Tema 2: conceptos avanzados de orientación a objetos:

Herencia

Reutilización y encapsulamiento: programación por responsabilidades, delegación.

"Una clase A hereda de otra B cuando los objetos de B son una especialización de A (relación **es-un**)"

Esto quiere decir:

B recoge la funcionalidad de A y, opcionalmente, la modifica.

- Normalmente amplía:
  La clase debe abstraer el comportamiento común a todos los objetos de esa clase con lo que las especializaciones lo serán porque hacen algo más.
- Será posible reducir. Esto es útil cuando alguna clase nos viene dada y necesitamos reducir su funcionalidad (Problema de semántica)
- También es posible no cambiarlas:
  Cambiar nombre e identificación: se puede restringir determinada comunicación a clases de objetos. Nos da expresividad para definir las relaciones.
  - Semántica
  - Futuros cambios
- Es posible sólo cambiar métodos: reescritura.

Clase abstracta:

Concepto
Ejemplos: redefinición

- Métodos diferidos

Clase final / método final

Concepto

Superclase

Concepto

- Tendremos acceso a las superclases
- Una variable podrá contener objetos de su clase o de sus subclases
  - ¿Por qué?
    + poder expresivo
    + consistente con la filosofía de orientación a objetos
  - ¿Por qué no con superclases? Error conceptual. Ejemplo…
- Constructores y destructores (Dibujo)

Ámbitos

- Público
- Privado
- Protegido

Herencia múltiple

Un objeto es especialización de varias clases distintas

Problemas de ambigüedad en llamadas:

Soluciones:

- interpretación explícita…
- mayor abstracción: dibujable

Constructores y destructores

- Ejecución en cascada

Interfaces: algunos lenguajes no permiten la herencia múltiple (JAVA)

Justificación: Mantenibilidad del código

Pérdida de expresividad. Redundancia…Peor reutilización…

Interfaz: conjunto de métodos públicos.

- Se pueden implementar varios interfaces (visto desde fuera es como la herencia múltiple)
- Desde dentro cambiará la implementación.
- Con los interfaces no es necesario crear una clase explícitamente: basta declarar los métodos públicos.
- Los interfaces también heredan y sí permiten la herencia múltiple. (caso del diamante): al eliminar los atributos, no hay problemas de copias…
- Se pueden declarar variables como de un interfaz en lugar de una clase, de modo que pueden contener objetos de cualquier clase que implemente ese interfaz o de las subclases.
- En lenguajes con herencia múltiple (dónde no existen los interfaces como tales), basta declarar una clase con sólo los métodos públicos y todos abstractos aunque hay problemas de nomenclatura explícita.

Polimorfismo

"Varias formas" = distintos comportamientos dependiendo de las clases. Principio de abstracción. Amplía el concepto de procedimiento y suaviza la asignación de tipos fuerte.

Significa al menos:

1) Sobrecarga de operadores
2) Sobrecarga de funciones
3) Tipos genéricos o parametrización de tipos
4) Ampliación de las reglas de compatibilidad de tipos y de la asignación en lenguajes de tipos
5) Polimorfismo de mensajes

Posible definición:

"Posibilidad de asociar a un mismo componente (identificador) sintáctico diferentes significados funcionales."

Contexto: En todos los casos existe un contexto que determina el valor semántico exacto de cada identificador. Esta determinación se puede realizar:

Estático: (en tiempo de compilación y enlazado)

Compilación:

Contexto = tipo definido en la declaración de objetos
Contexto = tipo de los identificadores
Contexto = tipo de operandos (en fuentes que se compilan)
Contexto = número y tipo de los argumentos (en fuentes que se compilan)
comprobación de asignaciones
comprobación de operaciones (funciones, operadores, métodos) implementados en las fuentes disponibles

Enlazado:

Contexto = tipo de operandos (en código objeto que se enlaza)
Contexto = número y tipo de los argumentos (en código objeto que se enlaza)
comprobación de operaciones (funciones, operadores, métodos) implementados en otras librerías

Dinámico: (en tiempo de ejecución)

Contexto = clase a la que pertenece el objeto
se resuelve en el momento de la utilización (de método o de variable).
Sólo es posible conocer la clase concreta en tiempo de ejecución.

La implementación del polimorfismo afectará básicamente al modo en que el lenguaje realiza **dos tareas**:

Verificación de tipos: determinar si una operación es válida en un contexto
Ligadura: determinar la semántica de una operación

Ambas pueden ser tanto estáticas como dinámicas…

Genéricos (dentro de estático)

Reutilización: definir plantillas sobre tipo genéricos (no se conocen a priori) y concretar los tipos en tiempo de compilación al instanciar la plantilla.
Justificación: eficiencia (en algunos como C++). Simplicidad en el código (no hay que hacer tantos moldes)
Cuidado: modificación de un genérico implica recompilación de todos los programas ya que el código se incluye en el programa que lo utiliza (no es verdad para JAVA…)

¿Qué pasaba antes con el polimorfismo?

C: punteros a funciones y punteros a void: a mano, pero se podía hacer…