Resumen Integral de Tecnología de Programación (TDP)

Diseñado como guía de estudio y referencia técnica.

Módulo 1: Principios de Modelado, Modularidad y Fundamentos de la POO

Clase 04: Principios de Modelado y Modularidad

El modelado es el acto fundamental de simplificar la realidad con el objetivo de comprender un sistema de software complejo, reduciendo costos y minimizando los riesgos de error durante la construcción. Para gestionar esta complejidad de forma efectiva, Bertrand Meyer establece cinco criterios esenciales de modularidad y cinco reglas de diseño indispensables:

Criterios de Modularidad de Meyer

- **Descomposición modular:** Capacidad de dividir un problema extremadamente complejo en subproblemas autónomos más simples que puedan resolverse de forma independiente.
- **Composición modular:** Facilidad para combinar elements de software ya existentes de forma libre y armónica para construir sistemas completamente nuevos (reutilización).
- **Continuidad modular:** Garantiza que un cambio pequeño en la especificación del problema afecte únicamente a un módulo o a un grupo muy reducido de ellos, evitando un efecto dominó.
- **Protección modular:** Capacidad de contener condiciones anormales o errores en tiempo de ejecución dentro de los límites de un módulo, evitando una propagación catastrófica al resto del sistema.
- **Comprensión modular:** Posibilidad de que un desarrollador entienda el funcionamiento interno de un módulo de forma completamente aislada, sin necesidad de inspeccionar el resto del software.

Reglas de Diseño Modular

1. **Pocas interfaces:** Reducir la cantidad de vías de comunicación entre módulos para mantener el acoplamiento al mínimo.
2. **Interfaces pequeñas:** Intercambiar la menor cantidad posible de información en cada comunicación.

3. **Interfaces explícitas:** Toda conversación entre el módulo A y el módulo B debe ser clara, visible y declarada, eliminando dependencias ocultas.
4. **Ocultamiento de información (Encapsulamiento):** Cada módulo debe exportar una interfaz clara y mantener todos sus detalles de implementación estrictamente privados.
5. **Mapeo directo:** Mantener una correspondencia clara y directa entre los conceptos del mundo real (dominio del problema) y las estructuras lógicas de software correspondientes.

Clase 05: Conceptos Fundamentales de la POO

El paradigma de Orientación a Objetos (POO) cumple de manera nativa con las reglas de Meyer al proponer los siguientes pilares estructurales y operativos:

- **Clase:** Descripción estática que actúa simultáneamente como un módulo y como un tipo de datos, especificando los atributos (estado) y las operaciones (servicios) aplicables a un conjunto de objetos.
- **Objeto:** Instancia dinámica y concreta de una clase en tiempo de ejecución que interactúa mediante el paso de mensajes (`receptor.mensaje()`). En todo momento de la ejecución existe una *instancia actual* representada por las palabras clave `this` o `self`.
- **Asociación:** Relación estructural básica donde una clase conoce y mantiene una referencia hacia otra clase para interactuar con ella.
- **Herencia:** Relación de generalización/especialización del tipo "es un" que permite a una subclase adoptar automáticamente las características de una superclase.
- **Instancias propias vs. de descendientes:** Las instancias propias de la clase A son los objetos creados directamente a partir de ella. Las instancias de descendientes abarcan a todos los objetos creados a partir de cualquier subclase derivada de A.
- **Polimorfismo y Ligadura Dinámica:** El polimorfismo permite enviar mensajes idénticos a objetos de tipos diferentes. La ligadura dinámica es el mecanismo que determina en tiempo de ejecución (y no en compilación) el método exacto a ejecutar basándose en el tipo dinámico (real) del objeto y no en el tipo estático de la referencia.

Clase 06: Survey de Diferentes Lenguajes Orientados a Objetos

Las ideas y fundamentos del paradigma de orientación a objetos no se limitan de forma exclusiva a las líneas de código, sino que pretenden gobernar y estructurar la totalidad del proceso de construcción, análisis y diseño del software. Bajo esta premisa, el lenguaje de programación elegido actúa simplemente como una herramienta de codificación e implementación que debe proveer los mecanismos técnicos necesarios para plasmar los conceptos previamente modelados en el diseño abstracto. Existe una amplia diversidad de lenguajes orientados a objetos, los cuales gestionan los tipos, la memoria y las jerarquías bajo filosofías radicalmente distintas:

- **Lenguajes Basados en Clases (Java, C++, Smalltalk):** Exigen la definición estática y explícita de una Clase para que actúe como el molde estructural invariable del cual nacerán las instancias u objetos en memoria.
- **Lenguajes Basados en Prototipos u Objetos (JavaScript):** Prescinden de la obligación de estructurar clases rígidas de forma inicial. Los objetos pueden ser declarados de forma dinámica y directa, y adoptan comportamientos o heredan rasgos delegando sus mensajes hacia otros objetos interconectados denominados prototipos.

Ejemplificaciones Técnicas en Semántica por Prototipos (JavaScript)

1. Declaración e instanciación directa de un objeto literal con estado interno y funciones de comportamiento asimiladas:

```
const usuario = {
    nombre: 'Homero',
    apellido: 'Simpson',
    getNombre: function() {
        return this.nombre + ' ' + this.apellido;
    }
};
```

2. Uso de funciones constructoras tradicionales combinadas con el operador `new` para emular tipos y comportamientos de instanciación dinámicos:

```
function imprimir() {
    document.write("<" + this.coordx + "," + this.coordy + ">");
}

function crearPunto(x, y) {
    this.coordx = x;
    this.coordy = y;
    this.mostrar = imprimir;
}

p1 = new crearPunto(2, 3);
p1.mostrar(); // Salida en Browser: <2,3>
```

3. Azúcar Sintáctico moderno (`class`): Las especificaciones contemporáneas del lenguaje incorporaron formalmente las palabras clave `class` y `constructor`. Es indispensable comprender que esto es puramente visual para asimilarse a lenguajes tradicionales, dado que internamente el motor del software continúa delegando y resolviendo las interacciones mediante su cadena nativa de prototipos físicos:

```
class Rectangle {
```

```
    constructor(height, width) {
        this.height = height;
        this.width = width;
    }
}
```

Módulo 2: Modelado Gráfico con UML, Semánticas de Objetos e Interacciones

Clase 07: UML - Diagrama de Clases (Parte 1)

UML (Unified Modeling Language) es el estándar gráfico para visualizar, especificar y documentar modelos de sistemas orientados a objetos. Una clase se representa mediante un rectángulo dividido en tres compartimientos: Nombre, Atributos y Operaciones.
La sintaxis formal para los atributos es: `:=` y para las operaciones: `():`. Las visibilidades estándares son Pública (`+`), Privada (`-`) y Protegida (`#`).

Relaciones Estructurales en UML

- **Asociación:** Conexión estructural simple entre objetos independientes. Incluye opcionalmente multiplicidad (ej. `1`, `0..*`, `1..*`) y direccionalidad.
- **Agregación:** Asociación especial que representa una relación jerárquica del tipo "es parte de" (todo-parte). Las partes son independientes y pueden sobrevivir a la destrucción del contenedor. Se grafica con un **rombo vacío** en el extremo del contenedor. *Ejemplo: Un `Equipo` y sus `Jugadores`.*
- **Composición:** Agregación restrictiva y fuerte donde el contenedor controla de forma estricta el ciclo de vida de sus partes. Las partes no pueden existir de forma independiente fuera de las fronteras del contenedor. Se grafica con un **rombo relleno**. *Ejemplo: Una `Casa` y sus `Habitaciones`.*

Clase 08: Tratamiento de Objetos, Semánticas y Diagramas de Interacción

El manejo físico de la memoria y la forma en que los lenguajes de programación gestionan la vida de las instancias se divide en dos semánticas fundamentales:

Característica	Semántica por Referencia (Java / Smalltalk)	Semántica por Valor (C++ / Objetos Incrustados)
Acceso	A través de punteros lógicos o direcciones de memoria.	Almacenamiento físico directo del subobjeto dentro del contenedor.
Creación	Explícita mediante operadores dedicados (`new`).	Implícita y automática junto con la instanciación del objeto padre.
Aliasing	Permitido (múltiples referencias pueden apuntar al mismo objeto).	Imposible (cada variable contiene su propia copia de datos).
Valores Nulos	Soportado (`null` o `nil`).	No aplicable (el espacio físico siempre está ocupado).

A partir de estas semánticas se definen las siguientes operaciones de manipulación estándar:

- **Comparación de referencias (`==`):** Evalúa si dos punteros apuntan exactamente a la misma posición física de memoria (identidad).
- **Comparación de objetos (`equals`):** Evalúa la igualdad de los campos o atributos lógicos internos de los objetos (igualdad superficial).
- **Copia (`copy`):** Sobreescribe los valores de los atributos de un objeto existente con los de otro.
- **Clonación (`clone`):** Crea una nueva instancia duplicando superficialmente los valores del objeto original. Si los campos son referencias, se copian las direcciones (shallow clone), requiriendo un clonado en profundidad (deep clone) si se busca duplicar las jerarquías de forma recursiva.

Modelos Dinámicos en UML

Permiten capturar y validar las interacciones dinámicas de los objetos en ejecución:

- **Diagrama de Secuencia:** Organiza la interacción basándose en una línea de tiempo estricta, priorizando el orden cronológico del envío de mensajes.
- **Diagrama de Colaboración o Comunicación:** Enfatiza las conexiones estructurales

entre los objetos de forma compacta, utilizando una numeración decimal estricta (`1`, `1.1`, `1.1.1`) para denotar el orden de despacho de los mensajes.

Módulo 3: Proceso Incremental de Modelado y Casos de Estudio Prácticos

Clases 09 a 12: Taller Práctico de Diseño - Caso "Base Starcraft"

Este conjunto de clases plasma de forma empírica el proceso iterativo que transforma un enunciado textual informal en una solución de software orientada a objetos robusta:

- **Clase 09 - Identificación de Conceptos:** Lectura exhaustiva para extraer los sustantivos del dominio del problema. La relevancia de las nociones candidatas se mide mediante su frecuencia de mención o por las relaciones directas que entablan en el texto (ej. los términos `Robot` y `Titanio` aparecen de forma reiterada, convirtiéndose en clases esenciales).
- **Clase 10 - Filtrado y Asignación de Roles:** Se descartan sustantivos circunstanciales o irrelevantes. Aquellos conceptos que solo describen características se transforman en atributos de las clases principales (ej. `capacidad_maxima` pasa a ser un atributo entero de `Robot`). Se confecciona la tabla de responsabilidades documentando rigurosamente qué datos almacena cada entidad y qué servicios provee.
- **Clase 11 - Refinamiento Estático:** Se plasman las tablas conceptuales en un Diagrama de Clases formal en UML, asignando tipos de datos y visibilidades. Se resuelven dilemas de diseño complejos para respetar el bajo acoplamiento: para evitar que `FabricaProcesamiento` retorne referencias directas o nulas a los robots, la Fábrica expone sus contenedores (`contenedores(): Coleccion`) y se delega la responsabilidad de evaluación al `Robot`.
- **Clase 12 - Validation Dinámica:** Se construye el diagrama de secuencia para la operación crítica `depositarTitanio()`. El flujo modelado demuestra cómo el `Robot` interactúa con la colección de contenedores mediante un bloque interactivo **loop**, enviando el mensaje `disponiblePara(compartimiento)` a cada `Contenedor`. En la alternativa (**alt**) afirmativa, el `Robot` despacha el mensaje `recibirCarga(compartimiento)`, logrando un diseño limpio, desacoplado y con mapeo directo.

Clases 19 y 20: Taller Práctico de Diseño - Caso "Road Fighter" (Parte 1)

El modelado inicial del clásico videojuego arcade *Road Fighter* sirve para ejercitar la clasificación taxonómica avanzada y la resolución de interacciones multidireccionales

complejas en el dominio del problema:

- **Clasificación Jerárquica de Vehículos:** Los vehículos se separan en dos ramas independientes bajo la superclase `Vehiculo`: vehículos de `Carrera` (con lógica de competencia y posiciones del 1 al 12, donde el `Jugador` es una especialización que recolecta extras) y vehículos de `Transito` (`Auto`, `Moto`, `Camion`) que actúan como obstáculos dinámicos en la ruta.
- **Modelado de Obstáculos y Extensiones:** Los elementos fijos en el mapa se consolidan bajo `Obstaculo` (`Bache`, `Anciana`, `Perro`). Se ejercita la especialización modelando al `Lobo` como subclase directa de `Perro`, heredando sus comportamientos pero alterando los constructores para duplicar el peso y redefinir el impacto, dejando el puntaje del jugador en cero.
- **Abstracción del Mecanismo de Colisiones:** Para evitar un acoplamiento directo y condicionales masivos cruzados entre todas las entidades móviles y fijas de la ruta, se diseñan las interfaces estratégicas `Colisionador` y `Colisionable`. El `Jugador` (actuando como Colisionador) despacha el mensaje `chocar(Colisionable c)`. Mediante este mecanismo de doble ligadura, el comportamiento exacto ante el impacto se delega de forma limpia al objeto colisionado (ej. el `Nitro` duplica la velocidad, la `Nafta` recarga combustible), eliminando condicionales rígidos.

Módulo 4: Mecanismos Avanzados de Tipos, Polimorfismo y Jerarquías

Clase 13: Generalización y Herencia

La generalización es el proceso mental de abstraer características, estados y comportamientos comunes de múltiples clases específicas para consolidarlos en una abstracción superior de carácter general. La herencia es la herramienta técnica que concreta esta generalización en los lenguajes de programación.

Para dar soporte y flexibilidad a las extensiones, se introduce el modificador de **visibilidad protegida (`#`)**, el cual restringe el acceso a los miembros para el entorno exterior, pero permite que las clases hijas accedan y manipulen dichos atributos de forma directa.

La **redefinición (overriding)** permite a una subclase declarar un método heredado de la superclase para alterar o complementar su algoritmo interno, manteniendo estrictamente intacta su firma (signature). La palabra clave `super` (o `base`/`parent` según el lenguaje) se utiliza para invocar de forma calificada la versión del método de la superclase y reutilizar su lógica:

```
Clase ProgramadorJefe hereda de Programador
Redefinición de sueldo(): entero {
    retornar super.sueldo() + (500 * proyectos_a_cargo);
```

```
}
```

Clase 14: Polimorfismo, Chequeo y Conformidad de Tipos

El polimorfismo se sustenta en la diferenciación rigurosa de dos conceptos temporales de tipos:

- **Tipo Estático:** El tipo asignado formalmente para declarar la entidad en el código fuente. Es inmutable y controlado estrictamente por el compilador para validar sintácticamente qué mensajes son válidos.
- **Tipo Dinámico:** La clase concreta del objeto real al cual apunta la referencia en memoria en un instante específico de la ejecución. Puede variar a lo largo del tiempo dentro de la jerarquía de herencia.

Una **asignación polimórfica** (`p = objetoReal`) es válida si y solo si existe **conformidad de tipos**, lo que exige que el tipo dinámico de la derecha sea un descendiente legítimo del tipo estático de la izquierda en la cadena de herencia.
Las **estructuras de contenido polimórfico** surgen de combinar colecciones genéricas con tipos base comunes (ej. un arreglo declarado como `Poligono[]` que aloja instancias de `Triangulo` y `Cuadrado`). Al iterar el arreglo y despachar mensajes genéricos, la ligadura dinámica ejecuta automáticamente el comportamiento particular de cada celda sin requerir condicionales de tipo.

Clase 15: Tipos, Subtipos y Herencia

Un tipo de datos se define formalmente como un conjunto de valores y las operaciones aplicables sobre ellos (enfocándose puramente en la especificación, el *qué hace*). La clase representa la implementación concreta (el *cómo lo hace*).
La relación de **subtipos** se rige por el Principio de Sustitución: el tipo B es subtipo de A si la especificación operativa de B incluye por completo a la de A, asegurando que un objeto de tipo B pueda reemplazar transparentemente a uno de tipo A en cualquier contexto de ejecución sin alterar la estabilidad.

- **Herencia de Interfaz:** Mecanismo puramente de especificación que asegura que un objeto responda a un conjunto de firmas (en Java modelado mediante `interface` e `implements`).
- **Herencia de Clase:** Mecanismo técnico para compartir código y representación de estado interno. Su abuso excesivo da lugar al **problema de la superclase frágil** (fragile base-class problem), donde cambios sutiles e internos en el código del padre rompen de forma imprevista el comportamiento de las clases hijas.
- **Clases Abstractas:** Son abstracciones incompletas que poseen al menos una operación declarada pero sin código de implementación (método abstracto o diferido). Debido a esto, **no pueden ser instanciadas directamente** mediante el operador `new`. Su

objetivo arquitectónico es servir de base común en la jerarquía para que las subclases concretas hereden y *efectivicen* obligatoriamente dichos métodos.

Módulo 5: Catálogo de Patrones de Diseño (GoF)

Clase 16: Patrones Creacionales

Los patrones creacionales encapsulan y abstraen el proceso de instanciación del sistema, ocultando cómo se crean y componen los objetos, y eliminando las referencias explícitas a clases concretas en el código de aplicación del cliente.

- **Abstract Factory (Fábrica Abstracta):**
  - *Intención:* Proveer una interfaz para crear familias de objetos dependientes o relacionados entre sí sin especificar sus clases concretas.
  - *Ejemplo:* Un constructor de niveles para un juego que invoca `factory.crearHabitacion()` y `factory.crearPuerta()`. Si se inicializa con la familia concreta `FabricaMagica`, se generarán habitaciones y puertas mágicas de forma transparente sin alterar el algoritmo de construcción principal.
- **Prototype (Prototipo):**
  - *Intención:* Especificar los tipos de objetos a crear utilizando una instancia prototipo (un modelo inicial) y generar nuevos objetos copiando o clonando esta instancia en memoria.
  - *Ejemplo:* Un sistema de generación de enemigos en un mapa dinámico. En lugar de evaluar condicionales rígidos para instanciar criaturas, se pre-cargan objetos prototipos en un registro y se invoca `enemigo = prototipoZombie.clonar()`, optimizando y flexibilizando la instanciación.

Clase 17: Patrones Estructurales

Los patrones estructurales se ocupan de cómo se componen y organizan las clases y los objetos para conformar estructuras de software más grandes y funcionales, priorizando la composición por sobre la herencia rígida.

- **Adapter (Adaptador / Wrapper):**
  - *Intención:* Convertir la interfaz de una clase existente en otra interfaz que los clientes esperan, permitiendo que cooperen clases con interfaces incompatibles.
  - *Ejemplo:* Un editor gráfico que manipula elementos mediante la interfaz `Figura` y su método `calcularArea()`. Para integrar una clase externa `TextoDeTerceros` que expone `obtenerTamañoFuente()`, se diseña una clase `AdaptadorTexto` que implemente `Figura` y, de forma interna, delega la llamada traduciendo el mensaje.
- **Composite (Objeto Compuesto):**
  - *Intención:* Componer objetos en estructuras de árbol para representar jerarquías de relación "todo-parte", permitiendo a los clientes tratar a objetos individuales y a composiciones de forma homogénea y uniforme.
  - *Ejemplo:* Una aplicación de diseño gráfico donde elementos simples (`Linea`, `Circulo`) e interactivos y elementos compuestos (`GrupoDeDibujo`) implementan

la interfaz común `Grafico`. Al invocar `grupo.dibujar()`, el compuesto itera internamente ejecutando de forma recursiva el método `dibujar()` sobre cada uno de sus hijos.

- **Decorator (Decorador / Wrapper):**
    - *Intención:* Adjuntar responsabilidades y comportamientos adicionales a un objeto de forma dinámica y transparente en tiempo de ejecución, ofreciendo una alternativa flexible a la herencia estática.
    - *Ejemplo:* Un componente de interfaz gráfica `AreaDeTexto`. Se puede envolver dinámicamente para añadirle bordes o barras de desplazamiento combinando envoltorios en memoria: `new DecoradorBorde(new DecoradorDesplazamiento(areaDeTexto))`.

Clase 18: Patrones de Comportamiento

Se centran de forma específica en la distribución de responsabilidades, el diseño de los algoritmos y los mecanismos de comunicación entre objetos interconectados.

- **Command (Comando):** Encapsula una petición o mensaje como un objeto independiente con su propia identidad, permitiendo parametrizar clientes con diferentes solicitudes, encolar peticiones y soportar operaciones reversibles (undo/redo). *Ejemplo: Botones de un menú gráfico que guardan instancias de `Comando`, ejecutando simplemente `comando.execute()` de forma desacoplada del receptor real.*
- **Observer (Observador):** Define una dependencia de uno-a-muchos entre objetos, de forma tal que cuando el objeto observado (*Sujeto*) sufre una modificación en su estado interno, todos sus dependientes (*Observadores*) son notificados de forma automática. *Ejemplo: Una estructura de datos lógicos que notifica de forma directa a múltiples componentes de paneles gráficos en pantalla para que se redibujen ante cambios de valores.*
- **Strategy (Estrategia):** Define una familia de algoritmos encapsulados de forma independiente, haciéndolos intercambiables en tiempo de ejecución para permitir que el algoritmo varíe de forma autónoma respecto a los clientes que lo consumen. *Ejemplo: Un personaje combatiente que conmuta dinámicamente su algoritmo de locomoción asignando objetos `EstrategiaCaminar` o `EstrategiaVolar`.*
- **State (Estado):** Permite que un objeto altere por completo su comportamiento operativo cuando cambia su estado interno, haciendo que el objeto parezca cambiar de clase. Elimina estructuras complejas de `if/else` delegando la lógica en objetos estados autónomos. *Ejemplo: Una clase `Soldado` cuyos métodos `atacar()` y `moverse()` delegan de forma directa en un objeto de estado actual (`Parado`, `Corriendo`, `Herido`).*
- **Visitor (Visitante):** Permite incorporar una nueva operación sobre una estructura jerárquica de objetos existente sin necesidad de modificar el código fuente interno de las clases de dichos elementos. Utiliza el mecanismo de doble ligadura para resolver la ejecución. *Ejemplo: Un sistema que recorre una red de cajeros automáticos; se pueden*

anexar objetos visitantes independientes como `VisitanteCalcularEfectivoTotal` o `VisitanteChequearErrores` sin alterar las clases concretas de los cajeros.

Módulo 6: Arquitectura de Software y Persistencia

Clase 21: Patrón Arquitectónico de Tres Capas

Es un patrón fundamental de la ingeniería de software enfocado en organizar y estructurar el sistema completo dividiendo el código en tres niveles lógicos independientes, aislando responsabilidades y controlando las dependencias:

Capa	Responsabilidades Primordiales
Vista (Presentación / Frontend)	Interfaz de usuario gráfica encargada de renderizar la información en pantalla y capturar las interacciones y periféricos del operador, traduciéndolos en eventos. En Java se implementa mediante frameworks como Swing o JavaFX.
Lógica (Negocio / Control)	El núcleo e inteligencia del sistema. Procesa los datos, aplica las reglas y restricciones del dominio del problema, realiza validaciones y actúa como puente de comunicación directo entre la Vista y los Datos.
Datos (Persistencia / Backend)	Se encarga del almacenamiento y recuperación física de la información a largo plazo. Abstrae la fuente técnica subyacente (archivos de texto plano, bases de datos relacionales, APIs de almacenamiento).

La ventaja crítica de este modelo arquitectónico es el bajo acoplamiento: permite modificar por completo la tecnología de una capa (ej. migrar la interfaz visual de Swing a una plataforma Web) de forma aislada, sin alterar ni una sola línea de código de la lógica de negocio ni de la persistencia de datos.

Clase 22: Aplicación Práctica de Arquitectura - Caso "Road Fighter"

(Parte 2)

La refactorización arquitectónica de *Road Fighter* demuestra la traducción concreta del modelo estático a una estructura limpia de paquetes:

- **Estructura de Paquetes:** Se divide el proyecto en los paquetes independientes `vista` (con clases de interfaz gráfica como `GUI` extendiendo `JFrame` y paneles dedicados para la ruta) y `logica` (coordinada por la clase `Juego` encargada de computar las posiciones matemáticas internas de las entidades).
- **Desacoplamiento mediante Controladores:** Para evitar que la lógica dependa de detalles visuales, la comunicación se media a través de las interfaces abstractas `ControladorJuego` y `ControladorVistas`.
- **Sincronización Gráfica con el Patrón Observer:** Se implementa el patrón *Observer* para mantener la coherencia gráfica. Clases visuales como `ObserverGrafico` actúan como observadores de las clases pertenecientes a la capa lógica (`EntidadLogica`). Cuando el auto del jugador altera sus coordenadas matemáticas en la lógica, el objeto simplemente emite una notificación y los observadores gráficos actualizan automáticamente la posición de los Sprites visuales en la pantalla Swing de forma transparente.

Clase 25: Serialización y Archivos de Configuración

La persistencia de los estados lógicos se complementa mediante el mecanismo de **Serialización**, el cual permite a la capa de datos transformar la estructura completa de un objeto en memoria en un flujo de bytes nativos aptos para ser almacenados en un archivo físico o transmitidos por red, logrando recuperar la instancia exacta posteriormente (Deserialización). Paralelamente, el uso de archivos de configuración externos de tipo **Properties** (estructuras clave-valor) permite parametrizar de forma limpia el entorno del software (ej. setear la velocidad base, cargar rutas de archivos o configurar conexiones) sin necesidad de modificar ni recompilar el código fuente binario de la aplicación.

Módulo 7: Concurrencia y Multiprocesamiento

Clase 23: Fundamentos de Concurrencia

Un programa secuencial clásico se ejecuta en memoria como un único *proceso* regido por un solo hilo de control, donde un *Instruction Pointer* (IP) avanza linealmente de sentencia en sentencia. La **concurrencia** es la capacidad de gestionar de forma simultánea o intercalada múltiples hilos de ejecución (*Threads*) que operan de forma coordinada compartiendo un mismo espacio de memoria.
Es crucial diferenciar la concurrencia lógica (intercalación temporal administrada por el

planificador del Sistema Operativo sobre un mismo núcleo) del **paralelismo físico** (ejecución estrictamente simultánea y real de múltiples procesos corriendo en diferentes núcleos de hardware independientes).

Riesgos Críticos en Entornos Concurrentes

- **Condición de Carrera (Data Race):** Surge cuando dos o más hilos acceden e intentan modificar de forma simultánea un dato compartido sin sincronización. Dado que operaciones simples como el incremento (`counter += 10`) involucran múltiples instrucciones no atómicas a nivel de CPU, los hilos intercalan sus pasos corrompiendo la consistencia de los datos.
- **Interbloqueo (Deadlock):** Bloqueo mutuo y permanente que ocurre cuando dos o más hilos se detienen indefinidamente esperando recursos que están siendo retenidos por el otro (el Hilo 1 reserva el Recurso A y espera por el B; el Hilo 2 reserva el Recurso B y espera por el A).
- **No determinismo:** Debido a la arbitrariedad del planificador del sistema operativo, el mismo código concurrente puede retornar resultados totalmente distintos en ejecuciones sucesivas.

Mecanismos Concurrentes en Java

Java ofrece soporte nativo para hilos compartiendo memoria a través de dos enfoques:

1. Heredar directamente de la clase `Thread`.
2. Implementar la interfaz funcional `Runnable`.

En ambos casos, el código concurrente se escribe obligatoriamente dentro del método `run()`, pero la ejecución asíncrona en un hilo independiente se inicia invocando de manera estricta al método **`start()`**.
Para asegurar la consistencia y el código seguro para hilos (*thread-safety*), se utiliza la palabra clave **`synchronized`**. Cada objeto en Java posee un monitor o cerrojo lógico implícito (*lock*); al sincronizar una operación o un bloque de sentencias (`synchronized(this)`), se restringe el acceso garantizando exclusión mutua: solo un hilo puede adueñarse del lock a la vez, obligando a los restantes a esperar en cola hasta su liberación.

Clase 24: Concurrencia Práctica - Caso "Road Fighter" (Parte 3)

La integración de hilos en el caso de estudio de *Road Fighter* fundamenta la necesidad de la concurrencia para evitar que la interfaz gráfica se congele y garantizar la fluidez en tiempo real:

- **Captura Asíncrona de Entradas:** Se asocian listeners de teclado en la capa visual que reaccionan inmediatamente a los comandos ingresados por el usuario, enviando señales directas a la capa de lógica para alterar el rumbo del vehículo sin detener la ejecución del juego.

- **Bucle de Juego Autónomo con Threads:** El bucle principal de simulación (hacer avanzar la ruta de forma constante, desplazar los vehículos de tránsito automáticos y recalcular las coordenadas físicas) se encapsula por completo dentro de un hilo concurrente dedicado (`Thread`). Este hilo corre de manera autónoma y periódica, ejecutando pausas controladas mediante `Thread.sleep(delay)` para regular los *frames* del juego, asegurando que el entorno se mueva de forma fluida e independiente de las acciones del teclado del jugador.

Módulo 8: Herencia Avanzada, Ambigüedades e Implementación de Lenguajes

Clase 28: Herencia Múltiple y Herencia Repetida

La herencia múltiple es la propiedad que permite a una subclase derivar simultáneamente de dos o más clases padres directas, adoptando todos sus miembros. El problema técnico intrínseco de este enfoque es la **colisión de nombres (ambigüedad)**, la cual se manifiesta cuando una clase hija hereda dos o más métodos implementados (*efectivos*) que poseen exactamente el mismo nombre o encabezado desde padres distintos.

Estrategias de Resolución de Colisiones por Lenguaje

- **Prohibición Absoluta (Java / Smalltalk):** Consideran que la herencia múltiple de clases es inherentemente peligrosa. La prohíben por completo en su modelo de objetos, forzando la utilización de herencia simple de clase combinada con herencia múltiple de interfaces puras para evitar la colisión de código efectivo.
- **Renombre Sintáctico (Eiffel):** Provee la cláusula `rename` en el acto de herencia. Permite cambiar el identificador de un método en conflicto de forma local en la subclase (ej. renombrar `m()` a `m_desde_X`), disolviendo la ambigüedad de forma explícita sin alterar los tipos ni las pre/postcondiciones originales.
- **Resolución de Ámbito Calificado (C++):** Permite la coexistencia de ambos métodos en la subclase y obliga al desarrollador a desambiguar cada invocación en el código de forma explícita utilizando el operador de resolución de ámbito (`objeto->ClaseX::m()`).
- **Linearización C3 por Prioridad de Declaración (Python):** Resuelve los conflictos calculando una lista plana de precedencia de la jerarquía de herencia mediante el algoritmo C3. Este mecanismo calcula el orden estricto respetando dos principios fundamentales:
    - *Monotonicidad:* Si la clase Base1 precede a la clase Base2 en la linearización de una subclase intermedia, ese mismo orden de precedencia se debe conservar inalterable en cualquier subclase posterior de la jerarquía.
    - *Preservación del ordenamiento local:* Se respeta el orden textual exacto de

izquierda a derecha con el que se listaron los padres directos en la declaración. Si una jerarquía compleja rompe estos principios, Python bloquea la compilación emitiendo un error de resolución de nombres ambiguo.

Herencia Repetida (El Problema del Diamante)

Surge cuando una subclase hereda de una superclase común a través de más de un camino de herencia independiente en la jerarquía (ej. la clase D hereda de B y C, los cuales a su vez heredan de la clase base A). Existen dos formas lógicas de resolver las propiedades heredadas repetidas:

1. **Compartir (Sharing):** Si el atributo o servicio describe el mismo concepto lógico, la subclase final conserva una única copia física unificada en memoria (ej. el atributo `edad` en una clase `ConductorFrancoIngles`).
2. **Replicación (Replication):** Si el atributo adquiere significados semánticos independientes por cada rama, se fuerzan copias físicas separadas en memoria aplicando renombre (ej. separar el contador de `infracciones` de la clase base en dos variables replicadas: `infracciones_Francia` e `infracciones_Inglaterra`).

Módulo 9: Artesanía del Software - Filosofía de Clean Code

Clase 26: Clean Code - Parte 1 (Nombres Significativos)

Basado en las directrices de Robert C. Martin, el código limpio es aquel que resulta legible, expresivo y fácil de mantener por cualquier desarrollador, reduciendo drásticamente la métrica de "WTFs por minuto" durante un code review. Para ir rápido en el desarrollo de software, la única estrategia real es mantener el código limpio en todo momento, aplicando de forma continua la **Regla del Boy Scout**: *"Deja siempre el código más limpio de como lo encontraste"*.

Directrices para Nombres Profesionales

- **Revelar la intención:** El nombre de una variable, función o clase debe responder explícitamente por qué existe, qué hace y cómo se utiliza. Si requiere un comentario aclaratorio a su lado, el nombre ha fallado. (Cambiar `int d;` por `int elapsedTimeInDays`).
- **Evitar la desinformación:** No dejar pistas falsas. Evitar abreviaturas ambiguas o usar nombres que evoquen tipos de datos incorrectos (no llamar `listaUsuarios` a una variable si su tipo real es un Arreglo o un Mapa). Evitar caracteres confusos como `l` minúscula u `O` mayúscula que se confunden con el `1` y el `0`.
- **Realizar distinciones significativas:** No diferenciar variables agregando números correlativos (`a1`, `a2`) ni rellenar nombres con palabras vacías (noise words) que no

aportan valor conceptual (ej. mezclar de forma inconsistente `ProductInfo`, `ProductData` y `Product` dentro del mismo módulo).

- **Nombres buscables y pronunciables:** Utilizar palabras claras del lenguaje en lugar de acrónimos crípticos (`generationTimestamp` por sobre `genymdhms`). Las variables de una sola letra (`i`, `j`) se reservan única y exclusivamente para contadores locales dentro de bucles muy cortos.
- **Evitar encodings y prefijos:** La notación húngara y los prefijos de atributos (`m_`) están obsoletos. En el diseño de interfaces, se desaconseja anteponer la letra "I" (`IShapeFactory`); es preferible mantener el nombre de la interfaz limpio (`ShapeFactory`) y añadir prefijos o sufijos a las clases de implementación concreta (`ShapeFactoryImpl`).
- **Clases y Métodos:** Las clases e interfaces deben ser sustantivos o frases sustantivas (`Customer`, `AccountRecord`), evitando verbos. Los métodos deben nombrarse con verbos o frases verbales (`save()`, `postPayment()`), utilizando los estándares `get`, `set` e `is` para accesores y predicados.
- **Una palabra por concepto:** Mantener consistencia terminológica en todo el proyecto (no mezclar sinónimos como `fetch()`, `retrieve()` y `get()` para describir la misma acción de búsqueda abstracta en clases distintas).

Clase 27: Clean Code - Parte 2 (Funciones, Comentarios y Formato)

Reglas de Calidad para Funciones

- **Extremadamente pequeñas y enfocadas:** Las funciones deben ser reducidas, idealmente de menos de 20 líneas de código y con un nivel mínimo de indentación (máximo uno o dos niveles de anidamiento).
- **Hacer una sola cosa (Do One Thing):** Una función cumple esta regla si todos sus pasos internos están exactamente un nivel de abstracción por debajo del nombre de la función.
- **Regla de Descenso (The Stepdown Rule):** El código debe leerse como una narrativa de arriba hacia abajo. Cada función debe estar seguida de forma inmediata por sus subfunciones pertenecientes al nivel inferior de abstracción.
- **Minimizar argumentos:** El número ideal de parámetros es cero (0), seguido por uno (1) y dos (2). Evitar funciones con tres o más argumentos. Eliminar parámetros de salida (es mejor retornar el estado modificado) y **argumentos tipo bandera (boolean flags)**, ya que su sola presencia explicita que la función toma caminos lógicos distintos y hace más de una cosa.
- **Separación de Comando y Consulta (CQS):** Una función debe hacer algo (cambiar el estado de una entidad) o responder algo (retornar un valor al cliente), pero nunca ambas cosas simultáneamente.

Filosofía de Comentarios y Formateo

Los comentarios se consideran fallas de nuestra expresividad en el código. Es preferible invertir esfuerzo en refactorizar un bloque complejo para hacerlo legible por sí mismo que escribir un comentario extenso explicativo. El código descartado mediante comentarios debe eliminarse de inmediato (para recuperar historial se utiliza Git).
El Formato Vertical se rige por la metáfora del periódico: archivos pequeños (menos de 500 líneas), con los conceptos de alto nivel y nombres claros arriba de todo, descendiendo progresivamente hacia los detalles técnicos de implementación abajo. Separar bloques lógicos con líneas en blanco. El Formato Horizontal exige líneas de código cortas (menos de 120 caracteres) y el uso estratégico de espacios para enfatizar asignaciones y precedencia de operadores.

Módulo 10: La Guía Definitiva de los Principios SOLID

Clase 29: Principios de Diseño SOLID

SOLID es la sigla nemotécnica que consolida las cinco directrices esenciales de diseño arquitectónico orientadas a objetos para garantizar la flexibilidad ante cambios y máxima reusabilidad:

1. SRP: Principio de Responsabilidad Única (Single Responsibility Principle)

Directriz: Una clase debe tener una, y solo una, razón para cambiar. Esto significa que debe asumir una única responsabilidad enfocada en el sistema. Cuando una clase acumula múltiples roles, las lógicas se acoplan y cualquier cambio menor para resolver una necesidad altera o rompe el funcionamiento de las restantes.

- **Violación común:** Una clase `Factura` que calcula el IVA, imprime la factura en formato PDF y guarda los registros en una base de datos MySQL. Reúne tres razones distintas para cambiar.
- **Solución correcta:** Mantener a `Factura` como entidad de puros datos, delegando la persistencia a una clase `RepositorioFactura` y la generación visual a un módulo `ImpresorFacturaPDF`.

2. OCP: Principio de Abierto / Cerrado (Open-Closed Principle)

Directriz: Las entidades de software deben estar abiertas a la extensión pero cerradas a la modificación. Debe ser completamente viable incorporar nuevas funcionalidades o alterar comportamientos incorporando código nuevo (subclases o implementaciones), manteniendo el código existente intacto y libre de modificaciones o recompilaciones.

- **Violación común:** Un método `procesarPago(MetodoPago m)` que utiliza estructuras rígidas `if (m.tipo == "TARJETA") ... else if (m.tipo == "EFECTIVO")`. Anexar el pago por criptomonedas obliga a modificar el código fuente del procesador.
- **Solución correcta:** Declarar `MetodoPago` como una interfaz abstracta con el servicio `pagar()`. Cada medio de pago concreto (`Tarjeta`, `Efectivo`, `Crypto`) implementa su propio algoritmo. El procesador simplemente ejecuta `m.pagar()`, cerrando el código a cambios futuros y abriéndose a infinitas extensiones.

3. LSP: Principio de Sustitución de Liskov (Liskov Substitution Principle)

Directriz: Las clases derivadas deben poder sustituir de forma completamente transparente a sus clases bases sin alterar la corrección ni la estabilidad del programa. Exige consistencia semántica: las subclases deben respetar los contratos, precondiciones y postcondiciones asumidos por el padre.

- **Violación común:** Diseñar que la clase `Cuadrado` sea una subclase heredera de `Rectangulo`. Si el código del cliente invoca los métodos de modificación fijando un ancho de 4 y un alto de 8 esperando un área de 32, la clase `Cuadrado` se ve forzada a alterar ambas dimensiones en conjunto para mantener sus lados iguales, retornando un área incorrecta de 64 y violando el contrato del padre.
- **Solución correcta:** Separar las estructuras. `Cuadrado` y `Rectangulo` deben ser clases independientes que hereden de una abstracción común superior llamada `Paralelogramo` o `FiguraGeometrica` que solo exponga el servicio de consulta `calcularArea()`.

4. ISP: Principio de Segregación de Interfaces (Interface Segregation Principle)

Directriz: Los clientes no deben verse obligados a depender de interfaces o métodos que no utilizan. Es preferible diseñar múltiples interfaces pequeñas, cohesionadas y altamente específicas en lugar de una única interfaz robusta o generalizada.

- **Violación común:** Una interfaz `Trabajador` que define las firmas `trabajar()`, `comer()` y `asistirAReuniones()`. Al implementar una clase `RobotInteligente`, se ve forzada a dejar métodos vacíos o lanzar excepciones en `comer()`.
- **Solución correcta:** Segregar en interfaces atómicas independientes: `Trabajable` (`trabajar()`), `Alimentable` (`comer()`), `Corporativo` (`asistirAReuniones()`). Cada clase concreta adopta únicamente el grupo de interfaces que corresponden a su naturaleza.

5. DIP: Principio de Inversión de Dependencia (Dependency Inversion Principle)

Directriz: Los módulos de alto nivel (lógica principal) no deben depender de módulos de bajo nivel (detalles técnicos e infraestructura); ambos deben depender exclusivamente de abstracciones (interfaces o clases abstractas). A su vez, las abstracciones no deben depender de los detalles concretos; los detalles deben depender de las abstracciones.

- **Violación común:** Una clase de alto nivel `Interruptor` que declara internamente una variable de tipo estático acoplada a la clase concreta de bajo nivel `LamparaLed` para encenderla. El interruptor queda encadenado a ese único artefacto físico.
- **Solución correcta:** Diseñar una interfaz abstracta intermedia llamada `DispositivoConectable` que exponga las firmas `encender()` y `apagar()`. La clase `Interruptor` pasa a depender puramente de esta interfaz. La clase concreta `LamparaLed` (o un `Motor`, o un `Ventilador`) implementa dicha interfaz. De esta forma, el detalle técnico pasa a depender de la abstracción, logrando una arquitectura desacoplada y flexible.