos de bucles, ambos son capaces de realizar exactamente las mismas funciones. El empleo de uno u otro depende, por lo general, del gusto del programador.

Funciones

Las funciones se crearon para evitar tener que repetir constantemente fragmentos de código. Una función podría considerarse como una variable que encierra código dentro de si. Por lo tanto cuando accedemos a dicha variable (la función) en realidad lo que estamos haciendo es ordenar al programa que ejecute un determinado código predefinido anteriormente.

Todos los lenguajes de programación tienen algunos elementos de formación primitivos para la descripción de los datos y de los procesos o transformaciones aplicadas a estos datos (tal como la suma de dos números o la selección de un elemento que forma parte de una colección). Estos elementos primitivos son definidos por reglas sintácticas y semánticas que describen su estructura y significado respectivamente.

Sintaxis

Con frecuencia se resaltan los elementos de la sintaxis con colores diferentes para facilitar su lectura. Este ejemplo está escrito en Python.

A la forma visible de un lenguaje de programación se le conoce como sintaxis. La mayoría de los lenguajes de programación son puramente textuales, es decir, utilizan secuencias de texto que incluyen palabras, números y puntuación, de manera similar a los lenguajes naturales escritos. Por otra parte, hay algunos lenguajes de programación que son más gráficos en su naturaleza, utilizando relaciones visuales entre símbolos para especificar un programa.

La sintaxis de un lenguaje de programación describe las combinaciones posibles de los símbolos que forman un programa sintácticamente correcto. El significado que se le da a una combinación de símbolos es manejado por su semántica (ya sea formal o como parte del código duro de la referencia de implementación). Dado que la mayoría de los lenguajes son textuales, este artículo trata de la sintaxis textual.

La sintaxis de los lenguajes de programación es definida generalmente utilizando una combinación de expresiones regulares (para la estructura léxica) y la Notación de Backus-Naur (para la estructura gramática). Este es un ejemplo de una gramática simple, tomada de Lisp:

expresión ::= átomo | lista

átomo ::= número | símbolo

número ::= [+-]? ['0'-'9']+

símbolo ::= ['A'-'Z'] ['a'-'z'].\*

lista ::= '(' expresión\* ')'

Con esta gramática se especifica lo siguiente:

* una expresión puede ser un átomo o una lista;
* un átomo puede ser un número o un símbolo;
* un número es una secuencia continua de uno o más dígitos decimales, precedido opcionalmente por un signo más o un signo menos;
* un símbolo es una letra seguida de cero o más caracteres (excluyendo espacios); y
* una lista es un par de paréntesis que abren y cierran, con cero o más expresiones en medio.

Algunos ejemplos de secuencias bien formadas de acuerdo a esta gramática:

'12345', '()', '(a b c232 (1))'

No todos los programas sintácticamente correctos son semánticamente correctos. Muchos programas sintácticamente correctos tienen inconsistencias con las reglas del lenguaje; y pueden (dependiendo de la especificación del lenguaje y la solidez de la implementación) resultar en un error de traducción o ejecución. En algunos casos, tales programas pueden exhibir un comportamiento indefinido. Además, incluso cuando un programa está bien definido dentro de un lenguaje, todavía puede tener un significado que no es el que la persona que lo escribió estaba tratando de construir.

Usando el lenguaje natural, por ejemplo, puede no ser posible asignarle significado a una oración gramaticalmente válida o la oración puede ser falsa:

* "Las ideas verdes y descoloridas duermen furiosamente" es una oración bien formada gramaticalmente pero no tiene significado comúnmente aceptado.
* "Juan es un soltero casado" también está bien formada gramaticalmente pero expresa un significado que no puede ser verdadero.

El siguiente fragmento en el lenguaje C es sintácticamente correcto, pero ejecuta una operación que no está definida semánticamente (dado que p es un apuntador nulo, las operaciones p->real y p->im no tienen ningún significado):

complex \*p = NULL;

complex abs\_p = sqrt (p->real \* p->real + p->im \* p->im);

Si la declaración de tipo de la primera línea fuera omitida, el programa dispararía un error de compilación, pues la variable "p" no estaría definida. Pero el programa sería sintácticamente correcto todavía, dado que las declaraciones de tipo proveen información semántica solamente.

La gramática necesaria para especificar un lenguaje de programación puede ser clasificada por su posición en la Jerarquía de Chomsky. La sintaxis de la mayoría de los lenguajes de programación puede ser especificada utilizando una gramática Tipo-2, es decir, son gramáticas libres de contexto. Algunos lenguajes, incluyendo a Perl y a Lisp, contienen construcciones que permiten la ejecución durante la fase de análisis. Los lenguajes que permiten construcciones que permiten al programador alterar el comportamiento de un analizador hacen del análisis de la sintaxis un problema sin decisión única, y generalmente oscurecen la separación entre análisis y ejecución. En contraste con el sistema de macros de Lisp y los bloques BEGIN de Perl, que pueden tener cálculos generales, las macros de C son meros reemplazos de cadenas, y no requieren ejecución de código.

Semántica estática

La semántica estática define las restricciones sobre la estructura de los textos válidos que resulta imposible o muy difícil expresar mediante formalismos sintácticos estándar. Para los lenguajes compilados, la semántica estática básicamente incluye las reglas semánticas que se pueden verificar en el momento de compilar. Por ejemplo el chequeo de que cada identificador sea declarado antes de ser usado (en lenguajes que requieren tales declaraciones) o que las etiquetas en cada brazo de una estructura case sean distintas. Muchas restricciones importantes de este tipo, como la validación de que los identificadores sean usados en los contextos apropiados (por ejemplo no sumar un entero al nombre de una función), o que las llamadas a subrutinas tengan el número y tipo de parámetros adecuado, puede ser implementadas definiéndolas como reglas en una lógica conocida como sistema de tipos. Otras formas de análisis estáticos, como los análisis de flujo de datos, también pueden ser parte de la semántica estática. Otros lenguajes de programación como Java y C# tienen un análisis definido de asignaciones, una forma de análisis de flujo de datos, como parte de su semántica estática.

Sistema de tipo

Un sistema de tipos define la manera en la cual un lenguaje de programación clasifica los valores y expresiones en tipos, cómo pueden ser manipulados dichos tipos y cómo interactúan. El objetivo de un sistema de tipos es verificar y normalmente poner en vigor un cierto nivel de exactitud en programas escritos en el lenguaje en cuestión, detectando ciertas operaciones inválidas. Cualquier sistema de tipos decidible tiene sus ventajas y desventajas: mientras por un lado rechaza muchos programas incorrectos, también prohíbe algunos programas correctos aunque poco comunes. Para poder minimizar esta desventaja, algunos lenguajes incluyen lagunas de tipos, conversiones explícitas no verificadas que pueden ser usadas por el programador para permitir explícitamente una operación normalmente no permitida entre diferentes tipos. En la mayoría de los lenguajes con tipos, el sistema de tipos es usado solamente para verificar los tipos de los programas, pero varios lenguajes, generalmente funcionales, llevan a cabo lo que se conoce como inferencia de tipos, que le quita al programador la tarea de especificar los tipos. Al diseño y estudio formal de los sistemas de tipos se le conoce como teoría de tipos.

Implementación

Código fuente de un programa escrito en el lenguaje de programación Java.

La implementación de un lenguaje es la que provee una manera de que se ejecute un programa para una determinada combinación de software y hardware. Existen básicamente dos maneras de implementar un lenguaje: compilación e interpretación.

* Compilación: es el proceso que traduce un programa escrito en un lenguaje de programación a otro lenguaje de programación, generando un programa equivalente que la máquina será capaz interpretar. Los programas traductores que pueden realizar esta operación se llaman compiladores. Éstos, como los programas ensambladores avanzados, pueden generar muchas líneas de código de máquina por cada proposición del programa fuente.
* Interpretación: es una asignación de significados a las fórmulas bien formadas de un lenguaje formal. Como los lenguajes formales pueden definirse en términos puramente sintácticos, sus fórmulas bien formadas pueden no ser más que cadenas de símbolos sin ningún significado. Una interpretación otorga significado a esas fórmulas.

Se puede también utilizar una alternativa para traducir lenguajes de alto nivel. En lugar de traducir el programa fuente y grabar en forma permanente el código objeto que se produce durante la compilación para utilizarlo en una ejecución futura, el programador sólo carga el programa fuente en la computadora junto con los datos que se van a procesar. A continuación, un programa intérprete, almacenado en el sistema operativo del disco, o incluido de manera permanente dentro de la máquina, convierte cada proposición del programa fuente en lenguaje de máquina conforme vaya siendo necesario durante el procesamiento de los datos. El código objeto no se graba para utilizarlo posteriormente.

La siguiente vez que se utilice una instrucción, se la deberá interpretar otra vez y traducir a lenguaje máquina. Por ejemplo, durante el procesamiento repetitivo de los pasos de un ciclo o bucle, cada instrucción del bucle tendrá que volver a ser interpretada en cada ejecución repetida del ciclo, lo cual hace que el programa sea más lento en tiempo de ejecución (porque se va revisando el código en tiempo de ejecución) pero más rápido en tiempo de diseño (porque no se tiene que estar compilando a cada momento el código completo). El intérprete elimina la necesidad de realizar una compilación después de cada modificación del programa cuando se quiere agregar funciones o corregir errores; pero es obvio que un programa objeto compilado con antelación deberá ejecutarse con mucha mayor rapidez que uno que se debe interpretar a cada paso durante una ejecución del código.

La mayoría de lenguajes de alto nivel permiten la programación multipropósito, aunque muchos de ellos fueron diseñados para permitir programación dedicada, como lo fue el Pascal con las matemáticas en su comienzo. También se han implementado lenguajes educativos infantiles como Logo mediante una serie de simples instrucciones. En la actualidad son muy populares algunos lenguajes especialmente indicados para aplicaciones web, como Perl, PHP, Ruby, Python o JavaScript.

Técnica

Libros sobre diversos lenguajes de programación.

Para escribir programas que proporcionen los mejores resultados, cabe tener en cuenta una serie de detalles.

Corrección. Un programa es correcto si hace lo que debe hacer tal y como se estableció en las fases previas a su desarrollo. Para determinar si un programa hace lo que debe, es muy importante especificar claramente qué debe hacer el programa antes de desarrollarlo y, una vez acabado, compararlo con lo que realmente hace.

Claridad. Es muy importante que el programa sea lo más claro y legible posible, para facilitar así su desarrollo y posterior mantenimiento. Al elaborar un programa se debe intentar que su estructura sea sencilla y coherente, así como cuidar el estilo en la edición; de esta forma se ve facilitado el trabajo del programador, tanto en la fase de creación como en las fases posteriores de corrección de errores, ampliaciones, modificaciones, etc. Fases que pueden ser realizadas incluso por otro programador, con lo cual la claridad es aún más necesaria para que otros programadores puedan continuar el trabajo fácilmente. Algunos programadores llegan incluso a utilizar Arte ASCII para delimitar secciones de código. Otros, por diversión o para impedir un análisis cómodo a otros programadores, recurren al uso de código ofuscado.

Eficiencia. Se trata de que el programa, además de realizar aquello para lo que fue creado (es decir, que sea correcto), lo haga gestionando de la mejor forma posible los recursos que utiliza. Normalmente, al hablar de eficiencia de un programa, se suele hacer referencia al tiempo que tarda en realizar la tarea para la que ha sido creado y a la cantidad de memoria que necesita, pero hay otros recursos que también pueden ser de consideración al obtener la eficiencia de un programa, dependiendo de su naturaleza (espacio en disco que utiliza, tráfico de red que genera, etc.).

Portabilidad. Un programa es portable cuando tiene la capacidad de poder ejecutarse en una plataforma, ya sea hardware o software, diferente a aquella en la que se elaboró. La portabilidad es una característica muy deseable para un programa, ya que permite, por ejemplo, a un programa que se ha desarrollado para sistemas GNU/Linux ejecutarse también en la familia de sistemas operativos Windows. Esto permite que el programa pueda llegar a más usuarios más fácilmente.

Paradigmas

Los programas se pueden clasificar por el paradigma del lenguaje que se use para producirlos. Los principales paradigmas son: imperativos, declarativos y orientación a objetos.

Los programas que usan un lenguaje imperativo especifican un algoritmo, usan declaraciones, expresiones y sentencias.3 Una declaración asocia un nombre de variable con un tipo de dato, por ejemplo: var x: integer;. Una expresión contiene un valor, por ejemplo: 2 + 2 contiene el valor 4. Finalmente, una sentencia debe asignar una expresión a una variable o usar el valor de una variable para alterar el flujo de un programa, por ejemplo: x := 2 + 2; if x == 4 then haz\_algo();. Una crítica común en los lenguajes imperativos es el efecto de las sentencias de asignación sobre una clase de variables llamadas "no locales".4

Los programas que usan un lenguaje declarativo especifican las propiedades que la salida debe conocer y no especifican cualquier detalle de implementación. Dos amplias categorías de lenguajes declarativos son los lenguajes funcionales y los lenguajes lógicos. Los lenguajes funcionales no permiten asignaciones de variables no locales, así, se hacen más fácil, por ejemplo, programas como funciones matemáticas.4 El principio detrás de los lenguajes lógicos es definir el problema que se quiere resolver (el objetivo) y dejar los detalles de la solución al sistema.5 El objetivo es definido dando una lista de sub-objetivos. Cada sub-objetivo también se define dando una lista de sus sub-objetivos, etc. Si al tratar de buscar una solución, una ruta de sub-objetivos falla, entonces tal sub-objetivo se descarta y sistemáticamente se prueba otra ruta.

La forma en la cual se programa puede ser por medio de texto o de forma visual. En la programación visual los elementos son manipulados gráficamente en vez de especificarse por medio de texto.

Programación automática

La Programación Automática pretende que sea el propio ordenador o el software el que escriba los programas que necesitan las personas, siguiendo las indicaciones de estas.

Historia

Desde que existen los ordenadores, los programadores han intentado que su trabajo sea lo más sencillo posible. Para ello han inventado y desarrollado diferentes tipos de técnicas y multitud de programas con la idea de resolver los diferentes tipos de problemas, que se les presentaban tanto a ellos mismos como a otras personas, con ayuda de ordenadores esperando que sus trabajos fueran más rentables o simplemente que pudieran solucionarlos con mayor facilidad y efectividad.

Existen muchos ejemplos de programas que escriben programas, por ejemplo, los traductores de lenguajes que toman un programa fuente escrito por el programador y producen como resultado un programa que puede ejecutarse, es decir que producen un ejecutable.

Sin embargo, cuando se habla de programación automática los objetivos son más ambiciosos y eso no es fácil de conseguir.

En Mitchell, T.M. (1997). Machine Learning. McGraw Hill. puedes hacerte una idea de los objetivos de la programación automática.

Dificultades

Existen grandes dificultades a la hora de conseguir estos objetivos. Una de las principales dificultades consiste en que los programas generados por los ordenadores son largos y difíciles de comprender. No son tan sencillos como los programas escritos por personas y en general tienen muchas instrucciones repetidas.

Actualidad

En la actualidad se están realizando esfuerzos en conseguir que el ordenador ayude en la construcción de programas. Algunas empresas están invirtiendo recursos materiales y humanos para conseguir el logro de estos objetivos.

España

En España, en concreto en la Universidad Politécnica de Valencia, se llevan varios años trabajando en este tema en colaboración con una empresa alemana y al parecer los resultados que están obteniendo son muy esperanzadores. En cualquier caso, todavía existe mucho que hacer hasta conseguir que los humanos podamos transmitir nuestras ideas a los ordenadores con el fin de que nos ayuden a llevarlas a la práctica.

Ejemplo: La grabadora de Excel

A continuación se muestra un ejemplo de una herramienta que permite grabar de forma automática subprogramas en VBA.

Descripción

Existe una herramienta en Excel que permite grabar subprogramas sin argumentos (mal llamadas macros en Excel) muy sencilla que es parecida en funcionamiento al típico grabador de cinta de casete que se utiliza para grabar sonido. En este caso lo que se graba son las acciones que realiza el usuario mediante el ratón y el teclado actuando sobre las celdas de la hoja de cálculo.

Existe otra herramienta similar en la hoja de cálculo Spreadsheet OpenOffice.org, aunque el código generado de forma automática en ambas aplicaciones no es diferente e incompatible.

Ejemplos

A continuación se proponen unos sencillos ejemplos con el fin de que el usuario pueda familiarizarse con la herramienta de grabación automática. Una vez generada la nueva macro debería analizarse y comparar el código generado automáticamente con el que nosotros hubiéramos escrito.

* Grabar un modelo de factura.
* Grabar un modelo de presupuesto.
* Grabar un modelo de albarán.

Analizar una macro

Analizar las macros generadas de forma automática y ver qué instrucciones hubiéramos utilizado nosotros para escribir un programa similar al generado de forma automática.

Aprender de la grabadora

Utilizando la herramienta de grabación automática de macros en Excel, escribir un subprograma que cambie el color de una celda de la hoja de cálculo.

Posteriormente, analizar el código generado de forma automática y a partir de él, escribir una nueva macro que recorra una matriz de la hoja de cálculo y cambie el color de todas las celdas que cumplan una determinada condición, tal como test(x), que será definida por el usuario.

También es útil observar qué es lo que va escribiendo la grabadora según vamos solicitando acciones a Excel. Para ello debe mostrarse tanto la pantalla de Excel como la pantalla del módulo en el que se esté grabando la macro. Téngase en cuenta que las macros siempre se graban en un nuevo módulo.

Programación dirigida por eventos

La programación dirigida por eventos es un paradigma de programación en el que tanto la estructura como la ejecución de los programas van determinados por los sucesos que ocurran en el sistema, definidos por el usuario o que ellos mismos provoquen.

Para entender la programación dirigida por eventos, podemos oponerla a lo que no es: mientras en la programación secuencial (o estructurada) es el programador el que define cuál va a ser el flujo del programa, en la programación dirigida por eventos será el propio usuario o lo que sea que esté accionando el programa el que dirija el flujo del programa. Aunque en la programación secuencial puede haber intervención de un agente externo al programa, estas intervenciones ocurrirán cuando el programador lo haya determinado, y no en cualquier momento como puede ser en el caso de la programación dirigida por eventos.

El creador de un programa dirigido por eventos debe definir los eventos que manejarán su programa y las acciones que se realizarán al producirse cada uno de ellos, lo que se conoce como el administrador de evento. Los eventos soportados estarán determinados por el lenguaje de programación utilizado, por el sistema operativo e incluso por eventos creados por el mismo programador.

En la programación dirigida por eventos, al comenzar la ejecución del programa se llevarán a cabo las inicializaciones y demás código inicial y a continuación el programa quedará bloqueado hasta que se produzca algún evento. Cuando alguno de los eventos esperados por el programa tenga lugar, el programa pasará a ejecutar el código del correspondiente administrador de evento. Por ejemplo, si el evento consiste en que el usuario ha hecho clic en el botón de play de un reproductor de películas, se ejecutará el código del administrador de evento, que será el que haga que la película se muestre por pantalla.

Un ejemplo claro lo tenemos en los sistemas de programación Lexico y Visual Basic, en los que a cada elemento del programa (objetos, controles, etcétera) se le asignan una serie de eventos que generará dicho elemento, como la pulsación de un botón del ratón sobre él o el redibujado del control.

La programación dirigida por eventos es la base de lo que llamamos interfaz de usuario, aunque puede emplearse también para desarrollar interfaces entre componentes de Software o módulos del núcleo.

En los primeros tiempos de la computación, los programas eran secuenciales, también llamados Batch. Un programa secuencial arranca, lee parámetros de entrada, procesa estos parámetros, y produce un resultado, todo de manera lineal y sin intervención del usuario mientras se ejecuta.

Con la aparición y popularización de los PC, el software empezó a ser demandado para usos alejados de los clásicos académicos y empresariales para los cuales era necesitado hasta entonces, y quedó patente que el paradigma clásico de programación no podía responder a las nuevas necesidades de interacción con el usuario que surgieron a raíz de este hecho.

Detección de eventos

En contraposición al modelo clásico, la programación orientada a eventos permite interactuar con el usuario en cualquier momento de la ejecución. Esto se consigue debido a que los programas creados bajo esta arquitectura se componen por un bucle exterior permanente encargado de recoger los eventos, y distintos procesos que se encargan de tratarlos. Habitualmente, este bucle externo permanece oculto al programador que simplemente se encarga de tratar los eventos, aunque en algunos entornos de desarrollo (IDE) será necesaria su construcción.

Ejemplo de programa orientado a eventos en pseudo lenguaje:

While (true){

Switch (event){

case mousse\_button\_down:

case mouse\_click:

case keypressed:

case Else:

}

}

Problemática

La programación orientada a eventos supone una complicación añadida con respecto a otros paradigmas de programación, debido a que el flujo de ejecución del software escapa al control del programador. En cierta manera podríamos decir que en la programación clásica el flujo estaba en poder del programador y era este quien decidía el orden de ejecución de los procesos, mientras que en programación orientada a eventos, es el usuario el que controla el flujo y decide.

Pongamos como ejemplo de la problemática existente, un menú con dos botones, botón 1 y botón 2. Cuando el usuario pulsa botón 1, el programa se encarga de recoger ciertos parámetros que están almacenados en un fichero y calcular algunas variables. Cuando el usuario pulsa el botón 2, se le muestran al usuario por pantalla dichas variables. Es sencillo darse cuenta de que la naturaleza indeterminada de las acciones del usuario y las características de este paradigma pueden fácilmente desembocar en el error fatal de que se pulse el botón 2 sin previamente haber sido pulsado el botón 1. Aunque esto no pasa si se tienen en cuenta las propiedades de dichos botones, haciendo inaccesible la pulsación sobre el botón 2 hasta que previamente se haya pulsado el botón 1.

Programación extrema

La programación extrema o eXtreme Programming (de ahora en adelante, XP) es una metodología de desarrollo de la ingeniería de software formulada por Kent Beck, autor del primer libro sobre la materia, Extreme Programming Explained: Embrace Change (1999). Es el más destacado de los procesos ágiles de desarrollo de software. Al igual que éstos, la programación extrema se diferencia de las metodologías tradicionales principalmente en que pone más énfasis en la adaptabilidad que en la previsibilidad. Los defensores de la XP consideran que los cambios de requisitos sobre la marcha son un aspecto natural, inevitable e incluso deseable del desarrollo de proyectos. Creen que ser capaz de adaptarse a los cambios de requisitos en cualquier punto de la vida del proyecto es una aproximación mejor y más realista que intentar definir todos los requisitos al comienzo del proyecto e invertir esfuerzos después en controlar los cambios en los requisitos.

Se puede considerar la programación extrema como la adopción de las mejores metodologías de desarrollo de acuerdo a lo que se pretende llevar a cabo con el proyecto, y aplicarlo de manera dinámica durante el ciclo de vida del software.

Valores

Los valores originales de la programación extrema son: simplicidad, comunicación, retroalimentación (feedback) y coraje. Un quinto valor, respeto, fue añadido en la segunda edición de Extreme Programming Explained. Los cinco valores se detallan a continuación:

Simplicidad

La simplicidad es la base de la programación extrema. Se simplifica el diseño para agilizar el desarrollo y facilitar el mantenimiento. Un diseño complejo del código junto a sucesivas modificaciones por parte de diferentes desarrolladores hacen que la complejidad aumente exponencialmente.

Para mantener la simplicidad es necesaria la refactorización del código, ésta es la manera de mantener el código simple a medida que crece.

También se aplica la simplicidad en la documentación, de esta manera el código debe comentarse en su justa medida, intentando eso sí que el código esté autodocumentado. Para ello se deben elegir adecuadamente los nombres de las variables, métodos y clases. Los nombres largos no decrementan la eficiencia del código ni el tiempo de desarrollo gracias a las herramientas de autocompletado y refactorización que existen actualmente.

Aplicando la simplicidad junto con la autoría colectiva del código y la programación por parejas se asegura que cuanto más grande se haga el proyecto, todo el equipo conocerá más y mejor el sistema completo.

Comunicación

La comunicación se realiza de diferentes formas. Para los programadores el código comunica mejor cuanto más simple sea. Si el código es complejo hay que esforzarse para hacerlo inteligible. El código autodocumentado es más fiable que los comentarios ya que éstos últimos pronto quedan desfasados con el código a medida que es modificado. Debe comentarse sólo aquello que no va a variar, por ejemplo el objetivo de una clase o la funcionalidad de un método.

Las pruebas unitarias son otra forma de comunicación ya que describen el diseño de las clases y los métodos al mostrar ejemplos concretos de como utilizar su funcionalidad. Los programadores se comunican constantemente gracias a la programación por parejas. La comunicación con el cliente es fluida ya que el cliente forma parte del equipo de desarrollo. El cliente decide qué características tienen prioridad y siempre debe estar disponible para solucionar dudas.

Retroalimentación (feedback)

Al estar el cliente integrado en el proyecto, su opinión sobre el estado del proyecto se conoce en tiempo real.

Al realizarse ciclos muy cortos tras los cuales se muestran resultados, se minimiza el tener que rehacer partes que no cumplen con los requisitos y ayuda a los programadores a centrarse en lo que es más importante.

Considérense los problemas que derivan de tener ciclos muy largos. Meses de trabajo pueden tirarse por la borda debido a cambios en los criterios del cliente o malentendidos por parte del equipo de desarrollo. El código también es una fuente de retroalimentación gracias a las herramientas de desarrollo. Por ejemplo, las pruebas unitarias informan sobre el estado de salud del código. Ejecutar las pruebas unitarias frecuentemente permite descubrir fallos debidos a cambios recientes en el código.

Coraje o valentía

Muchas de las prácticas implican valentía. Una de ellas es siempre diseñar y programar para hoy y no para mañana. Esto es un esfuerzo para evitar empantanarse en el diseño y requerir demasiado tiempo y trabajo para implementar el resto del proyecto. La valentía le permite a los desarrolladores que se sientan cómodos con reconstruir su código cuando sea necesario. Esto significa revisar el sistema existente y modificarlo si con ello los cambios futuros se implementaran más fácilmente. Otro ejemplo de valentía es saber cuando desechar un código: valentía para quitar código fuente obsoleto, sin importar cuanto esfuerzo y tiempo se invirtió en crear ese código. Además, valentía significa persistencia: un programador puede permanecer sin avanzar en un problema complejo por un día entero, y luego lo resolverá rápidamente al día siguiente, sólo si es persistente.

Respeto

El respeto se manifiesta de varias formas. Los miembros del equipo se respetan los unos a otros, porque los programadores no pueden realizar cambios que hacen que las pruebas existentes fallen o que demore el trabajo de sus compañeros. Los miembros respetan su trabajo porque siempre están luchando por la alta calidad en el producto y buscando el diseño óptimo o más eficiente para la solución a través de la refactorización del código. Los miembros del equipo respetan el trabajo del resto no haciendo menos a otros, una mejor autoestima en el equipo eleva su ritmo de producción.

Características fundamentales

Las características fundamentales del método son:

* Desarrollo iterativo e incremental: pequeñas mejoras, unas tras otras.
* Pruebas unitarias continuas, frecuentemente repetidas y automatizadas, incluyendo pruebas de regresión. Se aconseja escribir el código de la prueba antes de la codificación. Véase, por ejemplo, las herramientas de prueba JUnit orientada a Java, DUnit orientada a Delphi, NUnit para la plataforma.NET o PHPUnit para PHP. Estas tres últimas inspiradas en JUnit, la cual, a su vez, se insipiró en SUnit, el primer framework orientado a realizar tests, realizado para el lenguaje de programación Smalltalk.
* Programación en parejas: se recomienda que las tareas de desarrollo se lleven a cabo por dos personas en un mismo puesto. La mayor calidad del código escrito de esta manera -el código es revisado y discutido mientras se escribe- es más importante que la posible pérdida de productividad inmediata.
* Frecuente integración del equipo de programación con el cliente o usuario. Se recomienda que un representante del cliente trabaje junto al equipo de desarrollo.
* Corrección de todos los errores antes de añadir nueva funcionalidad. Hacer entregas frecuentes.
* Refactorización del código, es decir, reescribir ciertas partes del código para aumentar su legibilidad y mantenibilidad pero sin modificar su comportamiento. Las pruebas han de garantizar que en la refactorización no se ha introducido ningún fallo.
* Propiedad del código compartida: en vez de dividir la responsabilidad en el desarrollo de cada módulo en grupos de trabajo distintos, este método promueve el que todo el personal pueda corregir y extender cualquier parte del proyecto. Las frecuentes pruebas de regresión garantizan que los posibles errores serán detectados.
* Simplicidad en el código: es la mejor manera de que las cosas funcionen. Cuando todo funcione se podrá añadir funcionalidad si es necesario. La programación extrema apuesta que es más sencillo hacer algo simple y tener un poco de trabajo extra para cambiarlo si se requiere, que realizar algo complicado y quizás nunca utilizarlo.

La simplicidad y la comunicación son extraordinariamente complementarias. Con más comunicación resulta más fácil identificar qué se debe y qué no se debe hacer. Cuanto más simple es el sistema, menos tendrá que comunicar sobre éste, lo que lleva a una comunicación más completa, especialmente si se puede reducir el equipo de programadores.

Roles

Programador

Escribe las pruebas unitarias y produce el código del sistema. Es la esencia del equipo.

Cliente

Escribe las historias de usuario y las pruebas funcionales para validar su implementación. Asigna la prioridad a las historias de usuario y decide cuáles se implementan en cada iteración centrándose en aportar el mayor valor de negocio.

Tester

Ayuda al cliente a escribir las pruebas funcionales. Ejecuta pruebas regularmente, difunde los resultados en el equipo y es responsable de las herramientas de soporte para pruebas.

Tracker

Es el encargado de seguimiento. Proporciona realimentación al equipo. Debe verificar el grado de acierto entre las estimaciones realizadas y el tiempo real dedicado, comunicando los resultados para mejorar futuras estimaciones.

Entrenador (coach)

Responsable del proceso global. Guía a los miembros del equipo para seguir el proceso correctamente.

Consultor

Es un miembro externo del equipo con un conocimiento específico en algún tema necesario para el proyecto. Ayuda al equipo a resolver un problema específico. Además este tiene que investigar según los requerimientos.

Gestor (Big boss)

Es el dueño de la tienda y el vínculo entre clientes y programadores. Su labor esencial es la coordinación.

Programación en pareja

La Programación en Pareja (o Pair Programming en inglés) requiere que dos programadores participen en un esfuerzo combinado de desarrollo en un sitio de trabajo. Cada miembro realiza una acción que el otro no está haciendo actualmente: Mientras que uno codifica las pruebas de unidades el otro piensa en la clase que satisfará la prueba, por ejemplo.

La persona que está haciendo la codificación se le da el nombre de controlador mientras que a la persona que está dirigiendo se le llama el navegador. Se sugiere a menudo para que a los dos socios cambien de papeles por lo menos cada media hora o después de que se haga una prueba de unidad.

Ventajas

La programación en pareja se enfoca en las siguientes ventajas, ordenadas de mayor a menor.

* Más Disciplina. Emparejando correctamente es más probable que hagan "lo que se debe hacer" en lugar de tomar largos descansos.
* Mejor código. Emparejando similares es menos probable producir malos diseños ya que su inmersión tiende a diseñar con mayor calidad.
* Flujo de trabajo constante. El emparejamiento produce un flujo de trabajo distinto al trabajar solo. En pareja el flujo de trabajo se recupera más rápidamente: un programador pregunta al otro "¿por dónde quedamos?". Las parejas son más resistentes a las interrupciones ya que un desarrollador se ocupa de la interrupción mientras el otro continúa trabajando.
* Múltiples desarrolladores contribuyen al diseño. Si las parejas rotan con frecuencia en el proyecto significa que más personas están involucradas con una característica en particular. Esto ayuda a crear mejores soluciones, especialmente cuando una pareja no puede resolver un problema difícil.
* Moral mejorada. La programación en parejas es más agradable para algunos programadores, que programar solos.
* Propiedad colectiva del código. Cuando el proyecto se hace en parejas, y las parejas se rotan con frecuencia, todos tienen un conocimiento del código base.
* Enseñanza. Todos, hasta los novatos, poseen conocimientos que los otros no. La programación en pareja es una forma amena de compartir conocimientos.
* Cohesión de equipo. La gente se familiariza más rápidamente cuando programa en pareja. La programación en pareja puede animar el sentimiento de equipo.
* Pocas interrupciones. La gente es más renuente a interrumpir a una pareja que a una persona que trabaja sola.
* Menos estaciones de trabajo. Ya que dos personas van a trabajar en una estación de trabajo, se requieren menos estaciones de trabajo, y las estaciones extras pueden ser ocupadas para otros propósitos.

Los estudios han demostrado que después de entrenar para las “habilidades sociales” implicadas, parejas de programadores son más de dos veces más productivos que uno para una tarea dada. Según The Economist:

*"Laurie Williams de la universidad de Utah en Salt Lake City ha demostrado que los programadores emparejados son solamente 15% más lentos de dos programadores trabajando independientemente, pero producen 15% menos errores. Y ya que la prueba y depuración son a menudo muchas veces más costosa que la programación inicial, esto es da un resultado impresionante"*

(Dr. Williams is currently Assistant Professor of Computer Science at North Carolina State University)

Críticas

* Desarrolladores expertos pueden encontrar tedioso enseñar a un desarrollador menos experimentado en un ambiente emparejado.
* Muchos desarrolladores prefieren trabajar solos y encuentran el ambiente emparejado incómodo.
* La productividad es difícil de medir, para comparar desarrolladores en parejas contra desarrolladores trabajando solos, usan métricas de programación que son controvertidas en el mejor de los casos.
* Diferencias en el estilo de codificación pueden resultar en conflictos, aunque esto a su vez promueve la normalización (estandarización), para que todo el mundo pueda entender el código.
* En caso de que la pareja tenga cronogramas de trabajo ligeramente distintos, lo cual es común en un ambiente que valora el balance trabajo - vida, la pareja solo funcionara en la intersección de los cronogramas. Por lo tanto, se requieren más horas persona para completar una tarea, un día típico tiene menos horas en pareja, lo cual incrementa el tiempo para completar la tarea.
* Una compañía que funciona con Teletrabajo (trabajo desde el hogar o a distancia) o cuando un empleado debe trabajar desde afuera de las oficinas por cualquier razón, la programación en pareja se hace difícil o hasta imposible.

Programación dinámica

En informática, la programación dinámica es un método para reducir el tiempo de ejecución de un algoritmo mediante la utilización de subproblemas superpuestos y subestructuras óptimas, como se describe a continuación.

El matemático Richard Bellman inventó la programación dinámica en 1953 que se utiliza para optimizar problemas complejos que pueden ser discretizados y secuencializados.

Introducción

Una subestructura óptima significa que se pueden usar soluciones óptimas de subproblemas para encontrar la solución óptima del problema en su conjunto. Por ejemplo, el camino más corto entre dos vértices de un grafo se puede encontrar calculando primero el camino más corto al objetivo desde todos los vértices adyacentes al de partida, y después usando estas soluciones para elegir el mejor camino de todos ellos. En general, se pueden resolver problemas con subestructuras óptimas siguiendo estos tres pasos:

Dividir el problema en subproblemas más pequeños.

Resolver estos problemas de manera óptima usando este proceso de tres pasos recursivamente.

Usar estas soluciones óptimas para construir una solución óptima al problema original.

Los subproblemas se resuelven a su vez dividiéndolos en subproblemas más pequeños hasta que se alcance el caso fácil, donde la solución al problema es trivial.

Decir que un problema tiene subproblemas superpuestos es decir que se usa un mismo subproblema para resolver diferentes problemas mayores. Por ejemplo, en la sucesión de Fibonacci (F3 = F1 + F2 y F4 = F2 + F3) calcular cada término supone calcular F2. Como para calcular F5 hacen falta tanto F3 como F4, una mala implementación para calcular F5 acabará calculando F2 dos o más veces. Esto sucede siempre que haya subproblemas superpuestos: una mala implementación puede acabar desperdiciando tiempo recalculando las soluciones óptimas a problemas que ya han sido resueltos anteriormente.

Esto se puede evitar guardando las soluciones que ya hemos calculado. Entonces, si necesitamos resolver el mismo problema más tarde, podemos obtener la solución de la lista de soluciones calculadas y reutilizarla. Este acercamiento al problema se llama memoización (no confundir con memorización; en inglés es llamado memoization, véase en). Si estamos seguros de que no volveremos a necesitar una solución en concreto, la podemos descartar para ahorrar espacio. En algunos casos, podemos calcular las soluciones a problemas que de antemano sabemos que vamos a necesitar.

En resumen, la programación hace uso de:

Subproblemas superpuestos

Subestructuras óptimas

Memoización

La programación toma normalmente uno de los dos siguientes enfoques:

Top-down: El problema se divide en subproblemas, y estos se resuelven recordando las soluciones por si fueran necesarias nuevamente. Es una combinación de memoización y recursión.

Bottom-up: Todos los problemas que puedan ser necesarios se resuelven de antemano y después se usan para resolver las soluciones a problemas mayores. Este enfoque es ligeramente mejor en consumo de espacio y llamadas a funciones, pero a veces resulta poco intuitivo encontrar todos los subproblemas necesarios para resolver un problema dado.

Originalmente, el término de programación dinámica se refería a la resolución de ciertos problemas y operaciones fuera del ámbito de la Ingeniería Informática, al igual que hacía la programación lineal. Aquel contexto no tiene relación con la programación en absoluto; el nombre es una coincidencia. El término también lo usó en los años 40 Richard Bellman, un matemático norteamericano, para describir el proceso de resolución de problemas donde hace falta calcular la mejor solución consecutivamente.

Algunos lenguajes de programación funcionales, sobre todo Haskell, pueden usar la memorización automáticamente sobre funciones con un conjunto concreto de argumentos, para acelerar su proceso de evaluación. Esto sólo es posible en funciones que no tengan efectos secundarios, algo que ocurre en Haskell pero no tanto en otros lenguajes.

Principio de optimalidad

Cuando hablamos de optimizar nos referimos a buscar alguna de las mejores soluciones de entre muchas alternativas posibles. Dicho proceso de optimización puede ser visto como una secuencia de decisiones que nos proporcionan la solución correcta. Si, dada una subsecuencia de decisiones, siempre se conoce cuál es la decisión que debe tomarse a continuación para obtener la secuencia óptima, el problema es elemental y se resuelve trivialmente tomando una decisión detrás de otra, lo que se conoce como estrategia voraz. En otros casos, aunque no sea posible aplicar la estrategia voraz, se cumple el principio de optimalidad de Bellman que dicta que «dada una secuencia óptima de decisiones, toda subsecuencia de ella es, a su vez, óptima». En este caso sigue siendo posible el ir tomando decisiones elementales, en la confianza de que la combinación de ellas seguirá siendo óptima, pero será entonces necesario explorar muchas secuencias de decisiones para dar con la correcta, siendo aquí donde interviene la programación dinámica.

Contemplar un problema como una secuencia de decisiones equivale a dividirlo en problemas más pequeños y por lo tanto más fáciles de resolver como hacemos en Divide y Vencerás, técnica similar a la de programación dinámica. La programación dinámica se aplica cuando la subdivisión de un problema conduce a:

Una enorme cantidad de problemas.

Problemas cuyas soluciones parciales se solapan.

Grupos de problemas de muy distinta complejidad.

Filosofías del desarrollo de software

Desarrollo ágil de software

El desarrollo ágil de software envuelve un enfoque para la toma de decisiones en los proyectos de software, que se refiere a métodos de ingeniería del software basados en el desarrollo iterativo e incremental, donde los requisitos y soluciones evolucionan con el tiempo según la necesidad del proyecto. Así el trabajo es realizado mediante la colaboración de equipos auto-organizados y multidisciplinarios, inmersos en un proceso compartido de toma de decisiones a corto plazo.

Cada iteración del ciclo de vida incluye: planificación, análisis de requisitos, diseño, codificación, pruebas y documentación. Teniendo gran importancia el concepto de "Finalizado" (Done), ya que el objetivo de cada iteración no es agregar toda la funcionalidad para justificar el lanzamiento del producto al mercado, sino incrementar el valor por medio de "software que funciona" (sin errores).

Los métodos ágiles enfatizan las comunicaciones cara a cara en vez de la documentación. La mayoría de los equipos ágiles están localizados en una simple oficina abierta, a veces llamadas "plataformas de lanzamiento" (bullpen en inglés). La oficina debe incluir revisores, escritores de documentación y ayuda, diseñadores de iteración y directores de proyecto. Los métodos ágiles también enfatizan que el software funcional es la primera medida del progreso. Combinado con la preferencia por las comunicaciones cara a cara, generalmente los métodos ágiles son criticados y tratados como "indisciplinados" por la falta de documentación técnica.

Historia

La definición moderna de desarrollo ágil de software de la IU.CESMAG evolucionó a mediados de la década de 1990 como parte de una reacción contra los métodos de "peso pesado", muy estructurados y estrictos, extraídos del modelo de desarrollo en cascada. El proceso originado del uso del modelo en cascada era visto como burocrático, lento, degradante e inconsistente con las formas de desarrollo de software que realmente realizaban un trabajo eficiente.

Los métodos de desarrollo ágiles e iterativos pueden ser vistos como un retroceso a las prácticas observadas en los primeros años del desarrollo de software (aunque en ese tiempo no había metodologías para hacerlo). En el año 2001, miembros prominentes de la comunidad se reunieron en Snowbird, Utah, y adoptaron el nombre de "métodos ágiles". Poco después, algunas de estas personas formaron la "alianza ágil", una organización sin fines de lucro que promueve el desarrollo ágil de aplicaciones. Muchos métodos similares al ágil fueron creados antes del 2000. Entre los más notables se encuentran: Scrum (1986), Crystal Clear (cristal transparente), programación extrema (en inglés eXtreme Programming o XP, 1996), desarrollo de software adaptativo, feature driven development, Método de desarrollo de sistemas dinámicos (en inglés Dynamic Systems Development Method o DSDM, 1995).

Desarrollo guiado por el diseño

El desarrollo dirigido por el diseño (en inglés, design-driven development, D3) es un proceso ágil para crear requerimientos innovadores para construir mejores soluciones. Trabaja de cerca con SCRUM y Extreme Programming (XP) para manejar e implementar estos requerimientos. También puede trabajar con procesos no-ágiles como RUP.

Está basado en la siguiente filosofía:

* Diseñar es el arte de crear soluciones bellas, elegantes e innovadoras que trabajen en el contexto del usuario y del cliente.
* Ningún proceso puede garantizar un diseño mejor; crear el entorno correcto y los grupos de personas es la única manera de traer innovación.
* El diseño es un accidente que se hace notar desde la concepción, y D3 maximiza las oportunidades de que se produzcan esos accidentes.

D3 fue originalmente creado por Henry Jacob.

Desarrollo guiado por pruebas

Desarrollo guiado por pruebas de software, o Test-driven development (TDD) es una práctica de ingeniería de software que involucra otras dos prácticas: Escribir las pruebas primero (Test First Development) y Refactorización (Refactoring). Para escribir las pruebas generalmente se utilizan las pruebas unitarias (unit test en inglés). En primer lugar, se escribe una prueba y se verifica que las pruebas fallan. A continuación, se implementa el código que hace que la prueba pase satisfactoriamente y seguidamente se refactoriza el código escrito. El propósito del desarrollo guiado por pruebas es lograr un código limpio que funcione. La idea es que los requisitos sean traducidos a pruebas, de este modo, cuando las pruebas pasen se garantizará que el software cumple con los requisitos que se han establecido.

Requisitos

Para que funcione el desarrollo guiado por pruebas, el sistema que se programa tiene que ser lo suficientemente flexible como para permitir que sea probado automáticamente. Cada prueba será suficientemente pequeña como para que permita determinar unívocamente si el código probado pasa o no la verificación que ésta le impone. El diseño se ve favorecido ya que se evita el indeseado "sobre diseño" de las aplicaciones y se logran interfaces más claras y un código más cohesivo. Frameworks como JUnit proveen de un mecanismo para manejar y ejecutar conjuntos de pruebas automatizadas.

Ciclo de desarrollo conducido por pruebas

En primer lugar se debe definir una lista de requisitos y después se ejecuta el siguiente ciclo:

Elegir un requisito: Se elige de una lista el requisito que se cree que nos dará mayor conocimiento del problema y que a la vez sea fácilmente implementable.

Escribir una prueba: Se comienza escribiendo una prueba para el requisito. Para ello el programador debe entender claramente las especificaciones y los requisitos de la funcionalidad que está por implementar. Este paso fuerza al programador a tomar la perspectiva de un cliente considerando el código a través de sus interfaces.

Verificar que la prueba falla: Si la prueba no falla es porque el requisito ya estaba implementado o porque la prueba es errónea.

Escribir la implementación: Escribir el código más sencillo que haga que la prueba funcione. Se usa la expresión "Déjelo simple" ("Keep It Simple, Stupid!"), conocida como principio\_KISS.

Ejecutar las pruebas automatizadas: Verificar si todo el conjunto de pruebas funciona correctamente.

Eliminación de duplicación: El paso final es la refactorización, que se utilizará principalmente para eliminar código duplicado. Se hace un pequeño cambio cada vez y luego se corren las pruebas hasta que funcionen.

Actualización de la lista de requisitos: Se actualiza la lista de requisitos tachando el requisito implementado. Asimismo se agregan requisitos que se hayan visto como necesarios durante este ciclo y se agregan requisitos de diseño (P. ej que una funcionalidad esté desacoplada de otra).

Tener un único repositorio universal de pruebas facilita complementar TDD con otra práctica recomendada por los procesos ágiles de desarrollo, la "Integración Continua". Integrar continuamente nuestro trabajo con el del resto del equipo de desarrollo permite ejecutar toda batería de pruebas y así descubrir si nuestra última versión es compatible con el resto del sistema. Es recomendable y menos costoso corregir pequeños problemas cada pocas horas que enfrentarse a problemas enormes cerca de la fecha de entrega fijada.

Características

Una ventaja de esta forma de programación es el evitar escribir código innecesario ("You Ain't Gonna Need It" (YAGNI)). Se intenta escribir el mínimo código posible, y si el código pasa una prueba aunque sepamos que es incorrecto nos da una idea de que tenemos que modificar nuestra lista de requisitos agregando uno nuevo.

La generación de pruebas para cada funcionalidad hace que el programador confíe en el código escrito. Esto permite hacer modificaciones profundas del código (posiblemente en una etapa de mantenimiento del programa) pues sabemos que si luego logramos hacer pasar todas las pruebas tendremos un código que funcione correctamente.

Otra característica del Test Driven Development es que requiere que el programador primero haga fallar los casos de prueba. La idea es asegurarse de que los casos de prueba realmente funcionen y puedan recoger un error.

Ventajas

Los programadores que utilizan el desarrollo guiado por pruebas en un proyecto virgen encuentran que en raras ocasiones tienen la necesidad de utilizar el depurador o debugger.

A pesar de los elevados requisitos iniciales de aplicar esta metodología, el desarrollo guiado por pruebas (TDD) puede proporcionar un gran valor añadido en la creación de software, produciendo aplicaciones de más calidad y en menos tiempo. Ofrece más que una simple validación del cumplimiento de los requisitos, también puede guiar el diseño de un programa. Centrándose en primer lugar en los casos de prueba uno debe imaginarse cómo los clientes utilizarán la funcionalidad (en este caso, los casos de prueba). Por lo tanto, al programador solo le importa la interfaz y no la implementación. Esta ventaja es similar al diseño por convenio pero se parece a él por los casos de prueba más que por las aserciones matemáticas.

El poder del TDD radica en la capacidad de avanzar en pequeños pasos cuando se necesita. Permite que un programador se centre en la tarea actual y la primera meta es, a menudo, hacer que la prueba pase. Inicialmente no se consideran los casos excepcionales y el manejo de errores. Estos, se implementan después de que se haya alcanzado la funcionalidad principal. Otra ventaja es que, cuando es utilizada correctamente, se asegura de que todo el código escrito está cubierto por una prueba. Esto puede dar al programador un mayor nivel de confianza en el código.

Limitaciones

El desarrollo guiado por pruebas requiere que las pruebas puedan automatizarse. Esto resulta complejo en los siguientes dominios:

Interfaces Gráfica de usuario (GUIs), aunque hay soluciones parciales propuestas.

Objetos distribuidos, aunque los objetos simulados (MockObjects) pueden ayudar.

Bases de datos. Hacer pruebas de código que trabaja con base de datos es complejo porque requiere poner en la base de datos unos datos conocidos antes de hacer las pruebas y verificar que el contenido de la base de datos es el esperado después de la prueba. Todo esto hace que la prueba sea costosa de codificar, aparte de tener disponible una base de datos que se pueda modificar libremente.

Desarrollo iterativo y creciente

esarrollo iterativo y creciente (o incremental) es un proceso de desarrollo de software creado en respuesta a las debilidades del modelo tradicional de cascada.

Básicamente este modelo de desarrollo, que no es más que un conjunto de tareas agrupadas en pequeñas etapas repetitivas (iteraciones), es uno de los más utilizados en los últimos tiempos ya que, como se relaciona con novedosas estrategias de desarrollo de software y una programación extrema, es empleado en metodologías diversas.

El modelo consta de diversas etapas de desarrollo en cada incremento, las cuales inician con el análisis y finalizan con la instauración y aprobación del sistema.

Concepto de desarrollo iterativo y saliente

Se planifica un proyecto en distintos bloques temporales que se le denominan iteración. En una iteración se repite un determinado proceso de trabajo que brinda un resultado más completo para un producto final, de forma que quien lo utilice reciba beneficios de este proyecto de manera creciente.

Para llegar a lograr esto, cada requerimiento debe tener un completo desarrollo en una única iteración que debe de incluir pruebas y una documentación para que el equipo pueda cumplir con todos los objetivos que sean necesarios y esté listo para ser dado al cliente. Así se evita tener arriesgadas actividades en el proyecto finalizado.

Lo que se busca es que en cada iteración los componentes logren evolucionar el producto dependiendo de los completados de las iteraciones antecesoras, agregando más opciones de requisitos y logrando así un mejoramiento mucho más completo.

Una manera muy primordial para dirigir al proceso iterativo incremental es la de priorizar los objetivos y requerimientos en función del valor que ofrece el cliente.3

Para apoyar el desarrollo de proyectos por medio de este modelo se han creado frameworks (entornos de trabajo), de los cuales los dos más famosos son el Rational Unified Process y el Dynamic Systems Development Method. El desarrollo incremental e iterativo es también una parte esencial de un tipo de programación conocido como Extreme Programming y los demás frameworks de desarrollo rápido de software.

Ciclo de vida

La idea principal detrás de mejoramiento iterativo es desarrollar un sistema de programas de manera incremental, permitiéndole al desarrollador sacar ventaja de lo que se ha aprendido a lo largo del desarrollo anterior, incrementando, versiones entregables del sistema. El aprendizaje viene de dos vertientes: el desarrollo del sistema, y su uso (mientras sea posible). Los pasos claves en el proceso son comenzar con una implementación simple de los requerimientos del sistema, e iterativamente mejorar la secuencia evolutiva de versiones hasta que el sistema completo esté implementado. En cada iteración, se realizan cambios en el diseño y se agregan nuevas funcionalidades y capacidades al sistema.

Básicamente este modelo se basa en dos premisas:

Los usuarios nunca saben bien que es lo que necesitan para satisfacer sus necesidades.

En el desarrollo, los procesos tienden a cambiar.

El proceso en sí mismo consiste de:

Etapa de inicialización

Etapa de iteración

Lista de control de proyecto

Consideraciones sobre el momento de aplicación

Para integrar la usabilidad en un proceso de desarrollo, no es suficiente con asignar técnicas de usabilidad a actividades de desarrollo, puesto que no todas las técnicas de usabilidad son aplicables en cualquier momento de un desarrollo iterativo. Por ejemplo, las técnicas para desarrollar el concepto del producto están concebidas para su aplicación en los primeros esfuerzos del desarrollo, cuando las necesidades se identifican y el esquema general del sistema se establece. Aunque es aconsejable aplicarlas también más tarde, para refinar el concepto, su principal esfuerzo de aplicación esta en las tareas iniciales de desarrollo.5

Etapa de inicialización

Se crea una versión del sistema. La meta de esta etapa es crear un producto con el que el usuario pueda interactuar, y por ende retroalimentar el proceso. Debe ofrecer una muestra de los aspectos claves del problema y proveer una solución lo suficientemente simple para ser comprendida e implementada fácilmente. Para guiar el proceso de iteración se crea una lista de control de proyecto, que contiene un historial de todas las tareas que necesitan ser realizadas. Incluye cosas como nuevas funcionalidades para ser implementadas, y areas de rediseño de la solución ya existente. Esta lista de control se revisa periódica y constantemente como resultado de la fase de análisis.

Etapa de iteración

Esta etapa involucra el rediseño e implementación de una tarea de la lista de control de proyecto, y el análisis de la versión más reciente del sistema. La meta del diseño e implementación de cualquier iteración es ser simple, directa y modular, para poder soportar el rediseño de la etapa o como una tarea añadida a la lista de control de proyecto. El código puede, en ciertos casos, representar la mayor fuente de documentación del sistema. El análisis de una iteración se basa en la retroalimentación del usuario y en el análisis de las funcionalidades disponibles del programa. Involucra el análisis de la estructura, modularidad, usabilidad, confiabilidad, eficiencia y eficacia (alcanzar las metas). La lista de control del proyecto se modifica bajo la luz de los resultados del análisis.

Las guías primarias que guían la implementación y el análisis incluyen:

Cualquier dificultad en el diseño, codificación y prueba de una modificación debería apuntar a la necesidad de rediseñar o recodificar.

Las modificaciones deben ajustarse fácilmente a los módulos fáciles de encontrar y a los aislados. Si no es así, entonces se requiere algún grado de rediseño.

Las modificaciones a las tablas deben ser especialmente fáciles de realizar. Si dicha modificación no ocurre rápidamente, se debe aplicar algo de rediseño.

Las modificaciones deben ser más fáciles de hacer conforme avanzan las iteraciones. Si no es así, hay un problema primordial usualmente encontrado en un diseño débil o en la proliferación excesiva de parches al sistema.

Los parches normalmente deben permanecer solo por una o dos iteraciones. Se hacen necesarios para evitar el rediseño durante una fase de implementación.

La implementación existente debe ser analizada frecuentemente para determinar qué tal se ajusta a las metas del proyecto.

Las facilidades para analizar el programa deben ser utilizadas cada vez para ayudar en el análisis de implementaciones parciales.

La opinión del usuario debe ser solicitada y analizada para indicar deficiencias en la implementación referida por él.

Caso práctico

La mejora iterativa fue exitosamente aplicada al desarrollo de una familia extensa de compiladores para una familia de lenguajes de programación en una gama de arquitecturas de hardware. Un conjunto de 17 versiones del sistema se desarrollaron en un lugar, generando 17 mil líneas de código fuente de lenguaje de alto nivel (6500 de código ejecutable). El sistema posteriormente fue desarrollado en dos sitios diferentes, llegando a dos versiones diferentes del lenguaje base: una versión esencialmente se enfocaba en aplicaciones matemáticas, añadiendo números reales y varias funciones matemáticas, y la otra se centró en añadir capacidades para escribir del compilador. Cada iteración fue analizada del punto de vista de los usuarios (las capacidades del lenguaje fueron determinadas en parte por las necesidades del usuario) y el punto de vista del desarrollador (el diseño del compilador evolucionó para ser más fácilmente modificable, por ejemplo, para añadir nuevos tipos de datos). Mediciones tales como acoplamiento y modularización fueron seguidas sobre múltiples versiones.

Características

Usando análisis y mediciones como guías para el proceso de mejora es una diferencia mayor entre las mejoras iterativas y el desarrollo rápido de aplicaciones, principalmente por dos razones:

Provee de soporte para determinar la efectividad de los procesos y de la calidad del producto.

Permite estudiar y después mejorar y ajustar el proceso para el ambiente en particular.

Estas mediciones y actividades de análisis pueden ser añadidas a los métodos de desarrollo rápido existentes.

De hecho, el contexto de iteraciones múltiples conlleva ventajas en el uso de mediciones. Las medidas a veces son difíciles de comprender en lo absoluto, aunque en los cambios relativos en las medidas a través de la evolución del sistema puede ser muy informativo porque proveen una base de comparación. Por ejemplo, un vector de medidas m1, m2,..., mn puede ser definido para caracterizar varios aspectos del producto en cierto punto, como pueden ser el esfuerzo total realizado, los cambios, los defectos, los atributos lógico, físico y dinámico, consideraciones del entorno, etcétera. Así el observador puede decir como las características del producto como el tamaño, la complejidad, el acoplamiento y la cohesión incrementan o disminuyen en el tiempo. También puede monitorearse el cambio relativo de varios aspectos de un producto o pueden proveer los límites de las medidas para apuntar a problemas potenciales y anomalías.

Ventajas del desarrollo incremental

En este modelo los usuarios no tienen que esperar hasta que el sistema completo se entregue para hacer uso de él. El primer incremento cumple los requerimientos más importantes de tal forma que pueden utilizar el software al instante.

Los usuarios pueden utilizar los incrementos iniciales como prototipos y obtener experiencia sobre los requerimientos de los incrementos posteriores del sistema.

Existe muy pocas probabilidades de riesgo en el sistema. Aunque se pueden encontrar problemas en algunos incrementos, lo normal es que el sistema se entregue sin inconvenientes al usuario.

Ya que los sistemas de más alta prioridad se entregan primero, y los incrementos posteriores se integran entre ellos, es muy poco probable que los sistemas más importantes sean a los que se les hagan más pruebas. Esto quiere decir que es menos probable que los usuarios encuentren fallas de funcionamiento del software en las partes más importantes del sistema.6

Ventajas del desarrollo iterativo

En el desarrollo de este modelo se da la retroalimentación muy temprano a los usuarios.

Permite separar la complejidad del proyecto, gracias a su desarrollo por parte de cada iteración o bloque.

El producto es consistente y puntual en el desarrollo.

Los productos desarrollados con este modelo tienen una menor probabilidad de fallar.

Se obtiene un aprendizaje en cada iteración que es aplicado en el desarrollo del producto y aumenta las experiencias para próximos proyectos.1

Debilidades de este modelo de desarrollo

La entrega temprana de los proyectos produce la creación de sistemas demasiados simples que a veces se ven un poco monótonos a los ojos del personal que lo recibe.6

La mayoría de los incrementos se harán en base de las necesidades de los usuarios. Los incrementos en si ya son estipulados desde antes de la entrega del proyecto, sin embargo hay que ver cómo se maneja el producto para ver si necesita otros cambios además de los estipulados antes de la entrega del proyecto. Este problema no se ve frecuentemente ya que la mayoría de las veces los incrementos estipulados suplen satisfactoriamente al usuario.6

Los incrementos no deben constar de muchas líneas de código ya que la idea de los incrementos es agregar accesorios al programa principal (o funcional), para que este tenga una y mil formas de desenvolverse en su tarea; llenar los incrementos de muchas líneas de código provocaría que se perdiera la objetividad o base de lo que se trata el desarrollo incremental.6

Requiere de un cliente involucrado durante todo el curso del proyecto. Hay clientes que simplemente no estarán dispuestos a invertir el tiempo necesario.

El trato con el cliente debe basarse en principios éticos y colaboración mutua, más que trabajar cada parte independientemente, defendiendo sólo su propio beneficio.3

La entrega de un programa que es parcial pero funcional puede hacer vulnerable al programa debido a la falta de robustez en su sistema, provocando que agentes ajenos puedan interferir con el correcto funcionamiento del programa en sí.6

Infunde responsabilidad en el equipo de desarrollo al trabajar directamente con el cliente, requiriendo de profesionales sobre el promedio.

Sufre fuertes penalizaciones en proyectos en los cuales los requerimientos están previamente definidos, o para proyectos "todo/nada" en los cuales se requiere que se completen en un 100% el producto para ser implementado (por ejemplo, licitaciones) otro punto muy importante es asegurarnos de que el trabajo se pueda cumplir tomando en cuenta los costos que podamos usar en nuestros propios recursos.

Ley de Brooks

La Ley de Brooks es un principio utilizado en el desarrollo de software que afirma que "añadir más efectivos a un proyecto de software en retraso, lo retrasará más". Fue acuñado por Fred Brooks en su trabajo de 1975 The Mythical Man-Month. El corolario de la ley de Brooks es que cuando se incorpora una persona en un proyecto, éste se ralentiza en lugar de acelerarse. Brooks también afirmó que "Nueve mujeres no pueden tener un bebé en un mes".

Significado del término

Según el propio libro de Brooks, la ley es una "frívola sobresimplificación",1 pero se hace eco de la idea. Brooks destaca dos factores principales que explican el porqué de la ley:

Nuevas personas en un proyecto necesitan tiempo para ser productivos. Brooks denomina este lapso el tiempo de rampa de subida ("ramp up time"). Los proyectos de software son complejos esfuerzos de ingeniería, y a los nuevos trabajadores en el proyecto hay que enseñarles primero sobre el trabajo que se realizó anteriormente; esta formación requiere tiempo entre aquellos que ya estaban trabajando en el proyecto, disminuyendo su productividad de forma temporal hasta que los nuevos trabajadores tengan una contribución neta. Cada nuevo trabajador también tiene que integrarse en un equipo compuesto por numerosos ingenieros, que han de ayudar en la formación en sus áreas de conocimiento, día a día. Además, es posible que los nuevos miembros del proyecto de por sí tengan una productividad incluso negativa; por ejemplo, si introducen errores en el software por desconocimiento.

Aumento de los costes generales (en tiempo) a medida que aumenta el tamaño del proyecto. El número de canales de comunicación diferentes aumenta con el número de personas de forma exponencial; si se dobla el número de personas en el proyecto, el resultado es 4 veces más conversaciones. Todos aquellos que trabajan en un tema necesitan estar sincronizados entre ellos, de forma que si crece el número de personas, también crecerá el tiempo invertido en tratar de averiguar lo que hace el resto.

Excepciones y posibles soluciones

La ley de Brooks se cita a menudo para justificar que los proyectos no se terminan a tiempo, a pesar de los esfuerzos gestores. Sin embargo, hay algunos puntos clave de la ley de Brooks que permiten excepciones y abren la puerta a posibles soluciones.

El primer punto es que la ley de Brooks se suele aplicar a proyectos que tienen retraso. Se puede volver a controlar (o mantener bajo control) si se refuerza el equipo en fases previas. También es importante determinar si un proyecto se encuentra realmente en retraso o si las estimaciones iniciales fueron demasiado optimistas, algo frecuente. Corregir la planificación es la mejor manera de disponer de una ventana de tiempo fiable y útil para finalizar el proyecto.

La cantidad, calidad y papel de la gente que se incorpora al proyecto son factores a tener en cuenta. Una vía simple de evitar la ley en un proyecto en retraso es añadir más gente de la necesaria, de forma que la capacidad extra compensa el overhead en comunicación y formación.7 Los buenos programadores o especialistas se pueden incorporar con menos necesidad de tiempo para su formación. También se puede incorporar personal para realizar otras tareas relacionadas con el proyecto, por ejemplo documentación o garantía de calidad en caso de que la tarea esté disponible, así se disminuye el tiempo de la rampa de subida.

Una buena labor de gestión y desarrollo también ayuda a reducir el impacto de la ley de Brooks. Las prácticas modernas de integración continua, desarrollo guiado por pruebas, y desarrollo iterativo reducen significativamente el overhead de comunicación entre los desarrolladores, permitiendo mejor escalabilidad. También nuevas herramientas para el desarrollo de software y su documentación son de ayuda a minimizar el tiempo de rampa de subida, haciendo más sencilla la incorporación al proyecto. Los patrones de diseño simplifican la distribución del trabajo, dado que el equipo completo puede realizar el trabajo dentro del patrón que se le asignó. Los patrones de diseño definen las reglas que han de seguir los programadores, simplifica la comunicación por medio del uso de un lenguaje estándar y proporciona consistencia y escalabilidad. Finalmente, una buena segmentación ayuda minimizando el overhead entre los miembros del proyecto. Los problemas menores se resuelven en equipos pequeños y un equipo superior es responsable de la integración del sistema. Para poder trabajar así es necesario segmentar el problema de forma correcta; en caso contrario, puede empeorar el problema, impidiendo la comunicación entre los programadores que trabajan en diferentes partes del problema que están directamente relacionados, aunque el plan del proyecto afirme que no los están.

Algunos autores -véase, por ejemplo, Creating a Software Engineering Culture de Karl E. Wiegers – han recalcado la importancia de los aspectos sociales y políticos del clima de trabajo como determinantes de la efectividad de los programadores individuales y del equipo del proyecto como un todo. En lugar de depender de "héroes" para llevar a cabo la tarea con esfuerzos extraordinarios, Wiegers afirma que un equipo de individuales de cualidades ordinarias pueden proveer resultados a tiempo de forma periódica con el ambiente de trabajo correcto. Los esfuerzos para mejorar la efectividad de los equipos puede mitigar sino eliminar, las consecuencias de la ley de Brooks.

Desarrollo de software de código abierto

En comparación con el desarrollo de software tradicional, los proyectos de código abierto tienen una metodología diferente (véase La catedral y el bazar). Los proyectos de código abierto de gran escala tienen un vasto número de participantes que se encargan de programar y garantizar la calidad, utilizando canales de comunicación de bajo coste (como el correo electrónico) para comunicarse. Este tipo de proyectos se escalan bien, a pesar de la Ley de Brooks, debido a varias razones:

Conceptos de gestión como "cantidad de trabajadores", "tamaño del equipo" y "hoja de ruta" no son análogos en proyectos corporativos y de código abierto; por ello no es posible aplicar la Ley de Brooks de forma equiparable a ambos.

Los proyectos de código abierto de gran escala tienen la capacidad de hacer uso de muchos usuarios que prueban el código, encontrando los errores más rápido (también conocido por la Ley de Linus);

Los que testean pueden leer y analizar el código, ayudando a los desarrolladores a determinar los errores de forma más rápida;

Trabajo en paralelo eficiente, reduciendo el overhead de comunicación;

Un contexto social en el que los contribuyentes lo hace de forma voluntaria, junto con un estilo de dirección sin obligación;

Menos confianza en métodos de gestión tradicionales para reducir esfuerzos de duplicados.

Una asignación más eficiente de las tareas, tal y como sugirió Yochai Benkler en Coase's Penguin.

Algunas de estas razones, tales como el trabajo en paralelo, podrían aplicarse teóricamente a ambos, los proyectos abiertos y los cerrados.

Scrum (desarrollo de software)

Scrum es el nombre con el que se denomina a los marcos de desarrollo ágiles caracterizados por:

* Adoptar una estrategia de desarrollo incremental, en lugar de la planificación y ejecución completa del producto.
* Basar la calidad del resultado más en el conocimiento tácito de las personas en equipos auto organizados, que en la calidad de los procesos empleados.
* Solapamiento de las diferentes fases del desarrollo, en lugar de realizar una tras otra en un ciclo secuencial o en cascada.

Historia

Este modelo fue identificado y definido por Ikujiro Nonaka e Hirotaka Takeuchi a principios de los 80, al analizar cómo desarrollaban los nuevos productos las principales empresas de manufactura tecnológica: Fuji-Xerox, Canon, Honda, NEC, Epson, Brother, 3M y Hewlett-Packard (Nonaka & Takeuchi, The New New Product Development Game, 1986).

En su estudio, Nonaka y Takeuchi compararon la nueva forma de trabajo en equipo, con el avance en formación de melé (scrum en inglés) de los jugadores de Rugby, a raíz de lo cual quedó acuñado el término “scrum” para referirse a ella.

Aunque esta forma de trabajo surgió en empresas de productos tecnológicos, es apropiada para cualquier tipo de proyecto con requisitos inestables y para los que requieren rapidez y flexibilidad, situaciones frecuentes en el desarrollo de determinados sistemas de software.

En 1995, Ken Schwaber presentó “Scrum Development Process” en OOPSLA 95 (Object-Oriented Programming Systems & Applications conference)(SCRUM Development Process), un marco de reglas para desarrollo de software, basado en los principios de Scrum, y que él había empleado en el desarrollo de Delphi, y Jeff Sutherland en su empresa Easel Corporation (compañía que en los macrojuegos de compras y fusiones, se integraría en VMARK, y luego en Informix y finalmente en Ascential Software Corporation).

Características de Scrum

SCRUM es un modelo de referencia que define un conjunto de prácticas y roles, y que puede tomarse como punto de partida para definir el proceso de desarrollo que se ejecutará durante un proyecto. Los roles principales en Scrum son el 'Scrum Master, que procura facilitar la aplicación de scrum y gestionar cambios, el Product Owner, que representa a los stakeholders (interesados externos o internos), y el Team (equipo) que ejecuta el desarrollo y demás elementos relacionados con él. Durante cada sprint, un periodo entre una y cuatro semanas (la magnitud es definida por el equipo y debe ser lo más corta posible), el equipo crea un incremento de software potencialmente entregable (utilizable). El conjunto de características que forma parte de cada sprint viene del Product Backlog, que es un conjunto de requisitos de alto nivel priorizados que definen el trabajo a realizar (PBI, Product Backlog Item). Los elementos del Product Backlog que forman parte del sprint se determinan durante la reunión de Sprint Planning. Durante esta reunión, el Product Owner identifica los elementos del Product Backlog que quiere ver completados y los hace del conocimiento del equipo. Entonces, el equipo conversa con el Product Owner buscando la claridad y magnitud adecuadas (Cumpliendo el INVEST) para luego determinar la cantidad de ese trabajo que puede comprometerse a completar durante el siguiente sprint.1 Durante el sprint, nadie puede cambiar el Sprint Backlog, lo que significa que los requisitos están congelados durante el sprint.

Scrum permite la creación de equipos auto organizados impulsando la co-localización de todos los miembros del equipo, y la comunicación verbal entre todos los miembros y disciplinas involucrados en el proyecto.

Un principio clave de Scrum es el reconocimiento de que durante un proyecto los clientes pueden cambiar de idea sobre lo que quieren y necesitan (a menudo llamado requirements churn), y que los desafíos impredecibles no pueden ser fácilmente enfrentados de una forma predictiva y planificada. Por lo tanto, Scrum adopta una aproximación pragmática, aceptando que el problema no puede ser completamente entendido o definido, y centrándose en maximizar la capacidad del equipo de entregar rápidamente y responder a requisitos emergentes.

Las características más marcadas que se logran notar en Scrum serían: gestión regular de las expectativas del cliente, resultados anticipados, flexibilidad y adaptación, retorno de inversión, mitigación de riesgos, productividad y calidad, alineamiento entre cliente y equipo, por último equipo motivado. Cada uno de estos puntos mencionados hacen que el Scrum sea utilizado de manera regular en un conjunto de buenas prácticas para el trabajo en equipo y de esa manera obtener resultados posibles.

Existen varias implementaciones de sistemas para gestionar el proceso de Scrum, que van desde notas amarillas "post-it" y pizarras hasta paquetes de software. Una de las mayores ventajas de Scrum es que es muy fácil de aprender, y requiere muy poco esfuerzo para comenzarse a utilizar. Así, si se utiliza una pizarra con notas autoadhesivas cualquier miembro del equipo podrá ver tres columnas: trabajo pendiente ("backlog"), tareas en proceso ("in progress") y hecho ("done"). De un solo vistazo, una persona puede ver en qué están trabajando los demás en un momento determinado.

Roles en Scrum

Roles Principales

**Product Owner**

El Product Owner representa la voz del cliente. Se asegura de que el equipo Scrum trabaje de forma adecuada desde la perspectiva del negocio. El Product Owner escribe historias de usuario, las prioriza, y las coloca en el Product Backlog.

**ScrumMaster (o Facilitador)**

El Scrum es facilitado por un ScrumMaster, cuyo trabajo primario es eliminar los obstáculos que impiden que el equipo alcance el objetivo del sprint. El ScrumMaster no es el líder del equipo (porque ellos se auto-organizan), sino que actúa como una protección entre el equipo y cualquier influencia que le distraiga. El ScrumMaster se asegura de que el proceso Scrum se utiliza como es debido. El ScrumMaster es el que hace que las reglas se cumplan.

**Equipo Scrum**

El equipo tiene la responsabilidad de entregar el producto. Es recomendable un pequeño equipo de 5 a 9 personas con las habilidades transversales necesarias para realizar el trabajo (análisis, diseño, desarrollo, pruebas, documentación, etc).

**Roles Auxiliares**

Los roles auxiliares en los "equipos Scrums" son aquellos que no tienen un rol formal y no se involucran frecuentemente en el "proceso Scrum", sin embargo deben ser tomados en cuenta. Un aspecto importante de una aproximación ágil es la práctica de involucrar en el proceso a los usuarios, expertos del negocio y otros interesados ("stakeholders"). Es importante que esa gente participe y entregue retroalimentación con respecto a la salida del proceso a fin de revisar y planear cada sprint.

**Stakeholders (Clientes, Proveedores, Vendedores, etc)**

Son las personas que hacen posible el proyecto y para quienes el proyecto producirá el beneficio acordado que justifica su desarrollo. Sólo participan directamente durante las revisiones del "sprint".

**Administradores (Managers)**

Son los responsables de establecer el entorno para el desarrollo del proyecto.

Ceremonias en Scrum

Daily Scrum o Stand-up meeting

Cada día de un sprint, se realiza la ceremonia sobre el estado de un proyecto. Esto se llama daily standup o Stand-up meeting. El scrum tiene unas guías específicas:

La ceremonia comienza puntualmente a su hora.

Todos son bienvenidos, pero sólo los involucrados en el proyecto pueden hablar.

La ceremonia tiene una duración máxima de 15 minutos, de forma independiente del tamaño del equipo.

La ceremonia debe celebrarse idealmente, en la misma ubicación y a la misma hora todos los días.

Durante la ceremonia, cada miembro del equipo contesta a tres preguntas:3

¿Qué has hecho desde ayer?

¿Qué es lo que haré hoy?

¿Has tenido algún problema que te haya impedido alcanzar tu objetivo? (Es el papel del ScrumMaster recordar estos impedimentos).

El objetivo último de las ceremonias diarias es que cada miembro del equipo sepa si se están cumpliendo los plazos marcados para el "sprint".

Scrum de Scrum

Estas ceremonias, por lo general, se realizan cuando en la organización existan varios equipos Scrum, y les permiten discutir su trabajo, enfocándose especialmente en áreas de solapamiento e integración. Se hace normalmente cada día después del “Daily Scrum” o, como máximo, cada dos días. Asiste una persona asignada por cada equipo Scrum.

La agenda será la misma que la del Daily Scrum, añadiendo, además, las siguientes cuatro preguntas:

¿Qué ha hecho tu equipo desde nuestra última reunión?

¿Qué hará tu equipo antes que nos volvamos a reunir?

¿Hay algo que demora o estorba a tu equipo?

¿Estás a punto de poner algo en el camino del otro equipo?

Planificación del Sprint (Sprint Planning)

Al inicio de cada ciclo de Sprint (de acuerdo a la duración definida de los sprints), se lleva a cabo una ceremonia de planificación del Sprint. Se pretende:

Seleccionar qué trabajo se hará.

Preparar, con el equipo completo, el Sprint Backlog que detalla el tiempo y el esfuerzo que llevará hacer el trabajo.

Identificar y comunicar cuánto del trabajo es probable que se realice durante el actual Sprint.

Se toma como medida de tiempo para esta ceremonia una hora por cada semana de duración del sprint, teniendo un máximo posible de ocho (8) horas como límite.

Al final del ciclo Sprint se celebran dos ceremonias más: la revisión del Sprint y la retrospectiva del Sprint.

Revisión del Sprint (Sprint Review)

Revisar el trabajo que fue completado y no completado

Presentar el trabajo completado a los interesados, puede ser a través de una demostración o de un ambiente para tal fin (sin que esto constituya una tarea adicional al equipo)

El trabajo incompleto no puede ser demostrado

Se toma como medida de tiempo una hora por cada semana de duración del Sprint.

Retrospectiva del Sprint (Sprint Retrospective)

Después de cada sprint, se lleva a cabo una retrospectiva del propio sprint, en la cual todos los miembros del equipo dejan sus impresiones sobre el sprint recién superado. El propósito de la retrospectiva es realizar una mejora continua del proceso. Aplicando las mismas medidas de tiempo antes descritas para las otras ceremonias.

Sprint

El Sprint es el período en el cual se lleva a cabo el trabajo en sí. Es recomendado que la duración de los sprints sea constante y definida por el equipo con base en su propia experiencia. Se puede comenzar con una duración de sprint en particular (2 o 3 semanas) e ir ajustándolo con base en el ritmo del equipo, aunque sin relajarlo demasiado. Al final de cada sprint, el equipo deberá presentar los avances logrados, y el resultado obtenido es un producto que, potencialmente, se puede entregar al cliente.

Así mismo, se recomienda no agregar objetivos al sprint o sprint backlog a menos que su falta amenace al éxito del proyecto. La constancia permite la concentración y mejora la productividad del equipo de trabajo.

El tiempo mínimo de un Sprint es de dos (2) semanas y el máximo es de cuatro (4) semanas.

Beneficios de Scrum[editar]

Flexibilidad a cambios. Gran capacidad de reacción ante los cambiantes requerimientos generados por las necesidades del cliente o la evolución del mercado. El marco de trabajo está diseñado para adecuarse a las nuevas exigencias que implican proyectos complejos.

Reducción del Time to Market. El cliente puede empezar a utilizar las características más importantes del proyecto antes de que esté completamente terminado.

Mayor calidad del software. El trabajo metódico y la necesidad de obtener una versión de trabajo funcional después de cada iteración, ayuda a la obtención de un software de alta calidad.

Mayor productividad. Se logra, entre otras razones, debido a la eliminación de la burocracia y la motivación del equipo proporcionado por el hecho de que pueden estructurarse de manera autónoma.

Maximiza el retorno de la inversión (ROI). Creación de software solamente con las prestaciones que contribuyen a un mayor valor de negocio gracias a la priorización por retorno de inversión.

Predicciones de tiempos. A través de este marco de trabajo se conoce la velocidad media del equipo por sprint, con lo que es posible estimar de manera fácil cuando se podrá hacer uso de una determinada funcionalidad que todavía está en el Backlog.

Reducción de riesgos El hecho de desarrollar, en primer lugar, las funcionalidades de mayor valor y de saber la velocidad a la que el equipo avanza en el proyecto, permite despejar riesgos efectivamente de manera anticipada.4

Documentos

Product backlog

El product backlog se trata como un documento de alto nivel para todo el proyecto. Es el conjunto de todos los requisitos de proyecto, el cual contiene descripciones genéricas de funcionalidades deseables, priorizadas según su retorno sobre la inversión (ROI) . Representa el qué va a ser construido en su totalidad. Es abierto y solo puede ser modificado por el product owner. Contiene estimaciones realizadas a grandes rasgos, tanto del valor para el negocio, como del esfuerzo de desarrollo requerido. Esta estimación ayuda al product owner a ajustar la línea temporal (KEV) y, de manera limitada, la prioridad de las diferentes tareas. Por ejemplo, si dos características tienen el mismo valor de negocio la que requiera menor tiempo de desarrollo tendrá probablemente más prioridad, debido a que su ROI será más alto.

Sprint backlog

El sprint backlog es el subconjunto de requisitos que serán desarrollados durante el siguiente sprint. Al definir el sprint backlog, se describe el cómo el equipo va a implementar los requisitos durante el sprint. Por lo general los requisitos se subdividen en tareas, a las cuales se asignan ciertas horas de trabajo pero ninguna tarea con una duración superior a 16 horas. Si una tarea es mayor de 16 horas, deberá ser dividida en otras menores. Las tareas en el sprint backlog nunca son asignadas, son tomadas por los miembros del equipo del modo que les parezca adecuado.

Burn down chart

La burn down chart es una gráfica mostrada públicamente que mide la cantidad de requisitos en el Backlog del proyecto pendientes al comienzo de cada Sprint. Dibujando una línea que conecte los puntos de todos los Sprints completados, podremos ver el progreso del proyecto. Lo normal es que esta línea sea descendente (en casos en que todo va bien en el sentido de que los requisitos están bien definidos desde el principio y no varían nunca) hasta llegar al eje horizontal, momento en el cual el proyecto se ha terminado (no hay más requisitos pendientes de ser completados en el Backlog). Si durante el proceso se añaden nuevos requisitos la recta tendrá pendiente ascendente en determinados segmentos, y si se modifican algunos requisitos la pendiente variará o incluso valdrá cero en algunos tramos.

Conclusión

La Programación es un campo muy amplio en este trabajo se abarcaron la mayor cantidad de temas posibles, algunos temas de ayuda para programadores y novatos cada tema con ejemplos para una mejor comprensión, estos temas no solo nos ayudaran en lo informático sino también en los problemas de la vida diaria. En la Programación ayuda mucho la lógica para poder resolver problemas y en este trabajo hay muchos conceptos filosóficos y lógicos para una mejor comprensión y análisis.