Asignatura: Entornos de programación

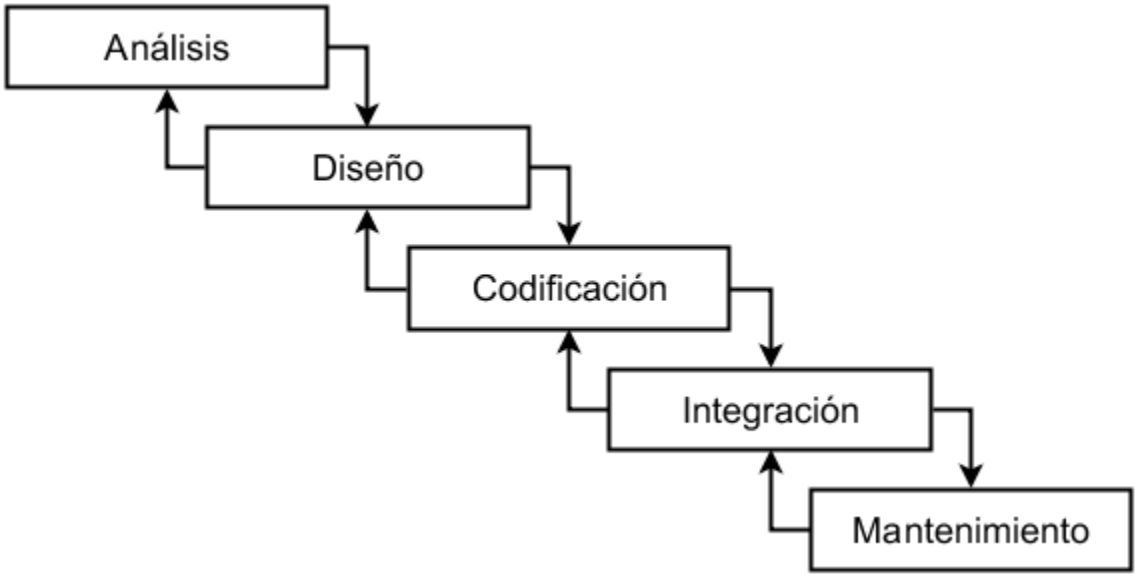
Entornos de programación

Concepto, funciones y tipos

Contenido

Actividades de desarrollo de software

En Ingeniería de Software se denomina "ciclo de vida" a una determinada organización en el tiempo de las actividades de desarrollo de software. Las principales actividades son las siguientes:



La figura representa el denominado "ciclo de vida en cascada", donde las flechas indican el orden en que se van realizando las actividades. Este modelo está en desuso, pero sigue siendo adecuado para identificar las actividades principales y el orden natural entre ellas.

Análisis de requisitos

Se estudian las necesidades de los usuarios, se decide qué debe hacer la aplicación informática para satisfacerlas en todo o en parte, y se genera un *Documento de Requisitos*.

Diseño de la arquitectura

Se estudia el Documento de Requisitos y se establece la estructura global de la aplicación, descomponiéndola en partes (módulos, subsistemas) relativamente independientes. Se genera un *Documento de Diseño*.

Diseño detallado

En esta segunda parte de la actividad de diseño se fijan las funciones de cada módulo, con el detalle de su interfaz. Se genera el código de declaración (o especificación) de cada módulo.

Codificación

Se desarrolla el código de cada módulo.

Pruebas de unidades

Como complemento de la codificación, cada módulo o grupo de módulos se prueba por separado. En las pruebas se comprueba si cada módulo cumple con su especificación de diseño detallado.

Pruebas de integración

Se hace funcionar la aplicación completa, combinando todos sus módulos. Se realizan ensayos para comprobar que el funcionamiento de conjunto cumple lo establecido en el documento de diseño.

Pruebas de validación

Como paso final de la integración se realizan nuevas pruebas de la aplicación en su conjunto. En este caso el objetivo es comprobar que el producto desarrollado cumple con lo establecido en el documento de requisitos, y satisface por tanto las necesidades de los usuarios en la medida prevista.

Fase de mantenimiento

No hay actividades diferenciadas de las anteriores. El mantenimiento del producto exige rehacer parte del trabajo inicial, que puede corresponder a cualquiera de las actividades de las etapas anteriores.

Entornos de desarrollo de Software

Un **entorno de desarrollo de software** es una combinación de herramientas que automatiza o soporta al menos una gran parte de la tareas (o fases) del desarrollo: análisis de requisitos, diseño de arquitectura, diseño detallado, codificación, pruebas de unidades, pruebas de integración y validación, gestión de configuración, mantenimiento, etc. Las herramientas deben estar bien integradas, pudendo interoperar unas con otras.

Están formados por el conjunto de instrumentos (*hardware*, *software*, procedimientos, ...) que facilitan o automatizan las actividades de desarrollo. En el contexto de esta asignatura se consideran básicamente los instrumentos software.

- **CASE**: *Computer-Aided Software Engineering*
  - Con este término genérico se denominan los productos software que dan soporte informático al desarrollo
  - Sería deseable automatizar todo el desarrollo, pero normalmente se automatiza sólo en parte
  - Productos CASE: son cada uno de los instrumentos o herramientas software de apoyo al desarrollo
- La tecnología CASE da soporte para **actividades verticales**
  - Son actividades verticales las específicas de una fase del ciclo de vida: análisis de requisitos, diseño de la arquitectura, edición y compilación del código, etc.
- También se necesita soporte para **actividades horizontales**
  - Son actividades horizontales las actividades generales: documentación, planificación, gestión de configuración, etc.

En [2] se expone una visión práctica de los que es un entorno de desarrollo.

Productos CASE en general

Los productos CASE facilitan el desarrollo organizado del software aplicando técnicas de **Ingeniería de Software**. En sentido amplio podemos englobar en la tecnología CASE toda la variedad de herramientas aplicables en el desarrollo de software: herramientas de análisis y diseño; editores de código, documentos, diagramas, etc.; compiladores y montadores de código ejecutable (linkers); depuradores; analizadores de consistencia; herramientas para obtención de métricas; generadores de código o de documentación; etc., etc.

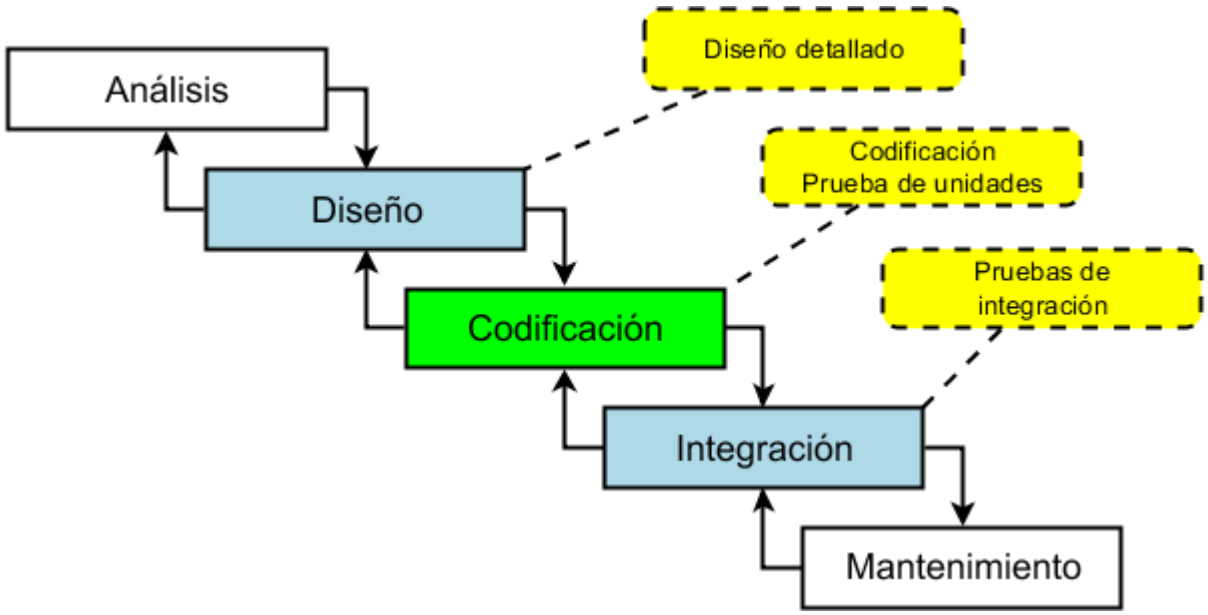
Debido a esa enorme variedad de productos, se han realizado diversos intentos para clasificarlos. Un punto de vista para su clasificación es el nivel de las funciones que realiza un producto determinado. En [3] (Table I) se sugiere la siguiente terminología para los **niveles funcionales**:

- **Servicio** (*service*): realiza automáticamente una determinada operación (atómica o unitaria).
  Ejemplo: compilación de un programa
- **Herramienta** (*tool*): ofrece los servicios necesarios para dar soporte a una tarea determinada (lo que hace un miembro del equipo de desarrollo en un momento dado).
  Ejemplo: edición de código fuente.
- **Banco de trabajo** (*workbench*): da soporte a todas las actividades correspondientes a un rol o perfil profesional propio de uno de los miembros del equipo de desarrollo.
  A veces se le llama también "herramienta" (*tool*)
  Ejemplo: "herramienta" CASE de análisis y diseño (OO, UML, ...)
- **Entorno** o **factoría** (*environment*, *factory*): da soporte a todo el proceso de desarrollo.
  A veces se le llama también "banco de trabajo" (*workbench*)

Entorno de programación

Las actividades mejor soportadas por herramientas de desarrollo son normalmente la centrales: codificación y pruebas de unidades. El conjunto de herramientas que soportan estas actividades constituyen lo que se llama un **entorno de programación**. A veces se utilizan las siglas **IDE** (*Integrated Development Environment*) para designar estos entornos, aunque no son un entorno de desarrollo completo, sino sólo una parte de él.

- Siguiendo la terminología anterior, de niveles funcionales, es el **banco de trabajo del programador**
- Da soporte a las actividades de la fase de codificación (preparación del código y prueba de unidades)
- Los mismos productos sirven también para el diseño detallado y para las pruebas de integración.
- Se sitúa, por tanto, en la parte central del ciclo de desarrollo



Funciones de un Entorno de Programación

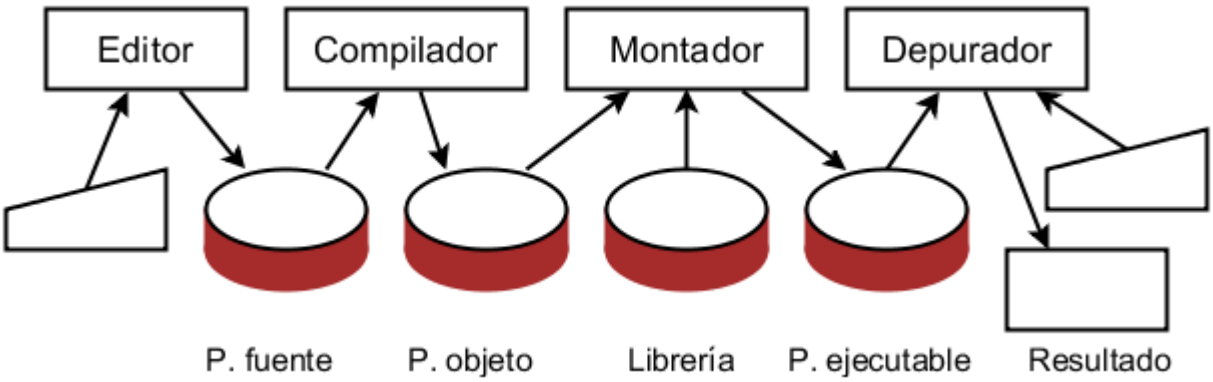
Como se ha dicho, la misión de un Entorno de Programación es dar soporte a la preparación de programas, es decir, a las **actividades de codificación y pruebas**.

- Las tareas esenciales de la fase de codificación son:
  - Edición (creación y modificación) del código fuente
  - Proceso/ejecución del programa
    - Interpretación directa (código fuente)
    - Compilación (código máquina) - montaje - ejecución
    - Compilación (código intermedio) - interpretación
- Otras funciones:
  - Examinar (hojear) el código fuente
  - Analizar consistencia, calidad, etc.
  - Ejecutar en modo depuración
  - Ejecución automática de pruebas
  - Control de versiones
  - Generar documentación, reformar código
  - ... y otras muchas más ...

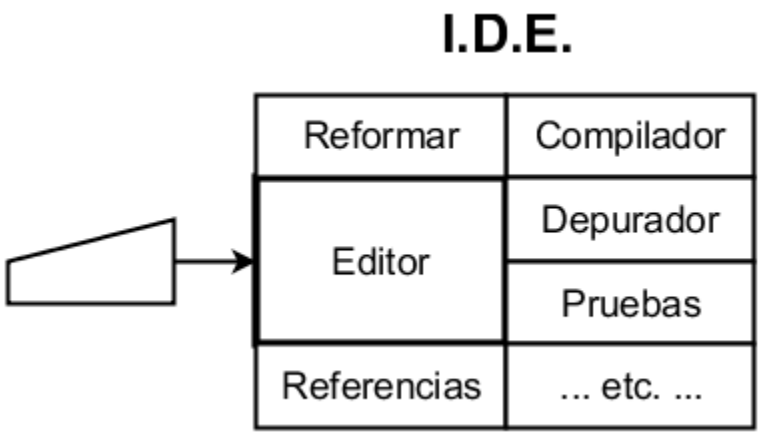
Tipos de Entornos de Programación

Un entorno de programación puede estar concebido y organizado de maneras muy diferentes. A continuación se mencionan algunas de ellas.

- En las primeras etapas de la informática la preparación de programas se realizaba mediante una cadena de operaciones tales como la que se muestra en la figura para un lenguaje procesado mediante compilador. Cada una de las herramientas debía invocarse manualmente por separado. En estas condiciones no puede hablarse propiamente de un entorno de programación



  - El editor es un editor de texto simple
  - El compilador traduce cada fichero de código fuente a código objeto
  - El montador (*linker* / *builder* / *loader*) combina varios ficheros objeto para generar un fichero ejecutable
  - El depurador maneja información en términos de lenguaje de máquina
- Un entorno de programación propiamente dicho combina herramientas como éstas, mejoradas y mejor integradas. A veces se nombra con las siglas **IDE** (*Integrated Development Environment*).



- Los componentes cuya evolución ha sido más aparente son los que realizan la interacción con el usuario:
  - El editor ya no es un simple editor de texto, sino que tiene una clara orientación al lenguaje de programación usado (reconoce y maneja determinados elementos sintácticos)
  - El depurador no presenta información en términos del lenguaje de máquina, sino del lenguaje fuente
  - El editor está bien integrado con las demás herramientas (se posiciona directamente en los puntos del código fuente en los que hay errores de compilación, o que se están ejecutando con el depurador en un momento dado.
- No es fácil establecer una clasificación dentro de la variedad de entornos de programación existentes. En algún momento [1]se describieron las siguientes clases de entornos, no excluyentes, usando un criterio esencialmente pragmático:
  - Entornos **centrados en un lenguaje**
  - Entornos **orientados a estructura**
  - Entornos **colección de herramientas**

  En [4] y [5] se exponen otras clasificaciones basadas en criterios más o menos formales.

Entornos centrados en un lenguaje

Presentan las siguientes características generales:

- Son específicos para un lenguaje de programación en particular
- Están fuertemente integrados. Aparecen como un todo homogéneo
- Se presentan como una herramienta única
- El editor tiene una fuerte orientación al lenguaje
- Son relativamente cómodos o fáciles de usar
- A veces son poco flexibles en lo referente a la interoperación con otros productos o a la ampliación de sus funciones
- Se basan en representar el código fuente como texto

Podemos encontrar ejemplos de estos entornos para todo tipo de lenguajes

- Lenguajes funcionales con interpretación directa

    - (Inter)Lisp, Haskell, etc.
- Lenguajes compilados a código de máquina nativo
    - Delphi, Visual C++, AdaGide/GNAT, GPS, etc.
- Lenguaje ejecutados sobre máquina virtual
    - Java (Visual Age, Eclipse), C# (Visual Studio .NET)
- Ejemplos especiales:
    - Entornos Ada (Stoneman, Cais, Asis)
    - Entornos Smalltalk
    - Entornos Oberon, Component Pascal

Entornos orientados a estructura

Podrían considerarse incluidos en la clase anterior, ya que suelen ser específicos para un lenguaje de programación, pero están concebidos de manera diferente:

- El editor de código fuente no es un editor de texto, sino un editor de estructura (editor sintáctico)
- Se basan en representar internamente el código fuente como una estructura:
    - Árbol de sintaxis abstracta: AST
- La presentación externa del código es en forma de texto
    - Plantillas (elementos sintácticos no terminales)
    - Texto simple (elementos terminales - a veces "frases" para expresiones)
- Compilación incremental (en algunos casos)
- Para desarrollo personal, no en equipo
- Ejemplos:
    - The Cornell Program Synthesizer (subconjunto de PL/I)
    - Mentor (Pascal)
    - Alice Pascal
    - Gandalf (intenta ser un entorno de desarrollo completo, para todo el ciclo de vida)

Estos entornos estuvieron de moda en los años 80. Los desarrollos fueron fundamentalmente académicos, y quedaron en desuso. En la actualidad los lenguajes de marcado (XML) pueden ser una buena forma de representar la estructura del código fuente con vistas a su manipulación. Existen editores y procesadores XML que podrían ser la base de nuevos entornos de programación orientados a estructura.

Entornos basados en combinación de herramientas

Consisten en una combinación de diversas herramientas capaces de interoperar entre ellas de alguna manera. Se denominan *entornos toolkit*. Presentan las siguientes características:

- Presentan integración débil
- Son un conjunto de elementos relativamente heterogéneos
- Son fáciles de ampliar o adaptar mediante nuevas herramientas
- Pueden ser construidos en parte por el propio usuario (programador): éste es más o menos el estilo UNIX original
- Ofrecen poco control de uso de cada herramienta
- El elemento frontal (*front-end*) para interacción con el usuario suele ser un editor configurable, con llamadas a herramientas externas. A veces estos editores configurables se designan también con las siglas IDE (que debería reservarse para el entorno completo)

- Ejemplos de editores configurables
  - Emacs, Vim, Gvim
  - Med, SciTE, jEdit
  - Eclipse (algo más que un editor)

Entornos multilenguaje

Hay aplicaciones que combinan piezas de código fuente escritas en diferentes lenguajes de programación. Algunas posibilidades de combinación son las siguientes:

- Entornos genéricos
  - No se combinan lenguajes en un mismo programa. Hay varios programas, cada uno en su propio lenguaje
  - Bastaría combinar las herramientas correspondientes a cada lenguaje (compiladores, etc.)
  - Se podría usar un frontal común: editor personalizable que soporte los lenguajes concretos
  - Ejemplos:
    - Emacs (con diferentes "modos")
    - Eclipse (con diferentes "plug-ins")
- Entornos específicos
  - Para una combinación concreta de lenguajes
  - Vienen a ser como los entornos centrados en un lenguaje, sólo que admiten más de uno
  - Usan un formato binario compatible que permite combinar en un mismo programa partes escritas en los diferentes lenguajes
  - Ejemplo: GPS permite combinar módulos en Ada y C++
- Lenguajes ejecutados sobre máquina virtual
  - La máquina virtual establece el formato del código binario
  - Pueden combinarse módulos escritos en diferentes lenguajes para los que exista el compilador apropiado
  - Cada lenguaje puede tener su entorno de programación separado, o bien existir un entorno de programación único
  - Ejemplos:
    - JVM (Java Virtual Machine). El lenguaje original es Java. El intérprete es el JRE (Java Runtime Environment). Hay compiladores a JVM para otros lenguajes además de Java: Ada, Fortran, Component Pascal (Oberon), etc. (incluso C#)
    - .Net (Microsoft). El lenguaje original es C#. El intérprete es el CLR (Common Language Runtime). Hay compiladores a .Net para otros lenguajes además de C#: Ada, Fortran, Component Pascal (Oberon), etc. (incluso Java)

Entornos para ingeniería de software

Un entorno de programación, tal como se ha definido anteriormente, serviría para dar soporte a las tareas de desarrollo de software realizadas por una persona. Para desarrollar proyectos de software no triviales se necesita trabajar en equipo usando las recomendaciones de la ingeniería de software.

Cada miembro del equipo de desarrollo puede disponer de una estación de trabajo con un entorno de programación adecuado para realizar su trabajo individual, y se necesita además

algún medio de combinar los trabajos individuales en una labor de conjunto, debidamente organizada.

Una manera intuitiva de organizar el entorno general de desarrollo es basarlo en un repositorio central de información, dotado de un sistema de gestión de configuración, y añadirle sistemas de mensajería, de gestión de incidencias, herramientas de modelado para análisis y diseño, de gestión del proyecto, etc.

Por ejemplo, hay plataformas generales que ofrecen este soporte como servicios web, incluso de manera gratuita para el desarrollo de software libre: SourceForge, Google Code, etc.

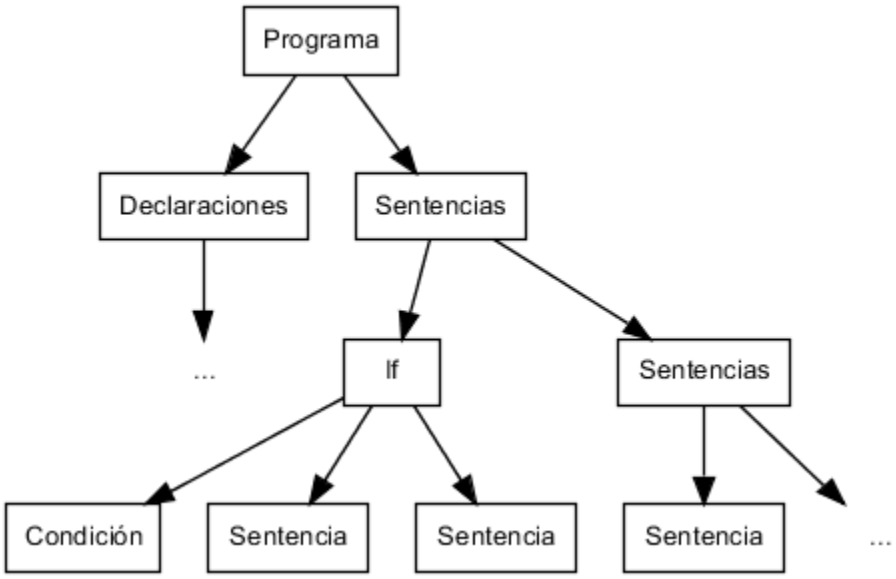
Entornos orientados a estructura

La idea de que un programa no es equivalente al texto de su código fuente, sino que lo esencial es la estructura lógica del cómputo que describe, ha llevado a la creación de los llamados entornos de programación orientados a estructura[1], en los que se manipula directamente la estructura lógica del código y no su representación como texto. Incluso hay casos en que el código del programa no se representa como texto sino en forma gráfica.

Editores de estructura

Los editores de estructura de código, llamados también editores sintácticos o más frecuentemente editores dirigidos por sintaxis (*sintax-directed editors*), permiten editar el código fuente manipulando directamente una representación interna de su estructura. A diferencia de la edición del código como texto, la edición de la estructura se hace sobre elementos sintácticos tales como expresiones, sentencias o funciones y no sobre elementos textuales tales como caracteres, palabras o líneas.

La representación habitual de la estructura del código es la de su árbol de sintaxis abstracta (AST). Ejemplo:



Los entornos de programación basados en un editor de estructura se denominan entornos orientados a estructura. Suelen tener las siguientes características:

- Soportan un único lenguaje de programación.
- Garantizan que el código es sintácticamente correcto.
- La compilación se realiza de manera incremental, a medida que se edita el código.
- Permite la ejecución inmediata del código editado, incluso aunque esté incompleto.
- Soportan el desarrollo de software a nivel individual, pero no el desarrollo en equipo a gran escala.

La mayoría de estos entornos se desarrollaron a finales del años 70 y a lo largo de los 80. Se emplearon habitualmente en ambientes académicos. Algunos ejemplos de entornos orientados a estructura son:

- El **Cornell Program Synthesizer (CPS)**. Es quizá el ejemplo más conocido y el más referenciado en la literatura [4]. Ha servido de ejemplo para desarrollar otros. Permite programar en un subconjunto del lenguaje PL/I denominado PL/CS.
- **Mentor** [7] es un entorno de programación en Pascal.
- **Gandalf** [8] es un conjunto de varios subproyectos. Su objetivo principal fue crear un entorno completo de desarrollo de software, y no sólo un entorno de programación.
- **Alice Pascal** [9] es otro entorno de programación en lenguaje Pascal compatible con TurboPascal. Sigue las ideas del CPS.
- **SDS** es un entorno de programación en Modula-2. Es un producto comercial desarrollado por Interface Technologies. Ha desaparecido.

Como complemento se han llegado a desarrollar también generadores de entornos similares a los generadores de compiladores. En particular existe el Synthesizer Generator [10], capaz de generar entornos similares al sintetizador de Cornell para otros lenguajes de programación a partir de una descripción de la sintaxis y semántica del lenguaje mediante una gramática de atributos.

Lenguajes y entornos visuales

Este es un caso especial de entornos orientados a estructura. La representación externa del código fuente no es en forma de texto, sino gráfica. El editor permite ir construyendo el grafo que representa la estructura del código. El programa construido de esta manera se ejecuta directamente mediante un intérprete, o bien se exporta como texto en un lenguaje formal para ser compilado o interpretado externamente. Algunos ejemplos de este tipo de entornos son:

- **Prograph**
- **Projector** (parte del meta-CASE DOME)
- **VFPE**

Tanto Prograph como Projector son lenguajes de flujo de datos. Un programa basado en flujo de datos se representa como un grafo en el que los nodos son operadores y los arcos son flujos de datos que conectan la salida de ciertos operadores con las entradas de otros. Una operación se ejecuta cuando hay datos presentes en todas las entradas requeridas. En ese momento se producen resultados que se transmiten por los arcos de salida, pudiendo entonces ejecutarse otras operaciones.

VFPE es un editor gráfico de la estructura (árbol sintáctico) de un programa funcional. El programa editado puede ejecutarse directamente o exportarse como código Haskell.

Discusión

Los entornos orientados a estructura presentan innegables ventajas respecto a los entornos basados en la edición del texto fuente. Entre ellas:

- Evitan los errores sintácticos
- Evitan tener que escribir los elementos fijos del código: Palabras clave, puntuación, etc.
- Presentan el código con un estilo uniforme, bien encolumnado.
- Guían al programador indicando qué elementos pueden insertarse en cada punto y recordándole la sintaxis de cada sentencia estructurada.
- Facilitan la reorganización del código al permitir la selección directa de secciones de código: funciones, bucles, etc.
- Facilitan trabajar con estructuras lógicas no contempladas directamente en el lenguaje de programación.

Si embargo estos entornos, tal como se concibieron inicialmente, no han llegado a utilizarse en la práctica habitual de desarrollo de software, ya que presentaban claros inconvenientes:

- No permitían el trabajo en equipo. En muchos casos sólo trabajaban con programas monolíticos.
- Era difícil realizar algunas operaciones de edición que resultan triviales sobre el texto.
- Exigen un cambio en la mentalidad del programador [11].
- Es difícil editar la estructura a nivel de grano fino (expresiones: operadores, operandos, ...)

Muchos de los productos citados como ejemplo han desaparecido. Su interés es fundamentalmente histórico. No obstante, la idea de trabajar directamente sobre la estructura del código sigue resultando atractiva, y parece posible aplicarla de nuevo siendo conscientes de sus dificultades y aprovechando los avances tecnológicos actuales.

Evolución de los entornos de programación

Otra circunstancia que quizá ha colaborado a impedir la difusión de los entornos orientados a estructura es el hecho de que los entornos de programación convencionales basados en la manipulación del código fuente como texto han ido mejorando a lo largo de los años, ofreciendo algunas funciones similares a las que se pretendían conseguir con los entornos orientados a estructura. A continuación se analizan algunas de estas funciones.

Lectura y navegación del código

Los entornos basados en texto fuente han ido incrementando su orientación al lenguaje, ofreciendo facilidades tales como las siguientes:

- **Resaltado de sintaxis**. Se destacan los elementos léxicos con diferentes colores o tipo de letra. También se pueden destacar los paréntesis o llaves que se emparejan simplemente poniendo el cursor sobre uno de ellos. El reconocimiento de estos elementos se basa en un sencillo análisis de cada línea de texto, que se realiza sobre la marcha. En la mayoría de los casos el resultado es totalmente satisfactorio, pero a veces se producen confusiones si se emplean construcciones sintácticas complejas. Un análisis sintáctico más completo para garantizar el correcto funcionamiento en todos los casos sería demasiado costoso.
- **Plegado/desplegado**. Se puede controlar el nivel de detalle con el que se presenta el texto fuente. Determinadas secciones de código pueden presentarse de manera abreviada, para facilitar la visión de conjunto. Por ejemplo, un subprograma puede reducirse a la línea de cabecera junto con una indicación visual de que parte del contenido está oculto. Ocasionalmente puede tener los inconvenientes indicados en el punto anterior.
- **Acceso directo entre elementos relacionados**. Son facilidades equivalentes a hipertexto para saltar rápidamente entre la declaración, definición, implementación y uso de un determinado identificador. Puede extenderse incluso a funciones de librería, y también al recorrido de todos los puntos en que usa un identificador. En ocasiones se dispone de una ventana emergente que muestra la cabecera de declaración de una función al poner el cursor sobre un punto en que se usa dicha función. Estas facilidades resultan extraordinariamente útiles para la lectura y comprensión del código.
- **Vistas múltiples**. Se usan varias ventanas para presentar simultáneamente diferentes vistas del código fuente. Es habitual tener un panel lateral con la estructura en árbol del código, bien del fichero en edición o de todo el proyecto. También se pueden tener a la vez los ficheros fuente de interfaz y de implementación del mismo módulo. Y ventanas adiciones con los resultados de aplicar determinadas herramientas: lista de errores de compilación, lista de puntos en que usa un identificador, mensajes de diagnóstico de una herramienta de análisis, representaciones gráficas, etc.

Las últimas facilidades de la lista pueden tener un coste elevado. En bastantes casos hay que construir unas tablas internas, bastante voluminosas, que describen la estructura del código y permiten indexar la aparición de cada identificador en el texto fuente.

Generación de documentación

Si se incluyen en el código fuente comentarios de descripción de cada elemento importante es posible generar automáticamente documentación de referencia a base de extraer las declaraciones de dichos elementos junto con el texto de descripción. Por ejemplo, se pueden

generar las páginas del manual de referencia de una librería, incluyendo índices alfabéticos o temáticos, o incluso incluir todo el código fuente en forma de páginas web enlazadas para consultarlas de manera muy cómoda.

Algunos ejemplos de esta tecnología son:

- **Doxygen**: Genera documentación de código C/C++ en forma de páginas web, incluyendo diversos índices, el código fuente coloreado e indexado, e incluso diagramas de dependencia entre módulos.
- **Javadoc**: Para lenguaje Java. Usa un formato prefijado de comentario para las descripciones, incluyendo marcas HTML embebidas y palabras clave introducidas con el símbolo @. Se generan documentos en forma de páginas web.
- **AdaBrowse/AdaDoc/Ada2html**: Son herramientas similares a las anteriores para código en lenguaje Ada. Igualmente generan páginas web.

A continuación se reproduce un ejemplo tomado de la documentación disponible en el sitio web de Sun Microsystems, Inc [12]. El fragmento de código fuente es:

```
/**
 * Returns an Image object that can then be painted on the
screen.
 * The url argument must specify an absolute {@link URL}.
The name
 * argument is a specifier that is relative to the url
argument.
 * <p>
 * This method always returns immediately, whether or not
the
 * image exists. When this applet attempts to draw the image
on
 * the screen, the data will be loaded. The graphics
primitives
 * that draw the image will incrementally paint on the
screen.
 *
 * @param  url  an absolute URL giving the base location of
the image
 * @param  name the location of the image, relative to the
url argument
 * @return      the image at the specified URL
 * @see         Image
 */
 public Image getImage(URL url, String name) {
    try {
        return getImage(new URL(url, name));
    } catch (MalformedURLException e) {
        return null;
    }
 }
```

A partir de él se genera el siguiente fragmento de documentación (*convertido a XHTML*):

---

getImage

```
public Image getImage(URL url,
                      String name)
```

Returns an `Image` object that can then be painted on the screen. The `url` argument must specify an absolute `URL`. The `name` argument is a specifier that is relative to the `url` argument.

This method always returns immediately, whether or not the image exists. When this applet attempts to draw the image on the screen, the data will be loaded. The graphics primitives that draw the image will incrementally paint on the screen.

Parameters:
`url` - an absolute URL giving the base location of the image
`name` - the location of the image, relative to the `url` argument

Returns:
the image at the specified URL

See Also:
`Image`

Análisis estático

En estas técnicas se analiza la estructura sintáctica y/o semántica del programa para realizar comprobaciones, extraer información y generar informes. Algunos ejemplos interesantes son:

- **Análisis de consistencia**: Se realizan comprobaciones adicionales a las que habitualmente hace un compilador. De esta manera se comprueba que el código no contiene irregularidades, ambigüedades, código inalcanzable, construcciones desaconsejadas, etc. Ejemplo en lenguaje C: uso de una asignación como condición, sentencia simple en un 'if', etc:

```
if (x = 3)
    printf( "ejemplo" );
    printf( "\n" );
```

- **Comprobación de estilo**: Se comprueba si el código cumple determinadas reglas de presentación o codificación. Puede realizarse conjuntamente con el análisis del punto anterior.
- **Cálculo de métricas**: Existen medidas bien establecidas de la complejidad o tamaño del código. Pueden utilizarse como indicadores de calidad o de productividad en el desarrollo del proyecto. Consisten habitualmente en recuentos de determinados elementos (número de líneas de código, de sentencias, de funciones, etc.) y obtención de parámetros estadísticos (número de líneas por función, porcentaje de comentarios, número de métodos por clase, etc.).

- **Seccionamiento (*slicing*) de programas**: Consiste el extraer la parte del código que interviene en el cálculo de una determinada variable. Esto ayuda a comprender el funcionamiento del programa, diagnosticar fallos, detectar dependencias entre partes del código o determinar el impacto de un posible cambio.
- **Otras ayudas a la comprensión del código**: En muchos casos se traducen en diagramas que muestran de manera clara determinadas características del código. Por ejemplo, el grafo de flujo de llamadas, un diagrama de estructura modular, etc.

Las operaciones de análisis mencionadas tienen partes en común, por lo que muchas veces una misma herramienta puede realizar simultánea o alternativamente varias de ellas.

Técnicas avanzadas de construcción de código

El desarrollo de metodologías de programación ha dado como resultado diversas técnicas y herramientas de ayuda a la construcción de nuevo código. Algunas de ellas se han incorporado a los entornos de programación integrados y otras están soportadas mediante herramientas independientes. A continuación se mencionan algunas de ellas.

- **Asistentes (*Assistants, Wizzards*)**: Ayudan al programador, bien guiando las operaciones de construcción, o bien generando automáticamente parte del código. Por ejemplo, los entornos de programación orientada a objetos pueden disponer de un *Class Wizzard* que presenta un índice de las clases existentes con sus miembros, y facilita la creación de nuevos elementos generando un esqueleto de código para la nueva clase o miembro.
- **Ingeniería inversa**: No es una técnica de generación de nuevo código, sino de reconstrucción de información de diseño a partir de código heredado no documentado.
- **Reingeniería**: Complementa la anterior, facilitando la reorganización del código antiguo para facilitar su mantenimiento.
- **Refactorización**: Viene a ser una forma de reingeniería continua del código en desarrollo, recomendada por la metodología de Programación Extrema (*Extreme Programming - XP*). El desarrollo se hace mediante cambios progresivos, y en cada uno de ellos se reorganiza el código, si es conveniente. Es decir, se modifica el diseño y no sólo el código fuente.
- **Desarrollo basado en componentes**: Es una forma intensiva de reutilización. Se construye una aplicación combinando componentes ya existentes en lugar de desarrollar nuevos elementos partiendo de cero. Por supuesto, sólo puede aplicarse de la forma indicada si existe previamente una base de componentes para el dominio de la aplicación. En general exige desarrollar algo de código para configurar la aplicación y a veces para actuar como intermediario entre componentes (*wrapper, middleware*), si estos no pueden conectarse directamente.
- **Composición invasiva**: A veces la simple combinación de componentes no es posible, sino que el código interno de los componentes debe ser modificado parcialmente para obtener la nueva aplicación. En este caso debería haber una herramienta que realice los cambios de acuerdo con una cierta especificación, en lugar de realizar los cambios manualmente. La acción de composición debe ser repetible, para facilitar el mantenimiento.
- **Programación orientada a aspectos (AOP)**: Es una forma de composición invasiva en que el código base se amplía para incluir nuevos "aspectos" inicialmente no tenidos en cuenta. Existen lenguajes para definir los nuevos fragmentos de código e indicar en qué puntos deben insertarse. La herramienta de soporte se denomina "tejedor" (*weaver*) y genera el código modificado a partir del código original y la descripción de los nuevos "aspectos".

- **Lenguajes específicos de un dominio (DSL)**: En lugar de crear librerías de funciones para implementar las operaciones frecuentes en un dominio de aplicación e invocarlas desde un lenguaje de programación de uso general, lo que se puede hacer es crear un lenguaje de programación especializado que incorpore directamente dichas funciones como predefinidas. Este lenguaje especializado puede facilitar la productividad de los programadores al trabajar a un nivel de abstracción superior y usar una notación más compacta o específica, que facilite la claridad del código. Es frecuente que estos lenguaje especializados no se procesen con un compilador específico, sino mediante un traductor o generador de código que produce código fuente en un lenguaje de programación de uso general, que posteriormente se compila de la forma habitual. Esto facilita además combinar en un mismo proyecto partes codificadas con el lenguaje del dominio y con el lenguaje de uso general.
- **Programación letrada (*Literate Programming* - LP)**: Se usa poco en la práctica, aunque puede resultar muy eficiente de cara al mantenimiento del software. La idea es redactar el código como si fuera un documento técnico, explicando cada parte. Los fragmentos de código aparecen entremezclados con las explicaciones, en el orden adecuado para entender el conjunto. No hay que seguir el orden del código final, sino que cada parte se introduce en el momento apropiado para facilitar la lectura y comprensión del documento. Se utiliza un lenguaje de autor especializado (denominado genéricamente *web*), y dos herramientas: una que genera el documento en formato legible con una buena presentación (*weave*), y otra que genera el código de la aplicación listo para ser compilado (*tangle*).

Discusión

Las técnicas y facilidades incorporadas en los entornos de programación y desarrollo a lo largo de estos años resultan extraordinariamente útiles para abordar la construcción de grandes sistemas de software, con cientos de miles o millones de líneas de código. Pero su aplicación sobre la base convencional de representar y manipular el código fuente como texto tiene un coste enorme en esfuerzo para la creación de las herramientas y en recursos para su aplicación. Cada herramienta o función necesita analizar el texto de código para reconocer su estructura lógica. Para facilitar los tratamientos es frecuente que en el entorno se tengan a la vez una representación de código como texto y unas tablas o índices con su estructura lógica, que si es detallada ocupa un espacio muy superior al del mismo código. Además a medida que se modifica el código hay que mantener sincronizadas ambas representaciones, lo que exige una gran potencia de cálculo.

Los antiguos entornos orientados a estructura no llegaron a desarrollar las facilidades ahora disponibles, pero tenían un enfoque mucho más simple, con una sola representación del código, más eficiente y sencilla de manipular. Recientemente han surgido desarrollos en los que se recuperan en parte algunas ideas o técnicas de orientación a estructura.

Referencias

1. S.A. Dart, R.J. Ellison, P.H. Feiler, A.N. Habermann: *Software Development Environments*. IEEE Computer, Vol.20 No.11 pp.18-28, Nov.1987.
2. M.B. Doar: *Practical Development Environments* (el Cap.2: *Project basics*, está disponible como muestra, así como una vista parcial del libro). O'Reilly Media, 2005.
3. C. Fernström, K-H Närfelt, L. Ohlsson: *Software Factory Principles, Architecture, and Experiments*. IEEE Software, Vol.9 No.2 pp.36-44, Mar.1992.
4. A. Fuggetta: A Classification of CASE Technology. IEEE Computer, Vol.26 No.12 pp.25-38, Dic.1993.

5. D.E. Perry, G.E. Kaiser: Models of Software Development Environments. IEEE Trans.Softw.Eng., Vol.17 No.3, pp.283-295, Mar.1991.
6. T. Teitelbaum, T. Repps: *The Cornell Program Synthesizer: A Syntax-Directed Programming Environment*. Comm. ACM, V.24 N.9 pp.563-673, Sep.1981.
7. V. Donzeau-Gouge, G. Huet, G. Kahn, B. Lang: *Programming environments based on structured editors: the MENTOR experience*. In *Interactive Programming Environments*, McGraw-Hill, 1984.
8. A. Habermann, D. Notkin: *Gandalf: Software development environments*. IEEE Transactions on Software Engineering SE-12, pp 1117-1127, 1986.
9. B. Templeton: *Alice Pascal website* (http://www.templetons.com/brad/alice.html)
10. T. Reps, T. Teitelbaum: *The Synthesizer Generator*. ACM Softw.Engin.Notes, May 1984.
11. L. R. Neal: *Cognition-Sensitive Design and User Modeling for Syntax-Directed Editors*. In Proceedings CHI+GI 1987 (Toronto, April 5-9, 1987) ACM, New York, pp 27-32.
12. Sun Microsystems: *Javadoc website* (http://java.sun.com/j2se/javadoc/writingdoccomments/index.html).