Proyecto final: Zoológico Virtual en Java Swing

Universidad de Concepción, 2° semestre 2023

Integrantes: Guillermo Oliva Orellana
Martín Llanos Fariña
Asignatura: [503212] PROGRAMACIÓN II
Docente: Geoffrey Jean-Pierre Hecht
Fecha: 09/12/2023

Enunciado de la tarea

La variación del proyecto escogida por nuestro grupo fue la siguiente, recogida desde la página de Canvas del curso:

Tema 3: Simulador de zoo

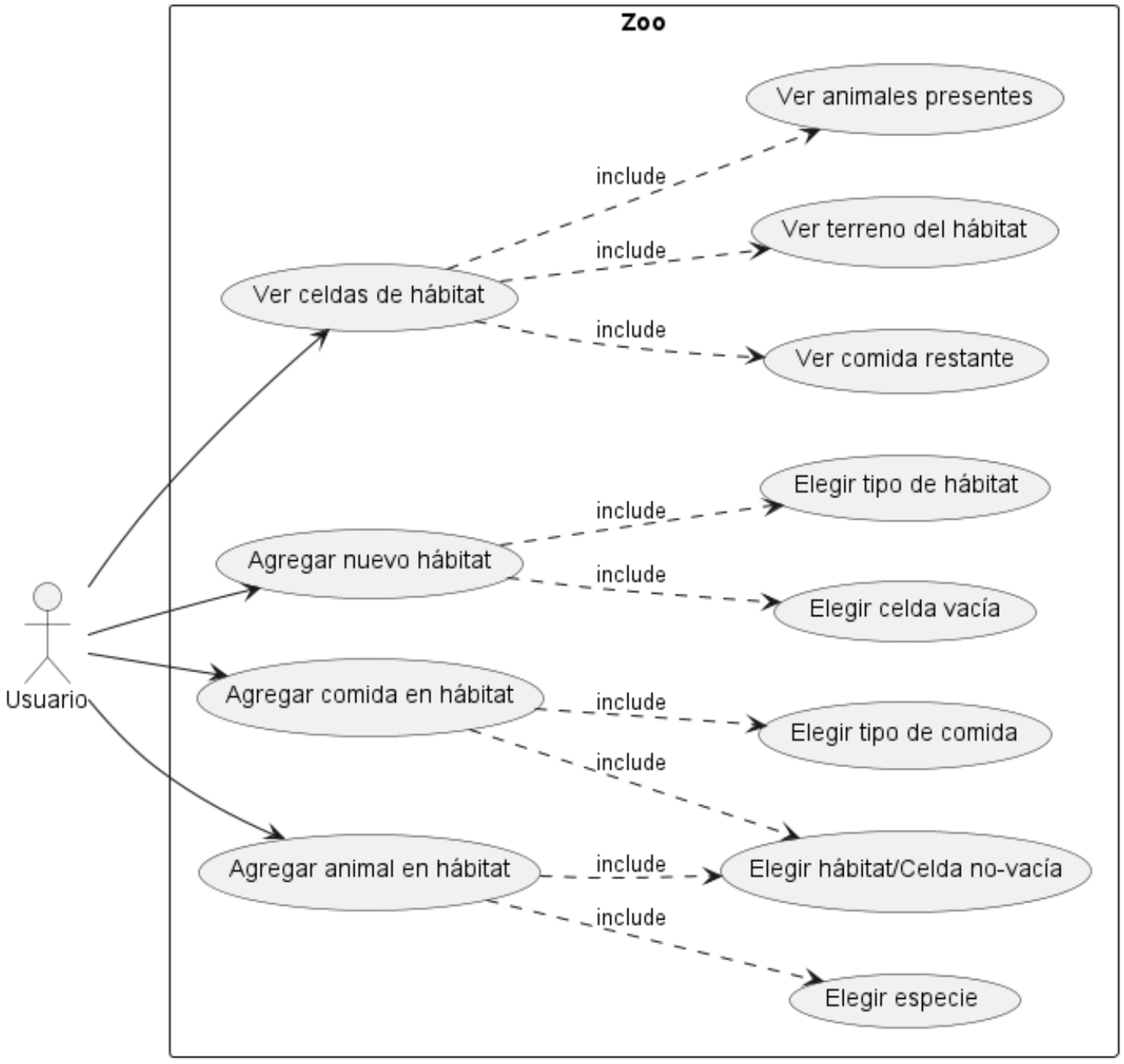
- El simulador de zoo permite a los usuarios crear y gestionar su propio zoológico virtual de manera interactiva. Los usuarios tienen la capacidad de construir diferentes hábitats para una variedad de animales.
- Los usuarios pueden seleccionar entre diferentes especies animales y colocarlas estratégicamente en sus respectivos hábitats. Cada animal tiene sus propias necesidades específicas, como alimentación, temperatura y compañeros de grupo posible.
- Los usuarios pueden colocar hábitats y animales de forma manual (desde un menú), también pueden colocar comida. Los animales (polígonos o imágenes) se mueven aleatoriamente dentro de los hábitats (mediante un temporizador) y a veces consumen comida. El usuario puede colocar comida en los hábitats.
- El software garantiza que los animales no puedan ser ubicados en hábitats inadecuados. Aparece una alerta cuando falta comida para uno de los hábitats.

Diagrama de clases UML

Este es bastante largo, por favor visite el siguiente link para ver el diagrama.

Temas como las relaciones entre clases (agregación, asociación, composición, extensión) fueron discutidos desde etapas tempranas de desarrollo, pero el desarrollo del diagrama UML empezó muy tarde, y debido al crunch time, el diagrama final quedó algo desordenado.

Diagrama de casos de uso



Patrones de diseño utilizados

Patrón: Proxy

Como su nombre lo indica, dentro del **package Controller** (en realidad, en el proyecto completo) se buscó aplicar el patrón Controller/Mediator con el objetivo de recibir actualizaciones desde el GUI, y que estas señales se transmitan al Model (backend), causando eventos como la instanciación de un nuevo objeto Animal al comunicar esta señal mediante la interfaz gráfica. De esta forma, se reduciría la comunicación directa entre interfaz gráfica y código en Modelo, tal que Model y View no "sepan" directamente toda la información de una u otra package. Además, el patrón Model-View-Controller es bastante popular en Java para proyectos de interfaz gráfica, por lo que inicialmente nos pareció muy intuitivo organizar el proyecto completo de esta forma.

Sin embargo, en la implementación final de este patrón, terminó siendo más similar a un Proxy, puesto que los tres métodos más importantes de esta clase (`nuevoHabitat`, `nuevoAnimal`, `addAlimento`) son todos `static` y son utilizados por el mismo usuario a través de la interfaz gráfica del programa. Las llamadas a estos métodos sólo ocurren cuando se comunica una señal de "añadir Animal" o "añadir Hábitat, Alimento…" a través de un clic del cursor, causando una comunicación de Interfaz sólo con un proxy del modelo encapsulado en el back end del programa al momento de crear nuevas instancias. Nos dimos cuenta de que este patrón era más apropiado durante el desarrollo, pero el nombre de la clase permaneció como ZooController.

A pesar de no haber sido la intención desde el principio, el Proxy fue crítico para la implementación del proyecto, pues necesitábamos enviar una señal tanto a View como a Model cuando ocurriese el `mouseEvent` para añadir elementos al zoológico.

Patrón: Factory

Dentro del **package Model,** se implementó un patrón "inspirado en" una Factory (pues en realidad no lo sigue al pie de la letra). Una de las clases es *ZooItemFactory,* la cual puede crear instancias nuevas de los tres objetos principales del zoológico: Hábitats, Animales y Alimentos. Nuestra motivación fue encapsular la creación de nuevos objetos que necesiten realizar otras clases del programa durante el runtime (principalmente Controller) detrás de una clase separada, que simplemente acepte como parámetro una constante representativa

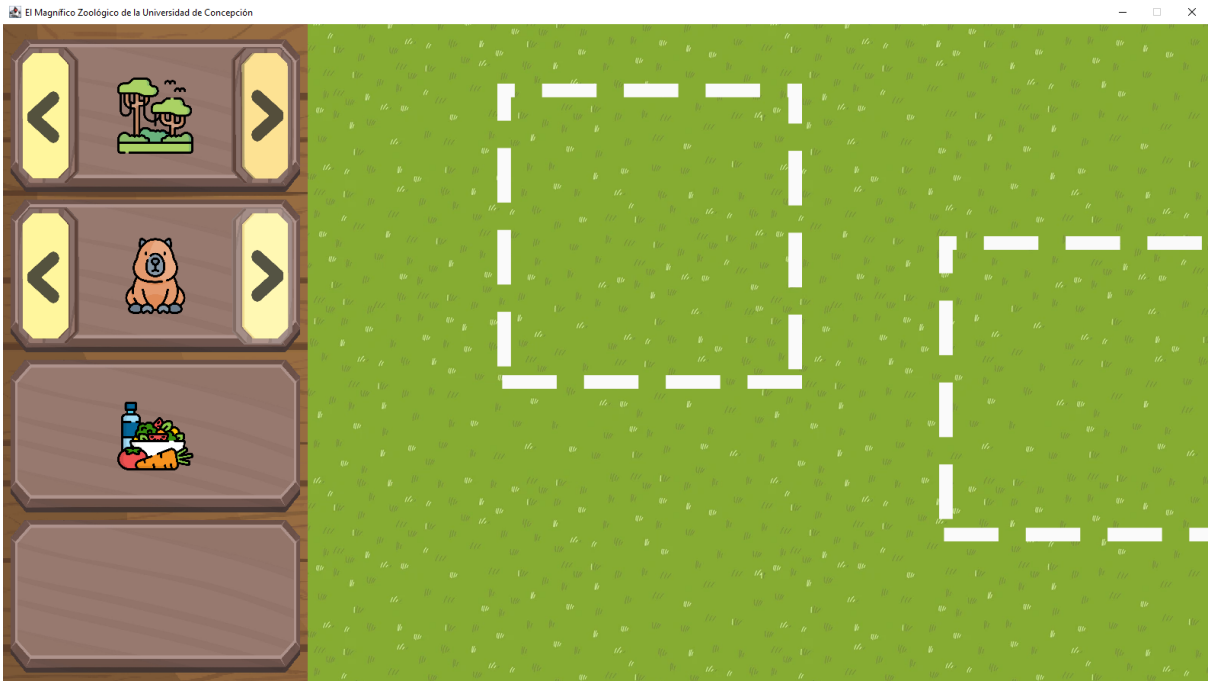
de alguno de los objetos que se desea crear (por ej., aceptar una constante AnimalEnum y devolver una nueva instancia Animal donde la especie difiere según la constante ingresada). En retrospectiva, no fue un patrón crítico para el desarrollo del proyecto, pero si fue beneficioso en términos de organización y accesibilidad de métodos para crear instancias de objetos.

Patrón: “Observer”

Dentro de **package View**, `VistaMenu` posee tres objetos que dibujan sobre un `JPanel`, pero no extienden a ninguna clase de Swing ellos mismos. Estos son `PanelSeleccionAnimal`, `PanelSeleccionComida`, y `PanelSeleccionHabitat`. Se creó un par de interfaces “inspiradas en” el patrón Observer: `ParentPanel` y `SubPanel`, que funcionan como un Observer/Subscriber y un “Updater” respectivamente. El *thought process* fue el siguiente:
Deseamos que el cursor cambie su apariencia dinámicamente según el último `PanelSeleccion` que fue presionado con el clic del mouse, es decir, que el cursor cambie a un “modo de creación” para Hábitats, Animales y Alimentos. Esto requería que los tres paneles de selección notifiquen al JFrame que re-dibuje el cursor con una textura diferente cada vez que han sido presionados, y que el cursor vuelva al estado default cuando se presione nuevamente o se haya completado la operación de creación deseada.

Debido a lo anterior, se eligió este patrón de diseño para este conjunto específico de componentes, con objetivo de reducir la abundancia de relaciones bidireccionales que se estaban presentando en el programa. En retrospectiva, se pudo haber implementado con un mejor uso de Listeners Swing, pero en su forma final cumplía con lo requerido sin problemas, por lo que se mantuvo esta implementación.

Capturas de la interfaz



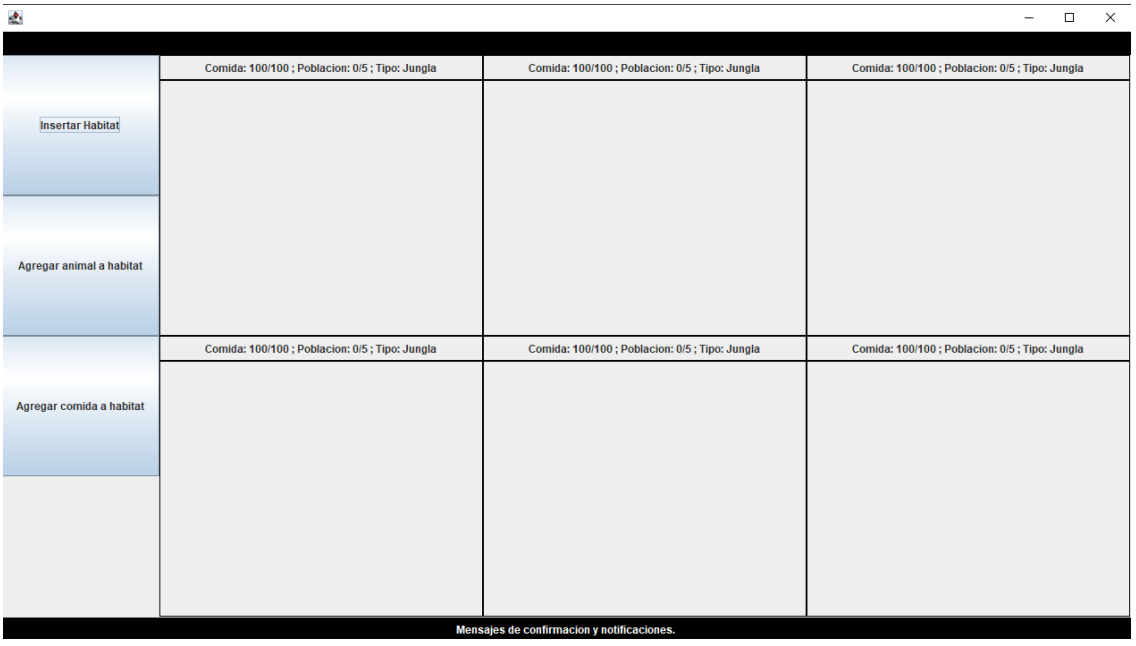
Vista inicial al ejecutar el programa.

Hacer click y drag en la textura del pasto revela más espacios para los hábitats.

Decisiones de desarrollo

Ya que la entrega pide una sección breve, se omiten los detalles más específicos de implementación.

Una de las primeras planificaciones hechas fue el **layout** de la interfaz gráfica. Nuestra entrega anterior en este curso presentaba una interfaz donde la mayoría, sino todas las interacciones, eran a través de botones que cambian gráficos en otro sector de la ventana. Considerando esto, decidimos que el modo de uso del proyecto final tuviese un carácter más interactivo e interacciones más directas con los gráficos dibujados por Swing, en particular con los sprites de hábitats y animales. Por ejemplo, al añadir animales, en vez de elegir animales desde una lista y luego seleccionar un hábitat desde otra lista, es más interactivo realizar acciones como posicionar hábitats nuevos en determinados lugares de la pantalla mediante la posición del mouse, agregar animales sobre los sprites de hábitats con clics del mouse, o posiblemente arrastrar elementos con el mouse (esto último, desafortunadamente, no se pudo realizar).



Bosquejo temprano de la interfaz.

Lo que siguió inmediatamente después fue segregar el proyecto en las dos packages mencionadas anteriormente, **Model** y **View**, con el objetivo adicional de implementar el patrón Model-View-Controller más adelante. No fue un acuerdo explícito, pero durante aproximadamente la primera mitad del transcurso de este proyecto, cada integrante del grupo trabajó por separado en una de estas dos divisiones del código, bajo la suposición de

que View solo reflejase lo que ocurra en Model durante el runtime del programa. Esto probaría ser un obstáculo más adelante, aunque no uno imposible de superar.

Se decidió manejar las excepciones producidas por casos de uso prohibidos por el enunciado (incompatibilidades de animales, hábitats, y otros casos en el contexto del zoológico) mediante clases que extienden a Exception, realizando throw desde Model cuando se intenta una acción ilegal como añadir un oso polar a un acuario, y catches principalmente en Controller.

En un momento se consideró ejecutar Model y View en Threads separados para maximizar el rendimiento del programa, pero nuestro grupo no poseía el conocimiento y experiencia suficiente para trabajar con multithreading, lo que nos llevó a realizar las actualizaciones del GUI y las actualizaciones de Model de forma **single-threaded**, usando Thread.sleep como una especie de “timer” que llame a métodos update() relevantes en Model.Hábitat y View.VistaPrincipal. A partir de esto, designamos la frecuencía de Thread.sleep a 60 veces por segundo, con el siguiente propósito: El GUI se actualiza con cada “ciclo” del UpdaterThread, con el objetivo de mantener la fluidez de las animaciones, mientras que las clases relevantes en Model (principalmente Animal y Hábitat) funcionan con un sistema de “ticks” donde cada Thread.sleep aumenta un contador hasta alcanzar un valor de “señalizar actualización” (por defecto 60, es decir, Model updates ocurren cada 1000 milisegundos aproximadamente) que luego causa una llamada a update() a todos los objetos de Model activos, así gatillando eventos como la ganancia de hambre en animales y cambios en su estado de movimiento aleatorio. A pesar de que en general las aplicaciones single-threaded tienen rendimiento mucho peor, esta decisión facilitó la comunicación entre Model y View, ya que de esta forma View puede acceder a información de Model sin la posibilidad de desincronizaciones de memoria.

Es necesario recalcar que no toda nuestra visión inicial pudo ser realizada. Esto se explica con un poco más detalle en la siguiente sección.

Problemas encontrados y aspectos a mejorar

Uno de los primeros problemas encontrados, y una consecuencia directa de nuestra delegación de trabajo dentro del grupo, fue **la dificultad al juntar Model y View** tal que funcionen de forma paralela una vez que ya ambas packages del programa se encontraban en un estado avanzado. Fue en este punto que hubo que tomar decisiones respecto a los threads y respecto a que timers conservar y cuáles no, pero aún así, no fue simple juntar ambas partes del trabajo.

Más adelante, al momento de implementar el **colocamiento manual de hábitats** en la interfaz por parte del usuario, se tuvo que descartar la idea de colocarlos en cualquier sector de VistaParque (el JPanel donde se dibuja el zoológico) debido a la dificultad que tuvimos al intentar diseñar un sistema de *placement* dinámico de este nivel. Como consecuencia, los hábitats sólo pueden colocarse en “slots” específicos dentro del panel, marcados con líneas blancas dibujadas sobre la textura de pasto. Como cierta “compensación” a esta pérdida de complejidad, se implementó un *click-and-drag* en el panel VistaParque que desplaza el área visible del parque total, revelando más “slots Hábitat” de los que podrían encajar dentro del área inicialmente visible al ejecutar el programa.

Ya en una etapa muy avanzada del desarrollo, nos dimos cuenta de un **memory leak** bastante desafortunado presente en el programa. Este no ha impedido el funcionamiento del programa en ninguna de nuestras pruebas, pero si causa un impacto significativo en el rendimiento del programa cuando hay muchos animales presentes en la pantalla. A la fecha de elaboración de este informe, no hemos podido localizar la fuente de este leak, aunque sospechamos que tiene que ver con cómo se dibujan las sprites de animales dentro de los hábitats (y, extrañamente, ¿el profiler utilizado [JProfiler] no detectó nada inusual? pero Administrador de Tareas detecta un uso de memoria demasiado alto).

En cuanto a los patrones de diseño, estamos conscientes de que los patrones presentados en este informe no están descritos con mucho detalle o su implementación podría ser mejor. Esto es debido a **la dificultad que hubo al identificar patrones de diseño apropiados para el proyecto**, o bien la dificultad envisionando una implementación “correcta” de los patrones sugeridos por nuestro referente. Creemos que este aspecto es bastante importante para mejorar, especialmente debido a la aplicabilidad que tienen los patrones de diseño a lo largo de la programación orientada a objetos en general, no solamente en Java.

Una de las funcionalidades que queríamos agregar y no tuvimos suficiente tiempo para implementar, fue **una respuesta visual a hacer mouseover por encima de los íconos de animal y hábitat** que consistía en lo siguiente: Al sostener el puntero del mouse por encima de uno de los íconos en VistaMenu un par de segundos (el panel izquierdo), mostrar un panel temporal de texto que informe al usuario de características relevantes del elemento apuntado. Por ejemplo, si se hace mouseover por encima del ícono de un Elefante en el panel selector de animales, entonces mostrar un panel que indique su rango de temperatura habitable, su apetito, y los animales con lo que puede compartir un hábitat. Toda esta información ya está presente en el código, sólo que nos hubiese gustado facilitar el acceso del usuario a ella.

La frecuencia de commits/organización del tiempo de trabajo fue apropiada, pero aún así imperfecta, pues a pesar de que el proyecto ya tenía un estado avanzado en la semana de entrega, aún así **hubo *crunch time* notable**, aunque esto también es un aspecto influido por los demás cursos de este semestre.