TEMA 1

INTRODUCCIÓN

- Historia
  - *Evolución de las técnicas de programación*
  - *¿Qué es orientado a objetos?*
- Factores cruciales que miden la calidad del software
  - *Externos*
  - *Internos*
- La familia Orientada a objetos

Historia

Ideas centrales de la Programación Orientada a Objetos: Simula (1967)

Smalltalk-80 (72-76)

- reutiliza estas ideas
- genera un entorno gráfico basado en ellas
- lo populariza en las universidades.

Con máquinas más potentes, aparecen nuevos lenguajes

- *C++*
- *Eifel*
- *ObjetiveC*
- *Pascal Orientado a Objetos (Object Pascal),*
- *Modula Orientado a Objetos, (Oberon).*
- *Java*
- *Entornos visuales Orientados a Objetos*

Evolución histórica

Antes de los 60

- El diseño y programación de aplicaciones era considerado como un arte.
- No había método para el diseño ni la implementación de software
- No había medios para mantener los programas
- Si algo iba mal, era mejor sustituirlo que revisarlo.
- La mayoría del software escrito en esta época tuvo que ser posteriormente reescrito.
- Esto llevo a la *crisis del software*

60 - . Metodologías

- Surge la programación procedural
  - *Aumentar la legibilidad*
  - *Bajar costes de mantenimiento*
  - *Acelerar el proceso de desarrollo*
- Programación estructurada
  - *Pone límites al programador*
  - *Abstracciones de datos*
  - *Abstracciones de control*
- Programación modular
  - *Software en módulos*
  - *Reutilización de módulos*
  - *Una aplicación es un conjunto de módulos*

- La programación modular introduce el concepto de Tipo abstracto de datos. Los datos empiezan a jugar un papel importante

- Los proyectos iban creciendo en tamaño

- Abarcaban más y más campos

- Un proyecto se realizaba por una equipo cada vez mayor de personas
  - Cuanto mayor era el equipo, menor era el rendimiento individual.
  - La razón: la cantidad de esfuerzo en coordinación.

La POO nace con la idea de que el trabajo pueda ser dividido de forma coherente en pequeñas unidades independientes de manera que la necesidad de coordinación sea mínima.

Brand J. Cox (Objetive-C) acuñó el término de software-IC

Ejemplo:

- Un diseñador hardware necesita construir una placa con cierta funcionalidad.
  - Estudia los componentes que necesita.
  - Si para cierta acción no existe un componente, tiene dos alternativas:
    - *Proponer a un equipo el diseño de un componente que resuelva el problema*
    - *Producir el efecto deseado por otros medios.*

- En cualquier caso, el trabajo del diseñador no es construir todos los componentes.

- La POO divide a la comunidad de programadores en dos tipos:
  - *Los productores de componentes*
  - *Los consumidores de componentes*
- Los equipos que crean componentes las someten a pruebas de depuración de manera que al usuario le llegan componentes en perfecto estado
- Los creadores de componentes ofrecen al usuario un interfaz por el que se pueden comunicar con el componente. Este interfaz impide conocer el interior del componente. Este permite que cambios internos en el componente no afecten a los consumidores de dichas componentes
- La misión del programador es conocer los componentes que existen en el mercado, y combinarlos adecuadamente para conseguir el fin.

¿Qué es Orientado a objetos?

El software mantiene

- I. Estructuras de datos
- II. Comportamientos

El término "Orientado a objetos" significa que el sofware se organiza como una colección de objetos. Los objetos representan abstracciones del mundo real

- Liga de forma robusta I y II

En oposición a la programación convencional

- Liga de forma debil I y II

Desarrollo orientado a objetos

Nueva forma de pensar

- Basada en abstracciones del mundo real. Los objetos representan a estas abstracciones

La esencia:

- La identificación y organización de conceptos del dominio de la aplicación y no de su representación final en un lenguaje de programación
- Es un proceso intelectual independiente de cualquier lenguaje
- Se tratan temas conceptuales de primer plano, no de implementaciones
- Aplicable a
  - *Diseño. Representación*
  - *Programación. Implementación*
  - *Bases de datos. Persistencia*

La consecuencia

- Posibilidad de representar directamente las entidades del mundo real en los entornos informáticos, sin necesidad de deformarlos ni descomponerlos.
- Posibilidad de reutilizar y extender las aplicaciones existentes, por ejemplo, a partir de librerías especializadas y fácilmente modificables.
- Trabajo con entornos de desarrollo potentes, por ejemplo para la depuración y el seguimiento de ejecuciones.
- Disponibilidad de herramientas de comunicación hombre-máquina visuales de gran calidad
- Facilidades de prototipado rápido de aplicaciones
- Facilidad de utilización de paralelismo en el momento de la implementación.

Factores cruciales que miden la calidad del software

☐ Factores externos

- Percibibles por el usuario (de cualquier nivel)

Corrección	Habilidad de ajustarse a los requerimientos
Robustez	Habilidad de funcionar aún bajo condiciones anormales
Extensibilidad	Facilidad de adatarse a cambios en los requerimientos
Reutilidad	Habilidad para ser reutilizado, todo, o en parte, en nuevos desarrollos
Eficiencia	Habilidad en el uso óptimo de los recursos hardware

Robustez

- La robustez frente a errores internos se puede compensar con la eficiencia
- La robustez frente a errores del usuario nunca se puede sacrificar
- Protección contra errores
  - *Toda función que admita valores del usuario, debe validar las entradas*
  - *Fallecer de forma "graciosa"*
  - *Un lenguaje con comprobación estática de tipos proporciona mayor protección*
- Optimizar después de que funcione
  - *A veces, la optimización va en detrimento de la extensibilidad y la reutilidad*
  - *Estudiar detenidamente el programa antes de optimizar*
  - *No incluya argumentos no verificables*

Robustez

- Evitar límites predefinidos
  - *Utilizar memoria dinámica si la estructura no tiene límites predefinidos*
  - *Tener cuidado con los límites de nombres de usuario, nombres de archivo, etc.*
  - *Los límites no se conocen en la etapa de diseño*
- Disponer el programa para la depuración y monitorización de rendimiento
  - *Poner puntos de depuración y rendimiento*
  - *El nivel de depuración dependerá del lenguaje usado*
  - *Incluir código para recoger estadísticas para comprender mejor el funcionamiento*

Extensibilidad

- "El software se extiende en formas que jamas podrían esperar los desarrolladores"
- Encapsular y ocultar estructuras de datos de una función
  - *Si exporta la estructura, limita la capacidad para cambiar el algoritmo en un futuro*
- Evite demasiadas indirecciones
- Evitar sentencias de caso basados en el tipo de un dato
- Distinguir funciones internas y externas

Reutilización

- Reduce costos de diseño, codificación y comprobación
  - *Se amortiza el esfuerzo en varios diseños*
- Incrementa la comprensión
  - *Disminuye la probabilidad de fallo*

Formas de reutilización

- Compartir un código recien escrito dentro de un proyecto en un proyecto nuevo
  - *Buscar rutinas comunes, etc. y utilizar las capacidades del lenguaje para compartirla*
- Utilizar código previamente escrito, en proyectos nuevos
  - *Requiere anticipación e inversión*
  - *Es difícil reutilizar un conjunto de código aislado*
  - *Se reutilizan subsistemas cuidadosamente pensados*

Reglas de estilo de la reutilización

- Mantener la coherencia de las funciones
  - *Debe realizar una única misión o un grupo de misiones relacionadas. Si no es así, partirla*
- Mantener pequeñas las funciones
  - *Si es grande, dividirlo en piezas. Quizás la función completa no sea reutilizable pero si una de sus piezas*
- Mantener la congruencia de las funciones
  - *Funciones similares deben tener parecido sus nombres, condiciones, orden de argumentos, tipos de datos, valor devuelto y condiciones de error*
  - *No coherente en C (puts y fputs)*

Reglas de estilo de la reutilización

- Separar la política de la implementación
  - *funciones políticas son las que toman decisiones y lanzan las funciones no políticas. Se encargan de comprobar estados y condiciones de error*
  - *funciones no políticas hacen cálculos e implementan algoritmos complejos sin decidir por qué lo hacen. Proporcionan estado pero no actúan por sí mismas*
- Proporcionar una cobertura uniforme
  - *Dar funciones para las necesidades actuales y futuras*
- Generalizar la función lo más posible
  - *generalizar los tipos de argumentos, las condiciones previas y restricciones*
  - *Controlar los valores extremos o nulos*

Reglas de estilo de la reutilización

- Evitar la información global
  - *Minimizar referencias externas. La alusión de un elemento global impone un contexto*
  - *Pasar la información como argumento*
- Evitar los modos
  - *Funciones que cambien bruscamente de comportamiento en función del contexto son difíciles de reutilizar*

Factores cruciales que miden la calidad del software

☐ Factores internos

- Percibibles por los programadores del producto

Formalidad

[Correción + Robustez]

- *Debe ser fácil crear software que funcione correctamente y fácil garantizar que esto es así*

Modularidad

[Reutilización + Extensibilidad]

- *Debe crearse el menor software posible*
- *El software debe ser fácil de modificar*

Modularidad

La construcción de programas por medio de la unión de pequeñas piezas de código.

No implica beneficio real en términos de extensibilidad y reutilización a menos que las piezas mencionadas (los módulos) sean:

- autónomos
- coherentes
- robustos

Para hacer un módulo deben tenerse presente:

- La capacidad de composición y descomposición del módulo
- La comprensión que ofrece cada módulo por separado
- La protección y continuidad (pequeños cambios en requerimientos producen pequeños cambios en el módulo)

Modularidad

Para satisfacer esto, un lenguaje debe incorporar mecanismos para que:

- Un módulo corresponda a una unidad sintáctica del lenguaje
- El número de entidades con las que se comunica un módulo pueda estar controlado.
- La información compartida sea mínima y debe hacerse de forma explícita
- La información de un módulo sea privada a menos que se desee lo contrario

Composición de un sistema software

Sofware: Conjunto de operaciones que realizan acciones sobre datos.

Dos formas de abordar la realización de un sistema:

1. Estructuras modulares basadas en procedimientos (acciones)
2. Estructuras modulares basadas en datos (objetos)

El criterio de reutilización defiende a la segunda en contra de la primera

"Es difícil reutilizar un módulo basado en procedimientos sin tener en cuenta los datos sobre los que actúa"

La familia Orientado a objetos

Programación

- Lenguajes
  - *Puros (Smalltalk, Eifell, Java, Actor, etc.)*
  - *Extensiones (C++, Object pascal, Cobol, etc.)*
  - *Códigos portables (Java)*

Bases de datos

- Extensiones de las relacionales (Ingres, Oracle, etc.)
- Nuevos sistemas (Object Store, Versant, O2, Poet, etc.)
- Intentos de normalización de objeto persistente

Sistemas operativos orientados a objetos

- Sistemas completos (Next)
- Interfaces (X/Windows, Motif, AWT)