Capítulo 3. IAGP 2005/06. Metodologías usadas en ingeniería del software

Buscar en la web

3.1 ¿Qué hay que saber para construir o elegir una metodología?

En el momento de adoptar un estándar o construir una metodología, se han de considerar unos requisitos deseables, por lo que seguidamente se proponen una serie de criterios de evaluación de dichos requisitos.

1. La metodología debe ajustarse a los objetivos
Cada aproximación al desarrollo de software está basada en unos objetivos. Por ello la metodología que se elija debe recoger el aspecto filosófico de la aproximación deseada, es decir que los objetivos generales del desarrollo deben estar implementados en la metodología de desarrollo.

2. La metodología debe cubrir el ciclo entero de desarrollo de software
Para ello la metodología ha de realizar unas etapas:

- Investigación
- Análisis de requisitos

- Diseño

3. La metodología debe integrar las distintas fases del ciclo de desarrollo

- **Rastreabilidad.** Es importante poder referirse a otras fases de un proyecto y fusionarlo con las fases previas. Es importante poder moverse no sólo hacia adelante en el ciclo de vida, sino hacia atrás de forma que se pueda comprobar el trabajo realizado y se puedan efectuar correcciones.

- **Fácil interacción entre etapas del ciclo de desarrollo**. Es necesaria una validación formal de cada fase antes de pasar a la siguiente. La información que se pierde en una fase determinada queda perdida para siempre, con un impacto en el sistema resultante.

4. La metodología debe incluir la realización de validaciones

La metodología debe detectar y corregir los errores cuanto antes. Uno de los problemas más frecuentes y costosos es el aplazamiento de la detección y corrección de problemas en las etapas finales del proyecto. Cuanto más tarde sea detectado el error más caro será corregirlo.

Por lo tanto cada fase del proceso de desarrollo de software deberá incluir una actividad de validación explícita.

5. La metodología debe soportar la determinación de la exactitud del sistema a través del ciclo de desarrollo.

La exactitud del sistema implica muchos asuntos, incluyendo la correspondencia entre el sistema y sus especificaciones, así como que el sistema cumple con las necesidades del usuario. Por ejemplo, los métodos usados para análisis y especificación del sistema deberían colaborar a terminar con el problema del entendimiento entre los informáticos, los usuarios, y otras partes implicadas.

Esto implica una comunicación entre usuario y técnico amigable y sencilla, exenta de consideraciones técnicas.

6. La metodología debe ser la base de una comunicación efectiva.

Debe ser posible gestionar a los informáticos, y éstos deben ser capaces de trabajar conjuntamente. Ha de haber una comunicación efectiva entre analistas, programadores, usuarios y gestores, con pasos bien definidos para realizar progresos visibles durante la actividad del desarrollo.

7. La metodología debe funcionar en un entorno dinámico orientado al usuario

A lo largo de todo el ciclo de vida del desarrollo se debe producir una transferencia de conocimientos hacia el usuario. La clave del éxito es que todas las partes implicadas han de intercambiar información libremente. La

participación del usuario es de importancia vital debido a que sus necesidades evolucionan constantemente. Por otra parte la adquisición de conocimientos del usuario la permitirá la toma de decisiones correctas.

Para involucrar al usuario en el análisis, diseño y administración de datos, es aconsejable el empleo de técnicas estructuradas lo más sencillas posible. Para esto, es esencial contar una buena técnica de diagramación.

8. La metodología debe especificar claramente los responsables de resultados

Debe especificar claramente quienes son los participantes de cada tarea a desarrollar, debe detallar de una manera clara los resultados de los que serán responsables.

9. La metodología debe poder emplearse en un entorno amplio de proyectos software

- **Variedad.** Una empresa deberá adoptar una metodología que sea útil para un gran número de sistemas que vaya a construir. Por esta razón no es práctico adoptar varias metodologías en una misma empresa.

- **Tamaño, vida.** Las metodologías deberán ser capaces de abordar sistemas de distintos tamaños y rangos de vida.

- **Complejidad.** La metodología debe servir para sistemas de distinta complejidad, es decir puede abarcar un departamento, varios de departamentos o varias empresas.

- **Entorno**. La metodología debe servir con independencia de la tecnología disponible en la empresa.

10. La metodología se debe de poder enseñar

Incluso en una organización sencilla, serán muchas las personas que la van a utilizar, incluso los que se incorporen posteriormente a la empresa. Cada persona debe entender las técnicas específicas de la metodología, los procedimientos organizativos y de gestión que la hacen efectiva, las herramientas automatizadas que soportan la metodología y las motivaciones que subyacen en ella.

11. La metodología debe estar soportada por herramientas CASE
La metodología debe estar soportada por herramientas automatizadas que mejoren la productividad, tanto del ingeniero de software en particular, como la del desarrollo en general.

El uso de estas herramientas reduce el número de personas requeridas y la sobrecarga de comunicación, además de ayudar a producir especificaciones y diseños con menos errores, más fáciles de probar, modificar y usar.

12. La metodología debe soportar la eventual evolución del sistema
Normalmente durante su tiempo de vida los sistemas tienen muchas versiones, pudiendo durar incluso más de 10 años. Existen herramientas CASE para la gestión de la configuración y otras denominadas "Ingeniería inversa" para ayudar en el mantenimiento de los sistemas no estructurados, permitiendo estructurar los componentes de éstos facilitando así su mantenimiento.

13. La metodología debe contener actividades conducentes a mejorar el proceso de desarrollo de software.
Para mejorar el proceso es básico disponer de datos numéricos que evidencian la efectividad de la aplicación del proceso con respecto a cualquier producto software resultante del proceso. Para disponer de estos datos, la metodología debe contener un conjunto de mediciones de proceso para identificar la calidad y coste asociado a cada etapa del proceso. Sería ideal el uso de herramientas CASE.

3.2 ¿Qué criterios son más importantes?

Se han realizado pocos estudios para determinar qué factores son los más importantes en el momento de seleccionar una metodología de desarrollo de software. Uno es el de Sachidanandam Sakthivec, donde la conclusión obtenida, es que no todos los requisitos son iguales de importantes para los profesionales de desarrollo de software.

Esta investigación se basó en requisitos de la metodología. Seguidamente se muestra un cuestionario para clasificar la importancia de los números relevantes

Cuestionarios	Números

Enviados	400
Contestados	54 (13.5%)
No válidos	27
Válidos	27

Posteriormente mediante un programa que utiliza algoritmos computacionales AHP (Analityc Hierarchy Process), identificó la importancia relativa de los requisitos de la metodología para cada participante en la encuesta. Las conclusiones de la investigación dieron el siguiente resultado:

- La capacidad de una metodología para desarrollar sistemas con la calidad requerida es el requisito más importante para los profesionales del desarrollo.
- Coincidencia con los investigadores en la importancia que tiene el criterio de satisfacción del usuario en la calidad del nuevo sistema
- La capacidad para desarrollar gran variedad de sistemas tiene mayor importancia que los aspectos relacionados con la productividad.
- Los profesionales prefieren como segundo requisito que la metodología soporte todas las etapas del desarrollo.

3.3 Paradigmas de programación y sus tipos

Un paradigma de programación es un modelo básico de diseño y desarrollo de programas, que permite producir programas con unas directrices específicas, tales como: estructura modular, fuerte cohesión, alta rentabilidad, etc.

Para algunos puede resultar sorprendente que existan varios paradigmas de programación. La mayor parte de los programadores están familiarizados con un único paradigma, el de la programación procedimental. Sin embargo hay multitud de ellos atendiendo a alguna particularidad metodológica o funcional, como por ejemplo el basado en reglas de gran aplicación en la ingeniería del conocimiento para el desarrollo de sistemas expertos, en que el núcleo del mismo son las reglas de producción del tipo "if then"; el de programación lógica, basado en asertos y reglas lógicas que define un entorno de programación de tipo conversacional, deductivo, simbólico y no determinista; el de programación funcional, basado en funciones, forma funcionales para crear funciones y mecanismos para aplicar los argumentos, y que define un entorno de programación interpretativo, funcional y aplicativo, el de programación heurística que aplica para la resolución de los problemas "reglas de buena lógica" que presentan visos de ser correctas aunque no se garantiza su éxito, modelizando el problema de una forma adecuada para aplicar estas heurísticas atendiendo a su representación, estrategias de búsqueda y métodos de resolución; el de programación paralela; el basado en restricciones; el basado en el flujo de datos; el orientado al objeto, etc.

Un paradigma de programación es una colección de modelos conceptuales que juntos modelan el proceso de diseño y determinan, al final, la estructura de un programa.

Esa estructura conceptual de modelos está pensada de forma que esos modelos determinan la forma correcta de los programas y controlan el modo en que pensamos y formulamos soluciones, y al llegar a la solución, ésta se debe de expresar mediante un lenguaje de programación. Para que este proceso sea efectivo las características del lenguaje deben reflejar adecuadamente los modelos conceptuales de ese paradigma.

Cuando un lenguaje refleja bien un paradigma particular, se dice que soporta el paradigma, y en la práctica un lenguaje que soporta correctamente un paradigma, es difícil distinguirlo del propio paradigma, por lo que se identifica con él.

Tipos de paradigmas

Floyd describió tres categorías de paradigmas de programación:

a) Los que soportan técnicas de programación de bajo nivel (ej.: copia de ficheros frente estructuras de datos compartidos)

b) Los que soportan métodos de diseño de algoritmos (ej.: divide y vencerás, programación dinámica, etc.)

c) Los que soportan soluciones de programación de alto nivel, como los descritos en el punto anterior

Floyd también señala lo diferentes que resultan los lenguajes de programación que soportan cada una de estas categorías de paradigmas. Sólo comentaremos los paradigmas relacionados con la programación de alto nivel.

Se agrupan en tres categorías de acuerdo con la solución que aportan para resolver el problema

a) Solución procedimental u operacional. Describe etapa a etapa el modo de construir la solución. Es decir señala la forma de obtener la solución.

b) Solución demostrativa. Es una variante de la procedimental. Especifica la solución describiendo ejemplos y permitiendo que el sistema generalice la solución de estos ejemplos para otros casos. Aunque es fundamentalmente procedimental, el hecho de producir resultados muy diferentes a ésta, hace que sea tratada como una categoría separada.

c) Solución declarativa. Señala las características que debe tener la solución, sin describir cómo procesarla. Es decir señala qué se desea obtener pero no cómo obtenerlo.

Paradigmas procedimentales u operacionales

La característica fundamental de estos paradigmas es la secuencia computacional realizada etapa a etapa para resolver el problema. Su mayor dificultad reside en determinar si el valor computado es una solución correcta del problema, por lo que se han desarrollado multitud de técnicas de depuración y verificación para probar la corrección de los problemas desarrollados basándose en este tipo de paradigmas.

Pueden ser de dos tipos básicos: Los que actúan modificando repetidamente la representación de sus datos (efecto de lado); y los que actúan creando nuevos datos continuamente (sin efecto de lado).

Los paradigmas con efecto de lado utilizan un modelo en el que las variables están estrechamente relacionadas con direcciones de la memoria del ordenador. Cuando se ejecuta el programa, el contenido de estas direcciones se actualiza repetidamente, pues las variables reciben múltiples asignaciones, y al finalizar el trabajo, los valores finales de las variables representan el resultado.

Existen dos tipos de paradigmas con efectos de lado:

- el imperativo

- el orientado a objetos

Los paradigmas sin efecto de lado no incluyen a los que tradicionalmente son denominados paradigmas funcionales. Sin embargo es importante distinguir la solución funcional procedimental de la solución funcional declarativa.

Los paradigmas procedimentales definen la secuencia explícitamente, pero esta secuencia se puede procesar en serie o en paralelo. En este segundo caso el procesamiento paralelo puede ser asíncrono (cooperación de procesos paralelos) o síncrono (procesos simples aplicados simultáneamente a muchos objetos).

Paradigmas declarativos
En este tipo, un programa se construye señalando hechos, reglas, restricciones, ecuaciones, transformaciones y otras propiedades derivadas del conjunto de valores que configuran la solución.

A partir de esta información el sistema debe de proporcionar un esquema que incluya el orden de evaluación que compute una solución. Aquí no existe la descripción de las diferentes etapas a seguir para alcanzar una solución, como en el caso anterior.

Estos paradigmas permiten el uso de variables para almacenar valores intermedios, pero no para actualizar estados de información.

Dado que estos paradigmas especifican la solución sin indicar cómo construirla, en principio eliminan la necesidad de probar que el valor calculado es el valor solución. En la práctica, mientras que muchos de los paradigmas secuencia de control y efecto de lado que requiera la noción de estado, las soluciones son todavía producidas como construcciones más bien que cómo especificaciones. Por lo que los paradigmas resultantes y los lenguajes que los soportan no son verdaderamente declarativos, sino pseudodeclarativos. En este grupo se encuentran: el funcional, el lógico y el de transformación.

En principio, los paradigmas declarativos no son soluciones inherentes de tipos serie o paralelo, ya que no dirigen la secuencia de control y no pueden alterar el natural no paralelismo del algoritmo. No obstante, los paradigmas pseudodeclarativos requieren al menos un limitado grado de secuencia, y por lo tanto admiten versiones en serie y paralelo.

Paradigmas demostrativos

Cuando se programa bajo un paradigma demostrativo (también llamada programación por ejemplos), el programador no especifica procedimentalmente cómo construir una solución. En su lugar, presentan soluciones de problemas similares y permite al sistema que generalice una solución procedimental a partir de estas demostraciones. Los esquemas individuales para generalizar tales soluciones van desde simular una secuencia procedimental o inferir intenciones.

Los sistemas que infieren, intentan generalizar usando razonamiento basado en el conocimiento. Una solución basada en la inferencia intenta determinar en qué son similares un grupo de datos u objetos, y, a partir de ello, generalizar estas similaridades.

Otra solución es la programación asistida: el sistema observa acciones que el programador ejecuta, y si son similares o acciones pasadas, intentará inferir cuál es la próxima acción que hará el programador. Las dos principales objeciones al sistema de inferencia son:

- Si no se comprueban exhaustivamente pueden producir programas erróneos que trabajan correctamente con los ejemplos de prueba, pero que fallen posteriormente en otros casos

- La capacidad de inferencia es tan limitada, que el usuario debe de guiar el proceso en la mayoría de los casos.

Los resultados más satisfactorios de los sistemas de inferencia son en áreas limitadas, donde el sistema tenía un conocimiento semántico importante de la aplicación.

El mayor problema que se presenta con estos sistemas, es conocer cuándo un programa es correcto. En el caso de los sistemas procedimentales, se consigue estudiando el algoritmo y el resultado de juegos de ensayo apropiados.

En el caso de los sistemas demostrativos el algoritmo se mantiene en una representación interna, y su estudio se sale del ámbito de estos sistemas. Por lo que la veracidad de la decisión se debe hacer exclusivamente sobre la base de la eficiencia del algoritmo sobre los casos específicos de prueba.

La programación demostrativa es del tipo "bottom-up" y se adapta bien a nuestra capacidad de pensar. Sin embargo en la mayor parte de los paradigmas la resolución del problema se efectúa aplicando métodos abstractos "top-down".

3.4 Paradigma imperativo

Este paradigma se caracteriza por un modelo abstracto de ordenador que consiste en un gran almacenamiento de memoria.

El ordenador almacena una representación codificada de un cálculo y ejecuta una secuencia de comandos que modifican el contenido de ese almacenamiento. Este paradigma viene bien representado por la arquitectura Von Neuman (1903-1957), ya que utiliza este modelo de máquina para conceptualizar las soluciones: "Existe un programa en memoria que se va ejecutando secuencialmente, y que toma unos datos de la memoria, efectúa unos cálculos y actualiza la memoria".

La programación en el paradigma imperativo consiste en determinar qué datos son requeridos para el cálculo, asociar a esos datos unas direcciones de memoria, y efectuar paso a paso una secuencia de transformaciones en los datos almacenados, de forma que el estado final represente el resultado correcto.

En su forma pura este paradigma sólo soporta sentencias simples que modifican la memoria y efectúan bifurcaciones condicionales e incondicionales. Incluso cuando se añade una forma simple de abstracción procedimental, el modelo permanece básicamente sin cambiar. Los parámetros de los procedimientos son "alias" de las zonas de memoria, por lo que pueden alterar su valor, y no retorna ningún tipo de cálculo. La memoria también se puede actualizar directamente mediante referencias globales.

El paradigma imperativo debe su nombre al papel dominante que desempeñan las sentencias imperativas. Su esencia es el cálculo iterativo, paso a paso, de valores de nivel inferior y su asignación a posiciones de memoria.

Si se analizan las características fundamentales de este tipo de paradigma se detectan las siguientes:

- Concepto de celda de memoria ("variable") para almacenar valores. El componente principal de la arquitectura es la memoria, compuesto por un gran número de celdas donde se almacenan los datos. Las

celdas tienen nombre (concepto de variable) que las referencian, y sobre los que se producen efectos de lado y definiciones de alias.

- Operaciones de asignación. Estrechamente ligado a la arquitectura de la memoria, se encuentra la idea de que cada valor calculado debe ser "almacenado", es decir asignado a una celda. Esta es la razón de la importancia de la sentencia de asignación en el paradigma imperativo. Las nociones de celda de memoria y asignación en bajo nivel, se tienden a todos los lenguajes de programación y fuerzan en los programadores un estilo de pensamiento basado en la arquitectura Von Neumann.

- Repetición. Un programa imperativo, normalmente realiza su tarea ejecutando repetidamente una secuencia de pasos elementales, ya que en este modelo computacional la única forma de ejecutar algo complejo es repitiendo una secuencia de instrucciones.

A este tipo de paradigma de programación se le suele llamar algorítmico, dado que el significado de algoritmo es análogo al de receta, método, técnica, procedimiento o rutina, y se define como "un conjunto finito de reglas diseñadas para crear una secuencia de operaciones para resolver un tipo específico de problemas". De esta forma para N. Wirth, un programa viene definido por la ecuación

Algoritmos + Estructura de Datos = Programas

No obstante, entendemos que aunque el concepto de algoritmo encaja en otros tipos de paradigmas, es privativo del tipo de programación procedimental en el que su característica fundamental es la secuencia computacional.

Atendiendo a los lenguajes imperativos, cabe clasificarlos en "orientados a expresiones" y "orientados a sentencias", según jueguen las expresiones o sentencias un papel más predominante en el lenguaje, respectivamente. Ambos son términos relativos y no se pueden aplicar de forma absoluta. Se puede decir que C, FORTRAN; Algol, Pascal, son lenguajes orientados a expresiones, mientras que COBOL y PL/1 están

orientados a sentencias.

Las expresiones se han encontrado útiles principalmente porque son simples y jerárquicas y pueden combinarse uniformemente para construir expresiones más complejas. Sí pues, las expresiones no sufren influencias de la arquitectura de Von Neumann.

Como ejemplos de programación imperativa, se muestran en lenguaje Pascal la generación de números primos mediante la criba de Eratóstenes, y en lenguaje C la ordenación de datos mediante el método de la burbuja.

Números Primos

```
(* Genera números primos en el rango 2..n, utilizando la criba de Eratóstenes *)

Program primos(input, output);

Const n=50;
Var i: 2..n;
    j: 2..25;
    iprimo: boolean;


Begin


for i:=2 to n do


Begin (* ¿ Es primo i ? *)
    j:=2; iprimo:=true;
While iprimo and (j<=i div 2) do
if ( (i mod j) <>0)
then j:=j+1 else iprimo:=false;
(* Si es primo imprime su valor *)
if iprimo then write (i:3)


End
```

```
void ordenar (int a[ ], int n)

{
int i,j, tem;

for (i=0; i<n; y++)

for (j=i+1; j<=n; j++)

if (a[i]>a[j])

{tem = a[i];
a[i]=a[j];
a[j]=tem;}
}

void escribir (int a[ ], int n)
{

int i;
for (y=0; i<n; y++)

{
if (!(i % 10)) printf ("\n");
printf("% 5d", a[i]);
}
}
```

La función ordenar compara parejas de datos contiguos e intercambia su contenido si el primero es mayor que el segundo. Y en cada iteración de la variable j, coloca en la cabeza de la lista que está ordenando el dato menor (la burbuja más ligera)

En este ejemplo, al igual que en anterior predominan los tres componentes descritos del paradigma imperativo.

3.5 Paradigma funcional

El paradigma funcional está basado en el modelo matemático de composición funcional. En este modelo, el resultado de un cálculo es la entrada del siguiente, y así sucesivamente hasta que una composición produce el valor deseado.

No existe el concepto de celda de memoria que es asignada o modificada. Más bien, existen valores intermedios que son el resultado de cálculos anteriores y las entradas a cálculos subsiguientes. Tampoco existen sentencias imperativas y todas las funciones tienen transparencia referencial.

La programación funcional incorpora el concepto de función como objeto de primera clase, lo que significa que las funciones se pueden tratar como datos (pueden pasar como parámetros, calculadas y devueltas como valores normales, y mezcladas en el cálculo con otras formas de datos).

En este paradigma el informático concibe la solución como una composición de funciones. Por ejemplo, para ordenar una lista, se puede diseñar la solución como una concatenación de listas más pequeñas, cada una de las cuales ya está clasificada. Esto reduce el problema a seleccionar las listas más pequeñas.

La forma en que se especifican las funciones puede variar. Se pueden especificar procedimentalmente o matemáticamente mediante su definición, sin secuencia de control.

Un ejemplo de lenguaje que soporta la forma procedimental del paradigma funcional, es el COMMONLISP, considerado como el estándar del lenguaje de programación LISP. Se ha desarrollado como un lenguaje funcional procedimental, y que admite ambos tipos de construcciones: la composición funcional con transferencia referencial y la asignación de variables con secuencias de control en la ejecución del programa.

El lenguaje LISP ("List Procesing") fue diseñado por John McCarthy, en 1959, en el Instituto Tecnológico de Masachussets, incorporando los siguientes elementos en su expresión más simple:

a) Funciones primitivas predefinidas:

Tratamiento de listas

CAR Devuelve la cabeza de la lista

CDR Devuelve la cola de la lista

CONS Construye una lista CONS a partir de otras dos

LIST Construye una lista a partir de varias sublistas

Asignación, relaciones booleanas y predicados

SET y SETQ Evalúan sus argumentos

EQ Comprueba si dos argumentos son iguales

GREATERP y LESSP Compara si el arg 1 es mayor/ menor que el 2

NULL Comprueba si la lista es nula

ATOM LISTP NUMBERP Averiguan el tipo de un objeto

OR, AND y NOT Predicados de suma, multiplicación y negación

Aritméticas

PLUS, DIFERENCE, TIMES, QUOTIENT, REMAINDER Suma, resta, multiplicación, cociente, resto

Condicionales e iterativas

COND Condicional con varios predicados

LOOP Función iterativa no declarativa

b) Formas funcionales para crear y combinar funciones

DEFUN Función que permite crear otras funciones

c) Objetos de datos

LISP dispone de esta estructura como fundamental

Un programa LISP utiliza:

Atomos Cadenas de números o caracteres

Identificadores Son átomos no numéricos

Seguidamente se presentan en LISP dos ejemplos. El primero sirve para comprobar si un número es primo o no; el segundo ordena de forma ascendente una lista de datos.

Primo
Determina si un número es primo o no

```
(DEFUN PRIMO(N)

COND((EQ N 2) T) ((EQ N 3) T)
(T (PRIMO1 N (QUOTIENT N 2)))))
(DEFUN PRIMO1 (N Y)

COND((EQ (REMAINDER N Y) 0) NIL)
((EQ Y 2) T)
(T (PRIMO1 N (DIFERENCE Y 1))))
```

Y al ejecutarlo, por ejemplo se tendría,

```
(PRIMO 3) T
(PRIMO 8) NUL
```

Ordena

```
Ordena de forma ascendente los elemento de una lista

DEFUN ORDENAR (LISTA)

COND((NULL LISTA) NIL)
(T (CONS (CAR (INVERSO (BURBUJA LISTA)))
(ORDENAR (CDR(INVERSO (BURBUJA LISTA))))))))

(DEFUN BURBUJA (LISTA)

(COND((NULL (CDR LISTA)) LISTA)
((GRATERP (CAR LISTA) (CADR LISTA))
(CONS (CAR LISTA) (BURBUJA (CDR LISTA))))
(T (CONS (CADR LISTA)
(BURBUJA (CONS (CAR LISTA) (CDDR LISTA)))))))
```

y al ejecutarse se tendría, por ejemplo,

(ORDENAR 20 8 1 3) (1,3,8,20)

la función INVERSO invierte la lista

3.6 Paradigma heurístico

Muchas de las tareas más interesantes y difíciles de programación implican utilizar el ordenador para resolver problemas del tipo: "¿Cuál es el camino más corto?" "Listar todos los casos posibles", "¿Existe una disposición de elementos que satisfaga?. Las características de estos problemas implica potencialmente una búsqueda exhaustiva de todas las posibles combinaciones de algún conjunto finito, que si no está controlado puede producir una "explosión combinatoria" (incremento exponencial del espacio de búsqueda con la dimensión del problema) imposible de tratar.

La Programación Heurística ha venido a significar el uso del conocimiento específico del dominio para cubrir esta explosión de posibilidades guiando la búsqueda por las direcciones más prometedoras. Se puede definir como "aquel tipo de programación computacional que aplica para la resolución de problemas reglas de buena lógica (reglas del pulgar). denominadas heurísticas, las cuales proporcionan entre varios cursos de acción uno que presenta visos de ser el más prometedor, pero no garantiza necesariamente el curso de acción más efectivo."

La Programación Heurística implica una forma de modelizar el problema en lo que respecta a la representación de su estructura, estrategias de búsqueda y métodos de resolución, que configuran el Paradigma Heurístico.

Este tipo de programación se aplica con mayor intensidad en el campo de la Inteligencia Artificial (IA), y en especial, en el de la Ingeniería del Conocimiento, dado que el ser humano opera la mayor parte de las veces utilizando heurísticas, un hecho cierto que una heurística es la conclusión del razonamiento humano en un dominio específico, por lo que es normal que este tipo de programación que encuadrado en el área de la I.A., ya que implementa el conocimiento humano, dado por la experiencia, utilizando reglas de buena lógica.

Como se ha señalado inicialmente, un paradigma de programación es un modelo básico de diseño e implementación de programas. Un modelo que permite producir programas de acuerdo con una metodología específica. Así, el paradigma de programación estructurada se basa en estructuras modulares, con fuerte cohesión en el módulo y bajo acoplamiento entre ellos, desarrollo "top-down", utilización de diagramas privilegiados, etc.

El Paradigma Heurístico define pues, un modelo de resolución de problemas en el que se incorpora alguna componente heurística sobre la base de:

- Una representación más apropiada de la estructura del problema para su resolución con técnicas heurísticas

- La utilización de métodos de resolución de problemas aplicando funciones de evaluación con procedimientos específicos de búsqueda heurística para la consecución de las metas.

Por otra parte, la Programación Heurística se presenta y utiliza desde diferentes puntos de vista:

- Como técnica de búsqueda para la obtención de metas en problemas no algorítmicos, o con algoritmos

que generan explosión combinatoria (ej. damas. ajedrez, etc.).

- Como un método aproximado de resolución de problemas utilizando funciones de evaluación de tipo heurístico (ej. algoritmos A*, AO*)

- Como método de poda para estrategias de programas que juegan, aunque estos métodos no son realmente heurísticos (ej. poda alfa-beta).

Se aconseja utilizar un modelo heurístico cuando:

- Los datos, limitados e inexactos, utilizados para estimar los parámetros modelo pueden contener errores inherentes muy superiores a los proporcionados con la solución de una buena heurística.

- Se utiliza un modelo simplificado, que por sí es una representación imprecisa de un problema real, por lo que la solución "óptima" es puramente académica.

- No se dispone de un método exacto que sea fiable para ser aplicado en un modelo del problema; o si existe, es intratable computacionalmente.

- Se desea mejorar la eficacia de un algoritmo optimizador aplicado al model por ejemplo, proporcionando buenas soluciones de inicio, guiando la búsqueda y reduciendo el número de soluciones candidatas

- Se tiene la necesidad de resolver el mismo problema frecuentemente, o una base de tiempo real, y el tratamiento heurístico significa un ahorro computacional

En general, un modelo heurístico es aconsejable si puede proporcionar resultados superiores a los del modelo actual.

Las especificaciones más relevantes del tratamiento heurístico deben tener en cuenta las características de la heurística, de la información y de las especificaciones del problema de acuerdo con las siguientes condiciones,

- Una buena heurística debe ser simple, con requerimientos razonables de memoria, con velocidad de búsqueda que no produzca incrementos polinomiales, ni exponenciales, precisa, robusta, que proporcione soluciones múltiples, y que disponga de un buen criterio de parada que incorpore el conocimiento obtenido durante la búsqueda.

- La información a tratar es fundamentalmente simbólica, inexacta o limitada, "incremental" y basada en el conocimiento.

- Las especificaciones del problema pueden ser: de optimización o de satisfación; que produzcan una o múltiples soluciones; con tratamiento en tiempo real o no; con decisión interactiva o no, etc.

La Programación Heurística no ha producido un lenguaje específico de programación, debido a que las heurísticas, al "ser reglas de sentido común", se pueden implementar con cualquiera de los lenguajes descritos en los diferentes paradigmas de programación. Por otra parte, al aplicarse este tipo de programación, fundamentalmente, en el campo de la I.A., las heurísticas están siendo implementadas mayoritariamente en herramientas de este área.

No obstante, si se tuviera que definir un lenguaje heurístico las características más destacables que debería incorporar son:

- Ser un lenguaje conversacional, que permitiera una interacción directa con el programador para la definición e implementación del problema.

- Tratamiento de estructuras "incrementales, que implementen programas que vayan ampliando el cuerpo de conocimiento que, en base a la experiencia, configure y refine el modelo heurístico.

- Tratamiento fundamentalmente simbólico, dado que la mayor parte de los problemas Jque precisan tratamiento heurístico tienen estructura simbólica.

- Unidades funcionales autónomas que posibiliten modelar una heurística y su mecanismo de ejecución, definiendo módulos independientes.

- Estructuras de datos que permitan describir estados de problemas y relaciones entre estados.

- Estructuras procedimentales de control y de proceso (o de definición) que permitan la ejecución coherente del modelo heurístico, y posibiliten la adquisición y utilización del conocimiento adquirido en el proceso de resolución del problema.

La Inteligencia Artificial Aplicada al Derecho.

La informática aplicada al derecho, se divide en distintos sectores o campos de aplicación, como: La informática jurídica documental, que se dedica a la aplicación de las técnicas informáticas, a la documentación jurídica en los aspectos atinentes al análisis, archivo y recuperación de información contenida en la legislación, jurisprudencia, doctrina, bibliografía o en cualquier otro documento con contenido jurídico relevante. La aplicación técnico - Jurídica se circunscribe a una especial metodología, diseñada para cada uno de los objetivos, es decir, al análisis de unidades de información de acuerdo con el sistema adoptado previamente; formación de bancos de datos, cuyo punto de partida pueden ser archivos manuales o sistematizados (sectorizados o integrales); y, finalmente, la utilización de lenguajes o mecanismos de recuperación de información guardada o grabada en bancos de datos. Estos mecanismos resultan de una organización coordinada de especialistas en el área escogida, quienes mediante palabras claves o descriptores, definen una estructura conceptual con un carácter normalizado, construyendo así el lenguaje que usuario y máquina deban manejar para encontrar la información requerida; son, en pocas palabras, los índices permutados.

La informática jurídica de gestión. Trata los aspectos gestiónales de tipo administrativo, registral, notarial, judicial, de labores parlamentarias, entre otras. Es la encargada de llevar a cabo y, en forma metódica el seguimiento, paso a paso, de los trámites y procedimientos, con la finalidad de obtener funcionalidad y rapidez en los resultados de la gestión. Ejemplos de este tipo, la sistematización de la administración de justicia, notarías, oficinas de registro, de propiedad industrial, de bufetes de abogados, entre otros. Por último, la informática jurídica decisional que es tal vez el sector más complejo de aplicación en cualquier tipo de aplicación de dominio. La función primordial de este tipo de aplicación, en el campo jurídico, consiste en el tratamiento particular que se le da a determinada cantidad de información que de ser adecuadamente configurada, resolverá automáticamente los casos que se le presenten a través de ordenadores que manejen la llamada "Inteligencia Artificial". La tecnología utilizada en estos casos es la de los "Sistemas Expertos", que parecen salir ya de la etapa puramente experimental para ingresar en aquella de inicial desarrollo en el mercado.

Múltiples pueden ser las definiciones sobre lo que es un sistema experto, si tenemos en cuenta que existen varios sectores o "dominios" de aplicación. El profesor Antonio A. Martino, director del instituto Per la Documentazione Giuridica del Consiglio Nazionale delle Picerche y autor de una teoría formalizada aplicable informáticamente, considera que un sistema experto "es un sistema que partiendo de ciertas informaciones proporcionadas por un especialista en la materia considerada, consiste en resolver los problemas que se presentan al interior de un específico "dominio", mediante la simulación de razonamientos que expertos han obtenido por sus conocimientos y experiencias adquiridos". Un sistema experto, hoy por hoy es un producto consolidado y reconocido en distintos paquetes aplicativos para sectores gestionales o en sistemas de auxilio a la proyectación.

Los sistemas expertos son programas inteligentes que usan el conocimiento y el procedimiento de inferencia para resolver problemas bastante difíciles, particularmente cuando los problemas requieren para ser resueltos una vasta experiencia de parte de una persona. Como sistemas informáticos, en grado de resolver problemas relativos a un "dominio" limitado, no requieren recurrir al típico acercamiento o contacto algoritmico de los programadores tradicionales, ya que imitan el razonamiento que sigue un experto humano. Es por esta razón que para realizar un sistema experto se necesita poder contar con el aporte de expertos en el sector aplicativo.

No existen muchos sistemas expertos jurídicos; esto se debe, según los estudiosos a una serie de motivaciones no siempre explícitamente declaradas. Dos pueden ser los tipos de motivaciones: Las que cada día empujan a un creciente número de "pioneros" a lanzarse en proyectos de desarrollo de sistemas expertos caracterizados por un alto nivel de riesgo y un largo periodo de retorno de lo invertido. La razón de quienes tienen este tipo de motivación, consiste principalmente en la imposibilidad de resolver de modo eficaz un específico problema aplicativo utilizando las normales tecnologías de programación, unidas desde luego a la facultad de encontrar expertos que verdaderamente sean capaces de resolver problemas aplicativos complejos.

La otra motivación nace de la posibilidad de efectuar una experiencia comprometiéndose con una tecnología innovativa y de crear al interior de la empresa un patrimonio de conocimiento en un campo de alto potencial de desarrollo y ventajas inimaginables. Consideran que con este tipo de aplicaciones pueden recuperar eficiencia y productividad frente a otras empresas. Las razones pueden ser entre otras: Responder a un objetivo de racionalización, de cualificación de recursos, entrar en el negocio de los sistemas expertos como proveedor, entre otros.

Entre los más importantes sistemas expertos legales tenemos:

1. El Taxman II de McCarty y Sridharan: Desarrollado en la Rutgers University hasta el estado de prototipo de investigación, provee una estructura idónea para la representación de los conceptos jurídicos y una metodología de transformación que permite reconocer las relaciones entre los conceptos. Las transformaciones del caso, una y otra vez, tomando en consideración las hipótesis relacionadas, vienen a constituir una base conceptual para el análisis de razonamiento y de la argumentación en el derecho. Taxman II contiene una sofisticada representación del conocimiento jurídico y se ocupa de un sector popular en el ámbito del derecho fiscal de los Estados Unidos.

2. El Sistema de la Rank Corporation L.D.S de Waterman y Paterson: Asiste a los expertos del derecho en la actividad de decisión de casos de responsabilidad por años derivados de productos, calculando la responsabilidad de lo convenido, el valor de la causa, y un importe equitativo por la liquidación. Su conocimiento se basa tanto sobre datos de tipo formal, derivados de la doctrina jurídica, como sobre nociones informales, como las estrategias de los abogados y los reclamos de los liquidadores.

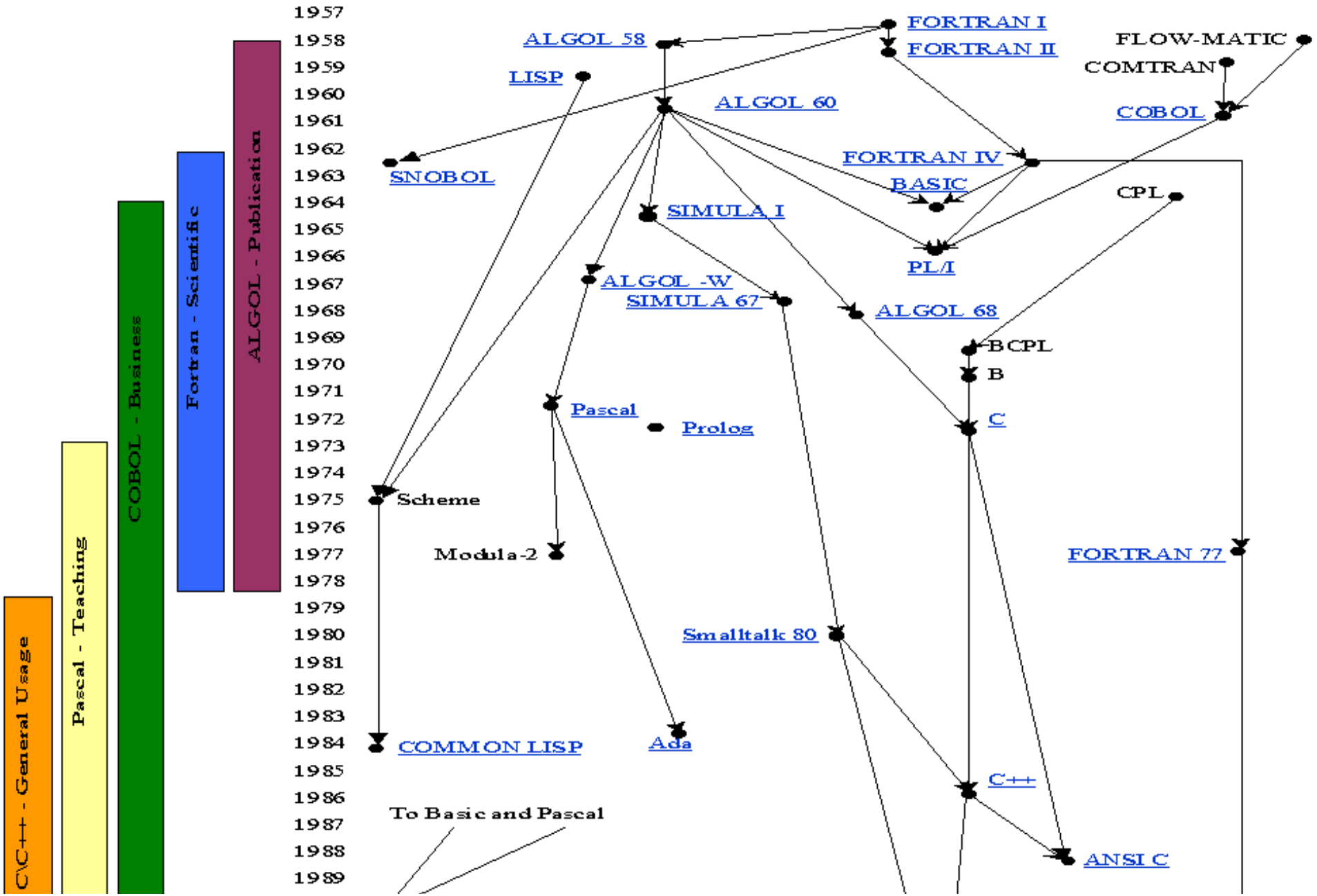
Posee la peculiaridad de basarse sobre reglas hipotéticas y es tal vez el más completo desde el punto de vista de la funcionalidad del motor de inferencia.

3. Sistema Legal Research System de Hafner: Tiene la función de ayudar a los operadores jurídicos a recuperar informaciones relativas a las decisiones judiciales y a la legislación en el campo del derecho de los títulos de comercio; contiene descriptores que vinculan los datos relativos a las preguntas de vez en vez sometidas por el usuario, con los conceptos atinentes a la materia implicada en el argumento tratado. Una red semántica basada sobre más de doscientos conceptos jurídicos constituye la estructura fundamental de la base de conocimiento.

4. Los Sistemas expertos jurídicos presentados en el cuarto congreso internacional de informática jurídica en Roma. Se trataron los temas relacionados con las tres aristas de la informática jurídica: Informar, conocer, decidir. Sobre este último aspecto, el congreso aportó entre otros asuntos los siguientes: La didáctica y los sistemas expertos; con respecto al derecho procesal de la informática se presentaron sistemas expertos aplicados a los procesos de conocimiento y procesos ejecutivos; se dieron igualmente

a conocer importantes aplicaciones de sistemas como PROLEG, y finalmente el control automatizado de la coherencia de los sistemas de normas.

El siguiente esquema muestra el desarrollo histórico de los lenguajes de programación más conocidos.



Enlaces de interés:

- Free Pascal. lenguaje gratuito.
- Paradigma de programación. Wikipedia
- Paradigmas de lenguajes de programación. Univ. Buenos Aires (República Argentina)
- Paradigmas de programación. Univ. Carlos III. Madrid
- Programación declarativa. Manuel Lucena EPS Univ. Jaen
- Software avanzado. Univ. Rey Juan Carlos . Madrid
- Tutorial básico de programación en Prolog
- Universidad Tecnológica Nacional. Santa Fe (Argentina).

anterior home siguiente