Programación Orientada a Objetos

Tema 3. Reutilización del Software

Reutilización del Software

- Desde los 60's se habla de reutilizar software
- En la actualidad se tiende firmemente hacia *“el desarrollo de software como industria basada en componentes”*
- La reutilización veremos que:
  - Beneficia desarrolladores
  - Beneficia consumidores
  - Beneficia la calidad del producto final

La reutilización beneficia a toda la industria

Beneficios (esperados)

Los beneficios más destacables

1. Rapidez de desarrollo:
   - La “oportunidad” de nuestro software mejorará
2. Fiabilidad
3. Eficiencia
4. Menor mantenimiento
5. Mayor consistencia en nuestros desarrollos
   - Pensarlo reutilizable nos hará hacerlo mejor
   - Haremos mejores diseños e implementaciones con idea de que van a ser reutilizados
6. Mejora costes. Y es una inversión rentable
7. **Se espera la mejoría de todos los factores de calidad**

Productor - Consumidor

La reutilización tiene 2 aspectos:

- Producción/desarrollo
  - Desarrollar componentes es una disciplina difícil
  - Antes de desarrollar hay que consumir y aprender
- Consumo
  - Estudia, aprende, ….. e imita:
    - Estilo
    - Diseño
    - Propiedades de un componente
    - Etc...

Cómo y qué reutilizar

- El personal de desarrollo (no propiamente software)
- Metodologías de programación:
  - POO: objetos, herencia, polimorfismo, plantillas, etc.
  - También en otras metodologías
- Tendencia a mayor reutilización de diseños y especificaciones
  - Es un proceso más complejo
  - Patrones de diseño (tema siguiente)
- Mejoras constantes en estructuras de reutilización:
  - En lenguajes de programación
  - Repositorios
  - Colecciones de componentes

copiar y pegar el código **no es suficiente**
el acceso al código fuente **no es suficiente**

La repetición

En una presencia habitual durante el desarrollo

- Detectarla (también factorizando comportamientos comunes)
- Entenderla en su sentido abstracto
- Describirla bien
- Para, a continuación, **desarrollarla como módulo**

Módulo ideal para la reutilización

- Desempeña tarea clara, concreta y bien descrita
- Suficiente nivel de abstracción que permita su instanciación en otros problemas: oculta detalles irrelevantes
- Facilidad de uso
- Tamaño razonable
- Debe ser buena pieza de desarrollo en si misma (de buena calidad)
- Debe cumplir los criterios, reglas y principios de una **buena descomposición modular** que ya hemos estudiado en el tema 2 (no confundir descomposición con reutilización)

Obstáculos

- Falta de formación
- Falsos mitos:
  - “Mejor hacerlo nosotros”
  - “Es más barato hacerlo nosotros”
  - “Nosotros todo, para que el cliente dependa de nosotros”
- Fallo en el diseño y gestión de un buen repositorio de componentes

Más obstáculos

- Reticencias a la reutilización más allá de nuestro propio equipo de desarrollo
  - Desconfianza de otros desarrollos
  - Reticencias a distribución del fuente
  - Muchas veces el esfuerzo es la interfaz no los módulos

Las **comunidades de desarrollo** tienen muchas ventajas: nos simplifica el desarrollo, detección/corrección de errores, portado a otras plataformas, mejora interfaz, mayor difusión, más posibles clientes, etc.

Comunidades de desarrolladores

- Ventajas:
  - Simplifican el desarrollo
  - Detección/corrección de errores y mantenimiento
  - Partabilidad a otras plataformas
  - Mejora de la interfaz
  - Mayor difusión… clientes
  - Aprendizaje, etc.

Estructuras de reutilización

- Rutinas, bibliotecas de rutinas o librerías
  - Unidad de software (subrutina, subprograma)
  - Adaptable solo con argumentos
  - Problemas muy concretos y complejos pero de reducido tamaño
  - No es suficiente

no incorporan elementos potentes de reutilización,
no facilitan la derivación de nuevos trabajos

Estructuras de reutilización

- Paquete
  - Más avanzado que rutina
  - Utilidades de todo tipo sobre un tema
  - Rutinas, variables, tipos, declaraciones, espacios de nombres, distintos componentes, etc.
  - También permiten un uso parcial de su funcionalidad
  - Solución completa o amplia a un problema
  - Ofrecen poco más allá de su funcionalidad
  - Ej: visión artificial, estadística, control de usuarios, encriptación, etc.

Estructuras de reutilización

- Framework/entorno de desarrollo
  - Facilitan el desarrollo de cualquier aplicación (dentro de su ámbito)
  - Puede incorporar elemento estructurales de diseño de aplicaciones.
  - Pueden especializarse en plataformas: web, disp. móviles, etc.
  - Ej.: Rails, Django, Symfony, Flask, JDK, Android, Apple, etc.

La POO y la reutilización

Existen diversos elementos específicos de la POO centrados en la reutilización.

En C++ hemos visto:

- Las clases y los objetos
- La herencia
- El polimorfismo: estático y dinámico
- Genericidad (*templates* de función y de clase)

Resumen

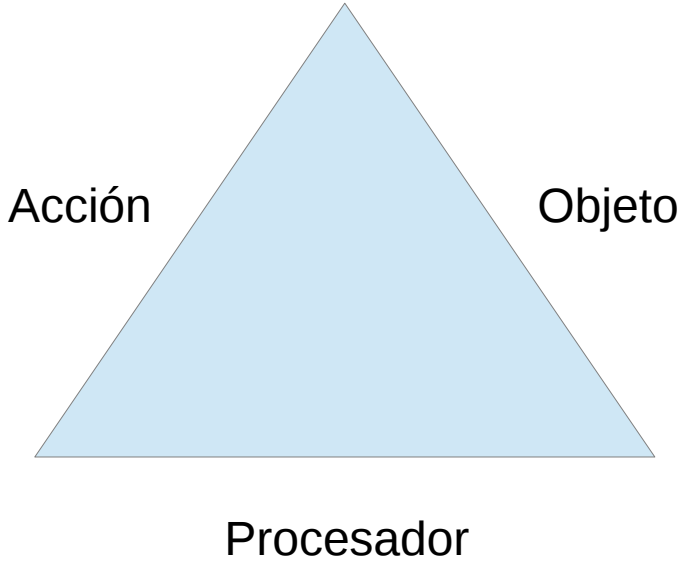
- *“el desarrollo de software como industria basada en componentes”*
- La POO:
  - Objetos y clases para una programación más efectiva y natural.
  - Estructuras de reutilización para sacar provecho de los beneficios de la reutilización.

Programación Orientada a Objetos

Tema 4: Tecnología OO

Tema 4: Tecnología OO

- Triángulo básico de la computación, las tres fuerzas de la computación:

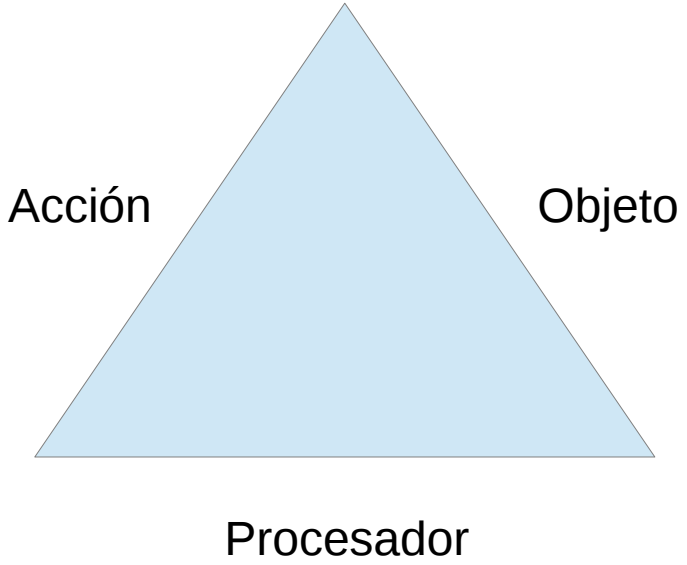


(A veces se denota como: acción = funciones, objeto=dato)

[Bertrand Meyer 99]

La visión orientada a objetos

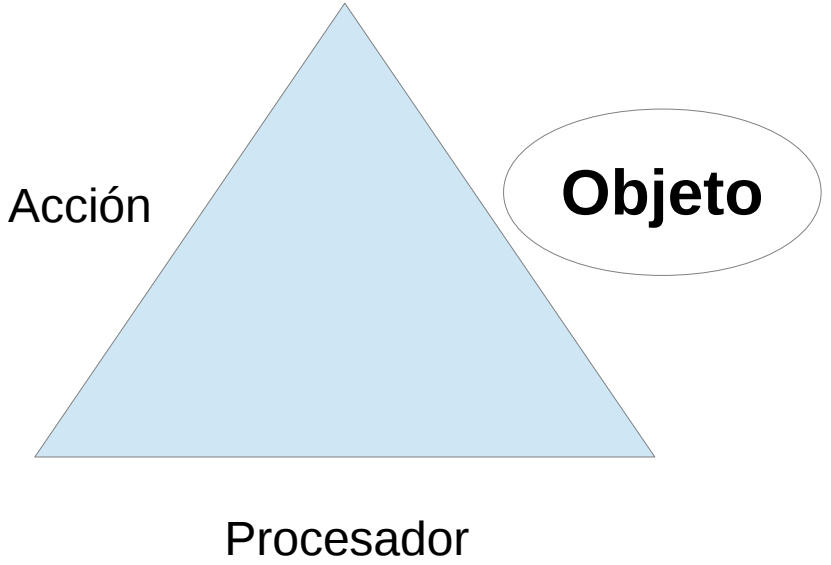
- Triángulo básico de la computación, las tres fuerzas de la computación:



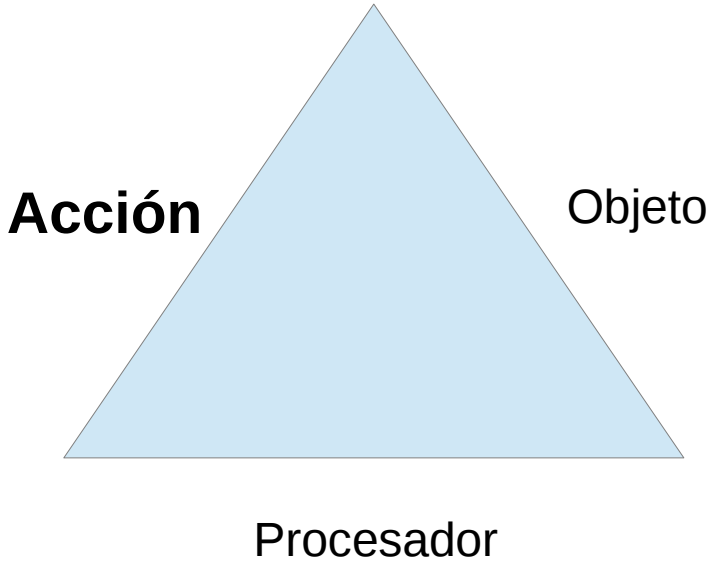
(A veces se denota como: acción = funciones, objeto=dato)

La visión orientada a objetos

... es la modularidad centrada en los objetos



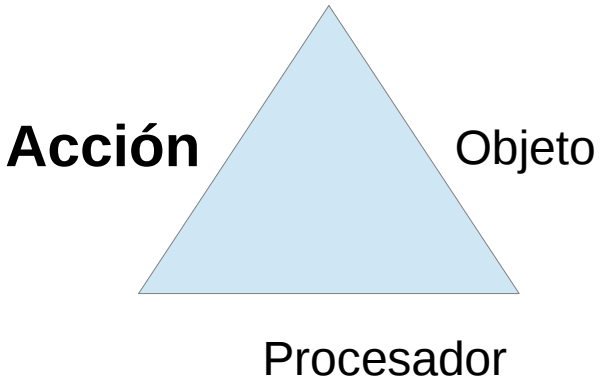
Enfoque tradicional



está basado en la acción

Enfoque tradicional

- Basada en funciones
- Decisiones de diseño prematuras y estratégicamente poco dispuestas a características de calidad deseables
- Excesiva importancia a la interfaz
- Se ha demostrado poco adecuado para sistemas de medio/gran tamaño



Enfoque OO

- Basada en objetos
- Los módulos se deducen de los objetos que se manipulan
  - No hay diseño prematuro basado en una función
  - La interfaz, una función aparte más
- **No es:** *qué hace el sistema,* **es:** *¡a qué se lo hace!*
- Diseño pensando en reutilización
- … y en la extensibilidad
- Más adecuado para grandes sistemas

Las clases y los objetos

- Clase: TAD equipado con una implementación
- Cliente de una clase
- Acceso a una clase
  - Métodos o mensajes
  - La interfaz (también se denomina exportar métodos o atributos)
  - Atributos vs funciones ¿conviene distinguir?

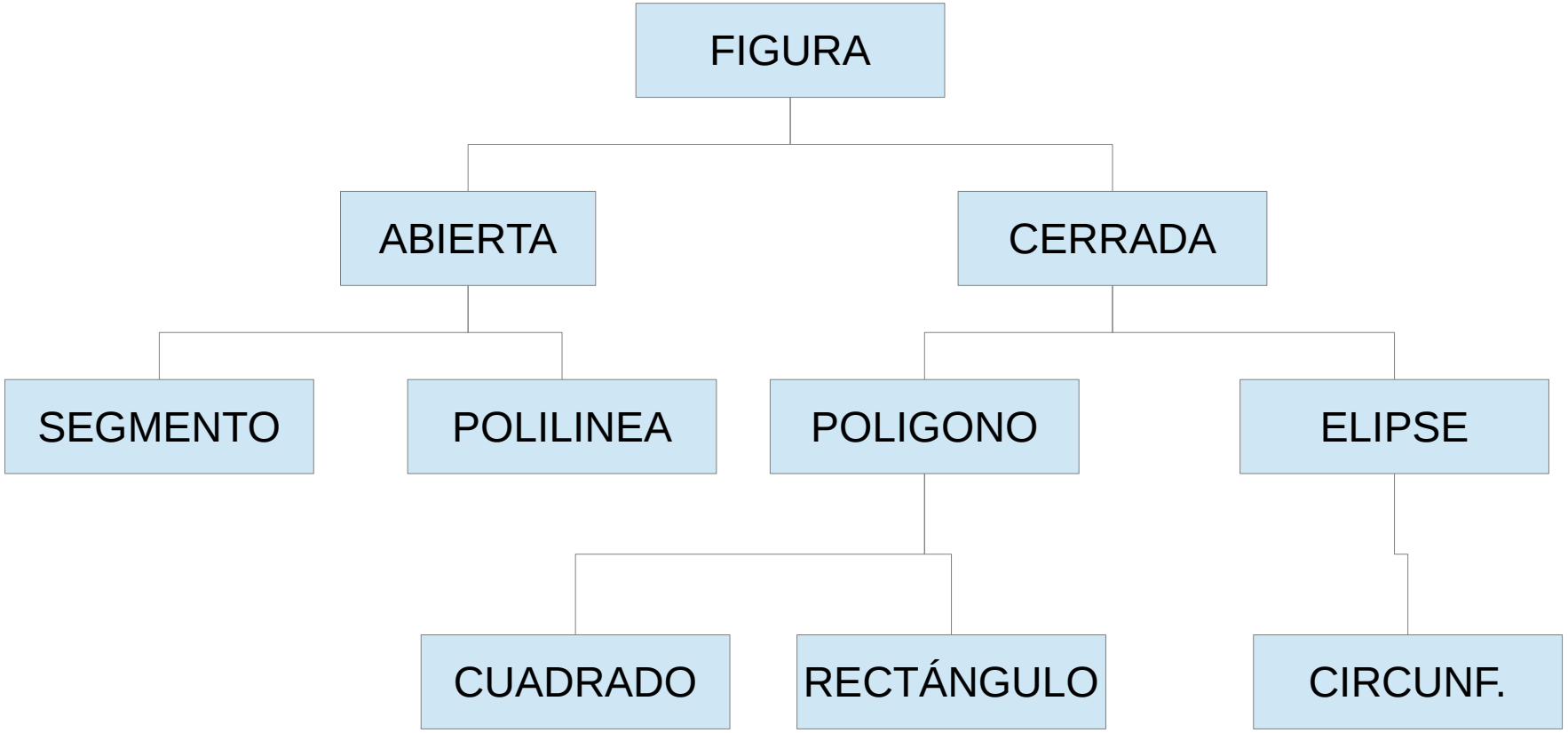
(→ siguiente figura.....)

Las clases y los objetos

- Atributos vs funciones ¿conviene distinguir?
  - Los atributos están más ligados a representación interna
  - Forzar al cliente a tener en cuenta esta diferencia añade dificultad. Para el cliente debe ser uniforme.
  - El acceso mediante atributos dificulta futuros cambios
  - Las funciones inline ayudan

Herencia

- Estructura de reutilización jerárquica
- Asigna comportamientos por niveles (descomposición)
- Clase base (ancestro, superclase, padre, madre...) y clase derivada (descendientes, subclase...).



Herencia

- Clase **diferida** (abstracta, ...):
  - Alguno de sus componentes es diferido: no está implementado en la propia clase
  - Determinan la interfaz/comportamiento de sus descendientes
  - Describen un interfaz genérico
  - C++: tiene funcionas virtuales o funciones virtuales puras (clases abstractas)
- La clase no diferida es una clase **efectiva**

Polimorfismo

- Definición: ocurre cuando se dispone del mismo interfaz sobre objetos de distinto tipo
- Hay varios tipos de polimorfismo
  - Estático (sobrecarga de funciones y op. en C++)
  - Estático paramétrico (plantillas/templates en C++)
  - Dinámico (de subtipo, polimorfismo en tiempo de ejecución en C++, vinculación dinámica)

Polimorfismo

- Polimorfismo estático (o *ad hoc*)
  - Forma más simple de polimorfismo
  - También llamado: sobrecarga de funciones, sobrecarga de operadores
  - Se define una función/operador diferente para cada tipo

la función/operador se hace caso a caso, *ad hoc*, para cada tipo

Polimorfismo

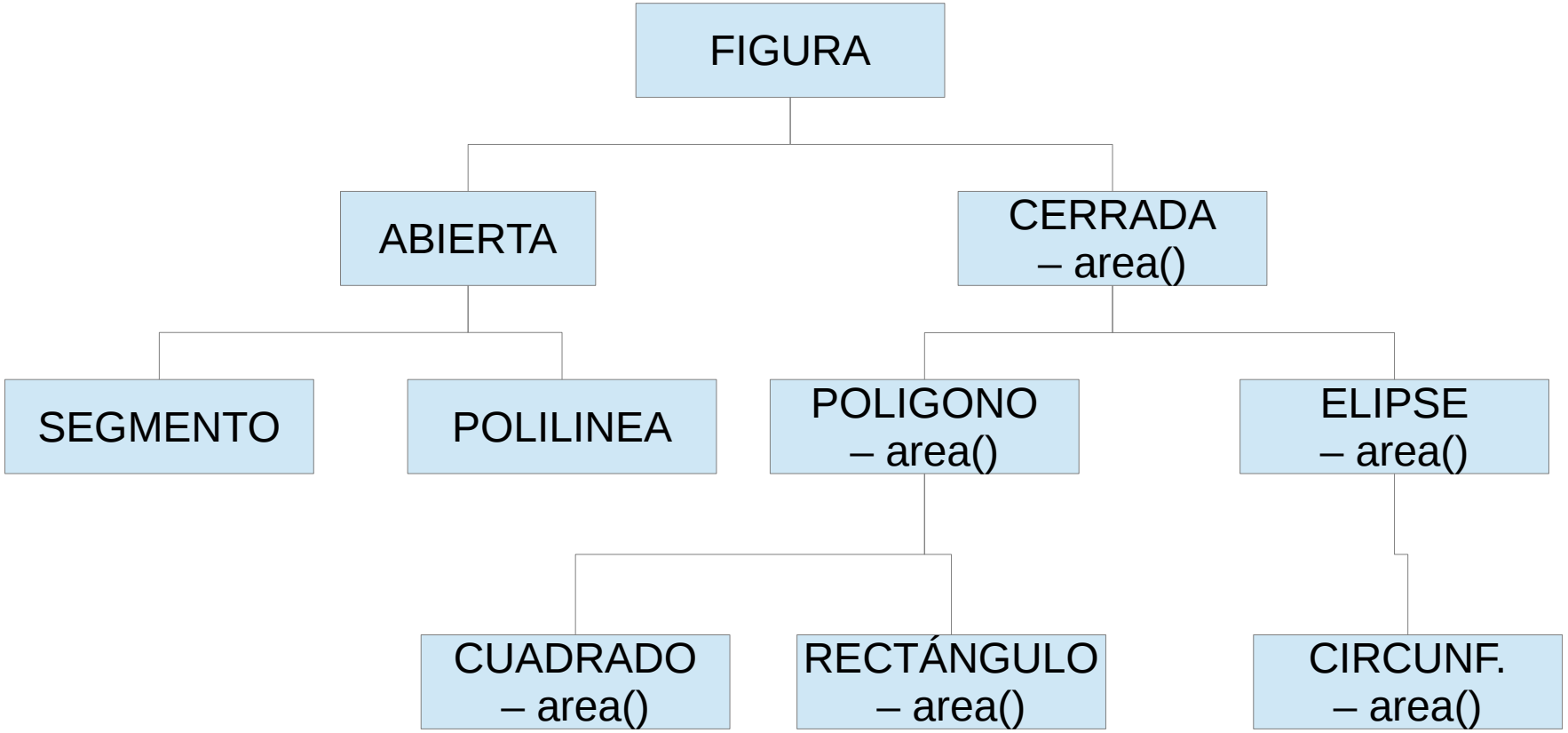
- Polimorfismo paramétrico (estático)
  - El código usa un tipo genérico que se especifica como parámetro cuando se instancia
  - La interfaz es la misma indep. del tipo instanciado
  - Plantillas de clase y de función en C++
    - C++ *Standard Template Library* (STL)
    - *Function templates*
    - *Class templates*

Polimorfismo

- Polimorfismo de subtipo (subtipado)
  - Vinculación dinámica
  - En tiempo de ejecución
  - Permite construir código genérico indep. del tipo
  - C++: punteros a la clase base y funciones virtuales
  - Ej: la clase *Figura* en los apuntes de C++

Polimorfismo

- En el polimorfismo de subtipo una familia de clases tienen el mismo comportamiento:
  - Los subtipos comparten interfaz
  - Potencia semántica, expresiva, sencillez
  - Acceso uniforme, etc.
  - Se puede escribir el mismo código para todas ellas



Sistema Software Orientado a Objetos

- Conjunto de clases (ensamblado de clases)
- Desde la **clase raíz** hasta el **despliegue** de todo el sistema
- ¿Y el programa principal?
  - En OO no juega un papel tan “principal”
  - Puede ser un simple método que inicie el despliegue
  - ¿cuál es el programa principal de un SO, una compleja aplicación, una web, etc. ?
  - Son más bien los servicios que ofrece, lo que describe el sistema
- Sistema **cerrado**: si contiene todas las clases que necesita la clase raíz

Gestión de memoria

- Memoria Estática
- Memoria basada en pila
- Memoria Basada en montículo (libre, dinámica)
- Desconexión/*detachment* de memoria
- Objetos inalcanzables:
  - Problemas de memoria
- LP modernos incorporan: **recolector automático de basura**
  - Todos los objetos que se recolecten deben ser inalcanzables
  - Todos los objetos inalcanzables deben ser recolectados

Conclusiones

- La reutilización clave del desarrollo de software actual
- Facilidad de la POO para la reutilización
- Formación
- Hay que explotar en profundidad la orientación a objetos para lograr la reutilización