Esta obra está bajo una licencia de Creative Commons.
Autor: Jorge Sánchez Asenjo (año 2008) http://www.jorgesanchez.net
e-mail:info@jorgesanchez.net

Esta obra está bajo una licencia de Reconocimiento-NoComercial-CompartirIgual de CreativeCommons
Para ver una copia de esta licencia, visite:
http://creativecommons.org/licenses/by-nc-sa/2.5/es/legalcode.es
o envíe una carta a:
Creative Commons, 559 Nathan Abbot

Reconocimiento-NoComercial-CompartirIgual 2.5 España

Usted es libre de:

copiar, distribuir y comunicar públicamente la obra

hacer obras derivadas

Bajo las condiciones siguientes:

Reconocimiento. Debe reconocer los créditos de la obra de la manera especificada por el autor o el licenciador (pero no de una manera que sugiera que tiene su apoyo o apoyan el uso que hace de su obra).

No comercial. No puede utilizar esta obra para fines comerciales.

Compartir bajo la misma licencia. Si altera o transforma esta obra, o genera una obra derivada, sólo puede distribuir la obra generada bajo una licencia idéntica a ésta.

- Al reutilizar o distribuir la obra, tiene que dejar bien claro los términos de la licencia de esta obra.
- Alguna de estas condiciones puede no aplicarse si se obtiene el permiso del titular de los derechos de autor
- Apart from the remix rights granted under this license, nothing in this license impairs or restricts the author's moral rights.

Advertencia

Los derechos derivados de usos legítimos u otras limitaciones reconocidas por ley no se ven afectados por lo anterior.
Esto es un resumen legible por humanos del texto legal (la licencia completa) disponible en los idiomas siguientes:
Catalán Castellano Euskera Gallego

Para ver una copia completa de la licencia, acudir a la dirección http://creativecommons.org/licenses/by-nc-sa/2.5/es/legalcode.es

(2) metodología de la programación

esquema de la unidad

(2.1) esquema de la unidad ____ 5
(2.2) metodologías ____ 6
(2.2.1) introducción 6
(2.2.2) fases de las metodologías 6
(2.2.3) prototipos 7
(2.3) fase de análisis ____ 10
(2.4) diseño ____ 12
(2.5) notaciones para el diseño de algoritmos ____ 13
(2.5.1) diagramas de flujo 13
(2.5.2) pseudocódigo 15
(2.6) UML ____ 32
(2.6.1) introducción 32
(2.6.2) diagramas UML 33
(2.6.3) diagrama de casos de uso 34
(2.6.4) diagramas de actividad 36
(2.7) índice de ilustraciones ____ 38

(2.1) metodologías

(2.1.1) introducción

Como ya se comentó en el tema anterior, la aplicación pasa por una serie de pasos relacionados con el ciclo de vida de la aplicación. En el tema anterior se indicaron los siguientes pasos:

(1) Análisis

(2) Diseño

(3) Codificación o implementación

(4) Prueba

(5) Mantenimiento

Esos pasos imponen un método de trabajo, lo que comúnmente se conoce como **metodología**. Las metodologías definen los pasos y tareas que hay que realizar para realizar un determinado proyecto. Aplicadas a la informática definen la forma exacta de desarrollar el ciclo de vida de una aplicación.

Una metodología marca las forma de realizar todas las fases de creación de un proyecto informático; en especial las relacionadas con el análisis y diseño.

Las metodologías marcan los siguientes elementos:

- **Pasos exactos a realizar en la creación de la aplicación**. Se indica de forma exacta qué fases y en qué momento se realiza cada una.
- **Protocolo**. Normas y reglas a seguir escrupulosamente en la realización de la documentación y comunicación en cada fase.
- **Recursos humanos**. Personas encargadas de cada fase y responsables de las mismas.

La metodología a elegir por parte de los desarrolladores es tan importante que algunas metodologías son de pago, pertenecen a empresas que poseen sus derechos. En muchos casos dichas empresas fabrican las herramientas (**CASE**) que permiten gestionar de forma informática el seguimiento del proyecto mediante la metodología empleada.

(2.1.2) fases en las metodologías

Todas las metodologías imponen una serie de pasos o fases a realizar en el proceso de creación de aplicaciones; lo que ocurre es que estos pasos varían de unas a otras. En general los pasos siguen el esquema inicial planteado en el punto anterior, pero varían los pasos para recoger aspectos que se consideran que mejoran el proceso. La mayoría de metodologías se diferencian fundamentalmente en la parte del diseño. De hecho algunas se refieren sólo a esa fase (como las famosas metodologías de **Jackson** o **Warnier** por ejemplo).

Por ejemplo la **metodología OMT** de **Rumbaugh** propone estas fases:

- **Análisis**
- **Diseño del sistema**
- **Diseño de objetos**
- **Implementación**

Naturalmente con esto no se completa el ciclo de vida, para el resto sería necesario completar el ciclo con otro método.

En la metodología **MERISE** desarrollada para el gobierno francés (y muy utilizada de forma docente en las universidades españolas), se realizan estas fases:

- **Esquema Director**
- **Estudio Previo**
- **Estudio Detallado**
- **Estudio Técnico**
- **Realización**
- **Mantenimiento**

Otra propuesta de método indicada en varios libros:

- **Investigación Preliminar**
- **Determinación de Requerimientos.**
- **Diseño del Sistema**
- **Desarrollo del Software**
- **Prueba del Sistema**
- **Implantación y Evaluación**

En definitiva las fases son distintas pero a la vez muy similares ya que el proceso de creación de aplicaciones siempre tiene pasos por los que todas las metodologías han de pasar.

(2.1.3) prototipos

Independientemente de la metodología utilizada, siempre ha habido un problema al utilizar metodologías. El problema consiste en que desde la fase de análisis, el cliente ve la aplicación hasta que esté terminada, independientemente de si el fallo está en una fase temprana o no.

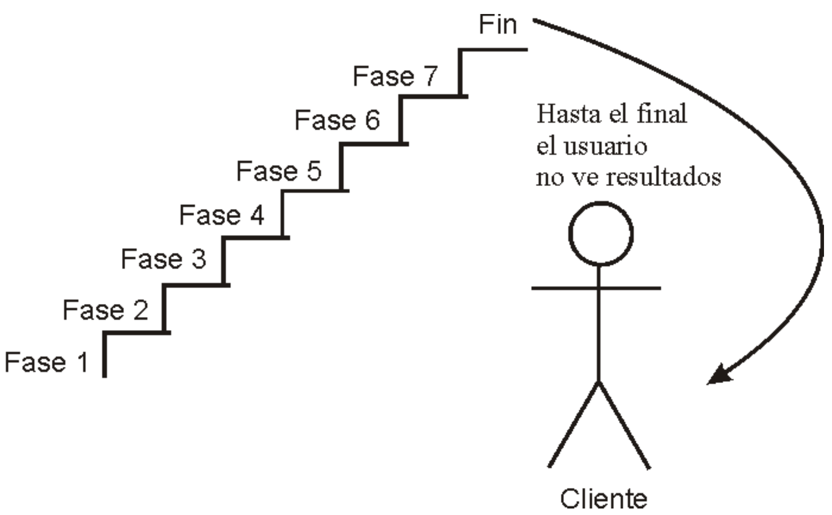


Ilustración 2-1, Fases clásicas de desarrollo de una aplicación

Esta forma de trabajo tiene el problema de que al usuario le puede llegar una aplicación distinta de la que deseaba de forma bastante habitual. El arreglo es complicado porque no sabemos en qué fase está el problema.

De ahí que apareciera el concepto de **prototipo**. Un prototipo es una forma de ver cómo va a quedar la aplicación antes de terminarla (como la maqueta de un edificio). Se suelen realizar varios prototipos; cuánto más alta sea la fase en la que se realiza, más parecido tendrá con la aplicación final.

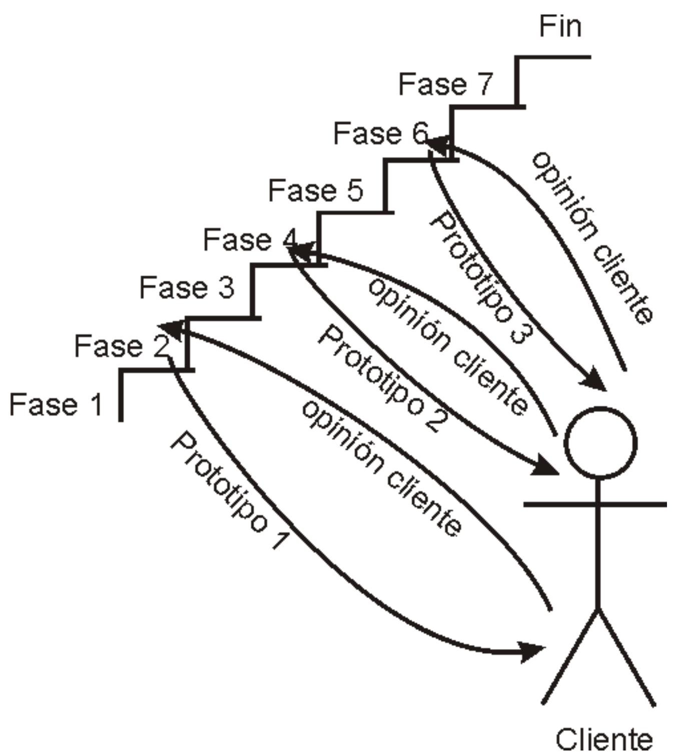


Ilustración 2-2, Prototipos en las metodologías modernas

En esta otra forma de trabajar, el cliente participa en todas las fases.

(2.1.4) metodologías iterativas

Uno de los problemas del ciclo de vida clásico consiste en que los problemas a veces se detectan en las fases finales del proyecto. De ser así hay que regresar a las primeras fases y esto es problemático.

Por eso se idearon metodologías que no fueran **lineales** (las metodologías lineales son las que cuando se acaba una fase, se da por concluida hasta la fase de mantenimiento).

Un ejemplo de ellas es por ejemplo las **metodologías en espiral**:

Ilustración 2-3, Ejemplo de desarrollo en espiral

En este tipo de metodologías, cuando se completa el ciclo, se vuelve a empezar tras analizar el cumplimiento de objetivos. Así se va refinando el sistema en cada vuelta. Aunque nunca se conseguirá el sistema óptimo, cada vuelta consigue un sistema mejor (él óptimo sería el centro de la espiral).

Otro modelo (utilizado en el **proceso unificado** del que hablaremos más adelante) es el iterativo. Con este método se realizan varias iteraciones del ciclo de vida (al estilo del modelo en espiral) de modo que las primeras iteraciones dedican más tiempo a las primeras fases del ciclo de vida y las últimas más a las últimas. Así cuando se entrega el sistema, ha sido depurado lo suficiente para que el producto tenga calidad.

Ejemplo:

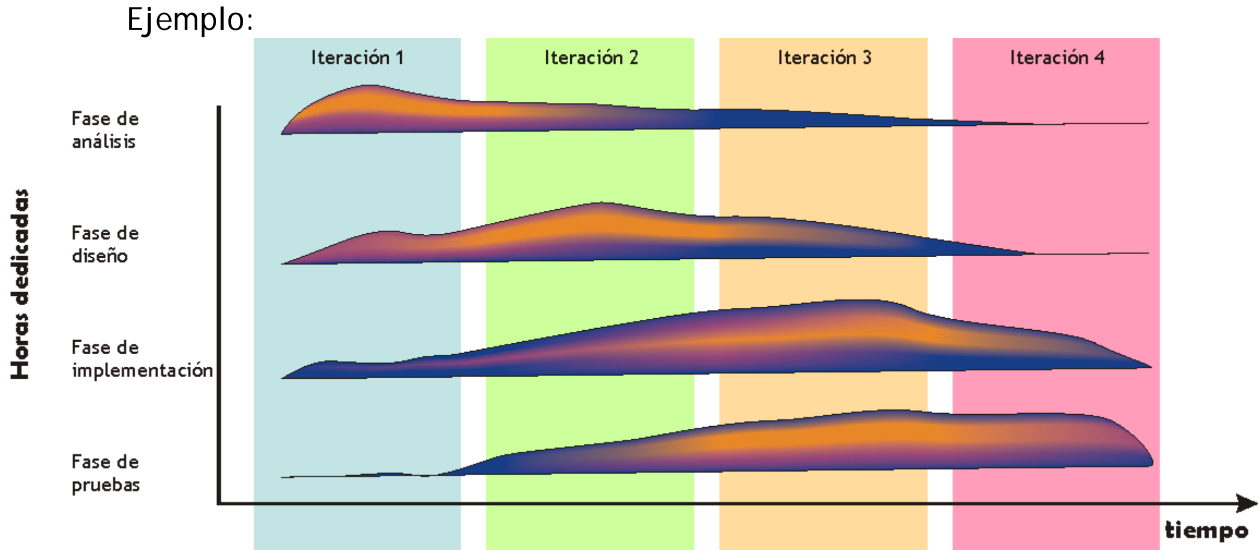


Ilustración 2-4, Metodología iterativa

(2.1.5) recursos humanos que intervienen en el ciclo de vida

Como se ha comentado antes, las metodologías no sólo indican los pasos y protocolos, sino también las personas que los realizan. Por tanto las metodologías imponen una serie de profesionales de las sistemas informáticos que intervienen en las distintas fases durante el desarrollo del software.

Evidentemente el personal depende de la empresa en sí. Las grandes empresas productores de software, como Adobe, Google o Microsoft por ejemplo, no tienen la misma estructura que los departamentos de informática de empresas dedicadas a otro fin o a los pequeños productores de software.

En cualquier caso suele haber los siguientes profesionales:

- **Directivos**. Desde al presidente de la empresa al gerente de informática, todos ellos son encargados de la gestión de la empresa. No tienen por qué tener conocimientos informáticos y sí de gestión empresarial. Su labor es coordinar todos los recursos necesarios para que funcione el proceso correctamente.
- **Analista de sistemas**. Responsable del proyecto y director de las fases de desarrollo. Tiene conocimientos avanzados de programación, análisis y diseño y además habilidades comunicativas. Es el encargado de realizar las entrevistas durante la fase de análisis
- **Analistas-programadores**. Se tratan de los profesionales que realizan la fase de diseño. Tienen conocimientos avanzados de programación, pero fundamentalmente realizan las fases de análisis y diseño a las órdenes de los anteriores.
- **Programadores**. Se trata de los profesionales encargados de realizar la fase de implementación a partir de los diseños de los analistas

- **Especialistas**. Se trata de profesionales conocedores de software y hardware importante en el proceso de desarrollo pero muy específico. Entre ellos están:
    - **Administradores de bases de datos**. Encargados de gestionar el sistema de base de datos que utilice la aplicación desarrollada
    - **Administradores de red**. Encargados de gestionar los recursos de red local para un correcto funcionamiento
    - **Programadores de sistemas**. Encargados del mantenimiento del Sistema Operativo y software necesario para el correcto funcionamiento de la aplicación.
- **Operadores**. Al servicio de los especialistas o bien del mantenimiento de la aplicación, especialmente en el apoyo al usuario.

(2.2) fase de análisis

Al programar aplicaciones siempre se debe realizar un análisis. El análisis estudia los requisitos que ha de cumplir la aplicación. El resultado del análisis es una **hoja de requerimientos** en la que aparecen las necesidades a cubrir por la aplicación. La habilidad para obtener correctamente la información necesaria de esta fase la realizan los/as **analistas**. La fase se cumple tras ser aprobada por el o la responsable del proyecto (normalmente un **jefe o jefa de proyecto**).

El análisis se suele dividir en estas subfases:

(1) **Estudio preliminar**. En el que se estudia la viabilidad del proyecto. Parte de una información general (la indicada por el cliente inicialmente). En ella se estiman los recursos (tanto humanos como materiales) necesarios y los costes.

(2) **Obtención de información**. Una vez aprobado el estudio preliminar, los y las analistas realizan entrevistas a los futuros usuarios para recabar la información imprescindible para realizar el proyecto.

(3) **Definición del problema**. Las entrevistas dan lugar a un primer estudio detallado del problema en el que se afina más el estudio preliminar a fin de determinar más exactamente los recursos y costes necesarios y las fases que se necesitarán.

(4) **Determinación de los requerimientos**. Se estudian los requerimientos de la aplicación a partir de los datos obtenidos de las fases anteriores.

(5) **Redacción de las hojas de requerimientos**. Se redactan las hojas de requerimientos en base al protocolo a utilizar.

(6) **Aprobación de las hojas**. Los jefes y jefas de proyecto se encargan de revisar y aprobar las hojas, a fin de proceder con las siguientes fases. En caso de no aprobar, habrá que corregir los errores o volver a alguna de las fases anteriores.

(7) **Selección y aprobación del modelo funcional.** O lo que es lo mismo, selección del modelo a utilizar en la fase de diseño. Se elegirá el idóneo en base al problema concreto que tengamos.

Como se comentó en el punto anterior, es habitual (y desde luego muy recomendable) el uso de prototipos. En esta fase bastaría con revisar los requerimientos con el usuario antes de aprobar las hojas de requerimientos.

(2.3) diseño

En esta fase se crean esquemas que simbolizan a la aplicación. Estos esquemas los elaboran analistas. Gracias a estos esquemas se facilita la implementación de la aplicación. Estos esquemas en definitiva se convierte en la documentación fundamental para plasmar en papel lo que el programador debe hacer.

En estos esquemas se pueden simbolizar: la organización de los datos de la aplicación, el orden de los procesos que tiene que realizar la aplicación, la estructura física (en cuanto a archivos y carpetas) que utilizará la aplicación, etc.

Cuanto más potentes sean los esquemas utilizados (que dependerán del modelo funcional utilizado), más mecánica y fácil será la fase de implementación.

La creación de estos esquemas se puede hacer en papel (raramente), o utilizar una **herramienta CASE** para hacerlo. Las herramientas CASE (*Computer Aided Software Engineering*, **Ingeniería de Software Asistida por Ordenador**) son herramientas software pensadas para facilitar la realización de la fase de diseño en la creación de una aplicación. Con ellas se realizan todos los esquemas necesarios en la fase de diseño e incluso son capaces de escribir el código equivalente al esquema diseñado.

En el caso de la creación de algoritmos, conviene en esta fase usar el llamado **diseño descendente**. Mediante este diseño el problema se divide en módulos, que, a su vez, se vuelven a dividir a fin de solucionar problemas más concretos. Al diseño descendente se le llama también **top-down**. Gracias a esta técnica un problema complicado se divide en pequeños problemas que son más fácilmente solucionables. Es el caso de las metodologías de **Warnier** y la de **Jackson**.

Hoy en día en esta fase se suelen utilizar modelos orientados a objetos como la metodología **OMT** o las basadas en **UML**.

Durante el diseño se suelen realizar estas fases:

(1) **Elaborar el modelo funcional**. Se trata de diseñar el conjunto de esquemas que representan el funcionamiento de la aplicación.

(2) **Elaborar el modelo de datos**. Se diseñan los esquemas que representan la organización de los datos. Pueden ser esquemas puros de datos (sin incluir las operaciones a realizar sobre los datos) o bien esquemas de objetos (que incluyen datos y operaciones).

(3) **Creación del prototipo del sistema**. Creación de un prototipo que simbolice de la forma más exacta posible la aplicación final. A veces

incluso en esta fase se realiza más de un prototipo en diferentes fases del diseño. Los últimos serán cada vez más parecidos a la aplicación final.

(4) Aprobación del sistema propuesto. Antes de pasar a la implementación del sistema, se debe aprobar la fase de diseño (como ocurre con la de análisis).

(2.4) notaciones para el diseño de algoritmos

En el caso de la creación de algoritmos. Se utilizan esquemas sencillos que permiten simbolizar el funcionamiento del algoritmo. Pasar de estos esquemas a escribir el algoritmo en código de un lenguaje de programación es muy sencillo.

(2.4.1) diagramas de flujo

introducción

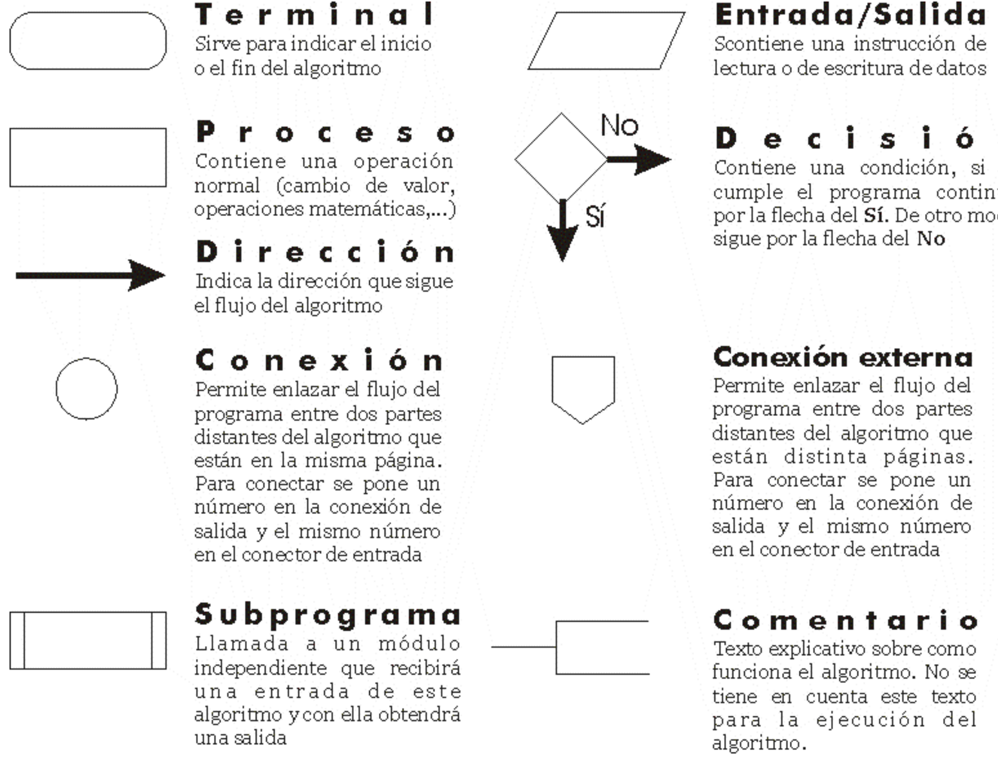
Es el esquema más viejo de la informática. Se trata de una notación que pretende facilitar la escritura o la comprensión de algoritmos. Gracias a ella se esquematiza el flujo del algoritmo. Fue muy útil al principio y todavía se usa como apoyo para explicar el funcionamiento de algunos algoritmos.

No obstante cuando el problema se complica, resulta muy complejo de realizar y de entender. De ahí que actualmente, sólo se use con fines educativos y no en la práctica. Desde luego no son adecuados para representar la fase de diseño de aplicaciones. Pero sus símbolos son tan conocidos que la mayoría de las metodologías actuales utilizan esquemas con símbolos que proceden en realidad de los diagramas de flujo.

Los símbolos de los diagramas están normalizados por organismos de estandarización como ANSI e ISO.

símbolos principales

La lista de símbolos que generalmente se utiliza en los diagramas de flujo es:



Ejemplo:

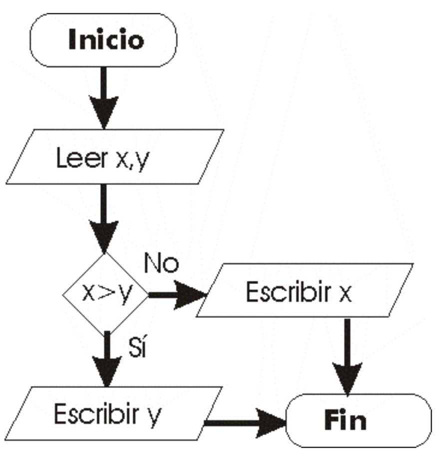


Ilustración 2-5, Diagrama de flujo que escribe el menor de dos números leídos

desventajas de los diagramas de flujo

Los diagramas de flujo son interesantes como primer acercamiento a la programación ya que son fáciles de entender. De hecho se utilizan fuera de la programación como esquema para ilustrar el funcionamiento de algoritmos sencillos.

Sin embargo cuando el algoritmo se complica, el diagrama de flujo se convierte en ininteligible. Además los diagramas de flujo no facilitan el aprendizaje de la programación estructurada, con lo que no se aconseja su uso a los programadores para diseñar algoritmos.

(2.4.2) pseudocódigo

introducción

Las bases de la programación estructurada fueron enunciadas por **Niklaus Wirdth**. Según este científico cualquier problema algorítmico podía resolverse con el uso de estos tres tipos de instrucciones:

- **Secuenciales.** Instrucciones que se ejecutan en orden normal. El flujo del programa ejecuta la instrucción y pasa a ejecutar la siguiente.
- **Alternativas.** Instrucciones en las que se evalúa una condición y dependiendo si el resultado es verdadero o no, el flujo del programa se dirigirá a una instrucción o a otra.
- **Iterativas.** Instrucciones que se repiten continuamente hasta que se cumple una determinada condición.

El tiempo le ha dado la razón y ha generado un estilo universal que insta a todo programador a utilizar sólo instrucciones de los tres tipos indicados anteriormente. Es lo que se conoce como **programación estructurada**.

El propio Niklaus Wirdth diseñó el lenguaje **Pascal** como el primer lenguaje estructurado. Lo malo es que el Pascal al ser un lenguaje completo y por tanto adaptado al ordenador, incluye instrucciones que se saltan la regla de que los algoritmos han de fabricarse independientes del ordenador.

Por ello se aconseja para el diseño de algoritmos estructurados el uso de un lenguaje especial llamado **pseudocódigo**, que además se puede traducir a cualquier idioma (Pascal está basado en el pseudocódigo en inglés).

El pseudocódigo además permite el **diseño modular** de programas y el diseño descendente (técnica que permite solucionar un problema descomponiéndole en problemas pequeños) gracias a esta posibilidad

Hay que tener en cuenta que existen multitud de *pseudocódigos*, es decir **no hay un pseudocódigo 100% estándar.** Pero sí hay gran cantidad de detalles aceptados por todos los pseudocódigos. Aquí se comenta el pseudocódigo que parece más aceptado en España.

El pseudocódigo son instrucciones escritas en un lenguaje orientado a ser entendible por un ordenador (con la ayuda de un **compilador** o un **intérprete**). Por ello en pseudocódigo sólo se pueden utilizar ciertas instrucciones. La escritura de las instrucciones debe cumplir reglas muy estrictas. Las únicas permitidas son:

- **De Entrada /Salida**. Para leer o escribir datos desde el programa hacia el usuario.
- **De proceso.** Operaciones que realiza el algoritmo (suma, resta, cambio de valor,...)

- **De control de flujo**. Instrucciones alternativas o iterativas (bucles y condiciones).
- **De declaración.** Mediante las que se crean variables y subprogramas.
- **Llamadas a subprogramas.**
- **Comentarios.** Notas que se escriben junto al pseudocódigo para explicar mejor su funcionamiento.

escritura en pseudocódigo

Las instrucciones que resuelven el algoritmo en pseudocódigo deben de estar encabezadas por la palabra **inicio** (en inglés **begin**) y cerradas por la palabra **fin** (en inglés **end**). Entre medias de estas palabras se sitúan el resto de instrucciones. Opcionalmente se puede poner delante del inicio la palabra **programa** seguida del nombre que queramos dar al algoritmo.
En definitiva la estructura de un algoritmo en pseudocódigo es:

```
programa nombreDelPrograma
inicio
   instrucciones
   ....
fin
```

Hay que tener en cuenta estos detalles:

- Aunque no importan las mayúsculas y minúsculas en pseudocódigo, se aconsejan las minúsculas porque su lectura es más clara y además porque hay muchos lenguajes en los que sí importa el hecho de hecho escribir en mayúsculas o minúsculas (C, Java, ...).
- Se aconseja que las instrucciones dejen un espacio (sangría) a la izquierda para que se vea más claro que están entre el inicio y el fin. Esta forma de escribir algoritmos permite leerlos mucho mejor.

comentarios

En pseudocódigo los comentarios que se deseen poner (y esto es una práctica muy aconsejable) se ponen con los símbolos **//** al principio de la línea de comentario (en algunas notaciones se escribe **). Un comentario es una línea que sirve para ayudar a entender el código. Cada línea de comentario debe comenzar con esos símbolos:

```
inicio
   instrucciones
   //comentario
   //comentario
   instrucciones
fin
```

instrucciones

Dentro del pseudocódigo se pueden utilizar los siguientes tipos de instrucción:

- **Declaraciones**. Sirven para declarar variables. Las variables son nombres identificativos a los que se les asigna un determinado valor y son la base de la programación. Un ejemplo sería la variable *salario* que podría servir por ejemplo para almacenar el salario de una persona en un determinado algoritmo. La instrucción de pseudocódigo es la sección **var**, que se coloca detrás del nombre del programa.
- **Constantes**. Es un tipo especial de variable llamada constante. Se utiliza para valores que no van a variar en ningún momento. Si el algoritmo utiliza valores constantes, éstos se declaran mediante una sección (que se coloca delante de la sección **var**) llamada **const** (de constante).

Ejemplo:

```
programa ejemplo1
const
    PI=3.141592
    NOMBRE=”Jose”
var
    edad: entero
    sueldo: real
inicio
....
```

Ilustración 2-6, Ejemplo de pseudocódigo con variables y constantes

- **Instrucciones primitivas**. Son las instrucciones que se realizan en el algoritmo. Se escriben entre las palabras inicio y fin. Sólo pueden contener estos posibles comandos:
  - Asignaciones (←)
  - Operaciones (+, -, * /,...)
  - Identificadores (nombres de variables o constantes)
  - Valores (números o texto encerrado entre comillas)
  - Llamadas a subprogramas
- **Instrucciones alternativas**. Permiten ejecutar una o más instrucciones en función de una determinada condición. Se trata de la instrucción **sí**.
- **Instrucciones iterativas**. Son una o más instrucciones cuya ejecución se realiza continuamente hasta que una determinada condición se cumpla. Son las instrucciones **mientras** (*while*), **repetir** (*repeat*) y **desde** (*for*).

(2.5) escritura de algoritmos usando pseudocódigo y diagramas de flujo

(2.5.1) instrucciones

Independientemente de la notación que utilicemos para escribir algoritmos, éstos contienen instrucciones, acciones a realizar por el ordenador. Lógicamente la escritura de estas instrucciones sigue unas normas muy estrictas. Las instrucciones pueden ser de estos tipos:

- **Primitivas.** Son acciones sobre los datos del programa. Son:
  - Asignación
  - Instrucciones de Entrada/Salida
- **Declaraciones**. Obligatorias en el pseudocódigo, opcionales en otros esquemas. Sirven para advertir y documentar el uso de variables y subprogramas en el algoritmo.
- **Control**. Sirven para alterar el orden de ejecución del algoritmo. En general el algoritmo se ejecuta secuencialmente. Gracias a estas instrucciones el flujo del algoritmo depende de ciertas condiciones que nosotros mismos indicamos.

(2.5.2) instrucciones de declaración

Sólo se utilizan en el pseudocódigo (en los diagramas de flujo se sobreentienden) aunque son muy importantes para adquirir buenos hábitos de programación. Indican el nombre y las características de las **variables** que se utilizan en el algoritmo. Las variables son nombres a los que se les asigna un determinado valor y son la base de la programación. Al nombre de las variables se le llama **identificador**.

identificadores

Los algoritmos necesitan utilizar datos. Los datos se identifican con un determinado **identificador** (nombre que se le da al dato). Este nombre:

- Sólo puede contener letras, números y el carácter _
- Debe comenzar por una letra
- No puede estar repetido en el mismo algoritmo. No puede haber dos elementos del algoritmo (dos datos por ejemplo) con el mismo identificador.
- Conviene que sea aclarativo, es decir que represente lo mejor posible los datos que contiene. *x* no es un nombre aclarativo, *saldo_mensual* sí lo es.

Los valores posibles de un identificador deben de ser siempre del mismo tipo (lo cual es lógico puesto que un identificador almacena un dato). Es decir no puede almacenar primero texto y luego números.

declaración de variables

Es aconsejable al escribir pseudocódigo indicar las variables que se van a utilizar (e incluso con un comentario indicar para qué se van a usar). En el caso de los otros esquemas (diagramas de flujo y tablas de decisión) no se utilizan (lo que fomenta malos hábitos).

Esto se hace mediante la sección del pseudocódigo llamada var, en esta sección se colocan las variables que se van a utilizar. Esta sección se coloca antes del inicio del algoritmo. Y se utiliza de esta forma:

```
programa nombreDePrograma
var
    identificador1: tipoDeDatos
    identificador2: tipoDeDatos
    ....
inicio
    instrucciones
fin
```

El tipo de datos de la variable puede ser especificado de muchas formas, pero tiene que ser un tipo compatible con los que utilizan los lenguajes informáticos. Se suelen utilizar los siguientes tipos:

- entero. Permite almacenar valores enteros (sin decimales).
- real. Permite almacenar valores decimales.
- carácter. Almacenan un carácter alfanumérico.
- lógico (o booleano). Sólo permiten almacenar los valores verdadero o falso.
- texto. A veces indicando su tamaño (texto(20) indicaría un texto de hasta 20 caracteres) permite almacenar texto. Normalmente en cualquier lenguaje de programación se considera un tipo compuesto.

También se pueden utilizar datos más complejos, pero su utilización queda fuera de este tema.

Ejemplo de declaración:

```
var
    numero_cliente: entero // código único de cada cliente
    valor_compra: real //lo que ha comprado el cliente
    descuento: real //valor de descuento aplicable al cliente
```

También se pueden declarar de esta forma:

```
var
    numero_cliente: entero // código único de cada cliente
    valor_compra, //lo que ha comprado el cliente
    descuento     :real //valor de descuento aplicable al cliente
```

La coma tras *valor_compra* permite declarar otra variable real.

constantes

Hay un tipo especial de variable llamada constante. Se utiliza para valores que no van a variar en ningún momento. Si el algoritmo utiliza valores constantes, éstos se declaran mediante una sección (que se coloca delante de la sección var) llamada const (de constante). Ejemplo:

```
programa ejemplo1
const
    PI=3.141592
    NOMBRE="Jose"
var
    edad: entero
    sueldo: real
inicio
....
```

A las constantes se las asigna un valor mediante el símbolo =. Ese valor permanece constante (pi siempre vale 3.141592). Es conveniente (aunque en absoluto obligatorio) utilizar letras mayúsculas para declarar variables.

(2.5.3) instrucciones primitivas

Son instrucciones que se ejecutan en cuanto son leídas por el ordenador. En ellas sólo puede haber:

- Asignaciones (←)
- Operaciones (+, -, * /,...)
- Identificadores (nombres de variables o constantes)
- Valores (números o texto encerrado entre comillas)
- Llamadas a subprogramas

En el pseudocódigo se escriben entre el inicio y el fin. En los diagramas de flujo y tablas de decisión se escriben dentro de un rectángulo

instrucción de asignación

Permite almacenar un valor en una variable. Para asignar el valor se escribe el símbolo ←, de modo que:

```
identificador←valor
```

El identificador toma el valor indicado. Ejemplo:

```
x←8
```

Ahora *x* vale 8. Se puede utilizar otra variable en lugar de un valor. Ejemplo:

```
y←9
x←y
```

x vale ahora lo que vale *y*, es decir x vale 9.

Los valores pueden ser:

- **Números.** Se escriben tal cual, el separador decimal suele ser el punto (aunque hay quien utiliza la coma).
- **Caracteres simples.** Los caracteres simples (un solo carácter) se escriben entre comillas simples: 'a', 'c',etc.
- **Textos.** Se escriben entre comillas doble "Hola"
- **Lógicos.** Sólo pueden valer **verdadero** o **falso** (se escriben tal cual)
- **Identificadores.** En cuyo caso se almacena el valor de la variable con dicho identificador.

Ejemplo:

```
x←9
y←x //y valdrá nueve
```

En las instrucciones de asignación se pueden utilizar expresiones más complejas con ayuda de los operadores.

Ejemplo:

```
x←(y*3)/2
```

Es decir x vale el resultado de multiplicar el valor de *y* por tres y dividirlo entre dos. Los operadores permitidos son:

+	Suma
-	Resta o cambio de signo
*	Producto
/	División
mod	Resto. Por ejemplo 9 **mod** 2 da como resultado 1
div	División entera. 9 **div** 2 da como resultado 4 (y no 4,5)
↑	Exponente. 9↑2 es 9 elevado a la 2

Hay que tener en cuenta la prioridad del operador. Por ejemplo la multiplicación y la división tienen más prioridad que la suma o la resta. Sí 9+6/3 da como resultado 5 y no 11. Para modificar la prioridad de la instrucción se utilizan paréntesis. Por ejemplo 9+(6/3)

(2.5.4) instrucciones de entrada y salida

lectura de datos

Es la instrucción que simula una lectura de datos desde el teclado. Se hace mediante la orden leer en la que entre paréntesis se indica el identificador de la variable que almacenará lo que se lea. Ejemplo (pseudocódigo):

```
leer x
```

El mismo ejemplo en un diagrama de flujo sería:

leer x

En ambos casos *x* contendrá el valor leído desde el teclado. Se pueden leer varias variables a la vez:

```
leer x,y,z
```

leer x, y, z

escritura de datos

Funciona como la anterior pero usando la palabra escribir. Simula la salida de datos del algoritmo por pantalla.

```
escribir x,y,z
```

escribir x, y, z

ejemplo de algoritmo

El algoritmo completo que escribe el resultado de multiplicar dos números leídos por teclado sería (en pseudocódigo)

```
programa mayorDe2
var
   x,y: entero
inicio
   leer x,y
   escribir x*y
fin
```

En un diagrama de flujo:

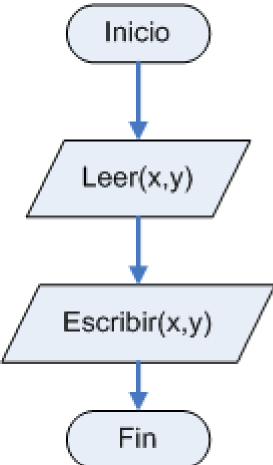


Ilustración 2-7, Diagrama de flujo con lectura y escritura

(2.5.5) instrucciones de control

Con lo visto anteriormente sólo se pueden escribir algoritmos donde la ejecución de las instrucciones sea secuencial. Pero la programación estructurada permite el uso de decisiones y de iteraciones.

Estas instrucciones permiten que haya instrucciones que se pueden ejecutar o no según una condición (instrucciones alternativas), e incluso que se ejecuten repetidamente hasta que se cumpla una condición (instrucciones iterativas). En definitiva son instrucciones que permiten variar el flujo normal del programa.

expresiones lógicas

Todas las instrucciones de este apartado utilizan expresiones lógicas. Son expresiones que dan como resultado un valor lógico (verdadero o falso). Suelen ser siempre comparaciones entre datos. Por ejemplo x>8 da como resultado verdadero si x vale más que 8. Los operadores de relación (de comparación) que se pueden utilizar son:

>	Mayor que
<	Menor que
≥	Mayor o igual
≤	Menor o igual
≠	Distinto
=	Igual

También se pueden unir expresiones utilizando los operadores **Y** (en inglés **AND**), el operador **O** (en inglés **OR)** o el operador **NO** (en inglés **NOT**). Estos operadores permiten unir expresiones lógicas. Por ejemplo:

Expresión	Resultado verdadero si...
a>8 Y b<12	La variable *a* es mayor que 8 y (a la vez) la variable *b* es menor que 12

Expresión	Resultado verdadero si...
a>8 Y b<12	O la variable *a* es mayor que 8 o la variable *b* es menor que 12. Basta que se cumpla una de las dos expresiones.
a>30 Y a<45	La variable *a* está entre 31 y 44
a<30 O a>45	La variable *a* no está entre 30 y 45
NO a=45	La variable *a* no vale 45
a >8 Y NO b<7	La variable *a* es mayor que 8 y *b* es menor o igual que 7
a>45 Y a<30	Nunca es verdadero

instrucción de alternativa simple

La alternativa simple se crea con la instrucción **si** (en inglés **if**). Esta instrucción evalúa una determinada expresión lógica y dependiendo de si esa expresión es verdadera o no se ejecutan las instrucciones siguientes. Funcionamiento:

```
si expresión_lógica entonces
   instrucciones
fin_si
```

Esto es equivalente al siguiente diagrama de flujo:

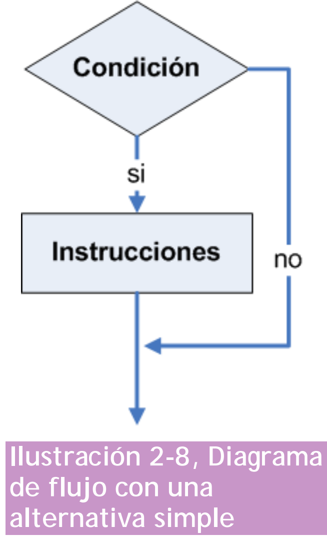


Ilustración 2-8, Diagrama de flujo con una alternativa simple

Las instrucciones sólo se ejecutarán si la expresión evaluada es verdadera.

instrucción de alternativa doble

Se trata de una variante de la alternativa en la que se ejecutan unas instrucciones si la expresión evaluada es verdadera y otras si es falsa. Funcionamiento:

```
si expresión_lógica entonces
    instrucciones //se ejecutan si la expresión es verdadera
si_no
    instrucciones //se ejecutan si la expresión es falsa
fin_si
```

Sólo se ejecuta unas instrucciones dependiendo de si la expresión es verdadera. El diagrama de flujo equivalente es:

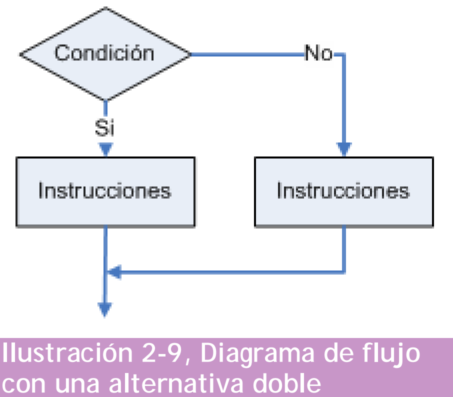


Ilustración 2-9, Diagrama de flujo con una alternativa doble

Hay que tener en cuenta que se puede meter una instrucción si dentro de otro si. A eso se le llama alternativas anidadas.

Ejemplo:

```
si a≥5 entonces
    escribe "apto"
    si a≥5 y a<7 entonces
            escribe "nota:aprobado"
    fin_si
    si a≥7 y a<9 entonces
            escribe "notable"
    si_no
            escribe "sobresaliente"
    fin_si
si_no
    escribe "suspenso"
fin_si
```

Al anidar estas instrucciones hay que tener en cuenta que hay que cerrar las instrucciones **si** interiores antes que las exteriores.

Eso es una regla básica de la programación estructurada:

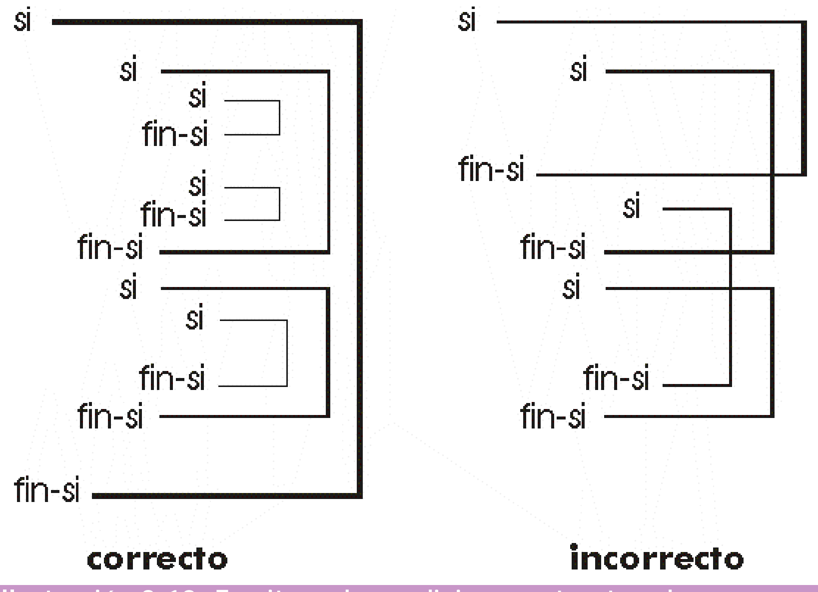


Ilustración 2-10, Escritura de condiciones estructuradas

alternativa compuesta

En muchas ocasiones se requieren condiciones que poseen más de una alternativa. En ese caso existe una instrucción evalúa una expresión y según los diferentes valores que tome se ejecutan unas u otras instrucciones.

Ejemplo:

```
según_sea expresión hacer
    valor1:
            instrucciones del valor1
    valor2:
            instrucciones del valor2
    ...
    si_no
            instrucciones del si_no
fin_según
```

Casi todos los lenguajes de programación poseen esta instrucción que suele ser un case (aunque C, C++, Java y C# usan switch). Se evalúa la expresión y si es igual que uno de los valores interiores se ejecutan las instrucciones de ese valor. Si no cumple ningún valor se ejecutan las instrucciones del *si_no*.

Ejemplo:

```
programa pruebaSelMultiple
var
    x: entero
inicio
    escribe "Escribe un número del 1 al 4 y te diré si es par o impar"
    leer x
    según_sea x hacer
            1:
                    escribe "impar"
            2:
                    escribe "par"
            3:
                    escribe "impar"
            4:
                    escribe "par"
            si_no
                    escribe "error eso no es un número de 1 a 4"
    fin_según
fin
```

El según sea se puede escribir también:

```
    según_sea x hacer
            1,3:
                    escribe "impar"
            2,4:
                    escribe "par"
            si_no
                    escribe "error eso no es un número de 1 a 4"
    fin_según
```

Es decir el valor en realidad puede ser una lista de valores. Para indicar esa lista se pueden utilizar expresiones como:

1..3	De uno a tres (1,2 o 3)
>4	Mayor que 4
>5 Y <8	Mayor que 5 y menor que 8
7,9,11,12	7,9,11 y 12. Sólo esos valores (no el 10 o el 8 por ejemplo)

Sin embargo estas últimas expresiones no son válidas en todos los lenguajes (por ejemplo el C no las admite).

En el caso de los diagramas de flujo, el formato es:

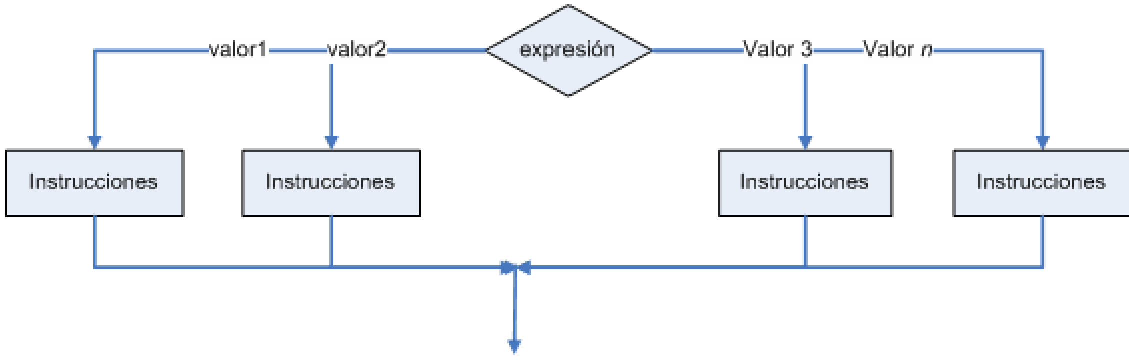


Ilustración 2-11, Diagrama de flujo con condiciones múltiples

instrucciones iterativas de tipo mientras

El pseudocódigo admite instrucciones iterativas. Las fundamentales se crean con una instrucción llamada mientras (en inglés while). Su estructura es:

```
mientras condición hacer
    instrucciones
fin_mientras
```

Significa que las instrucciones del interior se ejecutan una y otra vez mientras la condición sea verdadera. Si la condición es falsa, las instrucciones se dejan de ejecutar. El diagrama de flujo equivalente es:

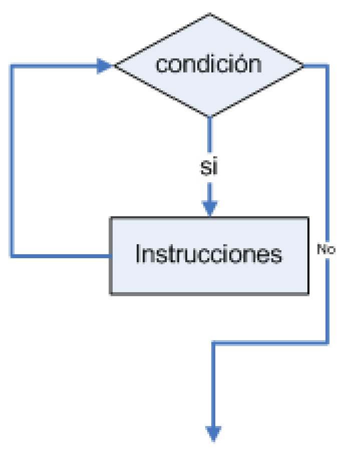


Ilustración 2-12, Diagrama de flujo correspondiente a la iteración *mientras*

Ejemplo (escribir números del 1 al 10):

```
x←1
mientras x<=10
    escribir x
    x←x+1
fin_mientras
```

Las instrucciones interiores a la palabra mientras podrían incluso no ejecutarse si la condición es falsa inicialmente.

instrucciones iterativas de tipo *repetir*

La diferencia con la anterior está en que se evalúa la condición al final (en lugar de al principio). Consiste en una serie de instrucciones que repiten continuamente su ejecución hasta que la condición sea verdadera (funciona por tanto al revés que el *mientras* ya que si la condición es falsa, las instrucciones se siguen ejecutando. Estructura

```
repetir
    instrucciones
hasta que condición
```

El diagrama de flujo equivalente es:

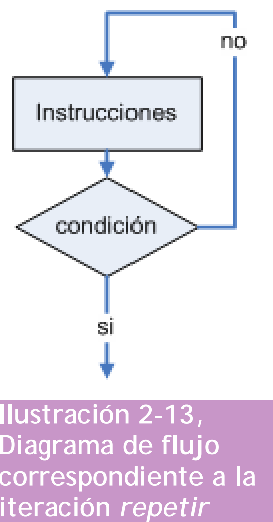


Ilustración 2-13, Diagrama de flujo correspondiente a la iteración *repetir*

Ejemplo (escribir números del 1 al 10):

```
x←1
repetir
    escribir x
    x←x+1
hasta que x>10
```

instrucciones iterativas de tipo *hacer...mientras*

Se trata de una iteración que mezcla las dos anteriores. Ejecuta una serie de instrucciones mientras se cumpla una condición. Esta condición se evalúa tras la ejecución de las instrucciones. Es decir es un bucle de tipo *mientras* donde las instrucciones al menos se ejecutan una vez (se puede decir que es lo mismo que un bucle repetir salvo que la condición se evalúa al revés). Estructura:

```
hacer
    instrucciones
mientras condición
```

Este formato está presente en el lenguaje C y derivados (C++, Java, C#), mientras que el formato de *repetir* está presente en el lenguaje Java.

El diagrama de flujo equivalente es:

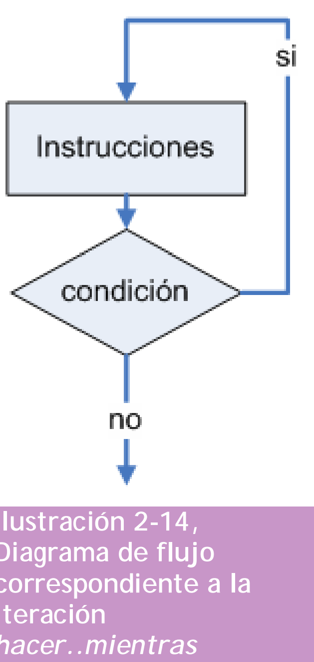


Ilustración 2-14, Diagrama de flujo correspondiente a la iteración *hacer..mientras*

Ejemplo (escribir números del 1 al 10):

```
x←1
hacer
    escribir x
    x←x+1
mientras x≤10
```

instrucciones iterativas *para*

Existe otro tipo de estructura iterativa. En realidad no sería necesaria ya que lo que hace esta instrucción lo puede hacer una instrucción *mientras*, pero facilita el uso de bucles con contador. Es decir son instrucciones que se repiten continuamente según los valores de un contador al que se le pone un valor de

inicio, un valor final y el incremento que realiza en cada iteración (el incremento es opcional, si no se indica se entiende que es de uno). Estructura:

```
para variable←valorInicial hasta valorfinal hacer
    instrucciones
fin_para
```

Si se usa el incremento sería:

```
para variable←vInicial hasta vFinal incremento valor hacer
    instrucciones
fin_para
```

El diagrama de flujo equivalente a una estructura *para* sería:

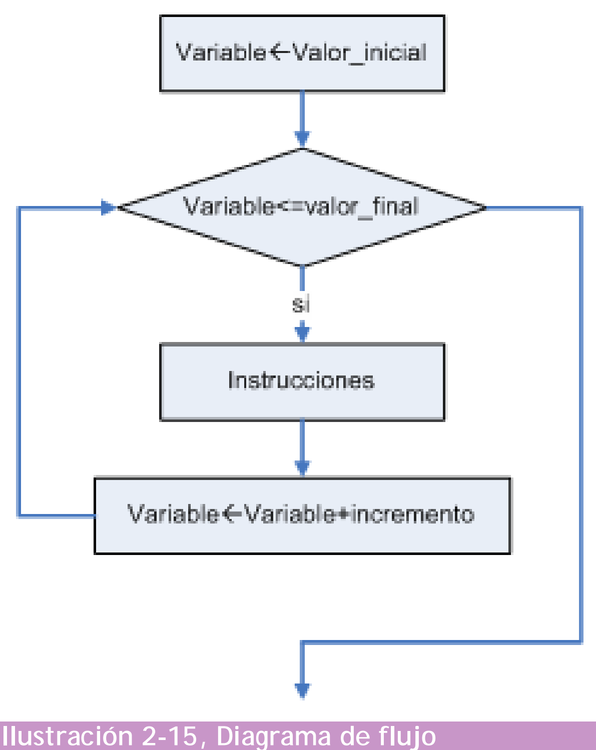


Ilustración 2-15, Diagrama de flujo correspondiente a la iteración *para*

También se puede utilizar este formato de diagrama:

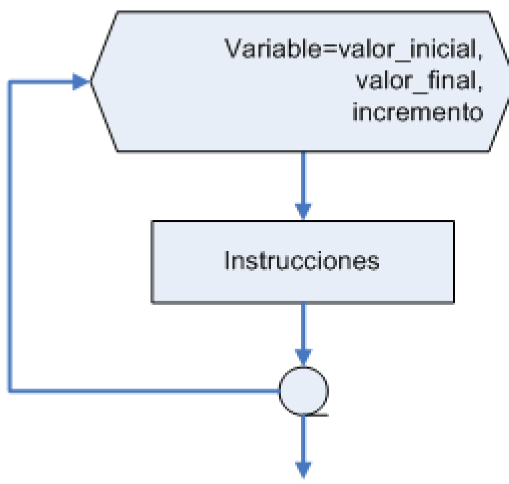


Ilustración 2-16, Diagrama de flujo con otra versión de la iteración *para*

Otros formatos de pseudocódigo utilizan la palabra **desde** en lugar de la palabra **para** (que es la traducción de **for**, nombre que se da en el original inglés a este tipo de instrucción).

estructuras iterativas anidadas

Al igual que ocurría con las instrucciones ***si***, también se puede insertar una estructura iterativa dentro de otra; pero en las mismas condiciones que la instrucción ***si***. Cuando una estructura iterativa esta dentro de otra se debe cerrar la iteración interior antes de cerrar la exterior (véase la instrucción ***si***).

(2.6) UML

(2.6.1) introducción

UML es la abreviatura de **Universal Modelling Language** (*Lenguaje de Modelado Universal*). No es una metodología, sino una forma de escribir esquemas con pretensiones de universalidad (que ciertamente ha conseguido).

La idea es que todos los analistas y diseñadores utilizaran los mismos esquemas para representar aspectos de la fase de diseño. UML no es una metodología sino una forma de diseñar (de ahí lo de lenguaje de modelado) el modelo de una aplicación.

Su vocación de estándar está respaldada por ser la propuesta de **OMG** (*Object Management Group*) la entidad encargada de desarrollar estándares para la programación orientada a objetos. Lo cierto es que sí se ha convertido en un estándar debido a que une las ideas de las principales metodologías orientadas a objetos: **OMT** de **Rumbaugh**, **OOD** de **Grady Booch y Objectory** de **Jacobson** (conocidos como **Los Tres Amigos**).

Los objetivos de UML son:

(1) Poder incluir en sus esquemas todos los aspectos necesarios en la fase de diseño.

(2) Facilidad de utilización.

(3) Que no se preste a ninguna ambigüedad, es decir que la interpretación de sus esquemas sea una y sólo una.

(4) Que sea soportado por multitud de herramientas CASE. Objetivo perfectamente conseguido, por cierto.

(5) Utilizable independientemente de la metodología utilizada. Evidentemente las metodologías deben de ser orientadas a objetos.

(2.6.2) proceso unificado

Los tres amigos llegaron a la conclusión de que las metodologías clásicas no estaban preparadas para asimilar las nuevas técnicas de programación.

Por ello definieron el llamado **Proceso Unificado de Rational**. Rational es una empresa dedicada al mundo del Análisis. De hecho es la empresa en la que recabaron Rumbaugh, Booch y Jacobson (actualmente pertenece a IBM). Dicha empresa fabrica algunas de las herramientas CASE más populares (como **Rational Rose** por ejemplo).

En definitiva el proceso unificado es una metodología para producir sistemas de información. Los modelos que utiliza para sus diagramas de trabajo, son los definidos por UML.

Es una metodología iterativa y por incrementos, de forma que en cada iteración los pasos se repiten hasta estar seguros de disponer de un modelo UML capaz de representar el sistema correctamente.

(2.6.3) diagramas UML

Lo que realmente define UML es la forma de realizar diagramas que representen los diferentes aspectos a identificar en la fase de diseño. Actualmente estamos en la versión 2.1.1 de UML, aunque el UML que se utiliza mayoritariamente hoy en día sigue siendo el primero. Los diagramas a realizar con esta notación son:

- **Diagramas que modelan los datos**.
    - **Diagrama de clases**. Representa las clases del sistema y sus relaciones.
    - **Diagrama de objetos**. Representa los objetos del sistema.
    - **Diagrama de componentes**. Representan la parte física en la que se guardan los datos.
    - **Diagrama de despliegue**. Modela el hardware utilizado en el sistema
    - **Diagrama de paquetes**. Representa la forma de agrupar en paquetes las clases y objetos del sistema.

- **Diagramas que modelan comportamiento** (para el modelo funcional del sistema).
    - **Diagrama de casos de uso.** Muestra la relación entre los usuarios (actores) y el sistema en función de las posibles situaciones (casos) de uso que los usuarios hacen.
    - **Diagrama de actividades**. Representa los flujos de trabajo del programa.
    - **Diagrama de estados**. Representa los diferentes estados por los que pasa el sistema.
    - **Diagrama de colaboración**. Muestra la interacción que se produce en el sistema entre las distintas clases.
    - **Diagramas de secuencia**. Muestran la actividad temporal que se produce en el sistema. Representa como se relacionan las clases, pero atendiendo al instante en el que lo hacen

Cabe decir que esto no significa que al modelar un sistema necesitemos todos los tipos de esquema.

A continuación veremos dos tipos de diagramas UML a fin de entender lo que aporta UML para el diseño de sistemas. Sin duda el diagrama más importante y utilizado es el de clases (también el de objetos). De hecho en la última fase del curso veremos en detalle como crear dichos diagramas.

(2.6.4) diagrama de casos de uso

Este diagrama es sencillo y se suele utilizar para representar la primera fase del diseño. Con él se detalla la utilización prevista del sistema por parte del usuario, es decir, los casos de uso del sistema.

En estos diagramas intervienen dos protagonistas:

- **Actores**. Siempre son humanos (no interesa el teclado, sino quién le golpea) y representan usuarios del sistema. La forma de representarlos en el diagrama es ésta:

Ilustración 2-17, Actor

- **Casos de uso**. Representan tareas que tiene que realizar el sistema. A dichas tareas se les da un nombre y se las coloca en una elipse.

El diagrama de casos de uso representa la relación entre los casos de uso y los actores del sistema. Ejemplo:

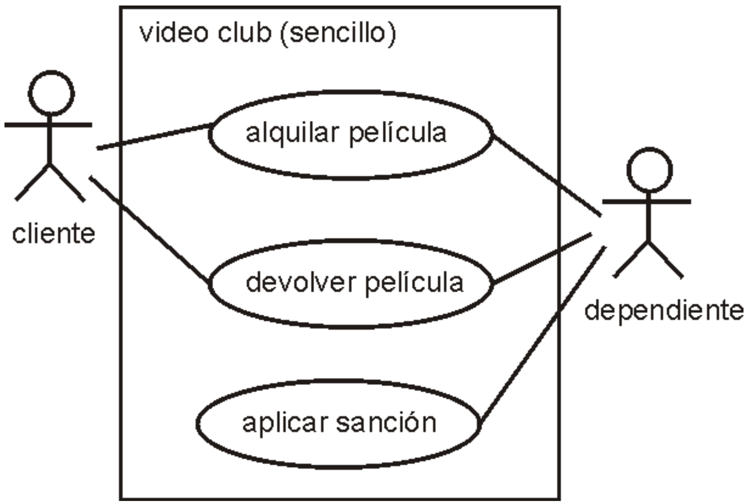


Ilustración 2-18, Ejemplo sencillo del funcionamiento de un videoclub con un diagrama de casos de uso.

En el dibujo el rectángulo representa el sistema a crear (video club). Fuera del sistema los actores y dentro los casos de uso que el sistema tiene que proporcionar.

Este tipo de diagramas se suele crear en la toma de requisitos (fase de análisis) con el fin de detectar todos los casos de uso del sistema.

Después cada caso de uso se puede subdividir (diseño descendente) en esquemas más simplificados. También se debe describir el caso de uso como texto en hojas aparte para explicar exactamente su funcionamiento. En un formato como éste:

Nombre del caso de uso:	Alquilar película
Autor:	Jorge Sánchez
Fecha:	14/10/2007

Descripción:	Permite realizar el alquiler de la película
Actores:	Cliente, dependiente
Precondiciones:	El cliente debe de ser socio
Funcionamiento normal:	1) El cliente deja la película en el mostrador junto con su carnet 2) El dependiente anota en el sistema el número de carnet para comprobar sanciones 3) Sin sanciones, se anota la película y el alquiler se lleva a cabo 4) El sistema calcula la fecha de devolución y el dependiente se lo indica al cliente
Funcionamiento alternativo	Si hay sanción, no se puede alquilar la película
Postcondiciones	El sistema anota el alquiler

(2.6.5) diagramas de actividad

Hay que tener en cuenta que todos los diagramas UML se basan en el uso de clases y objetos. Puesto que manejar clases y objetos no es el propósito de este tema, no se utilizan al explicar los diagramas de estado y actividad.

Se trata de un tipo de diagrama muy habitual en cualquier metodología. Representa el funcionamiento de una determinada tarea del sistema (normalmente un caso de uso). En realidad el diagrama de estado UML y el de actividad son muy parecidos, salvo que cada uno se encarga de representar un aspecto del sistema. Por ello si se conoce la notación de los diagramas de actividad es fácil entender los diagramas de estado.

Elementos de los diagramas de actividad

Actividad: nombre	Representada con un rectángulo de esquinas redondeadas dentro del cual se pone el nombre de la actividad. Cada actividad puede representar varios pasos y puede iniciarse tras la finalización de otra actividad.
Transición:	Representada por una flecha, indican el flujo de desarrollo de la tarea. Es decir unen la finalización de una actividad con el inicio de otra (es decir, indican qué actividad se inicia tras la finalización de la otra
Barra de sincronización:	Barra gruesa de color negro. Sirve para coordinar actividades. Es decir si una actividad debe de iniciarse tras la finalización de otras, la transición de las actividades a finalizar irán hacia la barra y de la barra saldrá una flecha a la otra actividad.
Diamante de decisión:	Representado por un rombo blanco, se utiliza para indicar alternativas en el flujo de desarrollo de la tarea según una determinada condición
Creación:	Indican el punto en el que comienza la tarea
Destrucción:	Indica el punto en el que finaliza el desarrollo del diagrama

Ejemplo de diagrama de actividad (para representar el funcionamiento del alquiler de una película del videoclub):

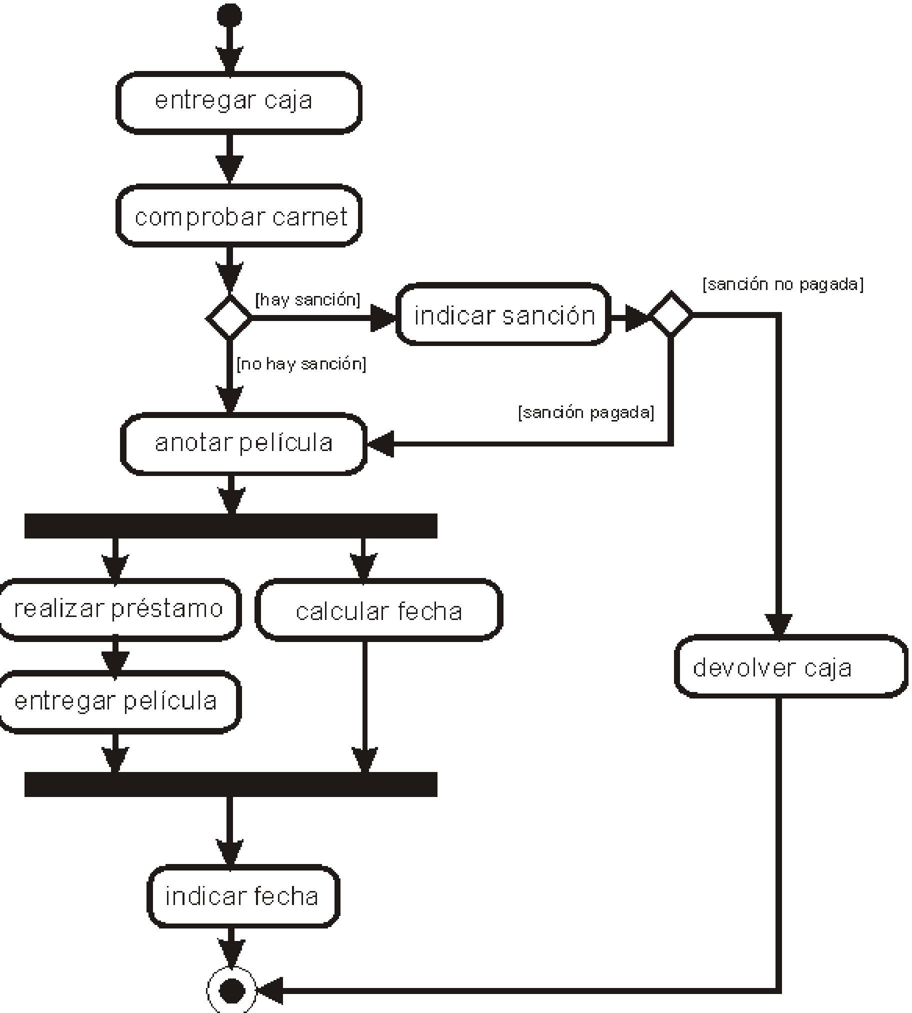


Ilustración 2-19, El diagrama de actividad del préstamo de una película

(2.7) índice de ilustraciones

Ilustración 2-1, Fases clásicas de desarrollo de una aplicación........8
Ilustración 2-2, Prototipos en las metodologías modernas........8
Ilustración 2-3, Ejemplo de desarrollo en espiral........9
Ilustración 2-4, Metodología iterativa........10
Ilustración 2-5, Diagrama de flujo que escribe el menor de dos números leídos........14
Ilustración 2-6, Ejemplo de pseudocódigo con variables y constantes........17
Ilustración 2-7, Diagrama de flujo con lectura y escritura........23
Ilustración 2-8, Diagrama de flujo con una alternativa simple........24
Ilustración 2-9, Diagrama de flujo con una alternativa doble........25
Ilustración 2-10, Escritura de condiciones estructuradas........26
Ilustración 2-11, Diagrama de flujo con condiciones múltiples........28
Ilustración 2-12, Diagrama de flujo correspondiente a la iteración *mientras*........28
Ilustración 2-13, Diagrama de flujo correspondiente a la iteración *repetir*........29
Ilustración 2-14, Diagrama de flujo correspondiente a la iteración *hacer..mientras*........30
Ilustración 2-15, Diagrama de flujo correspondiente a la iteración *para*........31
Ilustración 2-16, Diagrama de flujo con otra versión de la iteración *para*........32
Ilustración 2-17, Actor........34
Ilustración 2-18, Ejemplo sencillo del funcionamiento de un videoclub con un diagrama de casos de uso........35
Ilustración 2-19, El diagrama de actividad del préstamo de una película........37