Tema 1

Introducción al paradigma Orientado a Objetos

- El progreso de la abstracción

Abstracción: supresión intencionada de algunos detalles de un proceso o artefacto, con el fin de destacar más claramente otros aspectos, detalles o estructuras.
Mediante la abstracción creamos MODELOS de la realidad.

- Lenguajes de programación y niveles de abstracción
    - Perspectiva funcional: Ensamblador, Procedimientos
    - Perspectiva de datos: Paquetes, tipos de datos abstractos (TAD)
    - Perspectiva de servicios: Objetos (mensajes, herencia y polimorfismo)

Espacio del problema	Espacio de la solución
Lenguaje Orientado a Objetos LOO PURO (Smalltalk, Effiel)	Lenguaje ensamblador Lenguaje imperativo (C, fortran, BASIC) Lenguajes específicos (LISP, PROLOG)

LOO Híbridos (Multiparadigma)
C++, Object Pascal, Java

- Principales paradigmas
    - Paradigma: Forma de entender y representar la realidad. Conjunto de teorías, estándares y métodos que, juntos, representan un modo de organizar el pensamiento.
    - Paradigma funcional: lenguajes que describen procesos (Lisp, Haskell, ML)
    - Paradigma lógico: prolog.
    - Paradigma imperativo: C, Pascal
    - Paradigma Orientado a Objetos: Java, C++, Smalltalk
- Mecanismos
    - Ocultación de información: Omisión intencionada de detalles de implementación tras una interfaz simple.
    - Encapsulación: División estricta entre la vista interna de un componente (objeto) y su vista externa.
        - Interfaz: Qué hace el objeto. Visión externa.
        - Implementación: Cómo lo hace. Visión interna.
        - Favorece la intercambiabilidad.
        - Favorece la comunicación entre miembros del equipo de desarrollo y la interconexión de los artefactos resultantes de cada uno.
- Paradigma Orientado a Objetos
    - Es una metodología de desarrollo de aplicaciones en la cual estas se organizan como colecciones cooperativas de objetos, cada uno representa una instancia de alguna clase y cuyas clases son miembros de jerarquías de clases, unidas mediante relaciones de herencia.
        - Herramienta para resolver la crisis del software
        - Paradigma de programación dominante

- Escala muy bien
- Proporciona un modelo de abstracción que razona con técnicas que la gente usa para resolver problemas.

  - Estructurado en:
    - Agentes y comunidades: Un programa orientado a objetos se estructura como una comunidad de agentes que interaccionan (objetos). Todo objeto juega un rol y proporciona un servicio que utiliza otro miembro.
    - Mensajes y métodos: Al objeto se le envían mensajes para que realice una acción y el objeto selecciona un método (paso de mensajes).
    - Responsabilidades: protocolo. El comportamiento de un objeto se describe en términos de responsabilidades.
    - Objetos y clases: un objeto es una encapsulación de un estado (datos) y comportamiento (operaciones). Se agrupan en clases. Un objeto es una instancia de una clase.
    - Jerarquía de clases: generalización – implementación -> herencia. Mamífero -> Humano -> Dependiente -> Florista
    - Enlace de método: instante en el cual una llamada a un método es asociada al código que se debe ejecutar
      - ❖ Enlace estático: tiempo de compilación
      - ❖ Enlace dinámico: en tiempo de ejecución.

- Características básicas de LOO
  - Todo es un objeto
  - Cada cual construido a partir de otros
  - Todo objeto es instancia de una clase
  - Objetos de la misma clase pueden recibir mismos mensajes, pues tienen el mismo comportamiento
  - Se organizan según una jerarquía de herencia
  - Se comunican mediante el paso de mensajes
- Características opcionales de LOO
  - Polimorfismo: capacidad de referenciar elementos de distinto tipo en diferentes instantes (enlace dinámico).
  - Generacidad: definición de clases parametrizadas que definen tipos genéricos. Templates(C++) – generics(Java)
  - Gestión de errores: tratamiento de condiciones de error mediante EXCEPCIONES.
  - Aserciones: expresiones que especifican qué hace el software en lugar de cómo
    - Precondiciones: propiedades que deben ser satisfechas cada vez que se invoca el servicio.
    - Postcondiciones: propiedades que deben ser satisfechas al finalizar la ejecución de un servicio.
    - Invariantes: Expresan restricciones para la consistencia global de sus instancias.

- Tipado estático: se asegura en tiempo de compilación que un objeto entiende los mensajes que se le envían. Evita errores en tiempo de compilación.
- Recogida de basura (garbage collection): libera automáticamente la memoria de los objetos que no se utilicen.
- Concurrencia: permite que diferentes objetos actuen al mismo tiempo usando diferentes threads o hilos de control.
- Persistencia: propiedad por la cual la existencia de un objeto trasciende la ejecución del programa (implica el uso de BBDD).
- Reflexión: capacidad de un programa para modificar su propio estado, estructura y comportamiento.

- Historia de los LOO

Año	**Lenguaje**	Creadores	Observaciones
1967	**Simula**	Norwegian Computer Center	***clase, objeto, encapsulación***
1970s	**Smalltalk**	Alan Kay	***método y paso de mensajes, enlace dinámico, herencia***
1985	**C++**	Bjarne Stroustrup	Laboratorios Bell. Extensión de C. Gran éxito comercial (1986->)
1986	**1ª Conf. OOPSLA**		**Objective C, Object Pascal, C++, CLOS**,... Extensiones de lenguajes no OO (C, Pascal, LISP,...)
`90s	**Java**	Sun	POO se convierte en el paradigma dominante. Java: Ejecución sobre máquina virtual
`00->	**C#, Python, Ruby,...**		Más de 170 lenguajes OO... Lista TIOBE (Del Top 10, 8 o 9 son OO)

  - Los más implementados son Java, C++ y PHP. También están C#, Python, Objective-C. Híbridos (LOO-Procedimental) PHP, C++, VisualBasic, JavaScript.
- Metas de la programación orientada a objetos
  - Mejorar la calidad de las aplicaciones. Parámetros para medir la calidad:
    - Parámetros extrínsecos: FIABILIDAD (corrección + robustez).
      - Corrección: capacidad del software para realizar sus tareas tal y como se definen.
      - Robustez: capacidad del software para reaccionar ante condiciones excepcionales.
    - Parámetros intrínsecos: MODULARIDAD (extensibilidad + reutilización)
      - Extensibilidad: facilidad de adaptar el software a cambios de especificaciones.
      - Reutilización: capacidad de los elementos de servir para la construcción de aplicaciones diferentes.
  - Producir aplicaciones más fáciles de cambiar: MANTENIBILIDAD.

Tema 2

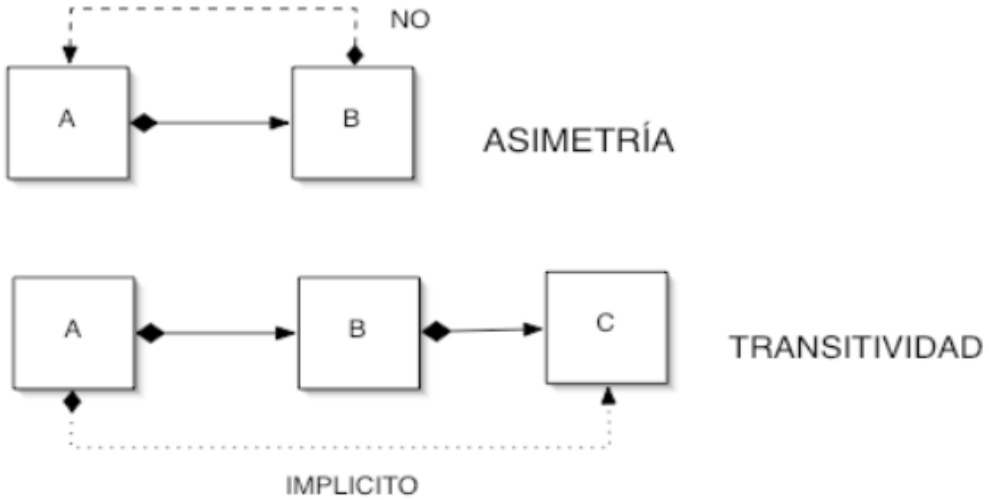
Conceptos básicos de la programación Orientado a Objetos

- Objeto: cualquier cosa que lo podamos asociar a unas determinadas propiedades y comportamientos. Según Grady Booch tienen:
  - Estado: conjunto de propiedades y valores actuales de esas propiedades.
  - Comportamiento: modo en el que el objeto actúa y reacciona ante los mensajes que se le envían (posibles cambios de estado). Viene determinado por la clase.
  - Identidad: distingue a un objeto de otro.
- Clase: representa un conjunto de objetos que comparten una estructura y comportamiento común. Todos los objetos son instancias de una clase.
  - Identificador: nombre.
  - Atributos: la combinación de estos determina el estado.
  - Roles: relaciones establecidas con otras clases.
  - Operaciones, métodos, servicios: Acciones que el objeto conocen como ejecutar.
- Atributos
  - Información que un objeto posee de sí mismo, suelen ser objetos y se declaran como campos.
  - Visibilidad:
    - + Interfaz
    - - Implementación (clase)
    - # Implementación (clases derivadas)
    - ~ en Java (paquete)
  - Tipos:
    - Constantes/Variables: constante (private final int) y variable (private int)
    - De instancia/clase:
      - De instancia: se guarda espacio para una copia de él por cada objeto creado.
      - De clase: características comunes a todos los objetos de la clase (private STATIC String fecha).
- Operaciones: de clase o de instancia.
  - De instancia: pueden acceder directamente a atributos de instancia o de clase.
  - De clase: acceder exclusivamente a atributos de clase, se ejecuta sin necesidad de que exista ninguna instancia.
  - Sobrecargadas: operación definida con el mismo nombre pero diferente número y tipo de argumentos (algunos LOO no lo soportan).
- En Java y C++ si no se define un constructor de clase de manera explícita, el compilador genera uno con visibilidad pública.
- Copia de objetos
  - Shallow copy (copia superficial): copia bit a bit de los atributos de un objeto.
  - Deep copy (copia completa): hay que implementarla explícitamente. Constructor de copia en C++ y Java o el método clone() de Java.

- Destructor de objetos
    - C++ -> Destructor
    - Java -> Métodos finalize() (para liberar recursos, cerrar ficheros abiertos, conexiones con BBDD) y recolector de basura (un programador no tiene control sobre cuándo exactamente libera la memoria de un objeto).
    - No es necesario definirlo (se crea por defecto).
- Forma canónica: conjunto de métodos que toda clase deberá definir. Suelen existir una definición "de oficio" proporcionada por el compilador y/o máquina virtual.

C++	Java
- Constructor por defecto - Constructor de copia - Operador de asignación - Destructor	- Constructor por defecto - public String toString() - public boolean equals(Object o) - public int hashCode()

- Relaciones entre Clases y Objetos
    - Persistentes: recogen caminos de comunicación entre objetos que se almacenan de algún modo y puede ser reutilizado.
    - No persistente: recogen caminos de comunicación que desaparecen tras ser utilizados.
    - Asociación: expresa una relación (uni o bidimensional) entre los objetos instanciados a partir de las clases conectadas. Los objetos existen independientemente.
    - Todo – Parte: relación en la que un objeto forma parte de otro. La diferencia con asociación es la Asimetría y la transitividad



        - Agregación: "tiene un", "es parte de", "pertenece a" (flexible).
        - Composición: agregación fuerte, cuando el todo es eliminado, se eliminan sus partes.
    - Uso: una clase A usa una clase B cuando no contiene datos miembro del tipo especificado por la clase B pero utiliza alguna instancia de B como parámetro, accede a sus variables privadas (clases amigas) o usa algún método.
    - Metaclase: existen métodos que se asocian con clases (new, delete, etc, métodos estáticos). En Smalltalk y en Java una clase es una instancia de otra clase, llamada metaclase.

Tema 3

Introducción al diseño orientado a objetos

- Pequeños proyectos: “programing in the small”. Pocos programadores. Uno abarca todos los aspectos del diseño. Problema: diseño y desarrollo.
- Grandes proyectos: “programing in the large”. Un programador no se responsabiliza de todo. Problema: manejo de detalles y comunicación.
- Interfaz e implementación
    - Principio de Parras: el desarrollador proporciona la información necesaria al usuario para usar el programa, y el desarrollador recibe la infomación necesaria para hacerlo del usuario.
- Métricas de calidad
    - Acoplamiento: Relación entre los componentes del software. Se consigue dando tareas a quien tiene la habilidad para hacerlas (interesa acoplamiento bajo).
    - Cohesión: Grado en que las responsabilidades de un solo componente forman una unidad significativa (interesa cohesión alta).
- UML: proporciona una vista estática del software.
- Diseño dirigido por responsabilidades (RDD – Responsibility-driven Design): proporciona técnicas formadas para el modelado de clases, responsabilidades y colaboración de objetos.
    - Principios
        - Maximizar abstracción
        - Distribución del comportamiento
        - Crear objetos inteligentes (que sepan operar).
        - Preservar flexibilidad (capacidad de ser modificado).
    - Artefactos
        - Aplicación: conjunto de objetos interactivos.
        - Objeto: implementación de una o más responsabilidades.
        - Rol: conjunto de responsabilidades.
        - Responsabilidad: obligación de realizar una tarea o conocer cierta información.
        - Colaboración: interacción entre roles y/u objetos.
    - Modelado de objetos
        - Colaboración: clases de la que se deben incluir instancias. Las que suministran servicio para realizar alguna acción y opcionalmente las que proporciona la clase actual.
        - Tarjetas CRC (Class/Responsibility/Colaborators)

Tema 4

Gestión de errores

- Excepción: evento que ocurre durante la ejecución de un programa e interrumpe el flujo normal de sentencias. Las excepciones lanzadas en Java son derivadas de la clase Throwable (throws new Exception).
    - Verificada: obliga a que se declare en su especificación, en tiempo de compliación.
    - No verificada: RuntimeException. Errores de programación, no se capturan ni se incluyen en la especificación.
    - Constructor: si se produce una excepción en un consructor (y no se captura dentro del mismo) el objeto no llega a ser construido.
    - Ventajas
        - Separar del manejo normal
        - Agrupar errores y diferenciarlos
        - Obliga al código a tratar las condiciones de error.
    - Inconvenientes
        - Sobrecarga del sistema para gestionar los flujos de control de excepciones. La colocación de los catch sí importa. Los bloques try/catch se pueden anidar.
- Java/C++ -> throw new Exception … try/catch/finally
    - Finally: se utiliza el bloque finally para cerrar funciones o liberar recursos. El bloque se puede ejecutar tras los bloques catch.
- Código Spaguetti: código que trata las tareas de error a la vez que sigue el flujo normal del programa.

```
if(variable == -1)
        error;
else
        seguir;
```

Tema 5

Herencia

- Herencia: las propiedades de la clase base son heredadas por la clase derivada.
    - Principales usos:
        - Herencia como reutilización de código: una clase derivada puede heredar el comportamiento de una clase base, por tanto, el código no se debe volver a escribir.
        - Herencia de implementación: la clase derivada implementa las funciones de la clase base.
- Tipos de herencia
    - Simple: una única clase base.
        - La clase derivada puede añadir más cosas.
        - La parte privada de una clase base no es accesible desde la clase derivada (sí mediante getters).
        - Protected: accesible por la propia clase y las derivadas.
        - Pública: Se hereda Interfaz e implementación (Java sólo soporta herencia pública).
        - Protegida y Privada (C++): esto tipos de herencia sólo heredan la implementación, la interfaz queda oculta desde objetos de la clase derivada.
        - La clase derivada puede añadir nuevos métidos y modificar los heredados de la clase base.
            - Refinamiento: se añade comportamiento antes y/o después del heredado. (C++, Java, constructor y destructor se refinan).
                - this: referencia al objeto actual usando implementación de la clase actual.
                - super: referencia al objeto actual usando implementación de la clase base.
            - Reemplazo: redefinición completa, sustituye el de la base.
        - Constructor: refinamiento, añadir al de la clase derivada el constructor de la clase base.
            - Ejecución implícita: del constructor por defecto de la clase base al invocar el constructor de la clase derivada.
            - Ejecución explícita: Circulo{super(); radio=1.0} ó super(col)
            - Orden de construcción: es el mismo que en C++, de la clase base a la clase derivada.
        - Destructor(C++) no se hereda.
            - Orden de destrucción: responsabilidad del programador (finally(), en C++ lo he definido antes).
                - Crear métodos propios para garantizar la finalización y liberación de recursos. Desventaja: el código cliente debe invocar explícitamente estos métodos.
        - Upcasting (Java): Convertir un objeto de tipo derivado a tipo base.

- Cuando se hace en C++, se hace object slicing, sin referencias, objetos enteros.

```
CuentaJoven tcj = new CuentaJoven();
Cuenta c;
c = (Cuenta) tcj; //explícito
c = tcj; //implícito
tcj.setedad(18); //OK
c.setEdad(18); //ERROR! -> ¿Por qué? Un objeto de la clase derivada al que se accede a través de una referencia a clase base, sólo se puede manipular usando la interfaz de la clase base.
```

  - Múltiple: cuando hay más de una clase base.
    - C++: soporta la herencia múltiple de implementación. Se heredan interfaces com… las implementaciones de la clase base.
    - Java: sólo soporta herencia múltiple de interfaz.
    - Colisión de nombres de herencia múltiple (C++)
      - Resolver los nombres mediante ámbitos.
    - Duplicación de propiedades en herencia múltiple
      - En C++ lo resolvemos usando herencia virtual.

  - De interfaz: la clase derivada no hereda código.
    - Objetivos: separar interfaz de implementación y garantizar la sustitución.
    - Principio de sustitución: Debe ser posible utilizar cualquier objeto instancia de una subclase en lugar de cualquier objeto instancia de su superclase sin que la semántica del programa escrito en los términos de la superclase se vea afectada.
      - Subtipo: una clase B, subclase de A, es un tipo de a si podemos sustituir instancias de A por instancias de B en cualquier situación y sin ningún efecto observable.
      - Los LOO soportan subtipos:
        - Lenguajes fuertemente tipados (tipado estático): caracteriza a los objetos por su clase.
        - Lenguajes débilmente tipados (tipado dinámico): caracteriza a los objetos por su comportamiento.
      - En Java el principio de sustitución se hace directamente.
      - En C++ sólo se hace a través de puntero a referencias.
- Herencia de interfaz: implementación mediante interfaces (Java/C#) o clases abstractas (C++) y enlace dinámico.
- Tiempo de Enlace: Momento en que se identifica el fragmento de código a ejecutar asociado a un mensaje (llamada a método) o el objeto concreto asociado a una variable.
  - Enlace estático (early or static binding): en tiempo de compilación. Ventaja: eficiencia. El tipo de objeto que contiene una variable se determina en tiempo de compilación.

- Enlace dinámico (late or dinamic binding): en tiempo de ejecución. Ventaja: flexibilidad. No está predefinido el tipo, la variable se gestionará en función de la naturaleza real del objeto.
  - Java usa enlace dinámico con objetos y estática con tipos escalados…
  - C++ sólo permite enlace dinámico con variables cuando estas son punteros o referencias y solo dentro de jerarquías de herencia.
- En métodos C++ usa enlace estático por defecto, y Java usa enlace dinámico por defecto.
- En C++ para que sea posible utilizar el enlace dinámico, el método debe ser creado como virtual.

- Clases Abstractas
  - Alguno de sus métodos no está definido.
  - Los métodos abstractos, tienen enlace dinámico por definición.
  - No se pueden crear objetos de esta clase.
  - Las clases derivadas de clases abstractas (o interfaces) están obligadas a implementar todos los métodos abstractos.
  - La clase derivada implementa la interfaz de la clase abstracta. Se guarda el principio de sustitución.
  - En Java: abstract <tipo> método();
  - En C++: contiene al menos un método virtual puro.
    - virtual void dibujar = 0; (la clase derivada lo implementará).
- Interfaz: declaración de un conjunto de métodos abstractos.

• Interfaces en Java

```
interface Forma
{
  // - Todos los métodos son abstractos por definición
  // - Visibilidad pública
  // - Sin atributos de instancia, sólo constantes estáticas
  void dibujar();
  int getPosicionX();
    ...
}
```

• Interfaces en C++

```
class Forma
{
  // - Sin atributos de instancia
  // - Sólo constantes estáticas
  // - Todos los métodos se declaran abstractos
 public:
  virtual void dibujar()=0;
  virtual int getPosicionX()=0;
  // resto de métodos virtuales puros...
}
```

- Herencia de implementación: habilidad para que una clase herede parte o toda su implementación de otra clase.
    - Uso seguro:
        - Especialización: la clase derivada es una especialización de la clase base, añade comportamiento pero no modifica nada.
        - Especificación: la clase derivada es una especificación de la clase base abstracta o interfaz. No añade ni elimina nada.
    - Uso inseguro:
        - Restricción (limitación): no toda clase base sirve para la clase derivada.
        - Generalización: se extiende el comportamiento de la clase base para obtener un tipo de objeto más general.
        - Varianza (herencia conveniente): conceptos no relacionados
- Herencia de construcción (herencia de implementación pura)
    - Modifica el interfaz heredado
    - La clase derivada un es una especificación de la clase base (No se cumple el principio de sustitución).
    - En C++: en la Herencia Privada se implementa un tipo de herencia de constantes que sí preserva el principio de sustitución.
- Beneficios y costes
    - Beneficios:
        - Responsabilidad software
        - Compartición de código
        - Consistencia de interfaz
        - Construcción de componentes
        - Prototipado rápido
        - Polimorfismo
        - Ocultación de información
    - Costes
        - Velocidad de ejecución
        - Tamaño del programa
        - Sobrecarga de paso de mensajes
        - Complejidad del programa
- Elección de técnica de reuso
    - Herencia IS-A: contener una clase
    - Composición HAS-A: contener un objeto
    - Regla del cambio: no se debe usar una herencia para describir una relación.
    - Regla del polimorfismo: la herencia es apropiada para describir una relación IS-A cuando las entidades o los componentes de las estructuras de datos del tipo más general pueden necesitar relacionarse con objetos del tipo más especializado
    - La composición, generalmente, es una técnica más sencilla que la herencia. La composición es más flexible.

Tema 6

Polimorfismo

- Polimorfismo-polisemia: un único nombre para muchos significados.
- Signatura de tipo. Notación: <argumentos> -> <tipo devuelto>
- Ámbito del nombre: porción del programa en la cual un nombre puede ser utilizado de una determinada manera.
    - Ámbitos activos: global, clase y método.
- Espacio de nombres:
    - Java: package (import).
    - C++: namespace (using).
    - .h: declaraciones agrupadas.
- Sistema de tipos
    - Estático(Java/C++)
    - Dinámico(Perl)
    - Fuerte: conversión de tipos estricta (int a=1; bool b = true; a=b; //ERROR)
    - Débil: permite conversión implícita.
    - Lenguajes procedimentales: sistemas de tipos estáticos y fuertes y no soportan el enlace dinámico (en general). C, Fortran, BASIC.
    - Lenguajes orientados a objetos
        - Sistemas de tipo estático: Sólo soportan enlace dinámico dentro de la jerarquía de tipo. C++, Java, C#, Objective-C, Pascal.
        - Sistemas de tipo dinámico: Javascript, PHP, Python, Ruby.
- Polimorfismo y reutilización: capacidad de una entidad de referenciar distintos elementos en distintos instantes de tiempo. Cuatro formas de polimorfismo: sobrecarga, sobreescritura, variables polimórficas y generacidad.
    - Sobrecarga (overloading): un solo nombre de método y muchas implementaciones distintas. Se distinguen en tiempo de compilación por tener diferentes parámetros.
    - Sobreescritura (Overrriding): ocurre dentro de relaciones de herencia en métodos con enlace dinámico. Métodos definidos en clases base, refinadas o remplazadas en las clases derivadas.
    - Variables polimórficas (polimorfismo de asignación): variable que se declara de un tipo pero que referencia un valor de distinto tipo (normalmente relacionado mediante herencia).
    - Generacidad (plantillas o templates): clases o métodos parametrizados (algunos se dejan sin definir). Es una forma de crear herramientas de propósito general y especializarlas.
- Sobrecarga. Tipos:
    - Basada en el ámbito: métodos con diferentes ámbitos de definición, independientemente de sus signaturas de tipo. (pej. toString() de Java). Diferentes ámbitos con mismo nombre de método, puede aparecer sin ambigüedad.

- Basada en la signatura: métodos con diferentes signaturas de tipo en el mismo ámbito. Siempre que difieran los argumentos en número, orden y tipo de argumentos (el tipo devuelto no se tiene en cuenta).
- C++: permite sobrecarga de métodos y operadores.
- Java, Python, Perl: permite sobrecarga de métodos pero NO de operadores.
- Eiffiel: permite sobrecarga de operaciones pero NO de métodos.
- Uso de operadores tradicionales con tipos definidos por el usuarios (<tipo devuelto> operator@(<argumentos>)).Para utilizar el operador con objetos definidos por el usuario, este debe ser sobrecargado.
- En la sobrecarga de operadores:
  - No se puede cambiar procedencia: qué operador se evalúa antes.
  - No se puede cambiar asociatividad: a=b=c -> a = (b=c).
  - No se puede cambiar aridad: operadores binarios para que actúen como unarios o viceversa.
  - No se pueden crear nuevos operadores.
  - No se pueden sobrecargar operadores para tipos predefinidos.
  - No se puede sobrecargar "." ".*" "::", sizeof, "? :".
  - Se realiza mediante
    - Funciones miembro: cuando el operando de la izquierda debe ser objeto de la case.
    - Función NO miembro: útil cuando el operando de la izquierda no es miembro de la clase.

- Alternativas a la sobrecarga
  - Funciones poliádicas: con variable número de argumentos (C, C++). Métodos poliádicos en Java
  - Coerción: cambio de tipo de manera implícita (B extends A; B pb = new B(); a pa = pb;)
  - Conversión: cambio de tipo de manera explícita. CAST (x = (double) //conversion). Java permite conversión entre tipos escalares, entre tipos relacionados por herencia y mediante métodos específicos. C++ permite conversión de un tipo externo al tipo definido por la clase (con un constructor de un solo parámetro) y del tipo definido por la clase a otro tipo (implementando un operador de conversión).
- Sobrecarga en jerarquías de herencia (sobreescritura)
  - Tiempo de enlace por defecto
    - Java: enlace dinámico en métodos de instancia públicos y protegidos. Y enlace estático para métodos privados, de clase (estáticos) y atributos
    - C++: todo tiene enlace estático.
  - Shadowing: refinadas. Misma signatura pero método enlazado en tiempo de compilación.
  - Redefinición: remplazados, redefinidos. merge (Java), los diferentes significados que se encuentran en todos los ámbitos actualmente activos, se unen para formar una sola colección de métodos. Jerárquico (C++), una

redefinición en clase derivada oculta el acceso directo a otras definiciones de la clase base.

- Sobreescritura: si los dos métodos tienen el mismo nombre, signatura de tipos y enlace dinámico (en tiempo de ejecución).
    - Podemos indicar que un método no puede ser sobrescrito mediante 'final'.
    - Covarianza: al sobrescribir un método en una clase derivada podemos cambiar el tipo de retorno del método a un subtipo del especificado en la clase base.
    - C++: La clase base debe indicar que el método tiene enlace dinámico (y puede sobrescribirse).
- Variables polimórficas: es aquello que puede referenciar más de un tipo de objeto.
    - Java: las referencias a objetos; Todas las clases son polimórficas.
    - C++: punteros o referencias a clases polimórficas; clase polimórfica -> es una clase con al menos un método virtual.
    - Podemos indicar que no se puede crear una clase derivada con final.
    - Variables receptoras: this y super. Hacen referencia al receptor del mensaje y cada clase representa un objeto de tipo diferente del super().
    - Downcasting (polimorfismo inverso): conversión de clase base a clase derivada. Deshacer el principio de sustitución.
        - Estático, tiempo de compilación.
        - Dinámico, tiempo de ejecución. Sólo dentro de una jerarquía de herencia. Si no es posible lanzar ClassCastException.
    - RTTI: Run Time Type Information. Proporciona información sobre tipos en tiempo de ejecución, podemos identificar subtipos mediante downcasting seguro.
        - Es una metaclase, cada clase tiene asociado un objeto Class.
        - instanceof: bool cierto si el objeto es del tipo indicado (es necesario conocer el nombre de la clase).
        - Class.isInstance() -> instanceof dinámico.
    - Polimorfismo Puro: alguno de sus argumentos es una variable polimórfica.
    - Tabla métodos. Apunta a la implementación más cercana. Cada objeto tiene un puntero oculto.
- Generacidad: propiedad que permite definir una clase o una función sin tener que especificar el tipo de todos o algunos de sus miembros o argumentos.
    - Listas, colas, pilas
        - En C++ apareció a finales de los 80, en Java desde la versión 1.5.
    - Métodos genéricos. Útiles para implementar funciones que aceptan argumentos de tipo arbitrario.
        - Un argumento genérico (public<T> void ___(){}).
        - Más de un argumento genérico (public<T,U> ___).
    - Clases genéricas: class vector<T> -> va a contener elementos de tipo genérico, no se conoce a priori.
    - Herencia en clases genéricas
        - Clase derivada genérica: class ___<T> extends Pila<T>{}
        - Clase derivada no genérica: class ___ extends Pila<T>

- Borrado de tipos: Java no guarda información RTTI sobre tipos genéricos. Son de tipo Object. La interfaz también puede ser genérica.
    - Sólo los métodos definidos en Object.
    - Comodines

Tema 7

Reflexión

- Reflexión: es una infraestructura del lenguaje que permite a un programa conocerse y manipularse a sí mismo en el tiempo de ejecución. Consiste en metadatos (datos que proporciona información sobre otro estado) más operaciones que los manipulen.
    - Puede usarse para: construir nuevos objetos o arrays, acceder y modificar atributos de instancia o de clase, invocar métodos de instancia y estáticos, accedes y modificar elementos de arrays, etc.
    - Cargar en memoria: Class c = Class.forName("Barco");
    - Invocar Constructor: Object obj = c.newInstance();
- API. Reflexión Java. Java.lang.Class / java.lang.reflect.* / java.lang.Classloader //clase abstracta.
- Objeto class: contiene información sobre una clase. Métodos, campos, superclase, interfaz, si es array (Class c = Class.forName("B"); -> si no está B en el CLASSPATH saltará la excepción ClassNotFoundException).
- Array: dos formas de declararlo.
    - Perro[] perrera = new Perro[10];
    - Class c1 = Class.forName("Perro"); Perro[] perrera = (Perro[]) Array.newInstance(c1, 10);
- Interface Member: representa un miembro de la clase. Implement x: Constructor, Method (Obtiene nombre y lista de parámetros e invocarlo. Obtener un método a partir de una signatura) y Field (signatura, obtener objetos).
- Mitos sobre la reflexión; Conclusiones.
    - La reflexión es una técnica común en otros lenguajes entados a objetos puros, como Smalltalk y Eiffel.
    - Ayuda a la robustez del software.
    - Permite ser flexible, extensible y uso de plug-ins.
    - No es demasiado compleja, hace falta llamar a unos pocos métodos.
    - Puede mejorar la usabilidad del código.
    - Puede aumentar el rendimiento del código.
    - La reflexión no puede usarse en aplicaciones certificadas con el estándar 100%. Pure Java. FALSO. Sólo hay algunas restricciones en la certificación: "El programa debe limitar las llamadas a métodos a clases que sean parte del programa o del JRE".
- LO QUE NO PODEMOS HACER
    - Saber las clases derivadas de una clase dada.
    - Qué método se está ejecutando en este momento.
- Sí podemos
    - Encontrar un método heredado.

Tema 8

Frameworks

- Frameworks y librerías: implementan funcionalidades útiles que el propio lenguaje del programa no incorpora.
    - Frameworks: esqueletos para fines específicos que debemos completar y personalizar mediante la implementación de interfaces, herencia de clases abstractas o ficheros de configuración. El framework llama a nuestros métodos.
    - Librerías: Conjunto de clases que realizan funciones más o menos concretas. Nosotros llamamos a los métodos de las librerías.
    - Utiliza:
        - Encapsulación. Ocultar implementación.
        - Polimorfismo. Mismo código para diferentes tipos de objetos.
        - Herencia. Reuso de funcionalidad.
    - Principio Hollywood: “no nos llames, ya te llamaremos nosotros”.
    - JUnit: es un framework. Funciona mediante inversión de control.
    - Java Collection Framework (JFC):
        - Conjunto de clases incluido el JDK representando tipos abstractos de datos básicos: pilas, colas, vectores, mapas.
        - Diseñado para que cualquiera pueda usarlo.
        - Si sólo usamos las implementaciones de referencia para cada interfaz, usaremos JFC como librería.
    - Apache Commons: conjunto de librerías incluidas en el programa apache. PEJ: Multimap.
    - JBCD. Java Data Base Conection: framework para conectar Java a la base de datos.
        - Utilizar drivers.
        - 1º crear conexión, 2º usar conexión para consultar/manipular la BBDD. 3º cerrar la conexión.
        - Prepared Statement: para mejorar el rendimiento.
    - Tratamiento de XML. JDK incluye:
        - DOM Parser: funciona como librería.
        - SAX Parser: funciona como framework.
    - Logging. Framework para la emisión eficiente de logs. Configurable mediante código o con ficheros.
    - Hibernate
        - Mapeo objeto/relacional (O/R)
        - A partir de unos ficheros de configuración genera clases que gestionan operaciones CRUD (créate, retrieve, update, delete) que hacen persistentes los objetos en la BBDD.

Tema 9

Refactorización

- Refactorización: proceso de mejora de la estructura interna de un sistema software de forma que su comportamiento externo no varía (minimiza posibilidad de bugs).
- Consta de dos pasos:
    - Introducir un cambio simple (refactorización).
    - Probar el sistema tras el cambio introducido.
- Consiste en realizar modificaciones como:
    - Añadir un argumento a un método.
    - Mover un atributo de una clase a otra.
    - Mover código arriba o abajo en una jerarquía de herencia.
- Principios de refracción.
    - Motivos:
        - Mejorar el diseño del software.
        - Hacer el código más fácil de leer.
        - Es más sencillo encontrar fallos.
    - Problemas:
        - Capa de persistencia (acoplamiento con BBDD).
        - Suele implicar cambios de interfaz.
- ¿Cuándo refractorizar?
    - Código con mal olor:
        - Código duplicado.
        - Métodos muy largos.
        - Clases muy grandes.
- Pruebas Unitarias y funcionales
    - Pruebas unitarias: prueban el correcto funcionamiento de un método. Automatizable, completa, repetibles o reutilizables, independientes y profesionales.
    - Funcionales: tratan a un componente software como una caja negra. Verifica una aplicación comprobando que su funcionalidad se ajusta a los requerimientos. JUnit (Java) y GNU LGPL Cxxtest y CppUnit (C++).
- Técnicas de refactorización: simples, comunes, refactorización y herencia.
    - Simples:
        - Añadir un parámetro. Evitar listas de argumentos demasiado grandes.
        - Quitar un parámetro. Si el método está sobrescrito, puede que otros la usen. En ese caso NO quitar. Considerar sobrecargar el mismo método sin ese parámetro.
        - Cambiar el nombre de un método. Cuando no indica su propósito. Hay que modificar todas las llamadas.
    - Comunes:
        - Mover un atributo. Si es público, encapuslar primero. Remplazar las referencias directas al atributo por llamadas al getter/setter correspondiente.
        - Mover un método. Es una de las refactorizaciones más comunes. Hacer que el antiguo método invoque al nuevo o bien eliminarlo. Comprobar la jerarquía de clases actual, si está definido no se puede mover.
        - Extraer una clase. Implica crear una relación entre la nueva clase y la antigua. Implica ‘mover atributo’ y ‘mover método’.

        - Cambiar condicionales por polimorfismo. Así solo necesita conocer la clase base.
        - Cambiar código de error por excepciones. Decidir si debe lanzar una excepción verificada o no verificada. Se deben manejar.
    - Refactorización y herencia
        - Generalización de un método: dos métodos, mismo comportamiento.
        - Especialización de un método. El comportamiento de la clase base solo es relevante para algunas clases derivadas.
        - Colapsar una jerarquía: cuando apenas hay diferencias entre la clase base y la clase derivada.
        - Extraer una clase derivada: Algunas características de una clase son usadas sólo por un grupo de instancias
        - Extraer una clase base (o interfaz): hay dos clases con características similares. Hay que mover el comportamiento común a la base.
        - Convertir herencia en composición: porque el principio de sustitución no se cumple. Los métodos de la clase base usadas en la clase derivada se implementan en la clase derivada y se convierten en una invocación al método base.
- Conclusiones. Forma sistemática y segura de realizar cambios en una aplicación con el fin de hacer más fácil comprender, modificar y extender. Algunas se hacen directamente con Eclipse. Lo importante es saber qué y cuándo refactorizar. Una de las innovaciones en el campo del software.

Tema 10

Principios de diseño

- Diseño de software: Proceso de solución de problemas cuyo objetivo es encontrar y describir una forma de implementar los requerimientos funcionales de un sistema respetando las restricciones impuestas los principios de diseño, la plataforma y los requisitos del proceso tales como el presupuesto y los plazos.
- D.O.O. se aplican, además de los diseños generales de software, los principios específicos de diseño de software orientado a objetos, utilizando una plataforma de desarrollo orientada a objetos. Damos por hecho: encapsulación, polimorfismo y herencia.
- Principios del diseño orientado a objetos
    - 1. Principio abierto – cerrado: las entidades software deben estar abiertas a su extensión, pero cerradas a su modificación. Las clases abstractas y sus derivadas son una buena forma de implementar conceptos fijos ilimitado de comportamientos. Heurísticas:
        - Encapsulación. Todos los atributos deben tener visibilidad privada.
        - No variables globales.
        - Usar RTTI es peligroso
    - 2. Principio de sustitución (Liskov)
        - Los métodos que usan referencias a la clase base deben ser capaces de utilizar el objeto de clase derivada sin saberlo.
    - 3. Principio de inversión de dependencias
        - Mal diseño: rigidez (difícil de cambiar), fragilidad (problemas al cambiarlo), inmovilidad (mala reutilización).
        - La interdependencia entre módulos es culpable de un mal diseño.
        - Los módulos de alto nivel no deben depender de los de bajo nivel. Ambos deben dependes de abstracciones. Las abstracciones no deben depender de detalles específicos de las abstracciones.
        - Este principio implica la separación de interfaz e implementación.
    - 4. Principio de segregación de interfaz
        - El cliente no debe ser forzado a depender de una interfaz que no utiliza:
            - Interfaces gruesas: muchos métodos, falta cohesión.
            - Polución de interfaz: la interfaz de una clase es contaminada por un interfaz que sólo necesitan algunas subclases, convirtiéndose en un interfaz público.
        - Conclusión:
            - Los interfaces gruesos o hinchados conducen a acoplamiento.
            - Estos interfaces pueden segregarse en diferentes clases abstractas o interfaces que rompen el acoplamiento no deseado entre clases.
    - 5. Principio de responsabilidad única
        - Nunca debe existir más de una razón para que una clase cambie.
            - Cada responsabilidad asociada a una clase es una razón para que la clase cambie.
            - Si asume más de una responsabilidad existirá más de una razón para que la clase cambie.
            - Si pasa eso, se acopla. Cambios en uno pueden impedir satisfacer otros: fragilidad.

- Vistazo a patrones de diseño
    - Patrón de diseño: es una solución a un problema de diseño. Características:
        - Se debe haber comprobado su efectividad resolviendo problemas similares en otras ocasiones.
        - Debe ser reutilizable, lo que significa que es aplicable a problemas de diseño similares en distintas circunstancias.
    - Acelera el proceso de encontrar una solución.