Universidad de Concepción
Facultad de Ingeniería
Departamento de Informática y Ciencias de la Computación

Informe de Proyecto Semestral: Simulador de Zoológico

Curso
Programación II (503212) S2-2023
Profesor
Geoffrey Hecht
Alumnos

Antonio Skorin García
2022402164
Ingeniería Civil Informática

Matías Olivas Henríquez
2022421061
Ingeniería Civil Informática

Índice

1. Introducción

El presente informe tiene la labor de documentar el trabajo realizado por los alumnos Matías Olivas Henríquez y Antonio Skorin García en el proyecto final del curso de Programación II, impartido por el profesor Geoffrey Hecht.

El proyecto a realizar se trataba de un simulador de zoológico, que le permitiera al usuario crear y mantener un entorno virtual que simulase hábitats, animales y sus diferentes requerimientos biológicos como temperatura, alimentación y cohabitación.

2. Casos de Uso e Interfaz

Al usuario se le permite la exploración y modificación del zoológico; donde la modificación concierne la creación de hábitats, la adición de animales a ellos y su alimentación a través de la interacción con un área desigada de comida; la exploración, la capacidad de observar los habitats dispuestos espacialmente en el zoo, junto a los animales que se encuentran en ellos, además de su movimiento, vida y posible muerte por inanición.

En vista de lo mencionado anteriormente, el siguiente diagrama de casos de uso representa las maneras que tiene el usuario para interactuar con el sistema.

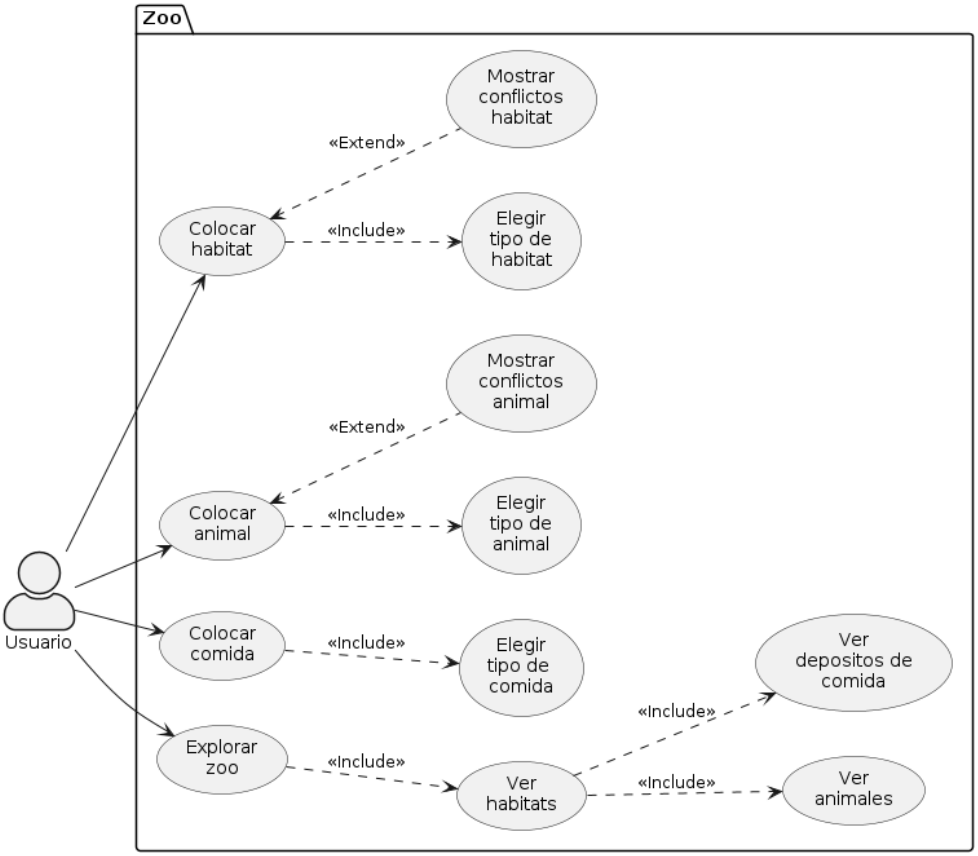
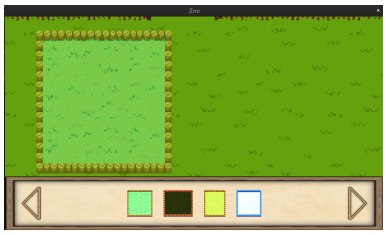


Figura 1: Diagrama UML de casos de uso, generado con *PlantUML*.

La interfaz de la aplicación contempla todo lo anterior, y es como se muestra a continuación:

(a) Inicio de la aplicación.



(b) Posicionamiento de un hábitat.

(c) Alimentando animales.

Figura 2: Interfaz de la aplicación.

3. Arquitectura de la Aplicación y Diagramas

3.1. Estructura General

La clase *App* instancia al modelo mediante la clase *EscenaZoo* y a la vista mediante la clase *VistaEscenaZoo*, luego genera un *game loop* que corre en un hebra, realizando un paso cada cierta cantidad fija de tiempo. En una iteración, el modelo y la vista se actualizan separadamente, y consecutivamente.

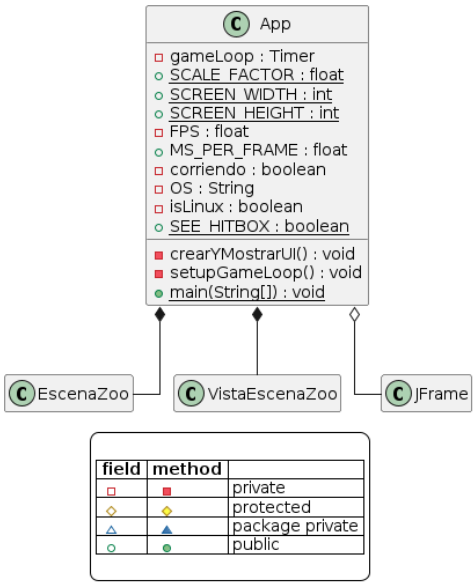


Figura 3: Diagrama UML de la estructura general del programa, generado con *PlantUML*.

3.2. Modelo

El zoológico es modelado a través de una clase *EscenaZoo*, que crea una instancia de *Zoo*, la cual está compuesta por hábitats que contienen animales, comida, y poseen las características propias de un habitat, como temperatura. Los animales, por su parte, tienen requisitos de alimentación y compatibilidad con otros animales, lo primero concierne el tipo de alimento que el animal necesita, y su capacidad para estar sin comida por un tiempo determinado.

Los animales también son capaces de encontrarse en distintos estados, a través de la clase *State*, dentro de los cuales está incluida el estado de hambre y de búsqueda de comida.

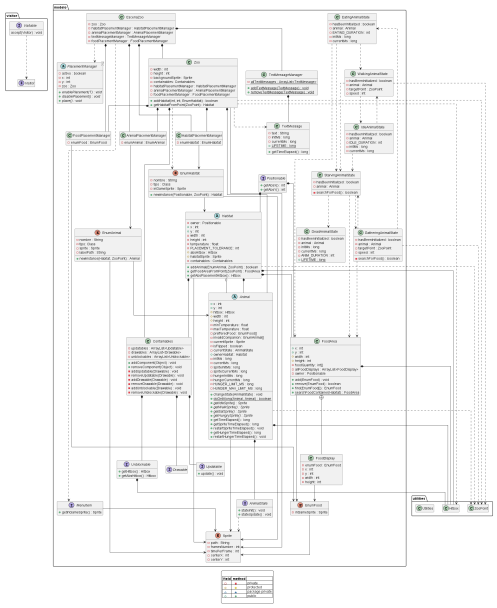


Figura 4: Diagrama UML del *package* modelo, sin relaciones de generalización, generado con *PlantUML*.

Más específicamente, tanto la clase *Animal* como la clase *Habitat* tienen la característica de ser abstractas. De ellas heredan los distintos tipos de animales y hábitats que pueden existir en el zoológico, tanto los animales como la comida y los hábitat son catalogados por sus respectivos *Enum*. Cada clase concreta tiene definida sus particularidades propias.

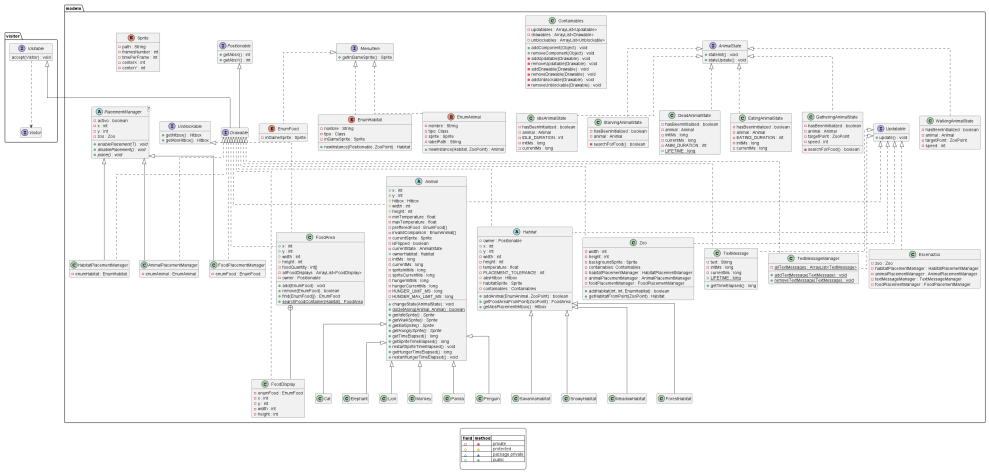


Figura 5: Diagrama UML del *package* modelo, con relaciones de generalización, generado con *PlantUML*.

3.3. Vista

La vista del zoológico se puede dividir en tres partes: la parte encargada de dibujar al zoológico en sí, una instancia de *DrawVisitor*; otra que gestiona los paneles, que le permiten al usuario añadir habitats, comida, y animales al zoológico; y las clases internas a *VistaEscenaZoo*: *ZooListener* y *PanelListener*, que se encargan del manejo de eventos.

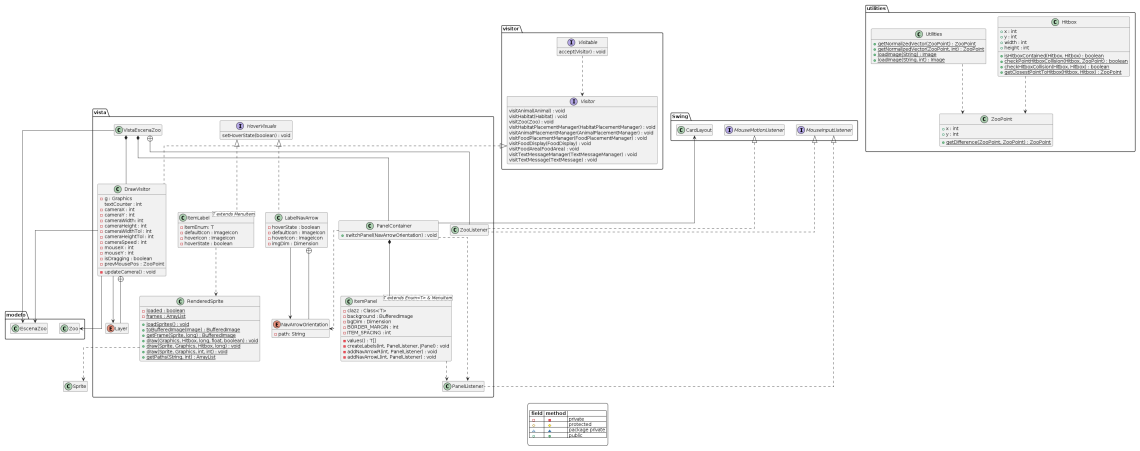


Figura 6: Diagrama UML del *package* vista y *utilities*, generado con *PlantUML*.

4. Decisiones Tomadas

Durante el transcurso del proyecto, se generaron diversas problemáticas que debían ser solucionadas, todas con distintos grados de dificultad e importancia. A continuación se detallarán las decisiones tomadas para solucionarlas, en especial las más relevantes, las que tuvieron un efecto más notorio en el estado final del proyecto, y las que más ponían a prueba lo aprendido durante el curso.

4.1. *Gameloop*

Se necesitaba una manera de estructurar la actualización y dibujado del simulador, se notó que los videojuegos en general ocupan un sistema que consiste en una estructura de control de iteración, dentro de la cual el modelo puedo ser actualizado, y la vista puede ser dibujada, por separado o en el orden que se desee, además de realizarse otras operaciones que se consideren pertinentes. Se decidió entonces, utilizar este sistema, más específicamente, se implementó mediante la creación de un *Timer*, que ejecutaría las instrucciones dentro de un ciclo cada cierta cantidad fija de tiempo. Este temporizador sería inicializado y hecho correr en el método *constructor* de la clase principal *App*.

La implementación de esta estructura fue simple, lo que permitió que el grupo de trabajo se pudiera enfocar en tareas de mayor prioridad, pero por otro lado, si se quisiera seguir trabajando sobre el simulador, el *gameloop* actual posiblemente probaría ser una limitación, ya que actualmente no permite funcionalidades que podrían resultar útiles, como:

- Intervalo de paso variable.
- Compensación por retraso entre cuadros.
- Interpolación en el dibujado.
- Diferentes hebras para separar modelo de vista, o de alguna otra funcionalidad, como el cargado de recursos.

4.2. Separación Modelo-Vista y *DrawVisitor*

En un comienzo, el simulador contaba con una lógica acoplada a la vista, lo que habría terminado siendo una desventaja seria en caso de mantenerse, afortunadamente, mediante la retroalimentación provista por el profesor, se decidió realizar una separación completa de la parte lógica del simulador y su interfaz gráfica. Esto se realizo mediante la creación de distintas clases especificadas en la sección de vista, dentro de las cuales se encuentra *DrawVisitor*, una clase que extiende a *JPanel*, y se encarga del dibujado del zoológico en sí, esto es, imagen de fondo, hábitats, animales, comida, etc... Además de implementar *Visitor*, una interfaz asociada a un patrón de diseño que cuya implementación detallada en la sección de visitor

4.3. Compatibilidades Animales

Es necesario que el programa sea capaz de discernir que animales no pueden compartir un mismo hábitat. Con este fin, nosotros decidimos que cada subclase de *Animal* debe definir en una variable *InvalidCompanion* una lista de animales con los cuales no puede convivir.

De la misma forma, el programa debe ser capaz de discernir cuales hábitats no permiten ciertos animales debido a sus temperaturas. En este caso decidimos que lo más natural es que cada subclase de *Habitat* tenga como atributo *temperature*, y que cada subclase de *Animal* defina el rango de temperaturas que permite.

Estas verificaciones se realizan al intentar colocar un animal en un hábitat. Si una de estas condiciones falla, se imprime un mensaje de texto en pantalla informando al usuario.

4.4. Manejo de Estados

Con el objetivo de crear una clase *Animal* compleja y dinámica, surge la necesidad de que un animal sea capaz de manifestar distintos comportamientos dependiendo de las circunstancias del programa.

Para ello optamos por encapsular cada comportamiento distintivo que puede tener un *Animal* en un **estado**, delegándole la responsabilidad de realizar los procesos pertinentes a esta nueva clase. Se ahonda más en como opera este patrón en la sección 5.2 de Patrones Utilizados.

4.5. Manejo de Eventos

Se entiende que en la vista del programa existen distintas maneras de interactuar con el modelo; a través de lo paneles, mediante los botones de animales, hábitats y comida; a través del mismo zoológico, arrastrando la cámara o posicionando elementos con el uso del puntero; para lograr lo anterior, existían distintas maneras de manejar los eventos:

- Permitir a cada panel, y por extensión, a cada *label* manejar sus propios eventos y la manera en la que respondían a aquellos. Esto tenía la ventaja de ser simple, pero su naturaleza descentralizada haría el código menos legible, adicionalmente, podrían ocurrir problemas de referencia, por ejemplo, en el caso de que un botón necesite acceder a un objeto que no conoce, esto habría causado métodos con demasiados parámetros o la necesidad de implementar atributos estáticos.
- Implementar una clase externa que maneje los eventos. Esta solución solucionaba el problema de la centralización de los eventos, pero aún había un problema con las referencias.
- Una clase interna a *VistaEscenaZoo* que manejara los eventos de los paneles y el zoológico. En este caso, se contaba con una manera centralizada de manejar los eventos, se tenía acceso a las referencias pertinentes al estar a un nivel alto de la jerarquía de la vista, dentro del panel principal de vista. Sin embargo, esta solución causaría métodos con varios condicionales, para evaluar el las fuentes de la interacción y casos específicos.

Finalmente se decidió por la última opción, ya que resultaba en código legible y manejable, y la problemática de los condicionales no era lo suficientemente seria en un proyecto de esta envergadura como para ser un factor decisivo.

5. Patrones Utilizados

5.1. Visitor

Visitor es un patrón de diseño de comportamiento que responde a la necesidad de exploración sistemática de elementos de un sistema, junto con la capacidad de separar la implementación de un método de los elementos sobre los que actúa. Esto obedece al principio de diseño abierto-cerrado, en efecto, el uso de este patrón permite generar código altamente extensible. La necesidad de separar el modelo de la vista, sumado a la necesidad de que nuestra solución de dibujado fuera extensible a más elementos del zoológico, llevó a la decisión de ocupar este esquema para el dibujado.

Más detalladamente, en el código del proyecto se hace poco uso de la librería de *Swing*; al menos en el sentido de que no se ocupa para el dibujado de objetos como animales, habitats, o comida; estas son partes del modelo que tienen la implementación de su dibujado en la clase *DrawVisitor*, que extiende *JPanel*. En términos más concretos, *DrawVisitor*, en su método *paintComponent* llama al método *accept*, sobre el elemento que se encuentra más arriba en la jerarquía del modelo, y este llama al método requerido de *visitor*, por ejemplo, en caso de ser un objeto *Zoo*, este haría uso del método *visitZoo* dentro de *accept*. Luego, en la implementación de este último, se haría uso de las funciones *accept* de las componentes de *Zoo*, y así, hasta llegar a todas las componentes del zoológico.

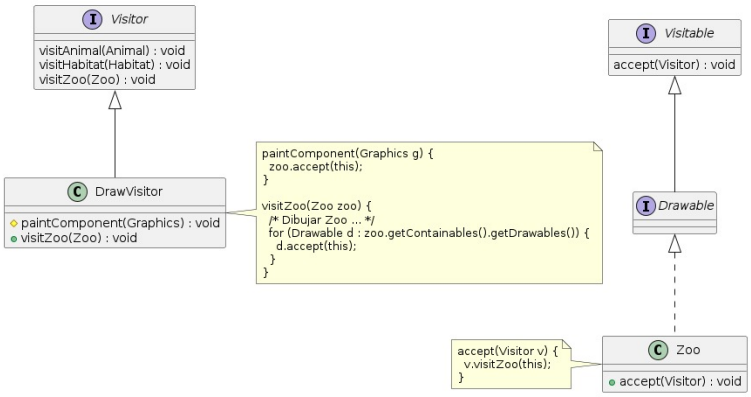


Figura 7: Diagrama UML simplificado de la implementación de *visitor*, generado con *PlantUML*.

5.2. State

State es un patrón de diseño de comportamiento que permite que un programa se comporte de maneras distintas dependiendo de su configuración actual. En este proyecto, nosotros consideramos que *State* es la solución más acorde con la problemática mencionada en la sección 4.4.

En la implementación general del patrón encontramos una clase *Contexto*, la cual tiene comportamientos distintos según el estado del programa. Luego, tenemos la interfaz *State* que describe métodos (vacíos) que tienen la función de realizar el comportamiento propio de los estados. Finalmente, tenemos clases que implementan la interfaz y especifican que realizar en los métodos de *State*, y así, determinan el comportamiento de *Contexto* en un estado determinado.

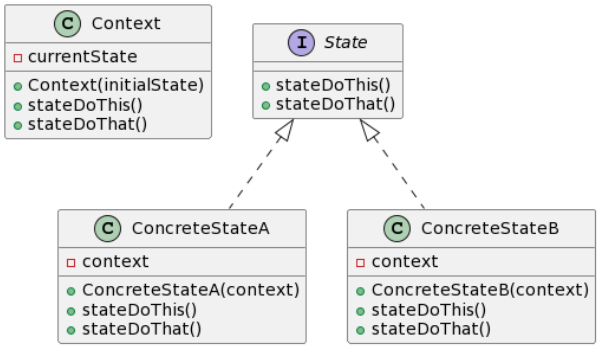


Figura 8: Diagrama del funcionamiento general del patrón *State*.

Aplicado al caso de nuestro proyecto, el contexto correspondería a la clase *Animal*, que va a transitar por diferentes estados dependiendo de distintas circunstancias, principalmente relacionadas a los ciclos de hambre de la instancia. Estos estados son todos implementaciones de una interfaz general *AnimalState*.

En cada cuadro lógico de ejecución del programa se va a ejecutar el método *update()* de *Animal*, y en este se va a llamar al método *stateUpdate()* del estado, donde se describe el comportamiento del *Animal* en dicho estado. Además, *AnimalState* tiene un método *stateInit()* que se debe ejecutar al inicializar un nuevo estado. Esto es con el motivo, entre otros, de qué un eventual estado puede llamar inmediatamente a otro.

Finalmente, son los propios estados que efectúan un cambio de estado mediante el método *changeState()* de la clase *Animal*.

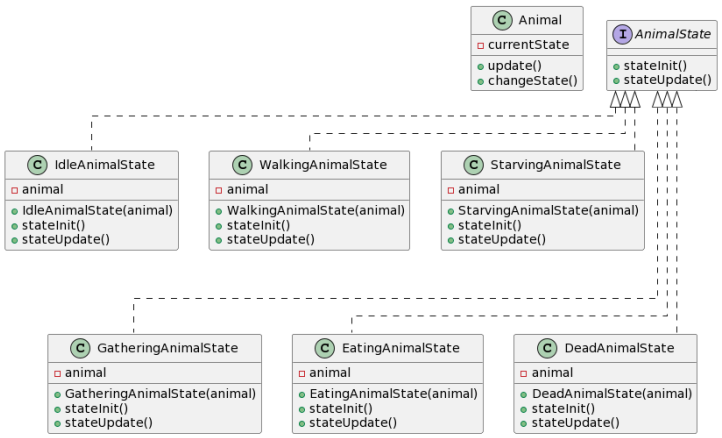


Figura 9: Diagrama de la implementación del patrón *State* en el proyecto.

6. Problemas Encontrados y Autocrítica

Durante todo el desarrollo del simulador, surgieron distintos tipos de problemáticas, la mayoría relacionadas al diseño de software. Particularmente, la falta de experiencia causa una incapacidad de ver un problema de diseño hasta que ya es demasiado tarde, ocurrió muchas veces que se planeaba e implementaba una solución a un problema, solo para después hacerse notorio que aquella solución era inefectiva, no era escalable, no contemplaba todos los casos, etc... En general, se aprendió que para escribir buen código, es necesario preguntarse en cada paso del proceso si es que lo que se está desarrollando es modular, mantenible, escalable... Con este propósito, en el curso se aprendió de los cincos de principios de diseño *SOLID*; aprender a seguir estas reglas, y más aún, entenderlas, resultó vital a medida que se avanzaba en el proyecto. De los principios de diseño *SOLID* se ignoraron muchos en el código del proyecto, en algunas partes, como en *DrawVisitor*, por ejemplo, se depende del detalle (no de la abstracción) para acceder a los componentes que pueden ser dibujados de la clase *Zoo*, se podrían haber ocupado interfaces intermedias o métodos más adecuados para evitar esto. En los manejadores de posicionamiento se hace uso del concepto de reflexión de manera inadecuada, definitivamente había una manera de solucionar esto, pero no se implementó. Los enumeradores implementados, que catalogan gran parte de los elementos distinguibles del zoológico, terminaron siendo clases con demasiadas responsabilidades, hasta el punto en el que se volvieron un obstáculo. Aún más allá, la arquitectura general del proyecto no fue pensada exhaustivamente, se experimentó con soluciones hasta que se llegó a una lo suficientemente adecuada como para ser pulida dentro de cierto margen de error.

Se puede concluir que la ingeniería de software es una disciplina sumamente complicada, pero a través de proyectos como este es que un desarrollador es capaz de adquirir la experiencia necesaria para identificar soluciones buenas de las malas.