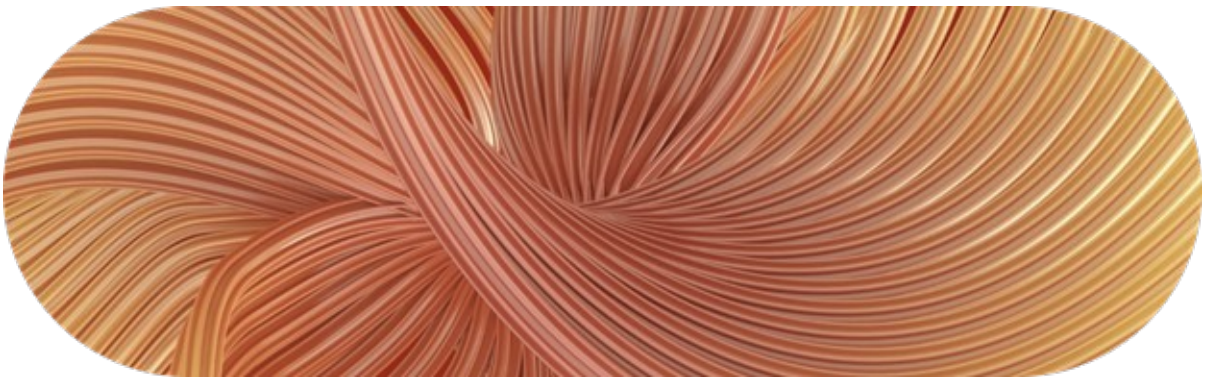
MANUAL TÉCNICO



Francisco Miser Junquera

Marcos Vidal González

David Torres Rial

ÍNDICE

1. Clase AlmacenCentral

1.1 Descripción

Se representa un almacén central con capacidad para almacenar comida animal y vegetal, permitiendo gestionar eficientemente los recursos necesarios para la piscifactoría. Se proporciona funcionalidad para aumentar la capacidad, añadir comida y distribuir los recursos a las piscifactorías registradas.

1.2 Funcionalidades

1. Se permite almacenar comida animal y vegetal en cantidades limitadas por la capacidad máxima del almacén.
2. Se ofrece la posibilidad de ampliar la capacidad máxima del almacén mediante el gasto de monedas.
3. Se distribuye comida animal y vegetal equitativamente entre las piscifactorías disponibles, garantizando el abastecimiento según sus necesidades actuales.
4. Se imprime un informe detallado de la capacidad, cantidades de comida almacenadas y porcentajes de ocupación.

1.3Constructor

- **public AlmacenCentral():**

  Se define el constructor de la clase AlmacenCentral. Se inicializan los siguientes valores predeterminados:
  Inicialización de atributos:
  capacidadMaxima: Se establece en 200, definiendo la capacidad máxima inicial del almacén.
  cantidadComidaAnimal: Se inicializa en 0, indicando que no hay comida animal almacenada al principio.
  cantidadComidaVegetal: Se inicializa en 0, indicando que no hay comida vegetal almacenada inicialmente.

1.4 Atributos

- **private int capacidadAlmacen:** Se indica la capacidad máxima que el almacén puede contener.
- **private int cantidadComidaAnimal**: Se registra la cantidad actual de comida animal almacenada.
- **private int cantidadComidaVegetal**: Se registra la cantidad actual de comida vegetal almacenada.
- **private final int costoMejora**: Se define el costo fijo en monedas necesario para mejorar la capacidad del almacén. Su valor es de 200.

1.5 Métodos

aumentarCapacidad

- **public void aumentarCapacidad()**

Se incrementa la capacidad del almacén central en 50 unidades si hay monedas suficientes. Si se realiza con éxito, se imprime un mensaje informando el nuevo nivel de capacidad.

añadirComidaAnimal

- **public void añadirComidaAnimal(int cantidad)**

Se añade comida animal al almacén en una cantidad especificada. Si la cantidad es válida y no supera la capacidad total, se actualiza el almacenamiento. En caso contrario, se muestra un mensaje indicando que la cantidad excede la capacidad.

añadirComidaVegetal

- **public void añadirComidaVegetal(int cantidad)**

Se añade comida vegetal al almacén en una cantidad especificada. Si la cantidad es válida y no supera la capacidad total, se actualiza el almacenamiento. En caso contrario, se informa que la capacidad ha sido superada.

distribuirComida

- **public void distribuirComida(List<Piscifactoria> piscifactorias)**

Piscifactoría **AD**

Se distribuye equitativamente la comida disponible entre las piscifactorías registradas. Se revisa cada piscifactoría para determinar sus necesidades de comida y se les asignan recursos según la cantidad disponible y el espacio libre en sus tanques.

2. Clase SistemaMonedas

2.1 Descripción

Se define la clase SistemaMonedas como un sistema de gestión de monedas virtuales dentro de la simulación. Se implementa como un patrón Singleton, asegurando que solo exista una instancia global para administrar la economía del juego. Se permite consultar, aumentar y gastar monedas según sea necesario, aplicando descuentos en ciertas transacciones.

2.2 Funcionalidades

1. Se permite consultar, aumentar y reducir la cantidad de monedas disponibles.
2. Se asegura que las transacciones solo se realicen cuando el saldo sea suficiente y las cantidades sean válidas.
3. Se aplica un descuento automático al calcular costos según la cantidad de recursos gestionados.

2.3 Constructor

- **private SistemaMonedas():**

  Se inicializa el sistema de monedas con un saldo inicial de 100 monedas. El constructor es privado para implementar el patrón Singleton, asegurando que solo una instancia sea creada.
  Inicialización de atributos:
  monedas: Se establece en 100, definiendo el saldo inicial de monedas disponibles en el sistema.

2.4 Atributos

- **private int monedas:** Se registra la cantidad actual de monedas disponibles en el sistema.
- **private static SistemaMonedas instance**: Se almacena la única instancia permitida del sistema de monedas, según el patrón Singleton.

2.5 Métodos

ganarMonedas

- **public boolean ganarMonedas(int cantidad)**

Se incrementa el saldo de monedas en una cantidad específica. Si la cantidad es válida (mayor que 0), se actualiza el saldo y se retorna true. En caso contrario, se retorna false.

gastarMonedas

- **public boolean gastarMonedas(int costo)**

Se reduce el saldo de monedas según un costo especificado. Si el saldo es suficiente y el costo es válido, se actualiza el saldo y se retorna true. Si el costo es mayor que el saldo disponible, se retorna false.

calcularDescuento

- **public int calcularDescuento(int cantidadComida)**

Se calcula el costo total aplicando un descuento de 5 monedas por cada 25 unidades de comida comprada. Si la cantidad de comida es inferior a 25, se retorna el costo completo sin descuento.

3. Clase DTOPedido

3.1 Descripción

Se define la clase DTOPedido como un objeto de transferencia de datos para un pedido. Esta clase encapsula la información esencial de un pedido, permitiendo el intercambio de datos entre las distintas capas del sistema sin exponer la lógica de negocio.

3.2 Funcionalidades

1. Representa la información de un pedido, incluyendo identificador, número de referencia, nombre del cliente, nombre del pez, cantidad total pedida y cantidad enviada.
2. 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)**

  Se inicializa un objeto DTOPedido con los valores proporcionados para cada atributo.
  Inicialización de atributos:
  id: Identificador único del pedido.
  numero_referencia: Número de referencia del pedido.
  nombre_cliente: Nombre del cliente asociado al pedido.
  nombre_pez: Nombre del pez asociado al pedido.
  cantidad: Cantidad total pedida.
  cantidad_enviada: Cantidad enviada hasta el momento.

3.4 Atributos

- **private int id:** Identificador único del pedido.
- **private String numero_referencia:** Número de referencia del pedido.
- **private int nombreCliente:** Nombre del cliente asociado al pedido.
- **private int nombrePez**: Nombre del pez asociado al pedido.
- **private int cantidad**: Cantidad total pedida.
- **private int cantidad_enviada**: Cantidad enviada hasta el momento.

3.5 Métodos

No posee ningún método solamente posee los getters de los atributos.

4. Clase DAOPedidos

4.1 Descripción

Se define la clase DAOPedidos como el Data Access Object (DAO) para la tabla *Pedido* de la base de datos. Esta clase se encarga de gestionar la comunicación con la base de datos utilizando Data Transfer Objects (DTOs) para encapsular y transportar la información de pedidos, clientes y peces. Permite 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

1. Permite la inserción, consulta, actualización y eliminación de pedidos, así como la generación automática de pedidos a partir de clientes y peces seleccionados aleatoriamente.

4.3 Constructor

- **public DAOPedidos()**

  Se inicializa la conexión a la base de datos y se preparan todos los *PreparedStatement* necesarios para ejecutar las consultas SQL requeridas. Durante este proceso se capturan y registran posibles excepciones de SQL.

4.4 Atributos

- **private Random random:** Generador de números aleatorios utilizado para seleccionar clientes y peces de forma aleatoria.
- **private static final String QUERY_INSERT_PEDIDO:** Consulta SQL para insertar un nuevo pedido.
- **private static final String QUERY_LISTAR_PEDIDOS_COMPLETADOS :** Consulta SQL para listar los pedidos completados.
- **private static final String QUERY_LISTAR_PEDIDOS_PENDIENTES** : Consulta SQL para listar los pedidos pendientes.
- **private static final String QUERY_SELECCIONAR_PEDIDO_POR_REFERENCIA** : Consulta SQL para seleccionar un pedido por su número de referencia.
- **private static final String QUERY_ACTUALIZAR_PEDIDO** : Consulta SQL para actualizar la cantidad enviada de un pedido.
- **private static final String QUERY_BORRAR_PEDIDOS** : Consulta SQL para eliminar todos los pedidos.
- **private static final String QUERY_RANDOM_CLIENTE** : Consulta SQL para obtener un cliente aleatorio.
- **private static final String QUERY_RANDOM_PEZ** : Consulta SQL para obtener un pez aleatorio.
- **private static final String QUERY_OBTENER_PEZ** : Consulta SQL para obtener los datos de un pez por su ID.

- **private static final String QUERY_OBTENER_NOMBRE** : Consulta SQL para obtener los datos de un cliente por su ID.
- **private Connection connection** : Conexión a la base de datos.
- ***PreparedStatement*** para cada operación:
  - **`pstInsertPedido`** para insertar un pedido.
  - **`pstListarPedidosPendientes`** para listar pedidos pendientes.
  - **`pstListarPedidosCompletados`** para listar pedidos completados.
  - **`pstSeleccionarPedidoPorReferencia`** para seleccionar un pedido por referencia.
  - **`pstActualizarPedido`** para actualizar la cantidad enviada de un pedido.
  - **`pstBorrarPedidos`** para borrar todos los pedidos.
  - **`ptsObtenerPez`** para obtener los datos de un pez por su ID.
  - **`ptsObtenerNombre`** para obtener los datos de un cliente por su ID.
  - **`pstRandomCliente`** para obtener un cliente aleatorio.
  - **`pstRandomPez`** para obtener un pez aleatorio.

4.5 Métodos

generarPedidoAutomatico

public DTOPedido generarPedidoAutomatico():

Se genera un pedido automático seleccionando aleatoriamente un cliente y un pez, e inserta el nuevo pedido en la base de datos.

listarPedidosPendientes
- **public List<DTOPedido> listarPedidosPendientes():**

Se lista los pedidos en los que la cantidad enviada es menor que la solicitada.

listarPedidosCompletados
- **public List<DTOPedido> listarPedidosCompletados():**

Se lista los pedidos en los que la cantidad enviada es igual o mayor que la solicitada.

obtenerPedidoPorReferencia

- **public DTOPedido obtenerPedidoPorReferencia(String numeroReferencia):**

Se obtiene un pedido a partir de su número de referencia.

actualizarPedido

- **public boolean actualizarPedido(DTOPedido pedido):**

Se actualiza la cantidad enviada de un pedido en la base de datos.

enviarPedido

- **public DTOPedido enviarPedido(DTOPedido pedido, int cantidadDisponible):**

Se envía una cantidad de peces para un pedido, actualizando la cantidad enviada y devolviendo el objeto actualizado.

borrarPedidos

- **public int borrarPedidos():**

Se elimina todos los pedidos de la base de datos y devuelve el número de registros eliminados.

obtenerClientePorId

- **public DTOCliente obtenerClientePorId(int idCliente):**

Se obtiene los datos de un cliente a partir de su ID.

obtenerPezPorId

- **public DTOPez obtenerPezPorId(int idPez):**

Se obtiene los datos de un pez a partir de su ID.

close

- **public void close():**

Se cierra la conexión a la base de datos y todos los PreparedStatement abiertos.

5. Clase Conexion

5.1 Descripción

Se define la clase Conexion para administrar la conexión a una base de datos MySQL. Esta clase se encarga de establecer y cerrar la conexión utilizando el DriverManager, permitiendo que la aplicación interactúe de forma centralizada con la base de datos. La conexión se maneja de manera estática, asegurando que exista una única instancia activa durante la ejecución de la aplicación.

5.2 Funcionalidades

1. Establece una conexión a la base de datos MySQL utilizando las credenciales y parámetros de conexión especificados.
2. Proporciona un método para obtener la conexión activa, creando la conexión si esta no existe.
3. Permite cerrar la conexión a la base de datos, liberando los recursos asociados.

5.3 Constructor

- No se requiere un constructor público ya que la clase utiliza métodos estáticos para gestionar la conexión a la base de datos.

5.4 Atributos

- **private static final String USER:** Usuario de la base de datos.
- **private static final String PASSWORD**: Contraseña del usuario 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 de la base de datos.
- **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():**

Se devuelve un objeto Connection. Si no existe una conexión activa, la crea utilizando los parámetros de conexión y el DriverManager. En caso de error, registra el fallo correspondiente.

closeConnection

- **public static void closeConnection():**

Se cierra la conexión a la base de datos si está activa, liberando los recursos asociados y estableciendo la conexión a null.

6. Clase GeneradorBD

6.1 Descripción

Se define la clase GeneradorBD como la encargada de crear la estructura de la base de datos para un sistema de pedidos de peces. Esta clase se responsabiliza de generar las tablas necesarias (Cliente, Pez y Pedido) si no existen, así como de insertar los datos iniciales en las tablas Cliente y Pez de forma condicional (verificando la existencia previa de los registros). De esta manera, se asegura que el sistema disponga de la estructura y los datos básicos requeridos para operar correctamente.

6.2 Funcionalidades

1. Se crean las tablas Cliente, Pez y Pedido en la base de datos, definiendo sus columnas, claves primarias y restricciones (incluyendo claves foráneas y acciones en cascada).
2. Se inserta registros iniciales en la tabla Cliente mediante la verificación de la existencia previa (evitando duplicados basados en el NIF).
3. Se inserta registros iniciales en la tabla Pez utilizando datos del sistema (nombre común y nombre científico), evitando duplicados según el nombre.
4. Se facilita la inicialización del sistema de pedidos de peces al asegurar que la base de datos cuente con la estructura y datos mínimos necesarios para su funcionamiento.

6.3 Constructor

- **No se define un constructor.**

6.4 Atributos

- **private static final String QUERY_AGREGAR_CLIENTES:** Consulta SQL que inserta un nuevo cliente en la tabla Cliente (con nombre, NIF y teléfono), comprobando previamente que no exista ya un registro con el mismo NIF.
- **private static final String QUERY_AGREGAR_PEZ**: Consulta SQL que inserta un nuevo pez en la tabla Pez (con nombre y nombre científico), verificando que no exista ya un pez con el mismo nombre.
- **private Connection connection:** Objeto de conexión a la base de datos obtenido mediante el método Conexion.getConnection(), que permite la ejecución de las instrucciones SQL necesarias para crear tablas e insertar datos.

6.5 Métodos

crearTablaCliente

- **public void crearTablaCliente():**

Se crea la tabla *Cliente* en la base de datos si no existe. Define la estructura de la tabla con las columnas:

- **id**: Clave primaria autoincrementable.
- **nombre**: Campo de texto que almacena el nombre del cliente.
- **nif**: Campo único que identifica al cliente.
- **telefono**: Campo que almacena el número de teléfono del cliente.

crearTablaPez

- **public void crearTablaPez():**

Se crea la tabla *Pez* en la base de datos si no existe. Define la estructura de la tabla con las columnas:

- **id**: Clave primaria autoincrementable.
- **nombre**: Campo de texto que almacena el nombre común del pez.
- **nombre_cientifico**: Campo de texto que almacena el nombre científico del pez.

crearTablaPedido

- **public void crearTablaPedido():**

Se crea la tabla *Pedido* en la base de datos si no existe. Define la estructura de la tabla con las columnas:

- **numero_referencia**: Clave primaria única que identifica cada pedido.
- **id_cliente**: Identificador del cliente asociado, con restricción de clave foránea que referencia a *Cliente(id)* y con eliminación en cascada.
- **id_pez**: Identificador del pez asociado, con restricción de clave foránea que referencia a *Pez(id)* y con eliminación en cascada.
- **cantidad**: Número total de peces solicitados en el pedido.
- **cantidad_enviada**: Número de peces enviados, inicializado en 0 por defecto.

agregarClientes

- **public void agregarClientes():**

Se inserta registros iniciales en la tabla *Cliente* utilizando un *PreparedStatement*.

Se utiliza arreglos de nombres, NIFs y teléfonos para agregar múltiples clientes, comprobando que no se inserten duplicados mediante la verificación del NIF.

agregarPeces

- **public void agregarPeces():**

Se inserta registros iniciales en la tabla *Pez* utilizando un *PreparedStatement*.
Se recorre la lista de peces implementados, asigna el nombre común y obtiene el nombre científico a través de las propiedades, e inserta estos datos en la tabla evitando duplicados basados en el nombre.

crearTablas

- **public void crearTablas():**

Se ejecuta en secuencia la creación de las tablas *Cliente*, *Pez* y *Pedido*, y posteriormente invoca los métodos para agregar los datos iniciales en las tablas *Cliente* y *Pez*.
Este método integra todas las operaciones necesarias para inicializar la estructura y los datos básicos del sistema de pedidos de peces.

7. Clase FileHelper

7.1 Descripción

Se define la clase *FileHelper* como una utilidad estática para gestionar operaciones relacionadas con archivos y directorios. Permite crear carpetas, verificar el contenido de directorios, listar archivos para la selección mediante un menú interactivo y extraer recompensas a partir de archivos XML.

7.2 Funcionalidades

1. Crear múltiples carpetas de forma segura.
2. Verificar si un directorio contiene archivos o subdirectorios.
3. Listar archivos en un directorio y mostrar un menú interactivo para que el usuario seleccione uno.
4. Obtener un arreglo con los nombres de los archivos en un directorio.
5. Extraer recompensas desde archivos XML en el directorio "rewards", procesando elementos y partes específicas.
6. Eliminar el contenido de corchetes en cadenas, devolviendo únicamente la parte base del nombre.

7.3 Constructor

- **No se define un constructor, ya que todos los métodos de la clase son estáticos.**

7.4 Atributos

- No se definen atributos de instancia.

7.5 Métodos

crearCarpetas

- **public static void crearCarpetas(String[] carpetas)**

Se crea múltiples carpetas especificadas en el arreglo de cadenas. Para cada carpeta, se verifica si ya existe; de no ser así, se intenta crear utilizando *mkdirs()*. Si ocurre algún error (por ejemplo, por permisos insuficientes o errores inesperados), se registra la incidencia a través del sistema de log.

hayContenidoEnDirectorio

- **public static boolean hayContenidoEnDirectorio(String rutaDirectorio)**

Se verifica si el directorio indicado existe y contiene archivos o subdirectorios. Si el directorio existe y tiene contenido, retorna *true*; si está vacío o no existe, se registra el error y retorna *false*.

mostrarMenuConArchivos

- **public static String mostrarMenuConArchivos(String rutaDirectorio)**

Se muestra un menú interactivo con los nombres de los archivos presentes en el directorio especificado. Los nombres se muestran sin sus extensiones. Se utiliza *InputHelper* para que el usuario seleccione una opción. Si el directorio no existe, no contiene archivos o ocurre un error, se registra la incidencia y se retorna *null*.

obtenerArchivosEnDirectorio

- **public static String[] obtenerArchivosEnDirectorio(String rutaDirectorio)**

Se devuelve un arreglo de cadenas con los nombres de los archivos que se encuentran en el directorio especificado. Los archivos se ordenan alfabéticamente. Si el directorio no existe o está vacío, se registra el error y se retorna un arreglo vacío.

getRewards

- **public static String[] getRewards()**

Se extrae las recompensas desde los archivos XML ubicados en el directorio "rewards". Para cada archivo XML, se procesa su contenido utilizando *SAXReader* para leer el documento y extraer el nombre de la recompensa, así como identificar y procesar partes específicas de edificios (por ejemplo, "Almacen central", "Piscifactoria de mar", "Piscifactoria de rio", "Tanque de mar" y "Tanque de rio"). Se aplican condiciones para determinar qué recompensas o partes se añaden, y se retorna un arreglo con los nombres resultantes.

getRewardsWithoutBrackets

- **public static String[] getRewardsWithoutBrackets(String[] opciones)**

Se recibe un arreglo de cadenas que contienen recompensas y elimina todo el contenido que aparece después del primer corchete "[" en cada opción. Retorna un nuevo arreglo con las recompensas sin los corchetes. Si una opción no contiene corchetes, se incluye tal cual en el resultado.

8. Clase InputHelper

8.1 Descripción

Se define la clase InputHelper para gestionar la entrada de datos desde la consola de manera segura y validada. Se permite leer cadenas, números enteros y seleccionar números dentro de un rango específico, asegurando que las entradas cumplan con los formatos esperados.

8.2 Funcionalidades

1. Se permite ingresar cadenas alfanuméricas, verificando que no estén vacías y que no contengan caracteres especiales.
2. Se permite ingresar números enteros válidos, mostrando mensajes de error si se ingresan valores no numéricos.
3. Se solicita un número dentro de un rango definido por valores mínimo y máximo, con validación automática.
4. Se cierra el Scanner utilizado para liberar los recursos del sistema.

8.3 Constructor

- No se define un constructor.

8.4 Atributos

- **private static final Scanner scanner**: Se utiliza para gestionar las entradas desde la consola de manera eficiente y reutilizable en toda la clase.

8.5 Métodos

readString

- **public static String readString(String prompt)**

Se lee una cadena desde la consola que no esté vacía y no contenga caracteres especiales. Si la entrada no es válida, se muestra un mensaje de error y se solicita una nueva entrada hasta que sea válida, devuelve la cadena inglesada.

readInt

- **public static int readInt(String prompt)**

Se solicita al usuario un número entero desde la consola. Si la entrada no es válida (vacía o no numérica), se muestra un mensaje de error y se solicita un nuevo número, devuelve el número entero ingresado.

solicitarNumero

- **public static int solicitarNumero(int min, int max)**

Se solicita al usuario un número entero dentro de un rango específico definido por valores mínimo y máximo. Si el número ingresado no está dentro del rango, se muestra un mensaje y se vuelve a solicitar.
Se devuelve el número entero válido dentro del rango especificado.

close

- **public static void close()**

Se cierra el Scanner utilizado en la clase para liberar recursos del sistema. Si el Scanner ya ha sido cerrado, se registra un error utilizando el sistema de registro de errores del Simulador. No devuelve nada.

9. Clase MenuHelper

9.1 Descripción

Se define la clase MenuHelper para facilitar la creación y visualización de menús en la consola. Se permite mostrar menús simples con opciones numeradas, incluyendo una opción especial para cancelar la operación.

9.2 Funcionalidades

1. Se muestra una lista de opciones numeradas en la consola para que el usuario seleccione una opción.
2. Se incluye una opción adicional para permitir cancelar operaciones 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)**

Se muestra un menú numerado en la consola según un array de opciones proporcionado. Se agrega automáticamente una opción 0. Cancelar, permitiendo al usuario cancelar la operación. No devuelve ningún valor.

mostrarMenu

- **public static void mostrarMenu(String[] opciones)**

Se muestra un menú numerado en la consola según un array de opciones proporcionado. No se incluye una opción de cancelación. No devuelve ningún valor.

10. Clase abstracta CarnivoroActivo

10.1 Descripción

Se define la clase abstracta CarnivoroActivo, que extiende de la clase Pez. Se utiliza para representar peces carnívoros con un comportamiento activo, lo que implica que pueden consumir más comida de lo habitual. Se utiliza un generador aleatorio para simular el comportamiento activo, haciendo que el pez pueda requerir una o dos unidades de comida animal según su actividad.

10.2 Funcionalidades

1. Se permite alimentar a los peces según su comportamiento activo, que es determinado aleatoriamente.
2. Se incrementa el consumo de comida animal según el nivel de actividad del pez. Si el pez está activo, consume dos unidades; de lo contrario, consume una.
3. Se actualiza el estado del pez dependiendo de si logró consumir comida o no, asegurando la correcta gestión de recursos.

10.3 Constructor

- **public CarnivoroActivo(boolean sexo, PecesDatos datos)**

  Se define el constructor de la clase CarnivoroActivo, el cual inicializa un pez carnívoro con comportamiento activo, heredando las propiedades de su clase base mediante la invocación del constructor de la superclase usando super(). Este constructor permite establecer los atributos principales del pez desde el momento de su creación.

  Parámetros:

  sexo: Se define el sexo del pez, donde true representa un sexo y false representa el otro según la implementación interna.

  datos: Se recibe un objeto de tipo PecesDatos, que proporciona información esencial sobre el pez, incluyendo características como tamaño, peso, hábitat y comportamientos específicos.

10.4 Atributos

- **Random rand**: Se define un generador de números aleatorios para simular el comportamiento activo del pez.

10.5 Métodos

alimentar

- **@Override public int alimentar(int cantidadComidaAnimal, int cantidadComidaVegetal)**

Se implementa el método alimentar, verificando el nivel de actividad del pez usando un generador aleatorio. Si el pez está activo (probabilidad del 50%), intenta consumir dos unidades de comida animal. Si la comida local no es suficiente, intenta consumir del almacén central. Si no está activo, consume una unidad en condiciones similares. Si no hay comida disponible, permanece sin alimentar. Se devuelve:

2: Si el pez está activo y consume dos unidades de comida animal.

1: Si el pez no está activo y consume una unidad de comida animal. 0:
Si no se consume comida por falta de recursos.

11. Clase abstracta Carnivoro

11.1 Descripción

Se define la clase abstracta Carnivoro, que extiende de la clase Pez. Se utiliza para representar peces con una dieta basada en comida animal. Se implementa un método de alimentación que permite consumir comida animal local o, en su defecto, obtenerla del almacén central, asegurando que el pez se mantenga alimentado correctamente.

11.2 Funcionalidades

1. Se permite alimentar a los peces utilizando comida animal disponible localmente o desde el almacén central.
2. Se actualiza el estado de alimentación del pez dependiendo de si logró consumir comida o no.

11.3 Constructor

- **public Carnivoro(boolean sexo, PecesDatos datos)**

  Se define el constructor de la clase Carnivoro, el cual inicializa un pez con características específicas heredadas de su clase base. Se utiliza el método super() para llamar al constructor de la clase padre, permitiendo establecer los atributos iniciales del pez.

  Parámetros:

  sexo: Se define el sexo del pez, donde true representa un sexo y false representa el otro según la implementación específica.

  datos: Se recibe un objeto de tipo PecesDatos, el cual contiene información detallada relacionada con las características y propiedades del pez, como su tamaño, peso, hábitat y requerimientos alimenticios.

11.4 Atributos

- No se definen atributos propios en esta clase

11.5 Métodos

alimentar

- **@Override public int alimentar(int cantidadComidaAnimal, int cantidadComidaVegetal)**

Se implementa el método alimentar, que permite alimentar al pez con comida animal disponible localmente o desde el almacén central. Si hay comida disponible localmente, se consume una unidad y se actualiza el estado del pez a alimentado = true. Si no hay comida, se intenta obtener una unidad del almacén central. Si no se encuentra comida, el pez permanece sin alimentar.Devuelve 1 Si el pez consume comida local, y devuelve 0 si consume comida del almacen central o no hay comida disponible.

12. Clase abstracta Filtrador

12.1 Descripción

Se define la clase abstracta Filtrador, que extiende de la clase Pez. Se utiliza para representar peces que se alimentan exclusivamente de comida vegetal. Se implementa un método de alimentación que permite consumir comida vegetal disponible localmente o desde el almacén central en caso de escasez. El comportamiento de alimentación se controla mediante un generador aleatorio, simulando la búsqueda de alimento en su entorno.

12.2 Funcionalidades

1. Se permite alimentar a los peces utilizando comida vegetal disponible localmente o desde el almacén central.
2. Se utiliza un generador aleatorio para determinar cuándo el pez intenta alimentarse.
3. Se actualiza el estado del pez según su nivel de alimentación, indicando si logró consumir comida vegetal o permaneció sin alimentar.

12.3 Constructor

- **public Filtrador(boolean sexo, PecesDatos datos)**

  Se define el constructor de la clase Filtrador, el cual inicializa un pez con comportamiento filtrador, heredando sus propiedades de la clase base mediante la invocación del constructor de la superclase usando super(). Esto permite establecer atributos esenciales relacionados con el pez desde su creación.

  Parámetros:

  sexo: Se define el sexo del pez, donde true representa un sexo y false representa

  el otro según la lógica implementada.

  datos: Se recibe un objeto de tipo PecesDatos, que contiene información relevante sobre el pez, como su tamaño, peso, tipo de alimento y características específicas de su comportamiento filtrador.

12.4 Atributos

- **Random rand**: Se define un generador de números aleatorios para simular el comportamiento de alimentación del pez filtrador.

12.5 Métodos

alimentar

- **@Override public int alimentar(int cantidadComidaAnimal, int cantidadComidaVegetal)**

Se implementa el método alimentar, que permite alimentar al pez usando comida vegetal disponible localmente o desde el almacén central. Si hay comida vegetal local, se consume una unidad. Si no hay comida local pero existe comida en el almacén central, se consume una unidad desde allí. Si no se encuentra comida en ninguno de los dos lugares, el pez permanece sin alimentar. El proceso de alimentación ocurre con una probabilidad del 50%. Si no se encuentra comida, el pez permanece sin alimentar. Se devuelve:

1: Si el pez consume comida vegetal local.

0: Si consume comida vegetal del almacén central o no consume comida por falta de recursos.

13.Clase abstracta Omnivoro

13.1 Descripción

Se define la clase abstracta Omnivoro, que extiende de la clase Pez. Se utiliza para representar peces con una dieta mixta, que consumen tanto comida animal como vegetal. Se implementa un método de alimentación que permite consumir cualquiera de los dos tipos de comida, dependiendo de la disponibilidad en el entorno local o en el almacén central. Se utiliza un generador aleatorio para simular la probabilidad de que el pez se alimente.

13.2 Funcionalidades

1. Se permite alimentar al pez con comida animal o vegetal, según la disponibilidad en el entorno local o el almacén central.
2. Se da prioridad a la comida disponible localmente y, en caso de no haber suficiente, se consume comida del almacén central.
3. Se actualiza el estado de alimentación según los recursos consumidos, indicando si el pez logró alimentarse correctamente.

13.3 Constructor

- **public Omnivoro(boolean sexo, PecesDatos datos)**

  Se define el constructor de la clase Omnivoro, el cual inicializa un pez con una dieta omnívora, heredando sus propiedades principales desde la clase base mediante la invocación del constructor de la superclase usando super(). Esto permite establecer atributos esenciales relacionados con el pez en el momento de su creación.

  Parámetros:

  sexo: Se define el sexo del pez, donde true representa un sexo y false representa el otro según la implementación interna.

  datos: Se recibe un objeto de tipo PecesDatos, que contiene información fundamental sobre el pez, incluyendo características como su tamaño, peso, hábitat y sus requerimientos alimenticios específicos.

13.4 Atributos

- **Random rand** Se define un generador de números aleatorios para determinar la probabilidad de que el pez se alimente en cada ciclo.

13.5 Métodos

alimentar

- **@Override public int alimentar(int cantidadComidaAnimal, int cantidadComidaVegetal)**

Se implementa el método alimentar, que permite consumir comida animal o vegetal según la disponibilidad. Con una probabilidad del 75%, el pez intenta alimentarse. Si hay comida disponible localmente, se consume una unidad de cualquiera de los dos tipos. Si no hay comida local, se intenta consumir comida del almacén central, priorizando el tipo de comida con mayor cantidad disponible. Si ninguno de los dos lugares dispone de comida, el pez permanece sin alimentar. Se devuelve:

1: Si el pez consume comida local, ya sea animal o vegetal.
0: Si consume comida del almacén central o no consume comida por falta de recursos.

14. Clase Dorada

14.1 Descripción

Se define la clase Dorada como una subclase de Omnivoro, representando un pez que consume tanto comida animal como vegetal. Se establece como un pez omnívoro con propiedades específicas definidas en la clase AlmacenPropiedades. Se permite crear nuevas instancias y realizar clonaciones según el sexo especificado.

14.2 Funcionalidades

1. Se hereda el comportamiento de alimentación de la clase Omnivoro, permitiendo consumir ambos tipos de comida según la disponibilidad.
2. Se permite crear instancias del pez con un sexo específico y generar copias exactas mediante un método de clonación.

14.3 Constructor

- **public Dorada(boolean sexo)**

  Se define el constructor de la clase Dorada, el cual inicializa un pez de tipo dorada estableciendo sus propiedades específicas mediante la invocación del constructor de la superclase usando super(). Se utiliza una constante predefinida AlmacenPropiedades.DORADA para asociar automáticamente las características específicas del pez dorada desde un almacén de datos centralizado.

  Parámetros:

  sexo: Se define el sexo del pez, (true para macho, false para hembra)

14.4 Atributos

- No se define ningún atributo.

14.5 Métodos

clonar

- **@Override public Dorada clonar(boolean nuevoSexo)**

Se crea una nueva instancia del pez Dorada con el sexo especificado. Este método permite replicar la instancia actual con un sexo diferente.
Devuelve: Una nueva instancia de Dorada con el sexo especificado.

15. Clase SalmonAtlantico

15.1 Descripción

Se define la clase SalmonAtlantico como una subclase de Carnivoro, representando un pez que se alimenta exclusivamente de comida animal. Se inicializa con propiedades específicas definidas en la clase AlmacenPropiedades. Se permite crear instancias del pez y realizar clonaciones con diferentes sexos.

15.2 Funcionalidades

1. Se hereda el comportamiento de alimentación de la clase Carnivoro, permitiendo consumir solo comida animal disponible localmente o desde el almacén central.
2. Se permite crear nuevas instancias del pez con un sexo específico y generar copias exactas mediante un método de clonación.

15.3 Constructor

- **public SalmonAtlantico(boolean sexo)**

  Se define el constructor de la clase SalmonAtlantico, el cual inicializa un pez de tipo salmón atlántico estableciendo sus propiedades específicas mediante la invocación del constructor de la superclase usando super(). Se utiliza una constante predefinida AlmacenPropiedades.SALMON_ATLANTICO para asignar automáticamente las características propias de esta especie desde un almacén central de datos.

  Parámetros:

  sexo: Se define el sexo del pez, (true para macho, false para hembra)

15.4 Atributos

- No se define ningún atributo.

15.5 Métodos

clonar

- **@Override public SalmonAtlantico clonar(boolean nuevoSexo)**

Se crea una nueva instancia de SalmonAtlantico con el sexo especificado. Este método permite replicar la instancia actual con un sexo diferente.
Devuelve: Una nueva instancia de SalmonAtlantico con el sexo especificado.

16. Clase TruchaArcoiris

16.1 Descripción

Se define la clase TruchaArcoiris como una subclase de Carnivoro, representando un pez que se alimenta exclusivamente de comida animal. Se configura utilizando propiedades específicas definidas en la clase AlmacenPropiedades. Se permite crear instancias del pez y realizar clonaciones especificando un nuevo sexo.

16.2 Funcionalidades

1. Se hereda el comportamiento de alimentación de la clase Carnivoro, permitiendo consumir comida animal local o desde el almacén central.
2. Se permite crear nuevas instancias del pez con un sexo definido y generar copias exactas mediante un método de clonación.

16.3 Constructor

- **public TruchaArcoiris(boolean sexo)**

  Se define el constructor de la clase TruchaArcoiris, el cual inicializa un pez de tipo trucha arcoíris estableciendo sus propiedades específicas mediante la invocación del constructor de la superclase usando super(). Se utiliza una constante predefinida AlmacenPropiedades.TRUCHA_ARCOIRIS para asignar automáticamente las características propias de esta especie desde un almacén central de datos.

  Parámetros:

  sexo: Se define el sexo del pez, (true para macho, false para hembra)

16.4 Atributos

- No se define ningún atributo.

16.5 Métodos

clonar

- **@Override public TruchaArcoiris clonar(boolean nuevoSexo)**

Se crea una nueva instancia de TruchaArcoiris con el sexo especificado. Este método permite replicar la instancia actual del pez con un sexo diferente.
Devuelve: Una nueva instancia de TruchaArcoiris con el sexo especificado.

17. Clase ArenqueDelAtlantico

17.1 Descripción

Se define la clase ArenqueDelAtlantico como una subclase de Filtrador, representando un pez que se alimenta exclusivamente de comida vegetal. Se configura utilizando propiedades específicas establecidas en la clase AlmacenPropiedades. Se permite crear instancias del pez y realizar clonaciones especificando un nuevo sexo.

17.2 Funcionalidades

1. Se hereda el comportamiento de alimentación de la clase Filtrador, permitiendo consumir comida vegetal disponible localmente o desde el almacén central.
2. Se permite crear nuevas instancias del pez con un sexo específico y generar copias exactas mediante el método de clonación.

17.3 Constructor

- **public ArenqueDelAtlantico(boolean sexo)**

  Se define el constructor de la clase ArenqueDelAtlantico, el cual inicializa un pez de tipo arenque del Atlántico estableciendo sus propiedades específicas mediante la invocación del constructor de la superclase usando super(). Se utiliza una constante predefinida AlmacenPropiedades.ARENQUE_ATLANTICO para asignar automáticamente las características propias de esta especie desde un almacén central de datos.

  Parámetros:

  sexo: Se define el sexo del pez, (true para macho, false para hembra)

17.4 Atributos

- No se define ningún atributo.

17.5 Métodos

clonar

- **@Override public ArenqueDelAtlantico clonar(boolean nuevoSexo)**

Se crea una nueva instancia de ArenqueDelAtlantico con el sexo especificado. Este método permite replicar la instancia actual con un sexo diferente.

Devuelve: Una nueva instancia de ArenqueDelAtlantico con el sexo especificado.

18. Clase Besugo

18.1 Descripción

Se define la clase Besugo como una subclase de Carnivoro, representando un pez que se alimenta exclusivamente de comida animal. Se configura utilizando propiedades específicas establecidas en la clase AlmacenPropiedades. Se permite crear instancias del pez y realizar clonaciones especificando un nuevo sexo.

18.2 Funcionalidades

1. Se hereda el comportamiento de alimentación de la clase Carnivoro, permitiendo consumir comida animal disponible localmente o desde el almacén central.
2. Se permite crear nuevas instancias del pez con un sexo definido y generar copias exactas mediante el método de clonación.

18.3 Constructor

- **public Besugo(boolean sexo)**

  Se define el constructor de la clase Besugo, el cual inicializa un pez de tipo besugo estableciendo sus propiedades específicas mediante la invocación del constructor de la superclase usando super(). Se utiliza una constante predefinida AlmacenPropiedades.BESUGO para asignar automáticamente las características propias de esta especie desde un almacén central de datos.

  Parámetros:

  sexo: Se define el sexo del pez, (true para macho, false para hembra)

18.4 Atributos

- No se define ningún atributo.

18.5 Métodos

clonar

- **@Override public Besugo clonar(boolean nuevoSexo)**

Se crea una nueva instancia de Besugo con el sexo especificado. Este método permite replicar la instancia actual con un sexo diferente.

Devuelve: Una nueva instancia de Besugo con el sexo especificado.

19. Clase LenguadoEuropeo

19.1 Descripción

Se define la clase LenguadoEuropeo como una subclase de Carnivoro, representando un pez que se alimenta exclusivamente de comida animal. Se configura utilizando propiedades específicas establecidas en la clase AlmacenPropiedades. Se permite crear instancias del pez y realizar clonaciones especificando un nuevo sexo.

19.2 Funcionalidades

1. Se hereda el comportamiento de alimentación de la clase Carnivoro, permitiendo consumir comida animal disponible localmente o desde el almacén central.
2. Se permite crear nuevas instancias del pez con un sexo definido y generar copias exactas mediante el método de clonación.

19.3 Constructor

- **public LenguadoEuropeo(boolean sexo)**

  Se define el constructor de la clase LenguadoEuropeo, el cual inicializa un pez de tipo lenguado europeo estableciendo sus propiedades específicas mediante la invocación del constructor de la superclase usando super(). Se utiliza una constante predefinida AlmacenPropiedades.LENGUADO_EUROPEO para asignar automáticamente las características particulares de esta especie desde un almacén central de datos.

  Parámetros:

  sexo: Se define el sexo del pez, (true para macho, false para hembra)

19.4 Atributos

- No se define ningún atributo.

19.5 Métodos

clonar

- **@Override public LenguadoEuropeo clonar(boolean nuevoSexo)**

Se crea una nueva instancia de LenguadoEuropeo con el sexo especificado. Este método permite replicar la instancia actual con un sexo diferente.
Devuelve: Una nueva instancia de LenguadoEuropeo con el sexo especificado.

20. Clase Robalo

20.1 Descripción

Se define la clase Robalo como una subclase de Carnivoro, representando un pez que se alimenta exclusivamente de comida animal. Se configura utilizando propiedades específicas establecidas en la clase AlmacenPropiedades. Se permite crear instancias del pez y realizar clonaciones especificando un nuevo sexo.

20.2 Funcionalidades

1. Se hereda el comportamiento de alimentación de la clase Carnivoro, permitiendo consumir comida animal disponible localmente o desde el almacén central.
2. Se permite crear nuevas instancias del pez con un sexo definido y generar copias exactas mediante el método de clonación.

20.3 Constructor

- **public Robalo(boolean sexo)**

  Se define el constructor de la clase Robalo, el cual inicializa un pez de tipo robalo estableciendo sus propiedades específicas mediante la invocación del constructor de la superclase usando super(). Se utiliza una constante predefinida AlmacenPropiedades.ROBALO para asignar automáticamente las características particulares de esta especie desde un almacén central de datos.

  Parámetros:

  sexo: Se define el sexo del pez, (true para macho, false para hembra)

20.4 Atributos

- No se define ningún atributo.

20.5 Métodos

clonar

- **@Override public Robalo clonar(boolean nuevoSexo)**

Se crea una nueva instancia de Robalo con el sexo especificado. Este método permite replicar la instancia actual con un sexo diferente.

Devuelve: Una nueva instancia de Robalo con el sexo especificado.

21. Clase LubinaRayada

21.1 Descripción

Se define la clase LubinaRayada como una subclase de Carnivoro, representando un pez que se alimenta exclusivamente de comida animal. Se configura utilizando propiedades específicas establecidas en la clase AlmacenPropiedades. Se permite crear instancias del pez y realizar clonaciones especificando un nuevo sexo.

21.2 Funcionalidades

1. Se hereda el comportamiento de alimentación de la clase Carnivoro, permitiendo consumir comida animal disponible localmente o desde el almacén central.
2. Se permite crear nuevas instancias del pez con un sexo definido y generar copias exactas mediante el método de clonación.

21.3 Constructor

- **public LubinaRayada(boolean sexo)**

  Se define el constructor de la clase LubinaRayada, el cual inicializa un pez de tipo robalo estableciendo sus propiedades específicas mediante la invocación del constructor de la superclase usando super(). Se utiliza una constante predefinida AlmacenPropiedades.LUBINA_RAYADA para asignar automáticamente las características particulares de esta especie desde un almacén central de datos.

  Parámetros:

  sexo: Se define el sexo del pez, (true para macho, false para hembra)

21.4 Atributos

- No se define ningún atributo.

21.5 Métodos

clonar

- **@Override public LubinaRayada clonar(boolean nuevoSexo)**

Se crea una nueva instancia de LubinaRayada con el sexo especificado. Este método permite replicar la instancia actual con un sexo diferente.
Devuelve: Una nueva instancia de LubinaRayada con el sexo especificado.

22. Clase CarpaPlateada

22.1 Descripción

Se define la clase CarpaPlateada como una subclase de Filtrador, representando un pez que se alimenta exclusivamente de comida vegetal. Se configura utilizando propiedades específicas establecidas en la clase AlmacenPropiedades. Se permite crear instancias del pez y realizar clonaciones especificando un nuevo sexo.

22.2 Funcionalidades

1. Se hereda el comportamiento de alimentación de la clase Filtrador, permitiendo consumir comida vegetal disponible localmente o desde el almacén central.
2. Se permite crear nuevas instancias del pez con un sexo definido y generar copias exactas mediante el método de clonación.

22.3 Constructor

- **public CarpaPlateada(boolean sexo)**

  Se define el constructor de la clase CarpaPlateada, el cual inicializa un pez de tipo carpa plateada estableciendo sus propiedades específicas mediante la invocación del constructor de la superclase usando super(). Se utiliza una constante predefinida AlmacenPropiedades.CARPA_PLATEADA para asignar automáticamente las características particulares de esta especie desde un almacén central de datos.

  Parámetros:

  sexo: Se define el sexo del pez, (true para macho, false para hembra)

22.4 Atributos

- No se define ningún atributo.

22.5 Métodos

clonar

- **@Override public CarpaPlateada clonar(boolean nuevoSexo)**

Se crea una nueva instancia de CarpaPlateada con el sexo especificado. Este método permite replicar la instancia actual con un sexo diferente.
Devuelve: Una nueva instancia de CarpaPlateada con el sexo especificado.

23. Clase Pejerrey

23.1 Descripción

Se define la clase Pejerrey como una subclase de Carnivoro, representando un pez que se alimenta exclusivamente de comida animal. Se configura utilizando propiedades específicas establecidas en la clase AlmacenPropiedades. Se permite crear instancias del pez y realizar clonaciones especificando un nuevo sexo.

23.2 Funcionalidades

1. Se hereda el comportamiento de alimentación de la clase Carnivoro, permitiendo consumir comida animal disponible localmente o desde el almacén central.
2. Se permite crear nuevas instancias del pez con un sexo definido y generar copias exactas mediante el método de clonación.

23.3 Constructor

- **public Pejerrey(boolean sexo)**

  Se define el constructor de la clase Pejerrey, el cual inicializa un pez de tipo pejerrey estableciendo sus propiedades específicas mediante la invocación del constructor de la superclase usando super(). Se utiliza una constante predefinida AlmacenPropiedades.PEJERREY para asignar automáticamente las características particulares de esta especie desde un almacén central de datos.

  Parámetros:

  sexo: Se define el sexo del pez, (true para macho, false para hembra)

23.4 Atributos

- No se define ningún atributo.

23.5 Métodos

clonar

- **@Override public Pejerrey clonar(boolean nuevoSexo)**

Se crea una nueva instancia de Pejerrey con el sexo especificado. Este método permite replicar la instancia actual con un sexo diferente.
Devuelve: Una nueva instancia de Pejerrey con el sexo especificado.

24. Clase PercaEuropea

24.1 Descripción

Se define la clase PercaEuropea como una subclase de Carnivoro Activo, representando un pez que se alimenta exclusivamente de comida animal. Se configura utilizando propiedades específicas establecidas en la clase AlmacenPropiedades. Se permite crear instancias del pez y realizar clonaciones especificando un nuevo sexo.

24.2 Funcionalidades

1. Se hereda el comportamiento de alimentación de la clase Carnivoro Activo, permitiendo consumir comida animal disponible localmente o desde el almacén central.
2. Se permite crear nuevas instancias del pez con un sexo definido y generar copias exactas mediante el método de clonación.

24.3 Constructor

- **public PercaEuropea(boolean sexo)**

  Se define el constructor de la clase PercaEuropea, el cual inicializa un pez de tipo PercaEuropea estableciendo sus propiedades específicas mediante la invocación del constructor de la superclase usando super(). Se utiliza una constante predefinida AlmacenPropiedades.PERCA_EUROPEA para asignar automáticamente las características particulares de esta especie desde un almacén central de datos.

  Parámetros:

  sexo: Se define el sexo del pez, (true para macho, false para hembra)

24.4 Atributos

- No se define ningún atributo.

24.5 Métodos

clonar

- **@Override public PercaEuropea clonar(boolean nuevoSexo)**

Se crea una nueva instancia de PercaEuropea con el sexo especificado. Este método permite replicar la instancia actual con un sexo diferente.
Devuelve: Una nueva instancia de PercaEuropea con el sexo especificado.

25. Clase SalmonChinook

25.1 Descripción

Se define la clase SalmonChinook como una subclase de Carnivoro, representando un pez que se alimenta exclusivamente de comida animal. Se configura utilizando propiedades específicas establecidas en la clase AlmacenPropiedades. Se permite crear instancias del pez y realizar clonaciones especificando un nuevo sexo.

25.2 Funcionalidades

1. Se hereda el comportamiento de alimentación de la clase Carnivoro, permitiendo consumir comida animal disponible localmente o desde el almacén central.
2. Se permite crear nuevas instancias del pez con un sexo definido y generar copias exactas mediante el método de clonación.

25.3 Constructor

- **public SalmonChinook(boolean sexo)**

  Se define el constructor de la clase SalmonChinook, el cual inicializa un pez de tipo salmón Chinook estableciendo sus propiedades específicas mediante la invocación del constructor de la superclase usando super(). Se utiliza una constante predefinida AlmacenPropiedades.SALMON_CHINOOK para asignar automáticamente las características particulares de esta especie desde un almacén central de datos.

  Parámetros:

  sexo: Se define el sexo del pez, (true para macho, false para hembra)

25.4 Atributos

- No se define ningún atributo.

25.5 Métodos

clonar

- **@Override public SalmonChinook clonar(boolean nuevoSexo)**

Se crea una nueva instancia de SalmonChinook con el sexo especificado. Este método permite replicar la instancia actual con un sexo diferente.

Devuelve: Una nueva instancia de SalmonChinook con el sexo especificado.

26. Clase TilapiaDelNilo

26.1 Descripción

Se define la clase TilapiaDelNilo como una subclase de Filtrador, representando un pez que se alimenta exclusivamente de comida vegetal. Se configura utilizando propiedades específicas establecidas en la clase AlmacenPropiedades. Se permite crear instancias del pez y realizar clonaciones especificando un nuevo sexo.

26.2 Funcionalidades

1. Se hereda el comportamiento de alimentación de la clase Filtrador, permitiendo consumir comida vegetal disponible localmente o desde el almacén central.
2. Se permite crear nuevas instancias del pez con un sexo definido y generar copias exactas mediante el método de clonación.

26.3 Constructor

- **public TilapiaDelNilo(boolean sexo)**

  Se define el constructor de la clase TilapiaDelNilo, el cual inicializa un pez de tipo tilapia del Nilo estableciendo sus propiedades específicas mediante la invocación del constructor de la superclase usando super(). Se utiliza una constante predefinida AlmacenPropiedades.TILAPIA_NILO para asignar automáticamente las características particulares de esta especie desde un almacén central de datos.

  Parámetros:

  sexo: Se define el sexo del pez, (true para macho, false para hembra)

26.4 Atributos

- No se define ningún atributo.

26.5 Métodos

clonar

- **@Override public TilapiaDelNilo clonar(boolean nuevoSexo)**

Se crea una nueva instancia de TilapiaDelNilo con el sexo especificado. Este método permite replicar la instancia actual con un sexo diferente.
Devuelve: Una nueva instancia de TilapiaDelNilo con el sexo especificado.

27. Clase Pez

27.1 Descripción

Se define la clase abstracta Pez como la clase base para todos los tipos de peces en la simulación. Se gestiona información esencial como el nombre, edad, sexo, estado de vida, alimentación, fertilidad y madurez. También se permite realizar operaciones como el crecimiento diario, la alimentación y el reinicio del estado del pez. Se implementan métodos para mostrar y devolver información relevante sobre los peces.

27.2 Funcionalidades

1. Se permite establecer y consultar atributos clave como nombre, edad, sexo, fertilidad y estado de vida del pez.
2. Se realiza el crecimiento diario del pez, actualizando su edad, fertilidad y estado de madurez según sus propiedades biológicas.
3. Se permite reiniciar todos los atributos a sus valores iniciales para reutilizar objetos existentes.
4. Se define un método abstracto para crear copias exactas de peces con un nuevo sexo especificado.

27.3 Constructor

- **public Pez(boolean sexo, PecesDatos datos)**

  Se define el constructor principal de la clase Pez, encargado de inicializar un pez con los datos específicos proporcionados mediante un objeto de tipo PecesDatos. Se configuran atributos clave relacionados con su especie, nombre científico, sexo y ciclo de vida utilizando la información contenida en el objeto recibido.

  Parámetros:

  sexo: Se define el sexo del pez, (true para macho, false para hembra)

  datos: Se recibe un objeto de tipo PecesDatos, el cual contiene información detallada sobre la especie del pez, incluyendo su nombre común, nombre científico, ciclo de vida y características específicas.

  Inicialización de atributos:

  nombre: Se establece utilizando el nombre común obtenido de datos.getNombre().

  nombreCientifico: Se asigna utilizando el nombre científico obtenido de datos.getCientifico().

sexo: Se asigna directamente a partir del parámetro recibido.

datos: Se guarda el objeto PecesDatos completo para acceder a sus propiedades específicas.

ciclo: Se inicializa utilizando datos.getCiclo(), definiendo el ciclo de vida del pez según su especie.

27.4 Atributos

- **private final String nombre:** Se almacena el nombre común del pez.
- **private final String nombreCientifico:** Se registra el nombre científico del pez.
- **private int edad:** Se almacena la edad actual del pez en días.
- **private boolean sexo:** Se define el sexo del pez (true para macho, false para hembra).
- **private boolean fertil:** Se indica si el pez es fértil.
- **private boolean vivo:** Se indica si el pez está vivo.
- **protected boolean alimentado:** Se indica si el pez ha sido alimentado.
- **private boolean maduro:** Se almacena el nombre común del pez.
- **private final String nombre:** Se indica si el pez ha alcanzado la madurez biológica.
- **protected int ciclo**: Se gestiona el ciclo reproductivo del pez.
- **private PecesDatos datos:** Se almacenan propiedades específicas del pez, como su ciclo de reproducción, edad de madurez y más.

27.5 Métodos

showStatus

- **public void showStatus()**

Se muestra el estado actual del pez, incluyendo edad, sexo, estado de vida, fertilidad, madurez y estado de alimentación.

grow

- **public void grow()**

Se realiza el crecimiento diario del pez. Si no está alimentado, existe una probabilidad de muerte. Si está vivo, su edad se incrementa y se verifican las condiciones de fertilidad y madurez según sus propiedades biológicas.

reset

- **public void reset()**

Se restablecen todos los atributos del pez a sus valores iniciales, permitiendo reutilizar la instancia en simulaciones futuras.

clonar

- **public abstract Pez clonar(boolean nuevoSexo)**

  Se define un método abstracto para crear una copia exacta del pez con el nuevo sexo especificado.

  Se devuelve una nueva instancia de la subclase correspondiente.

alimentar

- **public abstract int alimentar(int cantidadComidaAnimal, int cantidadComidaVegetal)**

  Se define un método abstracto para alimentar al pez, especificando la cantidad de comida animal y vegetal disponible.

  Se devuelve un int que indica la cantidad de comida consumida.

28. Clase Piscifactoria

28.1 Descripción

Se define la clase abstracta Piscifactoria, que representa una instalación de cría y mantenimiento de peces. Se gestiona una lista de tanques, la cantidad de comida disponible y la capacidad máxima de los tanques. Se incluyen métodos para mostrar el estado general de la piscifactoría, alimentar a los peces y realizar simulaciones diarias para actualizar su ciclo de vida.

28.2 Funcionalidades

1. Se permite agregar y manejar tanques que contienen peces.
2. Se controla la cantidad de comida animal y vegetal almacenada en la piscifactoría.
3. Se realiza la simulación del crecimiento y desarrollo de los peces, gestionando su alimentación, fertilidad y supervivencia.
4. Se permite mostrar información detallada sobre la piscifactoría, los tanques y los peces que contiene.

28.3 Constructor

- **public Piscifactoria(String nombre)**

  Se define el constructor de la clase Piscifactoria, encargado de inicializar una piscifactoría asignándole un nombre específico proporcionado como parámetro. Este nombre permite identificar la piscifactoría dentro del sistema de gestión.

  Párametros:

  nombre: Se recibe una cadena de texto que representa el nombre asignado a la piscifactoría. Este nombre se utiliza para identificar de manera única la piscifactoría en el sistema.

  Inicialización de atributos:

  nombre: Se establece con el valor recibido en el parámetro nombre, permitiendo identificar la piscifactoría en registros y operaciones.

28.4 Atributos

- **protected String nombre:** Se almacena el nombre de la piscifactoría.
- **protected List<Tanque> tanques:** Se guarda la lista de tanques disponibles en la piscifactoría.

- **protected final int numeroMaximoTanques:** Se define el número máximo de tanques permitidos.
- **private int cantidadComidaAnimal:** Se registra la cantidad actual de comida animal disponible.
- **private int cantidadComidaVegetal:** Se almacena la cantidad actual de comida vegetal disponible.
- **protected int capacidadMaximaComida:** Se gestiona la capacidad máxima del almacén de comida.

28.5 Métodos

showStatus

- **public void showStatus()**

Se muestra el estado general de la piscifactoría, incluyendo el número de tanques, la ocupación, el total de peces vivos, alimentados y fértiles, así como la cantidad de comida disponible.

showTankStatus

- **public void showTankStatus()**

Se muestra el estado de cada tanque en la piscifactoría.

showFishStatus

- **public void showFishStatus(int numeroTanque)**

Se muestra información detallada de los peces dentro de un tanque específico.

showCapacity

- **public void showCapacity(int numeroTanque)**

  Se muestra la capacidad actual de un tanque determinado.

showFood

- **public void showFood()**

  Se muestra la cantidad de comida animal y vegetal disponible en la piscifactoría.

nextDay

- **public int[] nextDay()**

  Se simula el avance de un día en la piscifactoría, alimentando a los peces y actualizando su ciclo de vida. Se retorna un array con el total de peces vendidos y monedas ganadas.

upgradeFood

- **public abstract void upgradeFood()**

  Se define un método abstracto para aumentar la capacidad máxima del almacén de comida.

addTanque

- **public abstract void addTanque()**

  Se define un método abstracto para añadir un nuevo tanque a la piscifactoría.

añadirComidaAnimal

- **public boolean añadirComidaAnimal(int cantidad)**

  Se añade una cantidad específica de comida animal, verificando que no se exceda la capacidad máxima. Devuelve:
  True si la comida fue añadida correctamente.
  False si no se pudo añadir la comida.

añadirComidaVegetal

- **public boolean añadirComidaVegetal(int cantidad)**

  Se añade una cantidad específica de comida vegetal, verificando que no se exceda la capacidad máxima.Devuelve:
  True si la comida fue añadida correctamente.
  False si no se pudo añadir la comida.

alimentarPeces

- **`private void alimentarPeces(Tanque tanque)`**

  Se alimenta a los peces de un tanque, gestionando la cantidad de comida disponible y actualizando el estado de los peces.

29. Clase PiscifactoriaDeMar

29.1 Descripción

Se define la clase PiscifactoriaDeMar como una subclase de Piscifactoria, diseñada para gestionar la cría y el mantenimiento de peces de mar. Se establece una capacidad máxima de comida limitada y un sistema de mejora y expansión de tanques basado en monedas virtuales.

29.2 Funcionalidades

1. Se permite mejorar la capacidad de almacenamiento de comida hasta un límite máximo.
2. Se permite añadir nuevos tanques, con un costo creciente basado en la cantidad de tanques ya instalados.
3. Se gestiona el gasto de monedas para realizar mejoras y ampliaciones en la piscifactoría.

29.3 Constructor

- **public PiscifactoriaDeMar(String nombre)**

  Se define el constructor de la clase PiscifactoriaDeMar, encargado de inicializar una piscifactoría especializada en peces de agua salada. Se invoca el constructor de la superclase Piscifactoria para establecer el nombre de la piscifactoría. Además, se inicializa un tanque con capacidad de 100 unidades y se define la capacidad máxima de almacenamiento de comida para la piscifactoría.

  Parámetros:

  nombre: Se recibe una cadena de texto que representa el nombre asignado a la piscifactoría. Este nombre permite identificarla en el sistema de gestión de piscifactorías.

  Inicialización de Atributos:

  super(nombre): Se llama al constructor de la superclase Piscifactoria, asignando el nombre proporcionado a la piscifactoría.

  tanques.add(new Tanque(tanques.size() + 1, 100)): Se crea un nuevo tanque con un identificador único y una capacidad de 100 unidades, añadiéndolo a la lista de tanques de la piscifactoría.

  capacidadMaximaComida = 100: Se establece la capacidad máxima de almacenamiento de comida en 100 unidades, permitiendo gestionar el suministro de alimentos en la piscifactoría.

29.4 Atributos

- **private static final int COSTO_MEJORA = 200:** Define el costo en monedas para mejorar la capacidad de almacenamiento de comida.
- **private static final int INCREMENTO_CAPACIDAD = 100:** Especifica el incremento de capacidad de comida en cada mejora.
- **private static final int CAPACIDAD_MAXIMA_PERMITIDA = 1000:** Establece la capacidad máxima permitida para el almacén de comida de la piscifactoría.

29.5 Métodos

upgradeFood

- **@Override public void upgradeFood()**

Se realiza una mejora en la capacidad máxima del almacén de comida, aumentando su capacidad en INCREMENTO_CAPACIDAD hasta alcanzar CAPACIDAD_MAXIMA_PERMITIDA. Si hay monedas suficientes, se realiza la mejora; de lo contrario, se muestra un mensaje informando la falta de monedas.

addTanque

- **@Override public void addTanque()**

Se agrega un nuevo tanque a la piscifactoría, aumentando su costo según la cantidad de tanques existentes. Si hay suficientes monedas, se instala el tanque; de lo contrario, se muestra un mensaje de error.

30. Clase PiscifactoriaDeRio

30.1 Descripción

Se define la clase PiscifactoriaDeRio como una subclase de Piscifactoria, diseñada para gestionar peces de río. Se configura un límite máximo de capacidad de comida y se permite realizar mejoras mediante monedas virtuales. Se incluye un sistema de expansión de tanques basado en costos crecientes según la cantidad de tanques existentes.

30.2 Funcionalidades

1. Se permite aumentar la capacidad de almacenamiento de comida hasta un límite máximo establecido.
2. Se permite agregar nuevos tanques, con costos crecientes según la cantidad de tanques existentes.
3. Se controla el gasto de monedas para realizar mejoras en la capacidad de comida y añadir tanques.

30.3 Constructor

- **public PiscifactoriaDeRio(String nombre)**

  Se define el constructor de la clase PiscifactoriaDeRio, encargado de inicializar una piscifactoría especializada en peces de agua dulce. Se invoca el constructor de la superclase Piscifactoria para establecer el nombre de la piscifactoría. Además, se crea un tanque con una capacidad inicial de 25 unidades y se establece la capacidad máxima de almacenamiento de comida en 25 unidades.

  Parámetros:

  nombre: Se recibe una cadena de texto que representa el nombre asignado a la piscifactoría, permitiendo identificarla dentro del sistema de gestión de piscifactorías.

  Inicialización de atributos:

  super(nombre): Se llama al constructor de la superclase Piscifactoria, asignando el nombre proporcionado a la piscifactoría.

  tanques.add(new Tanque(tanques.size() + 1, 25)): Se crea un nuevo tanque con un identificador único y una capacidad de 25 unidades, añadiéndolo a la lista de tanques de la piscifactoría.

capacidadMaximaComida = 25: Se establece la capacidad máxima de almacenamiento de comida en 25 unidades, permitiendo gestionar el suministro de alimentos en la piscifactoría.

30.4 Atributos

- **private static final int COSTO_MEJORA = 50:** Define el costo en monedas para mejorar la capacidad de almacenamiento de comida.
- **private static final int INCREMENTO_CAPACIDAD = 25:** Especifica el incremento de capacidad de comida en cada mejora.
- **private static final int CAPACIDAD_MAXIMA_PERMITIDA = 1000:** Establece la capacidad máxima permitida para el almacén de comida de la piscifactoría.

30.5 Métodos

upgradeFood

- **@Override public void upgradeFood()**

Se mejora la capacidad máxima de almacenamiento de comida en INCREMENTO_CAPACIDAD hasta alcanzar CAPACIDAD_MAXIMA_PERMITIDA. Si hay monedas suficientes, se realiza la mejora; de lo contrario, se muestra un mensaje indicando la falta de monedas.

addTanque

- **@Override public void addTanque()**

Se agrega un nuevo tanque a la piscifactoría, aumentando su costo según la cantidad de tanques existentes. Si hay monedas suficientes, se instala un nuevo tanque; de lo contrario, se muestra un mensaje informando la falta de monedas.

31. Clase GestorEstado

31.1 Descripción

Se define la clase *GestorEstado* como la encargada de gestionar la persistencia del estado del simulador mediante archivos en formato JSON. Esta clase permite guardar el estado actual del simulador en un archivo ubicado en la carpeta "saves" y, de igual forma, cargar los datos de una partida guardada para reanudar el juego. El proceso de guardado y carga abarca información detallada, incluyendo datos de la entidad, día, monedas, estadísticas, estado del almacén, piscifactorías, tanques y peces.

31.2 Funcionalidades

1. Permite guardar el estado completo del simulador en un archivo JSON, estructurando la información mediante mapas y listas para mantener el orden y la coherencia de los datos.
2. Facilita la carga de una partida guardada, leyendo y parseando el archivo JSON para actualizar el objeto *Simulador* con la información previamente almacenada.
3. Gestiona la persistencia de datos complejos (edificios, tanques, peces, etc.) de forma centralizada, asegurando que se preserve el progreso del simulador entre sesiones.

31.3 Constructor

- No se define un constructor, ya que todos los métodos de la clase son estáticos y operan sin necesidad de instanciar la clase.

31.4 Atributos

- No se definen atributos de instancia en la clase *GestorEstado*, ya que el manejo del estado se realiza mediante métodos estáticos.

31.5 Métodos

guardarEstado

- **public static void guardarEstado(Simulador simulador):**

Guarda el estado actual del simulador en un archivo JSON en la carpeta "saves". Se utiliza la biblioteca J*son* para formatear los datos en formato legible (pretty printing). La información guardada incluye:

- La lista de peces implementados.
- El nombre de la entidad y el día actual.
- El saldo de monedas.
- Estadísticas exportadas.
- Datos del almacén (disponibilidad, capacidad y cantidades de comida).
- Detalles de las piscifactorías, tanques y peces (incluyendo datos específicos como edad, estado, ciclo, etc.).

Si se produce un error durante el guardado, se captura la excepción y se registra el fallo correspondiente.

load

- **public static void load(Simulador simulador, String archivoPartida):**

Carga los datos de una partida guardada desde un archivo JSON ubicado en la carpeta "saves" y actualiza el objeto *Simulador* con dicha información. El método realiza lo siguiente:

- Lee el archivo JSON y parsea su contenido utilizando *JsonParser*.
- Actualiza el nombre de la entidad, el día y el saldo de monedas del simulador.
- Configura el estado del almacén, creando un objeto *AlmacenCentral* si la información lo permite.
- Crea y asigna un objeto *Estadisticas* a partir de los datos exportados.
- Procesa las piscifactorías, tanques y peces, recreando la estructura original del simulador (incluyendo la distinción entre piscifactorías de río y de mar).

Si ocurre algún error durante la lectura o el parseo del archivo, se captura la excepción y se muestra un mensaje de error en la consola, registrando la incidencia en el sistema de log.

32. Clase CrearRecompensa

32.1 Descripción

Se define la clase *CrearRecompensa* como la encargada de gestionar la creación de recompensas en formato XML para el simulador. Esta clase permite generar diversos tipos de recompensas (pienso, algas, comida multipropósito, monedas) y recompensas relacionadas con la construcción de infraestructuras (tanques, piscifactorías y almacén central) basadas en un nivel de recompensa o tipo específico. Utiliza la biblioteca DOM4J para la creación y modificación de documentos XML, y registra las operaciones realizadas mediante *Simulador.registro*.

32.2 Funcionalidades

1. Crear y actualizar archivos XML de recompensas para diferentes elementos (pienso, algas, comida, monedas).
2. Generar recompensas para estructuras de construcción (tanque, piscifactoría, almacén central) en función del tipo y la parte especificada.
3. Incrementar la cantidad de recompensa en el archivo XML si éste ya existe, o crearlo con la estructura completa si no existe.
4. Registrar la creación o actualización de recompensas y gestionar errores durante el proceso.

32.3 Constructor

- La clase *CrearRecompensa* no define un constructor público, ya que todos sus métodos son estáticos y se utilizan sin instanciar la clase.

32.4 Atributos

- **private static final String REWARDS_DIRECTORY:** Directorio donde se almacenan los archivos XML de recompensas (definido como "rewards/").

32.5 Métodos

createPiensoReward

- **public static void createPiensoReward(int type):**

Genera una recompensa de pienso en formato XML basada en el nivel de recompensa especificado.

- Si el archivo correspondiente ya existe, se incrementa la cantidad almacenada en el elemento *quantity*.

- Si el archivo no existe, se crea uno nuevo con elementos que incluyen *name*, *origin*, *desc*, *rarity* y un subelemento *give* que contiene la cantidad de comida animal determinada por *getFoodAmount(type)*.

Registra la creación de la recompensa mediante *Simulador.registro.registroCrearRecompensa*.

createAlgasReward

- **public static void createAlgasReward(int type):**

Genera una recompensa de algas en formato XML basada en el nivel indicado.

- Actualiza la cantidad si el archivo existe, o lo crea con la estructura XML que incluye *name*, *origin*, *desc*, *rarity* y un elemento *give* con el atributo *type* "algae" y la cantidad definida por *getFoodAmount(type)*.

Registra la operación en el log.

createComidaReward

- **public static void createComidaReward(int type):**

Crea una recompensa de comida multipropósito en formato XML basada en el nivel de recompensa.

- Emplea el método *getSharedFoodAmount(type)* para determinar la cantidad y configura el XML con el atributo *type* "general" en el elemento *food*.

Actualiza o crea el archivo según corresponda y registra la acción.

createMonedasReward

- **public static void createMonedasReward(int type):**

Genera una recompensa de monedas en formato XML basada en el nivel especificado.

- Utiliza *getCoinsAmount(type)* para obtener la cantidad de monedas y actualiza o crea el archivo XML, incrementando la cantidad en caso de existencia previa.

Registra la creación de la recompensa.

createTanqueReward

- **public static void createTanqueReward(int type):**

Crea una recompensa para un tanque de piscifactoría en formato XML, basándose en el nivel de recompensa.

- Determina el tipo de tanque (río o mar) mediante el método *getTypeAmount(type)* y configura la rareza, el código del edificio y la parte (fija como "A") en el XML.

Actualiza el archivo existente o lo crea nuevo, y registra la acción.

createPiscifactoriaReward

- **public static void createPiscifactoriaReward(int type, String part):**

Genera una recompensa para una piscifactoría en formato XML, utilizando el nivel de recompensa y la parte especificada (por ejemplo, "A" o "B").

- Determina el tipo (río o mar) con *getTypeAmount(type)* y establece en el XML la rareza, el código del edificio y la parte indicada.

Actualiza o crea el archivo según corresponda, y registra la recompensa.

createAlmacenReward

- **public static void createAlmacenReward(String part):**

Crea una recompensa para un almacén central en formato XML basada en la parte especificada (A, B, C o D).

- Define la estructura XML con elementos como *name*, *origin*, *desc*, *rarity* y un subelemento *give* que contiene la información del edificio y la parte.

Actualiza la cantidad si el archivo existe o lo crea nuevo, y registra la operación.

romanize

- **private static String romanize(int number):**

Convierte un número entero (del 1 al 5) en su representación en números romanos ("I", "II", "III", "IV", "V").

Se utiliza para generar el nombre de la recompensa en función de su nivel.

getFoodAmount

- **private static int getFoodAmount(int type):**

Devuelve la cantidad de comida (para recompensas de pienso y algas) según el nivel de recompensa especificado.

Utiliza un *switch* para asignar valores: 100, 200, 500, 1000 o 2000.

getSharedFoodAmount

- **private static int getSharedFoodAmount(int type): :**

Retorna la cantidad de comida multipropósito asignada según el nivel de recompensa, con valores de 50, 100, 250, 500 o 1000.

getCoinsAmount

- **private static int getCoinsAmount(int type):**

Obtiene la cantidad de monedas asignada para la recompensa, basada en el nivel (1-5), retornando valores específicos (100, 300, 500, 750 o 1000).

getTypeAmount

- **private static char getTypeAmount(int type):**

Determina el tipo de recompensa para elementos de construcción (tanque o piscifactoría) basándose en el nivel indicado.

Retorna 'r' para indicar tipo "río" o 'm' para "mar".

Lanza una excepción si el nivel proporcionado es inválido.

33. Clase UsarRecompensa

33.1 Descripción

Se define la clase *UsarRecompensa* como la encargada de gestionar el uso de recompensas almacenadas en archivos XML. Esta clase procesa recompensas de distintos tipos (comida, monedas, tanque y piscifactoría) actualizando los archivos XML correspondientes: decrementa la cantidad de recompensas disponibles y, cuando esta llega a cero, elimina el archivo. Además, integra las recompensas al simulador (por ejemplo, sumando comida al almacén central o aumentando las monedas) y registra las operaciones realizadas mediante *Simulador.registro*.

33.2 Funcionalidades

1. Procesar recompensas de tipo "comida", "monedas" y "tanque" a partir de archivos XML.
2. Verificar y procesar las partes de recompensas necesarias para formar una piscifactoría (de río o de mar) y para completar un Almacén Central.
3. Actualizar la cantidad de recompensas en los archivos XML, eliminándolos cuando se agota la cantidad disponible.
4. Integrar las recompensas al estado del simulador, actualizando valores como comida y monedas.
5. Registrar el uso de cada recompensa a través del sistema de log.

33.3 Constructor

- No se define un constructor público, ya que todos los métodos de la clase son estáticos y se utilizan sin necesidad de instanciar la clase.

33.4 Atributos

- No se definen atributos de instancia, puesto que la clase opera de forma completamente estática.

33.5 Métodos

readFood

- **public static void readFood(String fileName):**

Procesa una recompensa de tipo "comida" desde un archivo XML ubicado en el directorio "rewards".

- Lee el archivo XML utilizando *SAXReader* y extrae el nombre de la recompensa y la lista de elementos *food* dentro del elemento *give*.

- Dependiendo del atributo *type* del elemento *food* ("algae", "animal" o "general"), añade la cantidad especificada a la comida vegetal, animal o ambos en el *AlmacenCentral*, o distribuye la comida a través del simulador si no hay almacén.

- Decrementa el valor del elemento *quantity* en el XML y elimina el archivo si la cantidad llega a cero.

- Registra el uso de la recompensa mediante *Simulador.registro.registroUsarRecompensa*.

readCoins

- **public static void readCoins(String fileName):**

Procesa una recompensa de tipo "monedas" desde un archivo XML en el directorio "rewards".

- Lee el archivo XML para extraer el valor de las monedas del elemento *coins* contenido en *give*.

- Suma la cantidad obtenida al saldo de monedas del simulador mediante *Simulador.monedas.ganarMonedas*.

- Decrementa la cantidad en el elemento *quantity* y elimina el archivo cuando ésta alcanza cero.

- Registra el uso de la recompensa en el sistema de log.

readTank

- **public static boolean readTank(String fileName):**

Procesa una recompensa de tipo "tanque" desde un archivo XML.

- Lee el archivo XML para obtener el nombre de la recompensa y actualiza el elemento *quantity* decrementándolo en uno.

- Escribe los cambios en el archivo XML y lo elimina si la cantidad se agota.

- Registra la operación realizada y retorna *true* si el procesamiento fue exitoso; en caso de que el archivo no exista, retorna *false*.

readPiscifactoria

- **public static Piscifactoria readPiscifactoria(boolean piscifactoriaRio):**

Verifica si se han reunido todas las partes necesarias para formar una nueva piscifactoría.

- Recorre los archivos XML que comienzan con "piscifactoria_r" o "piscifactoria_m" según corresponda, acumulando las partes obtenidas a partir del elemento *part* en el XML.

- Si la longitud de las partes acumuladas coincide con el total esperado (obtenido del elemento *total*), actualiza cada archivo (decrementando su cantidad y eliminándolos si se agota) y registra el uso de la recompensa.

- Solicita al usuario un nombre para la nueva piscifactoría mediante *InputHelper.readString* y crea un objeto *PiscifactoriaDeRio* o *PiscifactoriaDeMar* según el parámetro *piscifactoriaRio*.

- Retorna el objeto piscifactoría creado o *null* si no se han reunido todas las partes.

readAlmacenCentral

- **public static boolean readAlmacenCentral():**

Verifica si se han reunido todas las partes necesarias para completar un Almacén Central.

- Recorre los archivos XML que comienzan con "almacen", acumulando las partes obtenidas a partir del elemento *part*.

- Compara la longitud de las partes acumuladas con el total esperado (definido en el elemento *total*).

- Si se han reunido todas las partes, actualiza los archivos XML (decrementa la cantidad y elimina los archivos cuando corresponda) y registra el uso de la recompensa.

- Retorna *true* si todas las partes están completas; de lo contrario, retorna *false*.

34. Clase Tanque

34.1 Descripción

Se define la clase Tanque, que representa un contenedor para almacenar peces con capacidades de gestión, monitoreo y reproducción. Se permite realizar operaciones de cría, alimentación y venta de peces, además de mostrar información detallada sobre su estado actual.

34.2 Funcionalidades

1. Se permite añadir peces, comprobar compatibilidad y gestionar su crecimiento, reproducción y venta.
2. Se permite mostrar el estado general del tanque y detalles específicos como ocupación, cantidad de peces vivos, alimentados, adultos y fértiles.
3. Se permite simular el crecimiento diario de los peces y registrar estadísticas relacionadas con la reproducción y venta.

34.3 Constructor

- **public Tanque(int numeroTanque, int capacidadMaxima)**

  Se define el constructor de la clase Tanque, encargado de inicializar un tanque de almacenamiento de peces y comida en la piscifactoría. Se asigna un número identificador único al tanque y se establece su capacidad máxima según los valores proporcionados como parámetros.

  numeroTanque: Se recibe un valor entero que representa el número identificador único del tanque dentro de la piscifactoría.

  capacidadMaxima: Se recibe un valor entero que indica la capacidad máxima de almacenamiento del tanque en términos de unidades de peces o comida.

  this.numeroTanque = numeroTanque: Se asigna el número de identificación proporcionado al tanque.

  this.capacidadMaxima = capacidadMaxima: Se establece la capacidad máxima del tanque según el valor recibido.

34.4 Atributos

- **private List<Pez> peces:** Se almacena la lista de peces que habitan el tanque.
- **private final int numeroTanque:** Se define el número identificador del tanque.

- **private final int capacidadMaxima:** Se establece la capacidad máxima de peces que puede contener el tanque.

34.5 Métodos

showStatus

- **public void showStatus()**

Se muestra el estado general del tanque, incluyendo ocupación, cantidad de peces vivos, alimentados, adultos, hembras, machos y fértiles.

showFishStatus

- **public void showFishStatus()**

Se muestra el estado individual de cada pez en el tanque, incluyendo su edad, estado de vida y alimentación.

showCapacity

- **public void showCapacity()**

  Se muestra la capacidad actual del tanque en porcentaje y número de peces almacenados.

nextDay

- **public int[] nextDay()**

  Se simula el avance de un día en el tanque, haciendo crecer a los peces y gestionando su reproducción y ventas. Se retorna un array con el total de peces vendidos y monedas ganadas.

reproduccion

- **public void reproduccion()**

  Se gestiona la reproducción de los peces en el tanque, generando nuevos peces si existen machos y hembras fértiles. Si se supera la capacidad máxima, se interrumpe la reproducción.

addFish

- **public boolean addFish(Pez pez)**
  Se agrega un pez al tanque, verificando compatibilidad de especie y disponibilidad de monedas. Devuelve:
  True si el pez se agregó correctamente.
  False si el tanque está lleno o la especie es incompatible.

sellFish

- **public int[] sellFish()**
  Se venden los peces maduros del tanque y se actualiza el saldo de monedas. Se registran estadísticas de ventas. Se devuelve:
  Un array con el número total de peces vendidos y monedas ganadas.

35. Clase Logger

35.1 Descripción

Se define la clase *Logger* para gestionar el registro de logs en archivos. Esta clase permite escribir mensajes de error y mensajes generales de eventos en dos archivos de log distintos: uno global para errores (ERROR_LOG_PATH) y otro específico para cada partida, basado en el nombre de la partida. La clase implementa el patrón Singleton, garantizando que solo exista una instancia única durante la ejecución de la aplicación.

35.2 Funcionalidades

1. Registrar errores generales en un archivo de log global, incluyendo un timestamp en cada entrada.
2. Registrar eventos y mensajes específicos de la partida en un archivo de log particular, generando y creando la carpeta de logs si es necesario.
3. Proveer métodos para registrar distintos tipos de acciones (inicio de partida, compras, ventas, mejoras, finalización de día, etc.) y para cerrar adecuadamente el escritor de logs.

35.3 Constructor

- **private Logger(String nombrePartida)**

  Se define el constructor privado de la clase *Logger*, el cual inicializa el escritor persistente para el log general de errores utilizando el archivo ubicado en *ERROR_LOG_PATH*. Asimismo, configura la ruta del log específico de la partida basándose en el nombre de la partida recibido como parámetro. Este constructor garantiza que tanto el log global como el de la partida estén listos para registrar mensajes desde el inicio.

35.4 Atributos

- **private final String ERROR_LOG_PATH:** Ruta del archivo de log de errores general, definida como "logs/0_errors.log".
- **private BufferedWriter errorWriter**: Escritor persistente que se utiliza para escribir en el archivo de log general de errores.
- **private static Logger instance**: Instancia única de la clase *Logger*, implementando el patrón Singleton.
- **private String logPath**: Ruta del archivo de log específico de la partida, configurado con el nombre de la partida.

35.5 Métodos

getInstance

- **public static Logger getInstance(String nombrePartida):**

Se devuelve la instancia única de *Logger*. Si aún no se ha creado, se invoca el constructor privado pasando el nombre de la partida para generar la ruta del log de la misma.

logError

- **void logError(String message):**

Se escribe un mensaje de error en el log global de errores, precedido de un timestamp. Se asegura de escribir y vaciar el buffer para que el mensaje se registre de inmediato.

log

- **void log(String message):**

Se escribe un mensaje en el log específico de la partida. Verifica que el directorio del archivo de log exista, creándolo de ser necesario, y escribe el mensaje junto con un timestamp.

close

- **void close():**

Se cierra el escritor persistente del log general, liberando los recursos asociados y estableciendo el escritor a *null*.

logInicioPartida

- **void logInicioPartida(String nombrePartida, String nombrePiscifactoria):**

Se registra el inicio de la partida, indicando el nombre de la partida y el nombre de la piscifactoría inicial.

logComprarComidaPiscifactoria

- **void logComprarComidaPiscifactoria(int cantidadComida, String tipoComida, String nombrePiscifactoria):**

Se registra la compra de comida para una piscifactoría, especificando la cantidad, el tipo de comida y el nombre de la piscifactoría destino.

logComprarComidaAlmacenCentral

- **void logComprarComidaAlmacenCentral(int cantidadComida, String tipoComida):**

Se registra la compra de comida para el almacén central, indicando la cantidad y el tipo de comida adquirida.

logComprarPeces

- **void logComprarPeces(String nombrePez, Boolean sexoPez, int numeroTanque, String nombrePiscifactoria):**

Se registra la compra de un pez, incluyendo su nombre, sexo (representado como "M" o "H"), el número de tanque donde se añade y el nombre de la piscifactoría correspondiente.

logVenderPeces

- **void logVenderPeces(int numeroPecesVendidos, String nombrePiscifactoria):**

Se registra la venta de peces desde una piscifactoría, especificando la cantidad vendida y el nombre de la piscifactoría.

logLimpiarTanque

- **void logLimpiarTanque(int numeroTanque, String nombrePiscifactoria):**

Se registra la acción de limpiar un tanque, indicando el número del tanque y el nombre de la piscifactoría donde se realizó la limpieza.

logVaciarTanque

- **void logVaciarTanque(int numeroTanque, String nombrePiscifactoria):**

Se registra el vaciado de un tanque, mencionando el número del tanque y el nombre de la piscifactoría correspondiente.

logComprarPiscifactoria

- **void logComprarPiscifactoria(String tipoPiscifactoria, String nombrePiscifactoria):**

Se registra la compra de una nueva piscifactoría, especificando el tipo (por ejemplo, "de mar" o "de río") y el nombre de la misma.

logComprarTanque

- **void logComprarTanque(String nombrePiscifactoria):**

Se registra la compra de un tanque adicional para una piscifactoría, indicando el nombre de la piscifactoría afectada.

logComprarAlmacenCentral

- **void logComprarAlmacenCentral():**

Se registra la compra del almacén central.

logMejorarPiscifactoria

- **void logMejorarPiscifactoria(String nombrePiscifactoria):**

Se registra la mejora de una piscifactoría, indicando el aumento en la capacidad de comida.

logMejorarAlmacenCentral

- **void logMejorarAlmacenCentral(int incrementoCapacidad, int capacidadComida):**

Se registra la mejora del almacén central, detallando el incremento en la capacidad y la nueva capacidad total de comida.

logFinDelDia

- **void logFinDelDia(int dia):**

Se registra el final de un día en la simulación, indicando el número del día finalizado.

logOpcionOcultaPeces

- **void logOpcionOcultaPeces(String nombrePiscifactoria):**

Se registra la adición de peces a través de una opción oculta, especificando el nombre de la piscifactoría afectada.

logOpcionOcultaMonedas

- **void logOpcionOcultaMonedas():**

Se registra la adición de monedas mediante una opción oculta.

logCierrePartida

- **void logCierrePartida():**

Se registra el cierre de la partida.

logCrearRecompensa

- **void logCrearRecompensa():**

Se registra la creación de una recompensa.

logUsarRecompensa

- **void logUsarRecompensa(String nombreRecompensa):**

Se registra el uso de una recompensa, especificando el nombre de la misma.

logGuardarSistema

- **void logGuardarSistema():**

Se registra la acción de guardar el sistema.

logCargarSistema

- **void logCargarSistema():**

Se registra la acción de cargar el sistema.

logGenerarPedidos

- **void logGenerarPedidos(String referenciaPedido):**

Se registra la generación de un pedido, indicando la referencia asignada.

logPedidoEnviado

- **void logPedidoEnviado(String nombrePez, String referenciaPedido):**

Se registra el envío de un pedido, especificando el nombre del pez y la referencia del pedido.

logEnviadosConReferencia

- **void logEnviadosConReferencia(int cantidadPeces, String nombrePez, String referenciaPedido):**

Se registra la cantidad de peces enviados a un pedido, incluyendo la referencia del pedido y el nombre del pez involucrado.

36. Clase Transcriptor

36.1 Descripción

Se define la clase *Transcriptor* como la encargada de gestionar transcripciones detalladas de acciones en el sistema. Esta clase registra, en un archivo de transcripción ubicado en el directorio "transcripciones", todas las acciones importantes y eventos ocurridos durante la partida. La clase implementa el patrón Singleton para asegurar una única instancia y utiliza un *BufferedWriter* para escribir en el archivo de forma eficiente.

36.2 Funcionalidades

1. Registrar de forma detallada las acciones y eventos críticos del simulador, tales como el inicio de la partida, compras, ventas, mejoras, acciones ocultas, etc.
2. Generar un archivo de transcripción específico para cada partida, cuyo nombre se deriva del nombre de la partida.
3. Permitir la escritura y almacenamiento persistente de mensajes de transcripción, facilitando la revisión de las acciones realizadas durante la simulación.

36.3 Constructor

- **private Transcriptor(String nombrePartida)**

  Se define el constructor privado de la clase *Transcriptor*, el cual inicializa la ruta del archivo de transcripción basándose en el nombre de la partida. El archivo se crea en el directorio "transcripciones" con extensión ".tr". Este constructor es invocado una única vez a través del método *getInstance*, garantizando la implementación del patrón Singleton.

36.4 Atributos

- **private static Transcriptor instance:** Se instancia única de la clase *Transcriptor* que asegura la existencia de un único objeto para la transcripción durante la partida.
- **private String transcripcionPartidaPath**: Ruta del archivo de transcripción, definida en función del nombre de la partida.

36.5 Métodos

getInstance

- **public static Transcriptor getInstance(String nombrePartida):**

Se obtiene la instancia única del *Transcriptor*. Si la instancia aún no existe, se crea una nueva utilizando el nombre de la partida para establecer la ruta del archivo de transcripción.

transcribir

- **public void transcribir(String mensaje):**

Se registra un mensaje en el archivo de transcripción. Abre el archivo en modo de adición, escribe el mensaje y añade una nueva línea. Si ocurre algún error durante la escritura, se muestra un mensaje de error en la consola.

transcribirInicioPartida

- **void transcribirInicioPartida(String nombrePartida, int monedas, String nombrePiscifactoria, int dia):**

Se transcribe el inicio de la partida con detalles iniciales, incluyendo el nombre de la partida, el dinero inicial, el listado de peces (obtenido de *AlmacenPropiedades*) y el inicio del día siguiente. También registra la piscifactoría inicial.

transcribirComprarComidaPiscifactoria

- **void transcribirComprarComidaPiscifactoria(int cantidadComida, String tipoComida, int costoComida, String nombrePiscifactoria):**

Se transcribe la compra de comida para una piscifactoría, detallando la cantidad, el tipo de comida, el costo y la piscifactoría donde se almacena la comida.

transcribirComprarComidaAlmacenCentral

- **void transcribirComprarComidaAlmacenCentral(int cantidadComida, String tipoComida, int costoComida):**

Se transcribe la compra de comida para el almacén central, registrando la cantidad, el tipo de comida y el costo en monedas.

transcribirComprarPeces

- **void transcribirComprarPeces(String nombrePez, Boolean sexoPez, int costePez, int numeroTanque, String nombrePiscifactoria):**

Se transcribe la compra de peces, incluyendo el nombre del pez, su sexo (indicando "M" para macho o "H" para hembra), el costo, el número del tanque y la piscifactoría destino.

transcribirVenderPeces

- **void transcribirVenderPeces(int numeroPecesVendidos, String nombrePiscifactoria, int monedasGanadas):**

Se transcribe la venta de peces, registrando la cantidad de peces vendidos, el nombre de la piscifactoría y las monedas ganadas por la venta.

transcribirLimpiarTanque

- **void transcribirLimpiarTanque(int numeroTanque, String nombrePiscifactoria):**

Se transcribe la limpieza de un tanque, indicando el número del tanque y el nombre de la piscifactoría donde se realizó la acción.

transcribirVaciarTanque

- **void transcribirVaciarTanque(int numeroTanque, String nombrePiscifactoria):**

Se transcribe el vaciado de un tanque, especificando el número del tanque y la piscifactoría correspondiente.

transcribirComprarPiscifactoria

- **void transcribirComprarPiscifactoria(String tipoPiscifactoria, String nombrePiscifactoria, int costoPiscifactoria):**

Se transcribe la compra de una nueva piscifactoría, indicando el tipo de piscifactoría, el nombre asignado y el costo en monedas.

transcribirComprarTanque

- **void transcribirComprarTanque(int numeroTanque, String nombrePiscifactoria):**

Se transcribe la compra de un tanque adicional para una piscifactoría, registrando el número del tanque y el nombre de la piscifactoría.

transcribirComprarAlmacenCentral

- **void transcribirComprarAlmacenCentral():**

Se transcribe la compra del almacén central.

transcribirMejorarPiscifactoria

- **void transcribirMejorarPiscifactoria(String nombrePiscifactoria, int capacidadComida, int costoMejoraPiscifactoria):**

Se transcribe la mejora de una piscifactoría, detallando el nuevo total de capacidad de comida y el costo de la mejora.

transcribirMejorarAlmacenCentral

- **void transcribirMejorarAlmacenCentral(int incrementoCapacidad, int capacidadComida, int costoMejoraAlmacenCentral):**

Se transcribe la mejora del almacén central, indicando el incremento en la capacidad de comida, la nueva capacidad total y el costo de la mejora.

transcribirFinDelDia

- **void transcribirFinDelDia(int dia, int pecesDeRio, int pecesDeMar, int totalMonedasGanadas, int monedasActuales):**

Se transcribe el fin del día, registrando el número del día finalizado, las cantidades de peces de río y de mar, el total de monedas ganadas y las monedas actuales. Finaliza la transcripción con una línea de separación y la indicación del inicio del siguiente día.

transcribirOpcionOcultaPeces

- **void transcribirOpcionOcultaPeces(String nombrePiscifactoria):**

Se transcribe la adición de peces a través de una opción oculta, indicando el nombre de la piscifactoría afectada.

transcribirOpcionOcultaMonedas

- **void transcribirOpcionOcultaMonedas(int monedasActuales):**

Se transcribe la adición de monedas mediante una opción oculta, mostrando la cantidad total de monedas actuales tras la operación.

transcribirCrearRecompensa

- **void transcribirCrearRecompensa(String nombreRecompensa):**

Se transcribe la creación de una recompensa, especificando su nombre.

transcribirUsarRecompensa

- **void transcribirUsarRecompensa(String nombreRecompensa):**

Se transcribe el uso de una recompensa, indicando el nombre de la recompensa que se ha utilizado.

transcribirGenerarPedidos

- **void transcribirGenerarPedidos(String referenciaPedido):**

Se transcribe la generación de un pedido, registrando la referencia asignada al pedido.

transcribirPedidoEnviado

- **void transcribirPedidoEnviado(String nombrePez, String referenciaPedido):**

Se transcribe el envío de un pedido, especificando el nombre del pez y la referencia del pedido enviado.

transcribirEnviadosConReferencia

- **void transcribirEnviadosConReferencia(int cantidadPeces, String nombrePez, String referenciaPedido):**

Se transcribe la cantidad de peces enviados a un pedido, incluyendo el nombre del pez y la referencia del pedido involucrado.

37. Clase Registros

37.1 Descripción

Se define la clase *Registros* como el componente central para gestionar y coordinar el registro de eventos y acciones de la partida. Esta clase actúa como una fachada que integra dos sistemas de registro: el *Logger*, encargado de los logs de eventos y errores en archivos, y el *Transcriptor*, responsable de transcribir detalladamente las acciones en archivos de transcripción. Con ello, se asegura que todas las operaciones críticas y eventos del simulador queden documentados tanto en el log general como en un registro específico de la partida.

37.2 Funcionalidades

1. Se registra el inicio de la partida, compras, ventas, mejoras y otras acciones clave del simulador.
2. Se transcriben detalladamente los eventos de las piscifactorías y del almacén, permitiendo una revisión completa de la evolución de la partida.
3. Se coordinan la escritura en archivos de log y transcripción, garantizando la persistencia y el orden de la información registrada.
4. Se proveen métodos específicos para cada tipo de acción, facilitando el mantenimiento y la escalabilidad del sistema de registros.

37.3 Constructor

- **public Registros(String nombrePartida)**

  Se inicializa el sistema de registros de la partida. Se obtiene la instancia única de *Logger* y de *Transcriptor* utilizando el nombre de la partida, lo que permite generar rutas específicas para el log global y el archivo de transcripción.

37.4 Atributos

- **public Logger logger:** Se instancia de *Logger* para gestionar el registro de eventos y errores en el sistema.
- **public Transcriptor transcriptor**: Se instancia de *Transcriptor* para registrar de forma detallada las acciones realizadas durante la partida en un archivo de transcripción.

37.5 Métodos

registroInicioPartida

- **public void registroInicioPartida(String nombrePartida, int monedas, String nombrePiscifactoria, int dia):**

Se registra y transcribe el inicio de la partida. Se documenta el nombre de la partida, la cantidad inicial de monedas, el nombre de la piscifactoría inicial y el día de inicio de la simulación, mediante las llamadas correspondientes a *logger.logInicioPartida* y *transcriptor.transcribirInicioPartida*.

registroComprarComidaPiscifactoria

- **public void registroComprarComidaPiscifactoria(int cantidadComida, String tipoComida, int costoComida, String nombrePiscifactoria):**

Se registra y transcribe la compra de comida para una piscifactoría. Se invocan los métodos de *Logger* y *Transcriptor* para documentar la cantidad de comida, el tipo de comida, el costo en monedas y el destino de la compra.

registroComprarComidaAlmacenCentral

- **public void registroComprarComidaAlmacenCentral(int cantidadComida, String tipoComida, int costoComida):**

Se registra y transcribe la compra de comida destinada al almacén central. Se registra el evento con la información de la cantidad, el tipo de comida y el costo asociado.

registroComprarPeces

- **public void registroComprarPeces(String nombrePez, Boolean sexoPez, int costePez, int numeroTanque, String nombrePiscifactoria):**

Se registra y transcribe la compra de peces y su asignación a un tanque específico en una piscifactoría. Se documentan el nombre del pez, su sexo (indicando "M" para macho o "H" para hembra), el costo, el número de tanque y la piscifactoría destino.

registroVenderPeces

- **public void registroVenderPeces(int numeroPecesVendidos, String nombrePiscifactoria, int monedasGanadas):**

Se registra y transcribe la venta de peces junto con las monedas obtenidas. Se registra la cantidad de peces vendidos, el nombre de la piscifactoría y el total de monedas ganadas.

registroLimpiarTanque

- **public void registroLimpiarTanque(int numeroTanque, String nombrePiscifactoria):**

Se registra y transcribe la acción de limpiar un tanque dentro de una piscifactoría, documentando el número del tanque y el nombre de la piscifactoría afectada.

registroVaciarTanque

- **public void registroVaciarTanque(int numeroTanque, String nombrePiscifactoria):**

Se registra y transcribe el vaciado de un tanque, indicando el número de tanque y la piscifactoría correspondiente.

registroComprarPiscifactoria

- **public void registroComprarPiscifactoria(String tipoPiscifactoria, String nombrePiscifactoria, int costePiscifactoria):**

Se registra y transcribe la compra de una nueva piscifactoría, especificando el tipo de piscifactoría, el nombre asignado y el costo de la compra en monedas.

registroComprarTanque

- **public void registroComprarTanque(int numeroTanque, String nombrePiscifactoria):**

Se registra y transcribe la compra de un tanque adicional para una piscifactoría. Se registra la operación indicando el número del tanque y el nombre de la piscifactoría.

registroComprarAlmacenCentral

- **public void registroComprarAlmacenCentral():**

Se registra y transcribe la compra del almacén central.

registroMejorarPiscifactoria

- **public void registroMejorarPiscifactoria(String nombrePiscifactoria, int capacidadComida, int costoMejoraPiscifactoria):**

Se registra y transcribe la mejora de una piscifactoría, incluyendo la nueva capacidad total de comida y el costo de la mejora.

registroMejorarAlmacenCentral

- **public void registroMejorarAlmacenCentral(int incrementoCapacidad, int capacidadComida, int costoMejoraAlmacenCentral):**

Se registra y transcribe la mejora del almacén central, detallando el incremento en la capacidad, la nueva capacidad total y el costo asociado.

registroFinDelDia

- **public void registroFinDelDia(int dia, int pecesDeRio, int pecesDeMar, int totalMonedasGanadas, int monedasActuales):**

Se registra y transcribe el fin del día en la simulación. Se documentan el número del día, las cantidades de peces de río y de mar, el total de monedas ganadas y las monedas actuales. Además, se indica el inicio del siguiente día.

registroOpcionOcultaPeces

- **public void registroOpcionOcultaPeces(String nombrePiscifactoria):**

Se registra y transcribe la adición de peces mediante una opción oculta, indicando el nombre de la piscifactoría afectada.

registroOpcionOcultaMonedas

- **public void registroOpcionOcultaMonedas(int monedasActuales):**

Se registra y transcribe la adición de monedas a través de una opción oculta, mostrando la cantidad total de monedas tras la operación.

registroCrearRecompensa

- **public void registroCrearRecompensa(String nombreRecompensa):**

Se registra y transcribe la creación de una recompensa, especificando el nombre de la recompensa creada.

registroUsarRecompensa

- **public void registroUsarRecompensa(String nombreRecompensa):**

Se registra y transcribe el uso de una recompensa, indicando el nombre de la recompensa utilizada.

registroGenerarPedidos

- **public void registroGenerarPedidos(String referenciaPedido):**

Se registra y transcribe la generación de un pedido, documentando la referencia única asignada al pedido.

registroPedidoEnviado

- **public void registroPedidoEnviado(String nombrePez, String referenciaPedido):**

Se registra y transcribe el envío de un pedido, indicando el nombre del pez involucrado y la referencia del pedido.

registroEnviadosConReferencia

- **public void registroEnviadosConReferencia(int cantidadPeces, String nombrePez, String referenciaPedido):**

Se registra y transcribe la cantidad de peces enviados en un pedido, incluyendo la referencia del pedido y el nombre del pez.

registroLogError

- **public void registroLogError(String message):**

Se registra un mensaje de error en el log general utilizando el método *logger.logError*.

registroCierrePartida

- **public void registroCierrePartida():**

Se registra el cierre de la partida en el sistema de logs.

registroGuardarSistema

- **public void registroGuardarSistema():**

Se registra la acción de guardar el estado actual del sistema en el log.

registroCargarSistema

- **public void registroCargarSistema():**

Se registra la acción de cargar un estado previamente guardado en el log.

closeLogError

- **public void closeLogError():**

Se cierra el sistema de logs de errores, liberando los recursos utilizados por el *Logger*.

38. Clase Simulador

38.1 Descripción

Se define la clase *Simulador* como el núcleo central del sistema, encargada de gestionar la simulación completa de una piscifactoría. En ella se integran los módulos de administración de peces, tanques, edificios, recursos monetarios, estadísticas y persistencia del estado. Asimismo, se proporciona un menú interactivo que permite navegar entre las diversas opciones de gestión, de modo que se puedan realizar operaciones como ver el estado general, operar sobre piscifactorías y tanques, efectuar compras y ventas, mejorar edificios, gestionar recompensas y simular el paso de días. Todas las acciones se registran mediante el sistema de registros.

38.2 Funcionalidades

1. Se inicializa el sistema creando las carpetas necesarias, generando la estructura de la base de datos y permitiendo cargar o iniciar una partida nueva.
2. Se administra y muestra el estado general y específico de la simulación, incluyendo el día actual, las monedas disponibles, la información de cada piscifactoría, el estado de los tanques y la existencia del almacén central.
3. Se facilita la interacción mediante menús interactivos para seleccionar piscifactorías, tanques y ejecutar diversas operaciones (compras, ventas, mejoras, gestión de recompensas, envío de pedidos, etc.).
4. Se simula el avance de uno o varios días, actualizándose los estados de las piscifactorías, procesándose las ventas y generándose pedidos automáticos, todo ello registrándose adecuadamente.
5. Se coordinan operaciones específicas como añadir comida, comprar peces, limpiar o vaciar tanques, mejorar edificios, generar y utilizar recompensas, y enviar pedidos manuales.
6. Se integra la persistencia del estado mediante métodos para cargar y guardar la simulación, y se registran todas las acciones mediante el sistema de registros.

38.3 Constructor

- **No se posee un constructor en la clase Simulador**

38.4 Atributos

- **private int dia:** Se almacena el número de días transcurridos en la simulación.
- **private static List<Piscifactoria> piscifactorias**: Se mantiene la lista de todas las piscifactorías creadas en el sistema.
- **public static String nombreEntidad:** Se define el nombre de la entidad o partida en la simulación.
- **private String nombrePiscifactoria**: Se guarda el nombre de la primera piscifactoría, ingresado al iniciar una nueva partida.
- **public final static String[] pecesImplementados**: Se establece un arreglo con los nombres de los peces implementados, obtenido a partir de *AlmacenPropiedades*.
- **public static SistemaMonedas monedas:** Se utiliza la instancia del sistema de monedas para gestionar las transacciones.
- **public static Estadisticas estadisticas:** Se utiliza el sistema de estadísticas para registrar la cría, venta y ganancias de peces.
- **public static AlmacenCentral almacenCentral:** Se define el almacén central de comida para abastecer las piscifactorías.
- **public static Registros registro:** Se emplea el sistema de registros, que integra *Logger* y *Transcriptor*, para documentar eventos y acciones del sistema.
- **public DAOPedidos pedidos:** Se instancia el objeto DAO encargado de gestionar las operaciones de pedidos en la base de datos.
- **public GeneradorBD generador:** Se instancia el objeto responsable de generar la estructura de la base de datos (creación de tablas e inserción de datos iniciales).

38.5 Métodos

init

- **public void init():**

Se inicia el simulador creando las carpetas necesarias, generando las tablas de la base de datos y permitiendo la carga de una partida existente o el inicio de una nueva. En el caso de una nueva partida se solicitan el nombre de la entidad y de la primera piscifactoría, se inicializan las estadísticas y se registra el estado inicial mediante *GestorEstado.guardarEstado(this)*.

menu

- **public void menu():**

Se muestra el menú principal con las opciones disponibles (estado general, estado de piscifactoría, estado de tanque, informes, Ictiopedia, pasar día, comprar comida, comprar peces, vender peces, limpiar tanque, vaciar tanque, mejorar, recompensas, pasar varios días, enviar pedido y salir).

menuPisc

- **public void menuPisc():**

Se muestra un menú con la lista de piscifactorías actuales, presentando para cada una información detallada (peces vivos, peces totales y capacidad total).

selectPisc

- **public Piscifactoria selectPisc():**

Se permite seleccionar una piscifactoría de la lista mostrada en *menuPisc()* y se retorna la piscifactoría seleccionada; en caso de cancelación se retorna *null*.

selectTank

- **public Map.Entry<Piscifactoria, Tanque> selectTank():**

Se permite seleccionar un tanque de una piscifactoría. Se muestra un menú con la lista de tanques y se retorna un par (piscifactoría, tanque) mediante *AbstractMap.SimpleEntry*; si no se selecciona ninguno se retorna (null, null).

showGeneralStatus

- **public void showGeneralStatus():**

Se muestra el estado general de la simulación, incluyendo el día actual, las monedas disponibles, el estado de cada piscifactoría y, de existir, la información del almacén central.

showSpecificStatus

- **public void showSpecificStatus():**

Se permite seleccionar una piscifactoría y se muestra su estado específico, enfocado en la información de sus tanques.

showTankStatus

- **public void showTankStatus():**

Se muestra el estado de los peces en un tanque seleccionado de una piscifactoría; si el tanque está vacío se informa al usuario.

showStats

- **public void showStats():**

Se muestra un desglose de las estadísticas registradas por cada tipo de pez, utilizando el objeto *estadisticas*.

showIctio

- **public void showIctio():**

Se presenta la Ictiopedia, que muestra de forma detallada las propiedades y características de los peces implementados (nombre, nombre científico, tipo, coste, alimentación, etc.).

nextDay

- **public void nextDay():**

Se simula el avance de un día en la simulación. Se incrementa el contador de días, se procesan las operaciones en cada piscifactoría (por ejemplo, venta de peces y generación de monedas), y se registra el fin del día mediante *registro.registroFinDelDia*. Además, cada 10 días se genera un pedido automático.

nextDay (int dias)

- **public void nextDay(int dias):**

Se simula el avance de varios días consecutivos invocando el método *nextDay()* la cantidad de veces especificada.

addFood

- **public void addFood():**

Se permite añadir comida al sistema. En ausencia de un almacén central se solicita seleccionar una piscifactoría para agregarle comida; en caso contrario, se añade al almacén central. Se muestran menús para seleccionar el tipo (Animal o Vegetal) y la cantidad, se calcula el costo mediante *monedas.calcularDescuento()* y se actualiza el estado de comida, registrándose la operación.

addFish

- **public void addFish():**

Se permite añadir un pez al tanque seleccionado. Si el tanque está vacío se muestran opciones de peces según el tipo de piscifactoría (de río o de mar); de lo contrario, se permite clonar el primer pez existente. Se determina el sexo del pez en función del equilibrio de machos y hembras en el tanque, se añade el pez y se registra la operación actualizando las estadísticas de nacimientos.

sell

- **public void sell():**

Se procede a vender todos los peces adultos del tanque seleccionado, incrementándose las monedas disponibles y registrándose la venta mediante *registro.registroVenderPeces*.

cleanTank

- **public void cleanTank():**

Se eliminan del tanque seleccionado todos los peces muertos. Se efectúa la limpieza y se registra la acción.

emptyTank

- **public void emptyTank():**

Se vacía el tanque seleccionado eliminando todos los peces y se registra la operación de vaciado.

upgrade

- **public void upgrade():**

Se muestra un menú de gestión de edificios que permite al usuario comprar o mejorar edificios (piscifactorías y, si está disponible, el almacén central). Se invocan métodos internos para gestionar cada operación y se registra la acción.

gestionarCompraEdificios

- **private void gestionarCompraEdificios():**

Se gestionan las compras de edificios mediante un menú que ofrece la opción de comprar una piscifactoría o el almacén central, según la disponibilidad. Se registra la operación.

gestionarMejoraEdificios

- **private void gestionarMejoraEdificios():**

Se gestionan las mejoras de edificios mostrando un menú para mejorar una piscifactoría o el almacén central, registrándose la acción correspondiente.

upgradePiscifactoria

- **public void upgradePiscifactoria():**

Se permite mejorar una piscifactoría seleccionada, ofreciendo opciones para comprar un tanque adicional o aumentar la capacidad de comida. Se registra la mejora en el sistema.

contarPiscifactoriasDeRio

- **public int contarPiscifactoriasDeRio():**

Se cuentan y retornan el número de piscifactorías de río existentes.

contarPiscifactoriasDeMar

- **public int contarPiscifactoriasDeMar():**

Se cuentan y retornan el número de piscifactorías de mar existentes.

addPiscifactoria

- **public void addPiscifactoria():**

Se permite agregar una nueva piscifactoría. Se solicita el nombre y se presentan opciones para elegir entre piscifactoría de río o de mar, calculándose el costo en función del número actual de cada tipo. Se registra la compra y se añade la nueva piscifactoría a la lista.

recompensas

- **private void recompensas():**

Se muestra un menú con las recompensas disponibles (obtenidas mediante *FileHelper.getRewards()*), permitiéndose al usuario seleccionar una recompensa. Según la opción elegida, se invocan los métodos de *UsarRecompensa* para aplicar la recompensa y se registra la acción.

generarRecompensas

- **private void generarRecompensas():**

Se generan diversas recompensas invocando los métodos correspondientes de *CrearRecompensa* para cada tipo (algas, pienso, comida multipropósito, monedas, tanque, piscifactoría y almacén central).

pecesRandom

- **public void pecesRandom():**

Se añaden 4 peces de forma aleatoria al primer tanque de la piscifactoría seleccionada que disponga de espacio suficiente. Se determina el sexo de cada pez en función del equilibrio actual en el tanque y se registra la operación como una opción oculta.

enviarPedidoManual

- **public void enviarPedidoManual():**

Se permite enviar un pedido de forma manual. Se muestran los pedidos pendientes y se solicita seleccionar uno, obteniéndose la cantidad de peces disponibles en el tanque seleccionado para actualizar el pedido en la base de datos. Si el pedido se completa, se registra el envío y se otorga una recompensa aleatoria (comida, monedas o tanque).

borrarPedidos

- **public void borrarPedidos():**

Se eliminan todos los pedidos almacenados en la base de datos, informándose al usuario del número de pedidos eliminados y registrándose la acción.

cerrarConexion

- **public void cerrarConexion():**

Se cierra la conexión a la base de datos, liberándose los recursos asociados al objeto *DAOPedidos*.

main

- **public static void main(String[] args):**

Se inicia la simulación creando una instancia de *Simulador*, inicializándola mediante *init()*, y entrando en un ciclo interactivo en el que se muestra el menú y se procesan las opciones del usuario. Al finalizar, se cierran las conexiones, se guarda el estado del simulador y se registra el cierre de la partida.