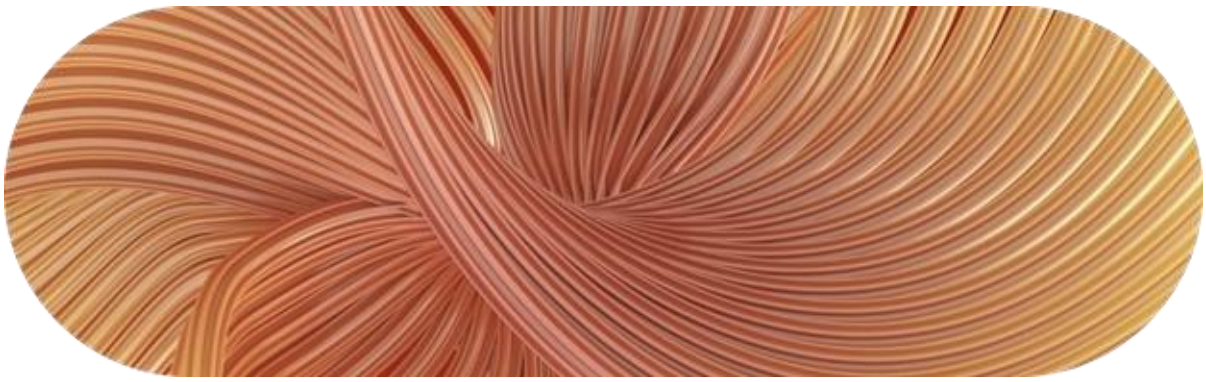
MANUAL TÉCNICO



Francisco Miser Junquera

Marcos Vidal González

David Torres Rial

1. Clase AlmacenCentral

1.1 Descripción

Representa un almacén central capaz de almacenar comida animal y vegetal, permitiendo gestionar eficientemente los recursos de la piscifactoría. Proporciona funcionalidades para aumentar la capacidad, añadir comida y distribuir los recursos a las piscifactorías registradas.

1.2 Funcionalidades

- Almacenar comida animal y vegetal en cantidades limitadas por la capacidad máxima del almacén.
- Ampliar la capacidad máxima mediante el gasto de monedas.
- Distribuir de forma equitativa la comida (animal y vegetal) entre las piscifactorías según sus necesidades.
- Imprimir un informe detallado de la capacidad, cantidades almacenadas y porcentaje de ocupación.

1.3 Constructor

- `public AlmacenCentral()`
  Inicializa el almacén con los siguientes valores predeterminados:
  - **capacidadMaxima:** 200 (capacidad inicial).
  - **cantidadComidaAnimal:** 0 (sin comida animal al inicio).
  - **cantidadComidaVegetal:** 0 (sin comida vegetal al inicio).

1.4 Atributos

- `private int capacidadAlmacen`: Capacidad máxima que puede contener el almacén.
- `private int cantidadComidaAnimal`: Cantidad actual de comida animal almacenada.
- `private int cantidadComidaVegetal`: Cantidad actual de comida vegetal almacenada.
- `private final int costoMejora`: Costo fijo (200 monedas) necesario para mejorar la capacidad del almacén.

1.5 Métodos

- **aumentarCapacidad**
  `public void aumentarCapacidad()`
  Incrementa la capacidad del almacén en 50 unidades si se dispone de suficientes monedas, mostrando un mensaje con el nuevo nivel.
- **añadirComidaAnimal**
  `public void añadirComidaAnimal(int cantidad)`
  Agrega la cantidad especificada de comida animal, validando que no se supere la capacidad total. Si la cantidad excede la capacidad, se muestra un mensaje de error.
- **añadirComidaVegetal**
  `public void añadirComidaVegetal(int cantidad)`
  Agrega la cantidad especificada de comida vegetal, siempre que la cantidad sea válida y no exceda la capacidad total; de lo contrario, se informa que la capacidad ha sido superada.
- **distribuirComida**
  `public void distribuirComida(List<Piscifactoria> piscifactorias)`
  Reparte equitativamente la comida disponible entre las piscifactorías registradas, evaluando las necesidades y el espacio libre en cada tanque.

2. Clase SistemaMonedas

2.1 Descripción
Gestiona las monedas virtuales dentro de la simulación, implementándose como un patrón Singleton para asegurar una única instancia global. Permite consultar, aumentar y gastar monedas, aplicando descuentos en determinadas transacciones.

2.2 Funcionalidades

- Consultar, aumentar y reducir la cantidad de monedas disponibles.
- Validar que las transacciones se realicen solo si el saldo es suficiente.
- Aplicar un descuento automático al calcular costos según la cantidad de recursos gestionados.

2.3 Constructor

- `private SistemaMonedas()`
  Inicializa el sistema con un saldo inicial de 100 monedas. Al ser privado, garantiza la existencia de una única instancia (Singleton).

2.4 Atributos

- `private int monedas`: Cantidad actual de monedas disponibles.
- `private static SistemaMonedas instance`: Única instancia permitida del sistema.

2.5 Métodos

- **ganarMonedas**
  `public boolean ganarMonedas(int cantidad)`
  Incrementa el saldo si la cantidad es válida (mayor que 0) y retorna `true`; en caso contrario, retorna `false`.
- **gastarMonedas**
  `public boolean gastarMonedas(int costo)`
  Disminuye el saldo según el costo especificado. Si el saldo es suficiente y el costo es válido, actualiza el saldo y retorna `true`; si no, retorna `false`.
- **calcularDescuento**
  `public int calcularDescuento(int cantidadComida)`
  Calcula el costo total aplicando un descuento de 5 monedas por cada 25 unidades de comida. Si la cantidad es menor a 25, se aplica el costo completo sin descuento.

3. Clase DTOPedido

3.1 Descripción

Objeto de transferencia de datos (DTO) para un pedido. Encapsula la información esencial (identificador, número de referencia, cliente, pez, cantidad total y cantidad enviada) para facilitar el intercambio de datos entre las capas del sistema sin exponer la lógica de negocio.

3.2 Funcionalidades

- Representa la información completa de un pedido.
- Facilita el transporte de datos entre los diferentes componentes del sistema.

3.3 Constructor

- `public DTOPedido(int id, String numero_referencia, int id_cliente, int id_pez, int cantidad, int cantidad_enviada)`
  Inicializa el objeto con los valores proporcionados para cada atributo:
  - **id:** Identificador único del pedido.
  - **numero_referencia:** Número de referencia del pedido.
  - **nombre_cliente:** Nombre del cliente asociado (aunque en la declaración aparece como entero, se entiende que representa el cliente).
  - **nombre_pez:** Nombre del pez asociado.
  - **cantidad:** Cantidad total solicitada.
  - **cantidad_enviada:** Cantidad enviada hasta el momento.

3.4 Atributos

- `private int id`
- `private String numero_referencia`
- `private int nombreCliente`
- `private int nombrePez`
- `private int cantidad`
- `private int cantidad_enviada`

3.5 Métodos

- No posee métodos adicionales, únicamente los *getters* para acceder a sus atributos.

4. Clase DAOPedidos

4.1 Descripción

Es el Data Access Object (DAO) para la tabla *Pedido* de la base de datos. Se encarga de gestionar la comunicación con la BD mediante DTOs para pedidos, clientes y peces, permitiendo la inserción, consulta, actualización y eliminación de pedidos, así como la generación automática de pedidos a partir de clientes y peces seleccionados aleatoriamente.

4.2 Funcionalidades

- Inserción, consulta, actualización y eliminación de pedidos.
- Generación automática de pedidos a partir de selecciones aleatorias de clientes y peces.

4.3 Constructor

- `public DAOPedidos()`
  Inicializa la conexión a la base de datos y prepara los *PreparedStatement* necesarios para ejecutar las consultas SQL, manejando posibles excepciones de SQL.

4.4 Atributos

- `private Random random`: Generador de números aleatorios para seleccionar clientes y peces.
- Constantes SQL para diversas operaciones, por ejemplo:
  - `QUERY_INSERT_PEDIDO`
  - `QUERY_LISTAR_PEDIDOS_COMPLETADOS`
  - `QUERY_LISTAR_PEDIDOS_PENDIENTES`
  - `QUERY_SELECCIONAR_PEDIDO_POR_REFERENCIA`
  - `QUERY_ACTUALIZAR_PEDIDO`
  - `QUERY_BORRAR_PEDIDOS`
  - `QUERY_RANDOM_CLIENTE`
  - `QUERY_RANDOM_PEZ`
  - `QUERY_OBTENER_PEZ`
  - `QUERY_OBTENER_NOMBRE`
- `private Connection connection`: Conexión a la base de datos.
- *PreparedStatement* para cada operación:
  - `pstInsertPedido`, `pstListarPedidosPendientes`, `pstListarPedidosCompletados`, `pstSeleccionarPedidoPorReferencia`, `pstActualizarPedido`, `pstBorrarPedidos`, `ptsObtenerPez`, `ptsObtenerNombre`, `pstRandomCliente` y `pstRandomPez`.

4.5 Métodos

- **generarPedidoAutomatico**
  `public DTOPedido generarPedidoAutomatico()`
  Genera un pedido automático seleccionando aleatoriamente un cliente y un pez, insertándolo en la base de datos.
- **listarPedidosPendientes**
  `public List<DTOPedido> listarPedidosPendientes()`
  Lista aquellos pedidos donde la cantidad enviada es menor que la solicitada.
- **listarPedidosCompletados**
  `public List<DTOPedido> listarPedidosCompletados()`
  Lista los pedidos en los que la cantidad enviada es igual o mayor que la solicitada.
- **obtenerPedidoPorReferencia**
  `public DTOPedido obtenerPedidoPorReferencia(String numeroReferencia)`
  Recupera un pedido mediante su número de referencia.
- **actualizarPedido**
  `public boolean actualizarPedido(DTOPedido pedido)`
  Actualiza la cantidad enviada de un pedido en la base de datos.
- **enviarPedido**
  `public DTOPedido enviarPedido(DTOPedido pedido, int cantidadDisponible)`
  Envía una cantidad de peces para un pedido, actualizando la cantidad enviada y retornando el objeto actualizado.
- **borrarPedidos**
  `public int borrarPedidos()`
  Elimina todos los pedidos de la base de datos y retorna el número de registros eliminados.
- **obtenerClientePorId**
  `public DTOCliente obtenerClientePorId(int idCliente)`
  Recupera los datos de un cliente mediante su ID.
- **obtenerPezPorId**
  `public DTOPez obtenerPezPorId(int idPez)`
  Recupera los datos de un pez mediante su ID.
- **close**
  `public void close()`
  Cierra la conexión a la base de datos y todos los *PreparedStatement* abiertos.

5. Clase Conexion

5.1 Descripción

Administra la conexión a una base de datos MySQL. Se encarga de establecer y cerrar la conexión mediante el *DriverManager*, permitiendo que la aplicación interactúe de forma centralizada con la base de datos. La conexión se maneja de forma estática para garantizar que exista una única instancia activa durante la ejecución.

5.2 Funcionalidades

- Establecer una conexión a la base de datos MySQL usando las credenciales y parámetros especificados.
- Proveer un método para obtener la conexión activa, creándola si aún no existe.
- Permitir cerrar la conexión y liberar los recursos asociados.

5.3 Constructor

No se requiere un constructor público, ya que la clase utiliza métodos estáticos para gestionar la conexión.

5.4 Atributos

- `private static final String USER`: Usuario de la base de datos.
- `private static final String PASSWORD`: Contraseña de la base de datos.
- `private static final String SERVER`: Dirección del servidor de la base de datos.
- `private static final String PORT`: Puerto del servidor.
- `private static final String DATABASE`: Nombre de la base de datos.

`private static Connection connection`: Objeto de conexión a la base de datos.

5.5 Métodos

- **getConnection**
  `public static Connection getConnection()`
  Retorna un objeto `Connection`. Si no existe una conexión activa, la crea

utilizando los parámetros y el *DriverManager*. En caso de error, registra el fallo.

- **closeConnection**
  `public static void closeConnection()`
  Cierra la conexión a la base de datos si está activa, liberando los recursos y estableciendo la conexión a `null`.

6. Clase GeneradorBD

6.1 Descripción

Esta clase se encarga de crear la estructura de la base de datos para el sistema de pedidos de peces. Genera las tablas necesarias (Cliente, Pez y Pedido) si no existen y, de manera condicional, inserta los datos iniciales en las tablas Cliente y Pez. De este modo se asegura que el sistema disponga de la estructura y los datos básicos requeridos para su funcionamiento.

6.2 Funcionalidades

- Crear las tablas *Cliente*, *Pez* y *Pedido* en la base de datos, definiendo columnas, claves primarias y restricciones (incluyendo claves foráneas y acciones en cascada).
- Insertar registros iniciales en la tabla *Cliente* comprobando que no se dupliquen (por ejemplo, verificando el NIF).
- Insertar registros iniciales en la tabla *Pez* utilizando datos del sistema (nombre común y nombre científico), evitando duplicados según el nombre.

Facilitar la inicialización completa del sistema mediante la ejecución secuencial de las operaciones de creación de tablas y la inserción de datos.

6.3 Constructor

No se define un constructor.

6.4 Atributos

- Constantes SQL para las operaciones:
  - `QUERY_AGREGAR_CLIENTES`: Consulta para insertar un nuevo cliente.
  - `QUERY_AGREGAR_PEZ`: Consulta para insertar un nuevo pez.
- `private Connection connection`: Objeto de conexión a la base de datos obtenido mediante `Conexion.getConnection()`.

6.5 Métodos

- **crearTablaCliente**
  Crea la tabla *Cliente* (con columnas como *id*, *nombre*, *nif* y *telefono*) si no existe.
- **crearTablaPez**
  Crea la tabla *Pez* (con columnas *id*, *nombre* y *nombre_cientifico*) si no existe.
- **crearTablaPedido**
  Crea la tabla *Pedido* (con columnas como *numero_referencia*, *id_cliente*, *id_pez*, *cantidad* y *cantidad_enviada*) si no existe.
- **agregarClientes**
  Inserta registros iniciales en la tabla *Cliente* mediante un *PreparedStatement*, utilizando arreglos de datos y verificando la existencia previa (por ejemplo, por NIF).
- **agregarPeces**
  Inserta registros iniciales en la tabla *Pez* recorriendo la lista de peces y evitando duplicados.
- **crearTablas**
  Ejecuta de forma secuencial la creación de las tablas y la inserción de los datos iniciales, integrando todas las operaciones necesarias para la inicialización del sistema.

7. Clase FileHelper

7.1 Descripción

Es una clase de utilidad estática destinada a gestionar operaciones con archivos y directorios. Permite crear carpetas, verificar el contenido de directorios, listar archivos para selección mediante un menú interactivo y extraer recompensas a partir de archivos XML.

7.2 Funcionalidades

- Crear múltiples carpetas de forma segura.
- Verificar si un directorio contiene archivos o subdirectorios.
- Listar los archivos de un directorio y mostrar un menú interactivo para que el usuario seleccione una opción.
- Obtener un arreglo con los nombres de los archivos en un directorio.
- Extraer recompensas desde archivos XML del directorio "rewards", procesando sus elementos y partes específicas.
- Eliminar contenido adicional (como texto entre corchetes) en las
  No se define, ya que todos sus métodos son estáticos.

7.4 Atributos

No se definen atributos de instancia.

7.5 Métodos

- **crearCarpetas**
  `public static void crearCarpetas(String[] carpetas)`
  Crea cada carpeta del arreglo especificado, verificando su existencia y utilizando `mkdirs()` en caso de no existir.
- **hayContenidoEnDirectorio**
  `public static boolean hayContenidoEnDirectorio(String rutaDirectorio)`
  Verifica si el directorio existe y contiene archivos o subdirectorios, retornando `true` o `false` según el caso.
- **mostrarMenuConArchivos**
  `public static String mostrarMenuConArchivos(String rutaDirectorio)`
  Muestra un menú interactivo con los nombres (sin extensión) de los archivos presentes en el directorio indicado, permitiendo la selección por parte del usuario.
- **obtenerArchivosEnDirectorio**
  `public static String[] obtenerArchivosEnDirectorio(String rutaDirectorio)`
  Retorna un arreglo con los nombres de los archivos del directorio, ordenados alfabéticamente.
- **getRewards**
  `public static String[] getRewards()`
  Extrae las recompensas desde los archivos XML ubicados en el directorio "rewards", procesando su contenido mediante un lector SAX.
- **getRewardsWithoutBrackets**
  `public static String[] getRewardsWithoutBrackets(String[] opciones)`
  Recibe un arreglo de cadenas con recompensas y elimina el contenido que aparece después del primer corchete, retornando un nuevo arreglo con las recompensas depuradas.

8. Clase InputHelper

8.1 Descripción

Gestiona la entrada de datos desde la consola de forma segura y validada, permitiendo leer cadenas de texto y números enteros, así como solicitar números dentro de un rango específico. A su vez, se encarga de cerrar el recurso utilizado (Scanner) para liberar recursos del sistema.

8.2 Funcionalidades

- Leer cadenas de texto alfanuméricas que no estén vacías y que no contengan caracteres especiales.
- Leer números enteros, mostrando mensajes de error en caso de entrada no válida.
- Solicitar un número entero dentro de un rango definido (mínimo y máximo) y validar la entrada.
- Cerrar el Scanner utilizado para liberar recursos.

8.3 Constructor

No se define un constructor.

8.4 Atributos

- `private static final Scanner scanner`: Objeto para gestionar las entradas por consola de manera eficiente.

8. 5 Métodos

- **readString**
  `public static String readString(String prompt)`
  Lee una cadena desde la consola, validándola para que no esté vacía ni contenga caracteres especiales.
- **readInt**
  `public static int readInt(String prompt)`
  Solicita al usuario un número entero y valida que la entrada sea numérica; en caso de error, se vuelve a solicitar.
- **solicitarNumero**
  `public static int solicitarNumero(int min, int max)`
  Solicita un número dentro de un rango específico, validando que la entrada se encuentre entre el valor mínimo y máximo.
- **close**
  `public static void close()`
  Cierra el Scanner utilizado, registrando un error si este ya había sido cerrado.

9. Clase MenuHelper

9.1 Descripción

Facilita la creación y visualización de menús en la consola. Permite mostrar opciones numeradas para que el usuario seleccione una acción, incorporando también una opción especial para cancelar la operación.

9.2 Funcionalidades

- Mostrar una lista numerada de opciones en la consola para la selección de una acción.
- Incluir una opción adicional para cancelar la operación y regresar a un menú anterior.

9.3 Constructor

No se define un constructor.

9.4 Atributos

No se definen atributos en esta clase.

9.5 Métodos

- **mostrarMenuCancelar**
  `public static void mostrarMenuCancelar(String[] opciones)`
  Muestra un menú numerado que incluye la opción "0. Cancelar", permitiendo al usuario cancelar la operación.
- **mostrarMenu**
  `public static void mostrarMenu(String[] opciones)`
  Muestra un menú numerado de opciones sin la opción de cancelación.

10. Clase abstracta CarnivoroActivo

10.1 Descripción

Clase abstracta que extiende de la clase *Pez* para representar peces carnívoros con un comportamiento activo. Su diseño permite simular un mayor consumo de comida animal cuando el pez se encuentra en un estado activo.

10.2 Funcionalidades

- Alimentar al pez basándose en un comportamiento activo, determinado de forma aleatoria.

- Incrementar el consumo de comida animal: si el pez está activo consume dos unidades; de lo contrario, consume una unidad.
- Actualizar el estado del pez en función de si logra alimentarse o no, según la disponibilidad de recursos locales o del almacén central.

10.3 Constructor

- `public CarnivoroActivo(boolean sexo, PecesDatos datos)`
  Inicializa un pez carnívoro con comportamiento activo, llamando al constructor de la superclase mediante `super()`. Permite definir el sexo (por ejemplo, `true` para macho y `false` para hembra) y establece los atributos principales a partir de un objeto `PecesDatos`.

10.4 Atributos

- `Random rand`: Generador de números aleatorios utilizado para simular la probabilidad del comportamiento activo.

10.5 Métodos

- **alimentar**
  `@Override public int alimentar(int cantidadComidaAnimal, int cantidadComidaVegetal)`
  Implementa el proceso de alimentación:
  - Con una probabilidad del 50%, el pez se considera activo y consume dos unidades de comida animal;
  - Si no está activo, consume una unidad.
  - Si la comida local es insuficiente, se intenta obtenerla del almacén central; si no hay recursos, el pez permanece sin alimentarse.
    Devuelve:
  - **2:** Si el pez está activo y consume dos unidades.
  - **1:** Si el pez consume una unidad en estado no activo.
  - **0:** Si no se logra alimentar por falta de recursos.

11. Clase abstracta Carnivoro

11.1 Descripción

Se define la clase abstracta *Carnivoro*, que extiende de la clase *Pez*, para representar aquellos peces cuya dieta se basa en comida animal. La implementación de su método de alimentación permite consumir comida animal disponible localmente o, en su defecto, obtenerla del almacén central, asegurando la correcta nutrición del pez.

11.2 Funcionalidades

- Permitir la alimentación del pez usando comida animal, ya sea la disponible localmente o del almacén central.
- Actualizar el estado de alimentación del pez en función de si logra o no consumir alimento.

11.3 Constructor

- `public Carnivoro(boolean sexo, PecesDatos datos)`
  Inicializa el pez con las características propias de un carnívoro, utilizando la llamada a `super()` para heredar las propiedades básicas definidas en la clase *Pez*.

11.4 Atributos

- No se definen atributos propios en esta clase.

11.5 Métodos

- **alimentar**
  `@Override public int alimentar(int cantidadComidaAnimal, int cantidadComidaVegetal)`
  Implementa el proceso de alimentación del pez. Si hay comida animal disponible localmente, se consume una unidad y se marca el pez como alimentado; en caso de no disponer de comida local, se intenta obtener una unidad del almacén central.
    - Devuelve **1** si el pez consume comida local.
    - Devuelve **0** si consume del almacén central o si no hay comida disponible.

12. Clase abstracta Filtrador

12.1 Descripción

La clase abstracta *Filtrador* extiende de *Pez* y se utiliza para representar peces que se alimentan exclusivamente de comida vegetal. Su método de alimentación permite consumir alimento vegetal disponible en el entorno local o, en caso de escasez, obtenerlo del almacén central.

12.2 Funcionalidades

- Alimentar al pez utilizando comida vegetal, priorizando la fuente local.
- Simular el proceso de búsqueda de alimento mediante un generador aleatorio.
- Actualizar el estado de alimentación en función del éxito al consumir alimento.

12.3 Constructor

- `public Filtrador(boolean sexo, PecesDatos datos)`
  Inicializa un pez filtrador, llamando al constructor de la superclase para establecer las propiedades básicas y utilizando los datos provistos por el objeto *PecesDatos*.

12.4 Atributos

- `Random rand`: Generador de números aleatorios empleado para simular el comportamiento de alimentación.

12.5 Métodos

- **alimentar**
  `@Override public int alimentar(int cantidadComidaAnimal, int cantidadComidaVegetal)`
  Intenta alimentar al pez consumiendo una unidad de comida vegetal, preferentemente desde el recurso local; si no está disponible, se recurre al almacén central.
    - Devuelve **1** si el pez consume comida vegetal local.
    - Devuelve **0** si debe consumir del almacén central o si no hay alimento disponible.

13. Clase abstracta Omnivoro

13.1 Descripción

La clase abstracta *Omnivoro* extiende de *Pez* y representa a aquellos peces con una dieta mixta, que consumen tanto comida animal como vegetal. Su método de alimentación evalúa la disponibilidad de ambos tipos de alimento, priorizando el recurso local y utilizando un generador aleatorio para simular la probabilidad de alimentación.

13.2 Funcionalidades

- Permitir que el pez se alimente con comida animal o vegetal, según la disponibilidad en el entorno.
- Priorizar el consumo de alimento local y, en caso de ausencia, recurrir al almacén central.
- Actualizar el estado del pez en función del tipo de alimento consumido.

13.3 Constructor

- `public Omnivoro(boolean sexo, PecesDatos datos)`
  Inicializa un pez omnívoro llamando al constructor de la superclase y estableciendo las características específicas a través de los datos proporcionados por *PecesDatos*.

13.4 Atributos

- `Random rand`: Generador de números aleatorios que determina la probabilidad de que el pez se alimente.

13.5 Métodos

- **alimentar**
  `@Override public int alimentar(int cantidadComidaAnimal, int cantidadComidaVegetal)`
  Con una probabilidad del 75%, el pez intenta alimentarse. Si hay comida disponible localmente, consume una unidad (ya sea animal o vegetal); de lo contrario, intenta consumir del almacén central.
    - Devuelve **1** si el pez se alimenta localmente.
    - Devuelve **0** si debe recurrir al almacén central o no consigue alimento.

14. Clase Dorada

14.1 Descripción

La clase *Dorada* es una subclase de *Omnivoro* que representa un pez dorado. Se caracteriza por consumir tanto comida animal como vegetal, heredando el comportamiento de alimentación de la clase *Omnivoro*. Además, permite crear instancias y clonar el objeto especificando el sexo.

14.2 Funcionalidades

- Heredar y utilizar el comportamiento de alimentación definido en *Omnivoro*.
- Permitir la creación de instancias con un sexo determinado.
- Facilitar la clonación del objeto, generando copias exactas con un posible cambio en el sexo.

14.3 Constructor

- `public Dorada(boolean sexo)`
  Inicializa un pez Dorada utilizando la constante predefinida `AlmacenPropiedades.DORADA` para asignar automáticamente sus características específicas, llamando a `super()`.

14.4 Atributos

- No se definen atributos adicionales en esta clase.

14.5 Métodos

- **clonar**
  `@Override public Dorada clonar(boolean nuevoSexo)`
  Crea una nueva instancia de *Dorada* con el sexo especificado, replicando las propiedades de la instancia actual.
    - Devuelve una nueva instancia de *Dorada* con el sexo indicado.

15. Clase SalmonAtlantico

15.1 Descripción
La clase SalmonAtlantico es una subclase de Carnivoro que representa al pez salmón atlántico, el cual se alimenta exclusivamente de comida animal. Se inicializa con propiedades específicas definidas en AlmacenPropiedades.

15.2 Funcionalidades

- Heredar el comportamiento de alimentación de *Carnivoro*, consumiendo únicamente comida animal disponible.
- Permitir la creación de instancias y la clonación del objeto, especificando el sexo.

15.3 Constructor

- `public SalmonAtlantico(boolean sexo)`
  Inicializa el pez salmón atlántico utilizando la constante `AlmacenPropiedades.SALMON_ATLANTICO`, llamando al constructor de la superclase para establecer las propiedades correspondientes.

15.4 Atributos

- No se definen atributos adicionales en esta clase.

15.5 Métodos

- **clonar**
  `@Override public SalmonAtlantico clonar(boolean nuevoSexo)`
  Crea una nueva instancia de *SalmonAtlantico* con el sexo indicado, replicando las propiedades de la instancia actual.
    - Devuelve una nueva instancia de *SalmonAtlantico* con el sexo especificado.

16. Clase TruchaArcoiris

16.1 Descripción

Representa un pez de tipo trucha arcoíris, diseñado como una subclase de *Carnivoro*. Este pez se alimenta exclusivamente de comida animal y se configura con propiedades específicas definidas en un almacén de datos centralizado.

16.2 Funcionalidades

- Hereda el comportamiento de alimentación de la clase *Carnivoro*.
- Permite la creación de nuevas instancias y la clonación del objeto, posibilitando replicar la instancia actual con un sexo distinto.

16.3 Constructor

- `public TruchaArcoiris(boolean sexo)`
  Inicializa un pez trucha arcoíris mediante la invocación de `super()`, utilizando la constante predefinida (por ejemplo, `AlmacenPropiedades.TRUCHA_ARCOIRIS`) para asignar sus características específicas.

16.4 Atributos

- No se definen atributos adicionales en esta clase.

16.5 Métodos

- **clonar**
  `@Override public TruchaArcoiris clonar(boolean nuevoSexo)`
  Crea y devuelve una nueva instancia de *TruchaArcoiris* con el sexo especificado, replicando las propiedades de la instancia actual.

17. Clase ArenqueDelAtlantico

17.1 Descripción

Se define como una subclase de *Filtrador* para representar un pez que se alimenta exclusivamente de comida vegetal. Está configurado con propiedades específicas a partir de un almacén central de datos y permite la clonación para generar instancias con diferentes sexos.

17.2 Funcionalidades

- Hereda el método de alimentación de la clase *Filtrador*, consumiendo comida vegetal localmente o desde el almacén central.
- Permite crear nuevas instancias y clonar el objeto con un sexo definido.

17.3 Constructor

- `public ArenqueDelAtlantico(boolean sexo)`
  Inicializa el pez utilizando la constante `AlmacenPropiedades.ARENQUE_ATLANTICO` para asignar sus características específicas, a través de la llamada a `super()`.

17.4 Atributos

- No se definen atributos adicionales.

17.5 Métodos

- **clonar**
  `@Override public ArenqueDelAtlantico clonar(boolean nuevoSexo)`
  Genera y retorna una nueva instancia de *ArenqueDelAtlantico* con el sexo indicado.

18. Clase Besugo

18.1 Descripción

Representa un pez de tipo besugo, definido como una subclase de *Carnivoro*. Se alimenta exclusivamente de comida animal y se configura mediante propiedades específicas establecidas en un almacén central de datos.

18.2 Funcionalidades

- Hereda el comportamiento de alimentación de la clase *Carnivoro*.
- Permite la creación de instancias y la clonación del objeto para replicar el pez con un sexo diferente.

18.3 Constructor

- `public Besugo(boolean sexo)`
  Inicializa el pez besugo utilizando la constante `AlmacenPropiedades.BESUGO` a través de una llamada a `super()`.

18.4 Atributos

- No se definen atributos adicionales.

18.5 Métodos

- **clonar**
  `@Override public Besugo clonar(boolean nuevoSexo)`
  Crea y devuelve una nueva instancia de *Besugo* con el sexo especificado.

19. Clase LenguadoEuropeo

19.1 Descripción

Se define como una subclase de *Carnivoro* que representa al pez lenguado europeo, el cual se alimenta exclusivamente de comida animal. Está configurado mediante propiedades específicas definidas en un almacén central.

19.2 Funcionalidades

- Utiliza el comportamiento de alimentación heredado de la clase *Carnivoro*.
- Permite la creación y clonación de instancias para generar copias con un sexo alternativo.

19.3 Constructor

- `public LenguadoEuropeo(boolean sexo)`
  Inicializa el pez lenguado europeo haciendo uso de la constante `AlmacenPropiedades.LENGUADO_EUROPEO` a través de la llamada a `super()`.

19.4 Atributos

- No se definen atributos adicionales.

19.5 Métodos

- **clonar**
  `@Override public LenguadoEuropeo clonar(boolean nuevoSexo)`
  Retorna una nueva instancia de *LenguadoEuropeo* con el sexo indicado, replicando la configuración de la instancia actual.

20. Clase Robalo

20.1 Descripción

Se define como una subclase de *Carnivoro* que representa al pez robalo. Este pez se alimenta exclusivamente de comida animal y se configura con propiedades específicas obtenidas de un almacén central de datos.

20.2 Funcionalidades

- Hereda el método de alimentación de la clase *Carnivoro*, consumiendo únicamente comida animal.
- Permite la creación de nuevas instancias y la clonación del objeto, permitiendo cambiar el sexo de la instancia clonada.

20.3 Constructor

- `public Robalo(boolean sexo)`
  Inicializa el pez robalo utilizando la constante `AlmacenPropiedades.ROBALO`, invocando a `super()` para establecer sus propiedades.

20.4 Atributos

- No se definen atributos adicionales.

20.5 Métodos

- **clonar**
  `@Override public Robalo clonar(boolean nuevoSexo)`
  Crea y devuelve una nueva instancia de *Robalo* con el sexo especificado, replicando las características de la instancia actual.

21. Clase LubinaRayada

21.1 Descripción

Representa un pez de tipo Lubina Rayada, definido como una subclase de *Carnivoro*. Se caracteriza por alimentarse exclusivamente de comida animal y se configura utilizando propiedades específicas definidas en el almacén de datos. Permite crear instancias del pez y realizar clonaciones con un sexo determinado.

21.2 Funcionalidades

- Hereda el comportamiento de alimentación de la clase *Carnivoro*, consumiendo comida animal disponible localmente o desde el almacén central.
- Permite la creación de nuevas instancias con un sexo específico y la generación de copias exactas mediante el método de clonación.

21.3 Constructor

- `public LubinaRayada(boolean sexo)`
  Inicializa el pez Lubina Rayada llamando al constructor de la superclase (utilizando `super()`) y asignando automáticamente las características específicas a través de la constante predefinida `AlmacenPropiedades.LUBINA_RAYADA`.
  - **Parámetro:**
    - `sexo`: Indica el sexo del pez (por ejemplo, `true` para macho y `false` para hembra).

21.4 Atributos

- No se definen atributos adicionales.

21.5 Métodos

- **clonar**
  `@Override public LubinaRayada clonar(boolean nuevoSexo)`
  Crea y devuelve una nueva instancia de *LubinaRayada* con el sexo especificado, replicando las propiedades de la instancia actual.

22. Clase CarpaPlateada

22.1 Descripción

Representa al pez Carpa Plateada como una subclase de *Filtrador*. Se alimenta exclusivamente de comida vegetal y se configura con propiedades específicas establecidas en el almacén de datos. Permite crear instancias del pez y clonarlas con un sexo determinado.

22.2 Funcionalidades

- Hereda el comportamiento de alimentación de la clase *Filtrador*, consumiendo comida vegetal ya sea del recurso local o del almacén central.
- Permite la creación de nuevas instancias y la clonación del objeto para replicar la instancia con un sexo alternativo.

22.3 Constructor

- `public CarpaPlateada(boolean sexo)`
  Inicializa el pez Carpa Plateada mediante la llamada a `super()`, utilizando la constante predefinida `AlmacenPropiedades.CARPA_PLATEADA` para asignar sus propiedades específicas.
  - **Parámetro:**
    - `sexo`: Define el sexo del pez (por ejemplo, `true` para macho y `false` para hembra).

22.4 Atributos

- No se definen atributos adicionales.

22.5 Métodos

- **clonar**
  `@Override public CarpaPlateada clonar(boolean nuevoSexo)`
  Genera y retorna una nueva instancia de *CarpaPlateada* con el sexo especificado, replicando las características de la instancia actual.

23. Clase Pejerrey

23.1 Descripción

Se define como una subclase de *Carnivoro* que representa al pez Pejerrey, el cual se alimenta exclusivamente de comida animal. Se configura utilizando propiedades específicas establecidas en el almacén de datos y permite crear instancias y clonarlas cambiando el sexo si se requiere.

23.2 Funcionalidades

- Hereda el comportamiento de alimentación de la clase *Carnivoro*, utilizando comida animal disponible localmente o del almacén central.
- Facilita la creación y clonación de instancias con un sexo determinado.

23.3 Constructor

- `public Pejerrey(boolean sexo)`
  Inicializa el pez Pejerrey llamando a `super()` y utilizando la constante predefinida `AlmacenPropiedades.PEJERREY` para asignar automáticamente sus propiedades específicas.
    - **Parámetro:**
        - `sexo`: Define el sexo del pez (por ejemplo, `true` para macho y `false` para hembra).

23.4 Atributos

- No se definen atributos adicionales.

23.5 Métodos

- **clonar**
  `@Override public Pejerrey clonar(boolean nuevoSexo)`
  Crea y devuelve una nueva instancia de *Pejerrey* con el sexo especificado, replicando las características de la instancia actual.

24. Clase PercaEuropea

24.1 Descripción

Se define como una subclase de *Carnivoro Activo* que representa al pez Perca Europea, el cual se alimenta exclusivamente de comida animal. Está configurado mediante propiedades específicas definidas en el almacén de datos, y permite crear instancias y clonarlas cambiando el sexo según se requiera.

24.2 Funcionalidades

- Hereda el comportamiento de alimentación de la clase *Carnivoro Activo*, consumiendo comida animal disponible localmente o del almacén central.
- Permite la creación y clonación de instancias con un sexo definido, facilitando la replicación del objeto.

24.3 Constructor

- `public PercaEuropea(boolean sexo)`
  Inicializa el pez Perca Europea mediante la llamada a `super()` y utilizando la constante predefinida `AlmacenPropiedades.PERCA_EUROPEA` para asignar automáticamente sus características específicas.
    - **Parámetro:**
        - `sexo`: Indica el sexo del pez (por ejemplo, `true` para macho y `false` para hembra).

24.4 Atributos

- No se definen atributos adicionales.

24.5 Métodos

- **clonar**
  `@Override public PercaEuropea clonar(boolean nuevoSexo)`
  Genera y retorna una nueva instancia de *PercaEuropea* con el sexo especificado, replicando las propiedades de la instancia actual.

25. Clase SalmonChinook

25.1 Descripción

Se define como una subclase de *Carnivoro* que representa al pez SalmonChinook, caracterizado por alimentarse exclusivamente de comida animal. Se configura utilizando propiedades específicas definidas en el almacén de datos.

25.2 Funcionalidades

- Hereda el comportamiento de alimentación de la clase *Carnivoro*, consumiendo únicamente comida animal disponible localmente o del almacén central.
- Permite la creación de nuevas instancias y la clonación del objeto para replicar la instancia con un sexo determinado.

25.3 Constructor

- `public SalmonChinook(boolean sexo)`
  Inicializa el pez SalmonChinook mediante la llamada a `super()`, utilizando la constante predefinida (por ejemplo, `AlmacenPropiedades.SALMON_CHINOOK`) para asignar automáticamente las propiedades específicas de la especie.
    - **Parámetro:**
        - `sexo`: Define el sexo del pez (por ejemplo, `true` para macho y `false` para hembra).

25.4 Atributos

- No se definen atributos adicionales.

25.5 Métodos

- **clonar**
  `@Override public SalmonChinook clonar(boolean nuevoSexo)`
  Crea y devuelve una nueva instancia de *SalmonChinook* con el sexo especificado, replicando las características de la instancia actual.

26. Clase TilapiaDelNilo

26.1 Descripción

Representa la especie de pez conocida como Tilapia del Nilo dentro de la simulación. Esta clase modela las características específicas de la especie, permitiendo la creación de instancias y la clonación para replicar el pez con posibles variaciones en su sexo.

26.2 Funcionalidades

- Crear instancias de Tilapia del Nilo asignándole un sexo definido.
- Facilitar la clonación para replicar el objeto con un sexo alternativo.

26.3 Constructor

- `public TilapiaDelNilo(boolean sexo)`
  Inicializa la instancia configurando el sexo del pez y asignándole automáticamente las propiedades específicas de la especie.

26.4 Atributos

- No se definen atributos adicionales en esta clase.

26.5 Métodos

- **clonar**
  `@Override public TilapiaDelNilo clonar(boolean nuevoSexo)`
  Crea y retorna una nueva instancia de *TilapiaDelNilo* con el sexo especificado, replicando las características de la instancia actual.

27. Clase Pez

27.1 Descripción

Es la clase base que representa a un pez en la piscifactoría. Define las propiedades y comportamientos comunes (como el estado, crecimiento, reinicio y alimentación) que comparten todas las especies, sirviendo de fundamento para las subclases específicas.

27.2 Funcionalidades

- Proveer una estructura común para gestionar atributos (estado, tamaño, peso, etc.) y comportamientos básicos de los peces.
- Permitir que las subclases especialicen o extiendan la funcionalidad común (por ejemplo, en el método de alimentación o clonación).

27.3 Constructor

- `public Pez(...)`
  Inicializa los atributos básicos del pez. (La firma exacta dependerá de la implementación, incluyendo parámetros como tamaño, peso, estado de alimentación, etc.)

27.4 Atributos

- Atributos generales que definen el estado del pez, tales como:
  - Estado de alimentación.
  - Tamaño, peso y otras características relevantes.
    (Los detalles exactos dependen de la lógica del sistema).

27.5 Métodos

- **showStatus**
  `public void showStatus()`
  Muestra el estado actual del pez, incluyendo información relevante sobre su alimentación y condición general.
- **grow**
  `public void grow()`
  Permite que el pez crezca, actualizando sus atributos correspondientes, se reproduzca y se emferme.
- **reset**
  `public void reset()`
  Reinicia el estado del pez, útil para iniciar un nuevo ciclo o día en la simulación.
- **clonar**
  `public Pez clonar(boolean nuevoSexo)`
  Método (posiblemente abstracto) que permite clonar la instancia actual, generando una copia con un sexo alternativo.
- **alimentar**
  `public int alimentar(int cantidadComidaAnimal, int cantidadComidaVegetal)`
  Procesa la alimentación del pez, actualizando su estado según la cantidad de comida consumida. El valor de retorno indica el éxito de la alimentación (por ejemplo, 0, 1 o 2).

28. Clase Piscifactoria

28.1 Descripción

Representa la entidad central de la piscifactoría, encargada de gestionar los tanques, peces y el estado general de la instalación. Coordina la supervisión y actualización de diversos aspectos como la alimentación, la capacidad y el estado de los tanques.

28.2 Funcionalidades

- Mostrar el estado general de la piscifactoría, de los tanques y de los peces.
- Gestionar recursos como la comida (animal y vegetal) y la capacidad de la instalación.
- Permitir la transición al siguiente día en la simulación, actualizando estados y recursos.
- Facilitar mejoras en la alimentación y la ampliación de la infraestructura mediante la adición de nuevos tanques.

28.3 Constructor

- `public Piscifactoria(...)`
  Inicializa la instancia estableciendo los valores iniciales para la capacidad, recursos y estructura (lista de tanques y peces).
  (La firma exacta dependerá de la implementación).

28.4 Atributos

- Atributos que representan:
  - El estado general de la piscifactoría (por ejemplo, nivel de comida, capacidad, etc.).
  - Colecciones o listas de tanques y peces.
    (Los detalles específicos varían según la implementación del sistema).

28.5 Métodos

- **showStatus**
  `public void showStatus()`
  Muestra un resumen del estado general de la piscifactoría.
- **showTankStatus**
  `public void showTankStatus()`
  Muestra el estado de cada tanque, incluyendo detalles sobre ocupación y condiciones de los peces.
- **showFishStatus**
  `public void showFishStatus()`
  Detalla el estado individual de los peces dentro de la instalación.
- **showCapacity**
  `public void showCapacity()`
  Indica la capacidad actual y el nivel de ocupación de la piscifactoría.
- **showFood**
  `public void showFood()`
  Muestra la cantidad de comida disponible para alimentar a los peces.
- **nextDay**
  `public void nextDay()`
  Avanza la simulación al siguiente día, actualizando el estado general y de los peces.
- **upgradeFood**
  `public void upgradeFood()`
  Permite mejorar o incrementar la calidad/cantidad de comida disponible.
- **addTanque**
  `public void addTanque()`
  Agrega un nuevo tanque a la infraestructura de la piscifactoría.
- **añadirComidaAnimal**
  `public void añadirComidaAnimal(int cantidad)`
  Permite añadir una cantidad específica de comida animal.
- **añadirComidaVegetal**
  `public void añadirComidaVegetal(int cantidad)`
  Permite añadir una cantidad específica de comida vegetal.
- **alimentarPeces**
  `public void alimentarPeces()`
  Ejecuta el proceso de alimentación de los peces, actualizando sus estados según la disponibilidad de recursos.

29. Clase PiscifactoriaDeMar

29.1 Descripción

Representa una variante de la piscifactoría especializada en ambientes marinos. Adapta las funcionalidades generales de la piscifactoría para operar en condiciones propias del entorno marino, gestionando recursos y tanques adecuados para peces de mar.

29.2 Funcionalidades

- Actualizar la comida adaptada a condiciones marinas.
- Permitir la incorporación de tanques diseñados para peces de mar.

29.3 Constructor

- `public PiscifactoriaDeMar(...)`
  Inicializa la instancia configurando parámetros específicos para un entorno marino, heredando las propiedades generales de la piscifactoría.

29.4 Atributos

- Atributos específicos que distinguen esta variante (por ejemplo, condiciones de agua, parámetros ambientales, etc.).
  (Los detalles exactos dependen de la implementación).

29.5 Métodos

- **upgradeFood**
  `public void upgradeFood()`
  Mejora la calidad o cantidad de comida disponible en la piscifactoría de mar, adaptándose a las condiciones del ambiente.
- **addTanque**
  `public void addTanque()`
  Agrega un nuevo tanque diseñado para peces de mar, ampliando la capacidad de la instalación.

30. Clase PiscifactoriaDeRio

30.1 Descripción

Representa la variante de la piscifactoría destinada a operar en ambientes fluviales. Adapta las funcionalidades generales a las condiciones específicas de un entorno de río, permitiendo gestionar tanques y recursos acorde a un ambiente de agua dulce.

30.2 Funcionalidades

- Actualizar la comida adaptada a las necesidades y características de un ambiente de río.
- Facilitar la adición de tanques diseñados específicamente para peces de río.

30.3 Constructor

- `public PiscifactoriaDeRio(...)`
  Inicializa la instancia configurando parámetros particulares para un entorno fluvial, heredando las propiedades generales de la piscifactoría.

30.4 Atributos

- Atributos específicos que distinguen la piscifactoría de río, como condiciones del agua y capacidad adaptada al entorno fluvial.
  (Los detalles dependen de la implementación).

30.5 Métodos

- **upgradeFood**
  `public void upgradeFood()`
  Permite mejorar o incrementar la calidad y cantidad de comida en la piscifactoría de río, ajustándose a las condiciones del entorno.
- **addTanque**
  `public void addTanque()`
  Agrega un nuevo tanque adaptado para peces de río, ampliando la infraestructura de la instalación.

31. Clase GestorEstado

31.1 Descripción

Gestiona el estado global del sistema, permitiendo guardar y cargar la configuración y datos actuales de la simulación.

31.2 Funcionalidades

- Guardar el estado actual del sistema.
- Cargar un estado previamente guardado para restaurar la simulación.

31.3 Constructor

- `public GestorEstado()`
  Inicializa el gestor de estado, configurando los parámetros necesarios para las operaciones de guardado y carga.

31.4 Atributos

- Atributos que almacenan información relevante de la simulación (por ejemplo, configuraciones, datos de la partida, etc.).

31.5 Métodos

- **guardarEstado**
  `public void guardarEstado()`
  Guarda el estado actual del sistema en un archivo o base de datos.
- **load**
  `public void load()`
  Carga un estado previamente guardado, restaurando la configuración y datos del sistema.

32. Clase CrearRecompensa

32.1 Descripción

Gestiona la creación de recompensas dentro de la simulación, permitiendo definir distintos tipos de recompensas basadas en las acciones realizadas en el sistema.

32.2 Funcionalidades

- Crear recompensas para pienso, algas, comida, monedas, tanques, piscifactoria y almacén, entre otras.
- Convertir y obtener las cantidades asociadas a cada tipo de recompensa.

32.3 Constructor

- `public CrearRecompensa()`
  Inicializa la clase, configurando los parámetros necesarios para la creación de recompensas.

32.4 Atributos

- Atributos para almacenar la información y cantidades asociadas a las recompensas creadas.

32.5 Métodos

- **createPiensoReward**
- **createAlgasReward**
- **createComidaReward**
- **createMonedasReward**
- **createTanqueReward**
- **createPiscifactoriaReward**
- **createAlmacenReward**
- **romanize**
- Métodos para obtener las cantidades: **getFoodAmount**, **getSharedFoodAmount**, **getCoinsAmount** y **getTypeAmount**.
  (Cada uno de estos métodos crea o retorna la recompensa correspondiente y sus valores asociados.)

33. Clase UsarRecompensa

33.1 Descripción

Se encarga de aplicar las recompensas obtenidas en la simulación, afectando el estado del juego (por ejemplo, actualizando comida, monedas, tanques o la piscifactoria).

33.2 Funcionalidades

- Leer y aplicar recompensas de distintos tipos.
- Actualizar el estado del sistema en función de la recompensa utilizada.

33.3 Constructor

- `public UsarRecompensa()`
  Inicializa la clase encargada de usar recompensas.

33.4 Atributos

- Atributos para almacenar la información necesaria para procesar y aplicar las recompensas.

33.5 Métodos

- **readFood**
  `public void readFood()`
  Lee y aplica la recompensa de comida.
- **readCoins**
  `public void readCoins()`
  Lee y aplica la recompensa de monedas.
- **readTank**
  `public void readTank()`
  Lee y aplica la recompensa relacionada con tanques.
- **readPiscifactoria**
  `public void readPiscifactoria()`
  Lee y aplica la recompensa para la piscifactoria.
- **readAlmacenCentral**
  `public void readAlmacenCentral()`
  Lee y aplica la recompensa destinada al almacén central.

34. Clase Tanque

34.1 Descripción

Representa un tanque en la piscifactoría, encargado de albergar peces y gestionar su entorno, capacidad y estado.

34.2 Funcionalidades

- Mostrar el estado general del tanque y el de los peces que contiene.
- Controlar la capacidad y nivel de ocupación.
- Permitir la alimentación, crecimiento, reproducción y venta de peces.

34.3 Constructor

- `public Tanque(...)`
  Inicializa un tanque con parámetros específicos (como capacidad, ubicación, etc.).

34.4 Atributos

- Atributos que definen la capacidad, contenido, estado y otras características relevantes del tanque.

34.5 Métodos

- **showStatus**
  `public void showStatus()`
  Muestra el estado actual del tanque.
- **showFishStatus**
  `public void showFishStatus()`
  Detalla el estado de cada pez en el tanque.
- **showCapacity**
  `public void showCapacity()`
  Muestra la capacidad y ocupación del tanque.
- **nextDay**
  `public void nextDay()`
  Actualiza el estado del tanque y sus peces para el siguiente día.
- **reproduccion**
  `public void reproduccion()`
  Gestiona la reproducción de los peces.
- **addFish**
  `public void addFish()`
  Agrega un nuevo pez al tanque.
- **sellFish**
  `public void sellFish()`
  Gestiona la venta de peces, removiéndolos del tanque.
- **propagarEmfermedad**

private void propagarEnfermedad()

Gestiona la propagacion de la enfermedad de los peces.

35. Clase Logger

35.1 Descripción

Proporciona servicios de registro (logging) de eventos, errores y transacciones durante la simulación, permitiendo el seguimiento detallado de las operaciones del sistema.

35.2 Funcionalidades

- Registrar mensajes de error y eventos importantes.
- Mantener un historial de operaciones y transacciones.

35.3 Constructor

- `private Logger()`
  Implementa el patrón Singleton para asegurar que solo exista una única instancia de Logger.

35.4 Atributos

- Atributos para gestionar la salida de log (por ejemplo, archivo, buffer) y la instancia única del Logger.

35.5 Métodos

- **getInstance**
  `public static Logger getInstance()`
  Retorna la instancia única del Logger.
- **logError**
  `public void logError(String mensaje)`
  Registra un mensaje de error.
- **log**
  `public void log(String mensaje)`
  Registra un mensaje general.
- **close**
  `public void close()`
  Cierra el Logger y libera los recursos asociados.
- Métodos adicionales para registrar acciones específicas (por ejemplo, `logInicioPartida`, `logComprarPeces`, `logFinDelDia`, etc.).

36. Clase Transcriptor

36.1 Descripción

Se encarga de transcribir eventos y acciones en la simulación, generando registros textuales legibles para el usuario.

36.2 Funcionalidades

- Transcribir el inicio de la partida, compras, ventas, limpieza, mejoras y otros eventos.
- Proporcionar un registro detallado y legible de cada operación realizada.

36.3 Constructor

- `private Transcriptor()`
  Implementa el patrón Singleton para asegurar una única instancia de Transcriptor.

36.4 Atributos

- Atributos para gestionar la instancia única y el almacenamiento temporal de los mensajes transcritos.

36.5 Métodos

- **getInstance**
  `public static Transcriptor getInstance()`
  Retorna la instancia única de Transcriptor.
- **transcribir**
  `public void transcribir(String evento)`
  Transcribe un evento genérico.
- Métodos específicos para transcribir acciones, como:
  - `transcribirInicioPartida`
  - `transcribirComprarComidaPiscifactoria`
  - `transcribirComprarPeces`
  - `transcribirVenderPeces`
  - `transcribirLimpiarTanque`
  - `transcribirFinDelDia`
  - `transcribirOpcionOcultaPeces`
  - `transcribirCrearRecompensa`
  - `transcribirUsarRecompensa`
  - `transcribirGenerarPedidos`
  - `transcribirPedidoEnviado`

- `transcribirEnviadosConReferencia`, etc.

37. Clase Registros

37.1 Descripción

Gestiona el registro histórico de eventos y acciones realizadas durante la simulación, facilitando el análisis y la depuración del sistema.

37.2 Funcionalidades

- Registrar eventos de inicio, compras, ventas, limpieza, mejoras y cierre de la partida.
- Mantener un historial detallado de todas las operaciones.

37.3 Constructor

- `public Registros()`
  Inicializa la clase encargada de gestionar los registros, configurando los mecanismos necesarios.

37.4 Atributos

- Atributos para almacenar el historial (por ejemplo, listas o buffers de registro).

37.5 Métodos

- **registroInicioPartida**
- **registroComprarComidaPiscifactoria**
- **registroComprarComidaAlmacenCentral**
- **registroComprarPeces**
- **registroVenderPeces**
- **registroLimpiarTanque**
- **registroVaciarTanque**
- **registroComprarPiscifactoria**
- **registroComprarTanque**
- **registroComprarAlmacenCentral**
- **registroMejorarPiscifactoria**
- **registroMejorarAlmacenCentral**
- **registroFinDelDia**
- **registroOpcionOcultaPeces**
- **registroOpcionOcultaMonedas**

- **registroCrearRecompensa**
- **registroUsarRecompensa**
- **registroGenerarPedidos**
- **registroPedidoEnviado**
- **registroEnviadosConReferencia**
- **registroLogError**
- **registroCierrePartida**
- **registroGuardarSistema**
- **registroCargarSistema**
- **closeLogError**
  (Cada uno de estos métodos registra la acción correspondiente en el historial.)

38. Clase Simulador

38.1 Descripción

Es la clase principal que orquesta la simulación de la piscifactoría, coordinando la interacción entre los distintos componentes y gestionando el ciclo de vida de la simulación.

38.2 Funcionalidades

- Inicializar y gestionar el ciclo de vida de la simulación.
- Presentar menús y opciones de interacción al usuario.
- Actualizar el estado de la piscifactoría, tanques y peces durante cada ciclo o día.
- Coordinar las transacciones y operaciones registradas en el sistema.

38.3 Constructor

- `public Simulador()`
  Inicializa la simulación configurando todos los componentes necesarios (piscifactoría, tanques, Logger, Registros, GestorEstado, etc.).

38.4 Atributos

- Atributos que representan el estado global de la simulación, incluyendo instancias de piscifactoría, tanques, listas de peces, etc.

38.5 Métodos

- **init**
  `public void init()`
  Inicializa el sistema y prepara la simulación para comenzar.
- **menu**
  `public void menu()`
  Presenta el menú principal de opciones para la interacción del usuario.
- **menuPisc**
  `public void menuPisc()`
  Presenta un menú específico para operaciones relacionadas con la piscifactoría.
- **selectPisc**
  `public void selectPisc()`
  Permite seleccionar una piscifactoría, en caso de haber varias.
- **selectTank**
  `public void selectTank()`
  Permite seleccionar un tanque específico para operar.
- **showGeneralStatus**
  `public void showGeneralStatus()`
  Muestra el estado general de la simulación y la piscifactoría.
- **showSpecificStatus**
  `public void showSpecificStatus()`
  Muestra detalles específicos de ciertos componentes.
- **showTankStatus**
  `public void showTankStatus()`
  Detalla el estado de cada tanque.
- **showStats**
  `public void showStats()`
  Presenta estadísticas y datos relevantes de la simulación.
- **showIctio**
  `public void showIctio()`
  Muestra información relacionada con la ictiofauna (peces).
- **nextDay**
  `public void nextDay()`
  Avanza la simulación al siguiente día, actualizando todos los estados.
- **nextDay (int dias)**
  `public void nextDay(int dias)`
  Avanza la simulación varios días según el número especificado.

- **addFood**
  `public void addFood()`
  Permite agregar comida a la piscifactoría.
- **addFish**
  `public void addFish()`
  Permite agregar nuevos peces.
- **sell**
  `public void sell()`
  Gestiona la venta de peces.
- **cleanTank**
  `public void cleanTank()`
  Ejecuta la limpieza de un tanque.
- **emptyTank**
  `public void emptyTank()`
  Vacía un tanque, removiendo peces y residuos.
- **upgrade**
  `public void upgrade()`
  Permite mejorar las instalaciones o recursos de la piscifactoría.
- **gestionarCompraEdificios**
  `public void gestionarCompraEdificios()`
  Gestiona la compra de nuevos edificios o instalaciones.
- **gestionarMejoraEdificios**
  `public void gestionarMejoraEdificios()`
  Gestiona la mejora de los edificios existentes.
- **upgradePiscifactoria**
  `public void upgradePiscifactoria()`
  Permite mejorar la piscifactoría (por ejemplo, aumentando capacidad o eficiencia).
- **contarPiscifactoriasDeRio**
  `public int contarPiscifactoriasDeRio()`
  Retorna el número de piscifactorías de río.
- **contarPiscifactoriasDeMar**
  `public int contarPiscifactoriasDeMar()`
  Retorna el número de piscifactorías de mar.
- **addPiscifactoria**
  `public void addPiscifactoria()`
  Permite agregar una nueva piscifactoría al sistema.
- **recompensas**
  `public void recompensas()`
  Gestiona y muestra las recompensas disponibles.

- **generarRecompensas**
  `public void generarRecompensas()`
  Genera nuevas recompensas para el usuario.
- **pecesRandom**
  `public void pecesRandom()`
  Selecciona peces aleatoriamente para ciertas operaciones.
- **enviarPedidoManual**
  `public void enviarPedidoManual()`
  Permite enviar un pedido de peces de forma manual.
- **borrarPedidos**
  `public void borrarPedidos()`
  Elimina los pedidos existentes en el sistema.
- **GestionarEnfermedad**
  public void gestionarEnfermedad()

Permite curar los peces enfermos de una piscifactoria seleccionada.

- **cerrarConexion**
  `public void cerrarConexion()`
  Cierra la conexión a la base de datos y libera los recursos asociados.
- **main**
  `public static void main(String[] args)`
  Método principal que arranca la simulación.