Capítulo

7

Codificación del Software

Centro Asociado Palma de Mallorca

Tutor: Antonio Rivero Cuesta

7 Codificación del Software.

7.1 Introducción.

Las etapas de Análisis y Diseño tienen como misión fundamental la de organizar y traducir los requisitos del Cliente en módulos de programa que puedan ser codificados de forma independiente. En la fase de Codificación se elabora el producto fundamental de todo el desarrollo: los programas fuente.

Un elemento esencial dentro de la codificación es el lenguaje de programación que se emplea. Antes de iniciar la fase de codificación es fundamental establecer por escrito cual será la metodología de programación que se empleará por todos los miembros del equipo de trabajo.

En la fase de codificación pueden intervenir un gran número de programadores y durante un tiempo muy largo. Como parte de la metodología de programación y para garantizar la adecuada homogeneidad en la codificación se deben establecer las normas y estilo de codificación.

Un estilo homogéneo hace posible un trabajo coordinado entre los programadores ya que el entendimiento entre todos ellos resulta mucho más fácil. El empleo de estas normas facilita de forma considerable el mantenimiento posterior y la reusabilidad del software codificado.

Cuando los resultados de las pruebas no sean satisfactorios habrá que realizar cambios en la codificación. Ambos aspectos, codificación y pruebas, están muy relacionados.

La codificación se debe estructurar para facilitar su depuración y las modificaciones derivadas de las pruebas.

7.2 Objetivos

Estudiar la tarea de codificación.

Analizar los lenguajes de programación más utilizados y agruparlos por sus características comunes.

Estudiar los recursos de programación que nos ofrecen los lenguajes y las técnicas de implementación.

Repasar algunos criterios habitualmente utilizados para elegir un determinado lenguaje de programación.

7.3 Lenguajes de programación.

Los lenguajes de programación son el medio fundamental que tenemos para realizar la codificación.

Con los lenguajes actuales resulta más sencillo obtener un software de calidad, mantenible y reutilizable.

El conocimiento de las prestaciones de los lenguajes permite aprovechar mejor sus posibilidades y también salvar sus posibles deficiencias. No existe un único lenguaje ideal para todo y a veces es necesario trabajar con varios lenguajes para las distintas partes de un mismo desarrollo.

Desarrollo histórico. El estudio de la evolución histórica de los lenguajes es una buena manera de adquirir una visión panorámica de los mismos.

7.3.1 Lenguajes de primera generación

Son los ensambladores con un nivel de abstracción muy bajo. La programación en ensamblador resulta compleja, da lugar a errores difíciles de detectar, exige que el programador conozca bastante bien la arquitectura del computador y se necesita adaptar la solución a las particularidades de cada computador concreto.

Solo está justificada la utilización del ensamblador en aquellos casos en los que no se puede programar con ningún lenguaje de alto nivel.

7.3.2 Lenguajes de segunda generación

Su característica más notable es que no dependen de la estructura de ningún computador en concreto. Algunos de los lenguajes más representativos de esta generación son: Fortran, Cobol, Algol y Basic.

7.3.3 Lenguajes de tercera generación

Podemos verlos bajo dos enfoques:

1º: Programación imperativa: son lenguajes fuertemente tapados, con redundancias entre la declaración y el uso de cada tipo de dato, facilitando así la verificación en la compilación de las posibles inconsistencias.

- **Pascal:** tiene una tipificación de datos rígida, una compilación y separada deficiente.
- **Modula-2:** se separan la especificación del modulo de su realización concreta. Con esto tenemos una compilación segura y permite la Ocultación de los detalles de la implementación.
- **C:** es flexible. No tiene restricción en los datos.
- **Ada:** permite la definición de elementos genéricos, la programación concurrente de tareas y su sincronización y cooperación.

2º: Orientado a objetos, funcional o lógico: están orientados hacia el campo para el cual han sido diseñados.

Smalltalk: precursor de los lenguajes orientados a objetos.

C++: se incorpora a C, la Ocultación, las clases, herencia y polimorfismo.

Eiffel: es un lenguaje nuevo completamente orientado a objetos.

Permite la definición de clases genéricas, herencia múltiple y polimorfismo.

Lisp: precursor de los lenguajes funcionales Los datos se organizan en listas que se manejan con funciones, normalmente recursivas.

Prolog: representante de los lenguajes lógicos. Se utiliza en la construcción de sistemas expertos.

7.3.4 Lenguajes de cuarta generación

Tienen un mayor nivel de abstracción. Normalmente no son de propósito general; No son aconsejables para desarrollar aplicaciones complejas, por lo ineficiente del código que generan. Según su aplicación concreta son:

- ***Bases de datos:*** el mismo usuario genera sus informes, listados, resúmenes cuando los necesita.
- ***Generadores de programas***. Se pueden construir elementos abstractos fundamentales en un cierto campo de aplicación sin descender a los detalles concretos que se necesitan en los lenguajes de tercera generación. La mayoría generan aplicaciones de gestión en Cobol, y últimamente se han desarrollado herramientas CASE para diseño orientado a objetos, que se pueden utilizar para aplicaciones de sistemas y que generan programas en C, C++, o Ada.
- ***Calculo*:** hojas de cálculo, simulación y diseño para control. etc.
- ***Otros***: herramientas para la especificación y verificación formal de programas, lenguajes de simulación, lenguajes de prototipos.

7.4 Criterios de selección del lenguaje.

Para seleccionar el lenguaje deberían prevalecer los criterios técnicos, pero existen otros factores de tipo operativo que deben ser tenidos muy en cuenta. Estos factores son:

Imposición del cliente: es el cliente el que fija el lenguaje que se debe utilizar. Con esto se consigue disminuir los costes de desarrollo y mantenimiento que se producen cuando se utilizan cientos de lenguajes diferentes.

En otras ocasiones el cliente no es tan drástico y simplemente establece una relación reducida de lenguajes que se pueden usar en sus desarrollos.

Tipo de aplicación: cada día aparecen nuevos lenguajes asociados a un campo de aplicación concreto. Solo en determinadas aplicaciones estaría justificado el empleo de lenguajes ensambladores. En este caso, la parte realizada en ensamblador debería quedar reducida exclusivamente a lo imprescindible.

Disponibilidad y entorno: hay que ver que compiladores existen para el computador elegido. Un factor muy importante a tener en cuenta en la selección, es el entorno que acompaña al compilador. El desarrollo será más sencillo cuanto más potentes sean las herramientas disponibles. Por otro lado también se deben considerar las facilidades de manejo de estas herramientas.

Experiencia previa: siempre que sea posible es más rentable aprovechar la experiencia previa del equipo de trabajo. Cuando las condiciones de trabajo no se modifican el rendimiento aumenta y se disminuyen las posibilidades de error. Hay que tener en cuenta que la formación de todo el personal es una inversión muy importante.

Reusabilidad: es interesante conocer las librerías disponibles y tener herramientas que organicen las mismas para facilitar la búsqueda y el almacenamiento de los módulos reutilizables.

Transportabilidad: está ligada a que exista un estándar del lenguaje que se pueda adoptar en todos los compiladores.

Uso de varios lenguajes: aunque no es aconsejable mezclar varios en un mismo proyecto, hay ocasiones en las que las distintas partes del mismo resulta mucho más sencillas de codificar si se utilizan diferentes lenguajes.

7.5 Aspectos metodológicos.

Se repasan aspectos metodológicos que pueden mejorar la codificación.

7.5.1 Normas y estilo de codificación

Se deben fijar normas concretando un estilo de codificación, para que todo el equipo lo adopte y respete, en el que se deberán concretar lo siguiente:

Formato y contenido de las cabeceras de cada modulo. Formato y contenido para cada tipo de comentario. Utilización del encolumnado.

Elección de nombres.

Además hay que fijar las siguientes restricciones: Tamaño máximo de subrutinas.

Tamaño máximo de argumentos en subrutinas. Evitar anidamiento excesivo.

7.5.2 Manejo de errores

Durante la ejecución de un programa se pueden producir fallos.

Conceptos a tener en cuenta:

Defecto: errata de software. Puede permanecer oculto durante un tiempo indeterminado, si los elementos defectuosos no intervienen en la ejecución del programa.

Fallo: un elemento del programa no funciona correctamente, produciendo un resultado (parcial) erróneo.

Error: estado inadmisible de un programa al que se llega como consecuencia de un fallo. Típicamente consiste en la salida o almacenamiento de resultados incorrectos.

Actitudes respecto al tratamiento de errores son:

No considerar los errores: se exige que los datos introducidos sean correctos y que el programa no tenga ningún defecto. Cuando no se cumpla alguno de estos requisitos, no será posible garantizar el resultado correcto del programa.

Prevención de errores: es la programación a la defensiva en que el programa desconfía sistemáticamente de los datos o argumentos introducidos. Se evitan resultados incorrectos a base de no

producir resultados en caso de fallo. Una ventaja es que se evita la propagación de errores, facilitando el diagnostico de los defectos.

Recuperación de errores: cuando no se pueden detectar todos los posibles fallos es inevitable que en el programa se produzcan errores. En este caso se puede hacer un tratamiento del error con el objetivo de restaurar el programa en un estado correcto y evitar que el error se propague. Para esto se hacen dos cosas:

Detección del error: hay que concretar que situaciones se consideran erróneas y realizar las comprobaciones adecuadas en ciertos puntos del programa.

Recuperación del error: se adoptan decisiones sobre como corregir el estado del programa para llevarlo a una situación consistente.

Tenemos:

Recuperación hacia adelante: se identifica la naturaleza o el tipo de error y se toman las acciones que corrijan el estado del programa y pueda continuar su ejecución. Este esquema se puede programar mediante el mecanismo de excepciones. Por ejemplo, error de fuera de rango, etc.

Recuperación hacia atrás: trata de corregir el estado del programa restaurándolo a un estado anterior correcto a la aparición del error. En la nueva operación se parte de ese último estado correcto para obtener otro nuevo estado. Si ese nuevo estado es también correcto, la operación se da por terminada satisfactoriamente. En caso de error, se restaura el estado anterior y se trata de realizar la misma operación por un camino o algoritmo diferente.

Esta es la forma de operar de los sistemas basados en transacciones. Una transacción es una operación que puede terminar con éxito, modificando el estado del sistema, o con fallo, en cuyo caso la transacción se aborta y se restaura el estado inmediatamente anterior, de manera que la operación no produce ningún efecto. Los esquemas de transacciones mantienen la consistencia en las bases de datos.

Los sistemas que realizan una previsión o recuperación de errores se llaman tolerantes a fallos.

7.5.3 Aspectos de eficiencia

Con la potencia actual no se debe sacrificar la claridad en la codificación por una mayor eficiencia. La eficiencia se puede analizar desde varios puntos de vista:

Eficiencia en memoria: resulta suficiente recurrir a un compilador con posibilidades de compresión de memoria, y adopción de algoritmos eficientes.

Eficiencia en tiempo: adquiere su mayor importancia en sistemas de tiempo real. Esta eficiencia la podemos conseguir mediante la adopción de algoritmos eficientes y haciendo a veces un perjuicio a la eficiencia en memoria y utilizando técnicas de codificación como: Tabular cálculos complejos, Empleo de macros, Desplegado de bucles, Sacar fuera de los bucles lo que no haya que repetir, Evitar operaciones en coma flotante. Etc.

7.5.4 Transportabilidad del software

La transportabilidad permite usar el mismo software en distintos computadores actuales y futuros. Como factores esenciales de la transportabilidad se pueden destacar los siguientes:

Utilización de Estándares: un producto software desarrollado exclusivamente sobre elementos estándar es teóricamente transportable sin ningún cambio. La falta de estándares es uno de los problemas que dificulta la transportabilidad.

Aislar las Peculiaridades: la mejor manera de aislar las peculiaridades es destinar un modulo especifico para cada una de ellas. El transporte se resolverá recodificando y adaptando solamente estos módulos específicos al nuevo computador. Las peculiaridades fundamentales de los computadores suelen estar vinculadas a la arquitectura del computador y el sistema operativo.