Trabajo Práctico: Programación II

Clínica Veterinaria “Cuidamos tu mascota”

Información de la Entrega

Profesor: Gustavo Javier Klippmüller
Curso: YA-2C
Año Lectivo: 2.025
Fecha de Entrega: 27-11-2025

Integrantes del Grupo

Nombre y Apellido	Email
Artecona Lautaro	lautaroartecona@gmail.com
Imizcoz Lucas	lucasleoimizcoz@gmail.com
Lorentti Lautaro	lautaroglorentti@gmail.com
Maggiaiuolo Máximo	maximomaggiaiuolo@gmail.com
Nuñez Valentin	valentin.ezequiel.nunez@gmail.com

ÍNDICE:

Introducción

El propósito de este documento es presentar el diseño conceptual de la aplicación solicitada por la clínica veterinaria “Cuidamos tu mascota”, cuyo objetivo principal es gestionar las mascotas internadas (perros, gatos y conejos), sus fichas médicas, tratamientos y el seguimiento en tiempo real de su estado de salud. A partir del enunciado del problema, se modelan las clases principales del dominio (mascotas, doctores, dueños, fichas médicas, tratamientos, collares antipulgas, etc.) y sus relaciones, así como los flujos de interacción más relevantes mediante diagramas de secuencia. La solución propuesta se centra en aplicar patrones de diseño orientados a objetos que permitan que el sistema sea flexible, extensible y fácil de mantener: el comportamiento de las mascotas debe poder cambiar según su estado clínico, los criterios de atención de los doctores deben poder configurarse dinámicamente, los tratamientos deben combinarse en paquetes y la creación de collares debe independizarse de sus marcas concretas. De esta forma, se busca que la clínica pueda incorporar nuevos comportamientos, tratamientos o productos sin impactar en el resto de la aplicación y manteniendo un diseño claro y robusto.

Patrones de Diseño Identificados

A continuación, se detallan los patrones de diseño que se han identificado como necesarios para resolver los requisitos del problema.

Patrón 1: State

- **Problema que Resuelve:** La clínica requiere que cada Mascota responda de manera diferente a las acciones del doctor (dar de comer, dar de tomar, dar medicina) dependiendo de su estado interno: Feliz, Enfermo, Hambriento o Sediento. Por ejemplo, si la mascota está hambrienta y el doctor le da de comer, pasa a estar feliz; mientras que si está feliz y le dan de comer de más, pasa a estar enferma. Esta lógica implica múltiples combinaciones de estado y acción, que complicaría el código si se manejara con condicionales dispersos en la clase Mascota.
- **Solución Propuesta:** Se define una interfaz Estado que declara las operaciones relacionadas con las acciones del doctor (comer(), tomar(), recibirMedicacion() y recibirAccion(accion, mascota)). Para cada posible estado de la mascota se crea una implementación concreta (EstadoHambriento, EstadoSediento, EstadoEnfermo, EstadoFeliz), que encapsula el comportamiento específico frente a cada acción y es responsable de devolver el nuevo estado que corresponde (por ejemplo, de EstadoHambriento a EstadoFeliz al comer). La clase Mascota mantiene una referencia a un objeto Estado (estadoActual) y delega en él el manejo de las acciones, pudiendo cambiar dinámicamente su estado interno sin modificar el código del resto del sistema.
- **Clases Involucradas:** Mascota, Estado (interfaz), EstadoFeliz, EstadoEnfermo, EstadoHambriento, EstadoSediento

Patrón 2: Command

- **Problema que Resuelve:** El doctor puede aplicar distintas acciones sobre una mascota (dar comida, dar agua, dar medicina) y el enunciado indica que “si las acciones que hago son distintas a las enunciadas arriba, el comportamiento no es importante, puede hacer un ruido, hacer que vomite, etc.”. Es decir, se requiere un mecanismo que permita encapsular cada acción como un objeto independiente, de manera que se puedan agregar nuevas acciones o combinarlas sin modificar la lógica de la clase Doctor ni de Mascota.
- **Solución Propuesta:** Se define la interfaz Command con el método ejecutar(Mascota). Cada acción concreta del doctor se implementa como una clase que implementa esa interfaz: DarComida, DarAgua, DarMedicamento. El Doctor mantiene una referencia a un Command actual (por ejemplo, la acción elegida para esa consulta) y, al atender a la mascota, invoca command.ejecutar(mascota). De este modo, la lógica de cada acción queda encapsulada en su propia clase y es posible agregar nuevas acciones sin alterar el código del doctor ni de la mascota.]
- **Clases Involucradas:** Command (interfaz), DarComida, DarAgua, DarMedicamento, Doctor, Mascota

Patrón 3: Observer

- **Problema que Resuelve:** El sistema debe notificar automáticamente a los dueños y a los doctores cuando cambia el estado de una mascota internada. El enunciado indica que “el dueño de la mascota podrá enterarse del estado de todas sus mascotas en tiempo real” y que los doctores pueden suscribirse para hacer “seguimiento continuo” incluso cuando no están de guardia. Se necesita entonces un mecanismo de suscripción y notificación desacoplado, donde múltiples interesados reciban avisos cada vez que la mascota cambie de estado.
- **Solución Propuesta:** La clase Mascota actúa como sujeto observado, manteniendo una colección de observadores (observadores). Dichos observadores implementan la interfaz Observer/NotificarPersona, que define el método update(mascota, mensaje, estadoAnterior, estadoNuevo). Cuando la mascota cambia de estado (por ejemplo, de Enferm@ a Feliz), invoca internamente a notificarObservadores(), recorriendo la lista de observadores y llamando a update(...) en cada uno. A su vez, para desacoplar el canal de comunicación (mail, WhatsApp, etc.), se incorporan estrategias de notificación mediante la interfaz MedioNotificacion (NotificadorEmail, NotificadorWhatsapp). Así, el modelo permite que dueños y doctores se suscriban o se den de baja sin afectar la lógica de Mascota.
- **Clases Involucradas:** Mascota, Observer / NotificarPersona (interfaces), Duenio, Doctor (como observadores concretos), MedioNotificacion, NotificadorEmail, NotificadorWhatsapp

Patrón 4: Strategy + Singleton

- **Problema que Resuelve:** Los Doctores “tienen distinto criterio a la hora de atender (Realizar una operación, inyectar medicina, Hacer masajes) a las mascotas, que pueden variar dependiendo de los síntomas que presente. El sistema debe soportar que el doctor cambie dicha manera en que atiende las mascotas a demanda” y además “dichos criterios deben ser únicos y compartidos por todos los doctores de la veterinaria”. Es decir, se debe poder elegir dinámicamente qué algoritmo de atención aplicar, y cada criterio debe existir una sola vez en todo el sistema.
- **Solución Propuesta:** Strategy: Se define la interfaz Criterio con la operación atender(Mascota). Cada forma de atender se implementa como una estrategia concreta: CriterioOperar, CriterioVacunar, CriterioMasajear. La clase Doctor mantiene una referencia a un objeto Criterio y expone el método setCriterio(Criterio) para poder cambiar dinámicamente la estrategia de atención (por ejemplo, pasar de vacunar a operar según el contexto). Al atender a una mascota, el doctor delega el comportamiento a criterio.atender(mascota).

  Singleton: Dado que “dichos criterios deben ser únicos y compartidos por todos los Doctores”, cada clase de criterio se implementa con el patrón Singleton: se declara un atributo estático instancia y un método estático getInstancia() que garantiza que en toda la aplicación exista una sola instancia de CriterioOperar, CriterioVacunar y CriterioMasajear. De esta manera, todos los doctores comparten los mismos objetos de criterio y se evita crear múltiples instancias innecesarias.

- **Clases Involucradas:** Criterio (interfaz), CriterioOperar, CriterioVacunar, CriterioMasajear, Doctor

Patrón 5: Composite

- **Problema que Resuelve:** La Veterinaria debe poder facturar “tratamientos individuales” y “conjuntos de tratamientos que incluyan varios procedimientos” (por ejemplo, el paquete “Tiene un año” que agrupa vacunación, masajes, etc.). Es necesario tratar de forma uniforme tanto a un tratamiento simple como a un paquete que agrupa otros tratamientos, de manera que ambos puedan utilizarse en los mismos lugares (por ejemplo, al calcular el total de una factura o al aplicar los procedimientos sobre una mascota).
- **Solución Propuesta:** Se define la clase abstracta o interfaz Tratamiento con operaciones comunes como getPrecio(), aplicar() y esCompuesto(). Un tratamiento individual se modela mediante la clase Individual, que representa un único procedimiento (por ejemplo, vacunar) y define su propio precio. Los paquetes de tratamientos se modelan con la clase Paquete, que mantiene una colección interna de Tratamiento y permite agregar o eliminar tratamientos

(agregar(Tratamiento), eliminar(Tratamiento)). El método getPrecio() de Paquete recorre la lista interna y suma los precios de todos los tratamientos contenidos, mientras que aplicar() delega en cada uno la ejecución de su lógica. Desde el punto de vista de Factura, ambos tipos (individual y paquete) se manejan mediante la misma abstracción Tratamiento, aprovechando la estructura de árbol típica del patrón Composite.

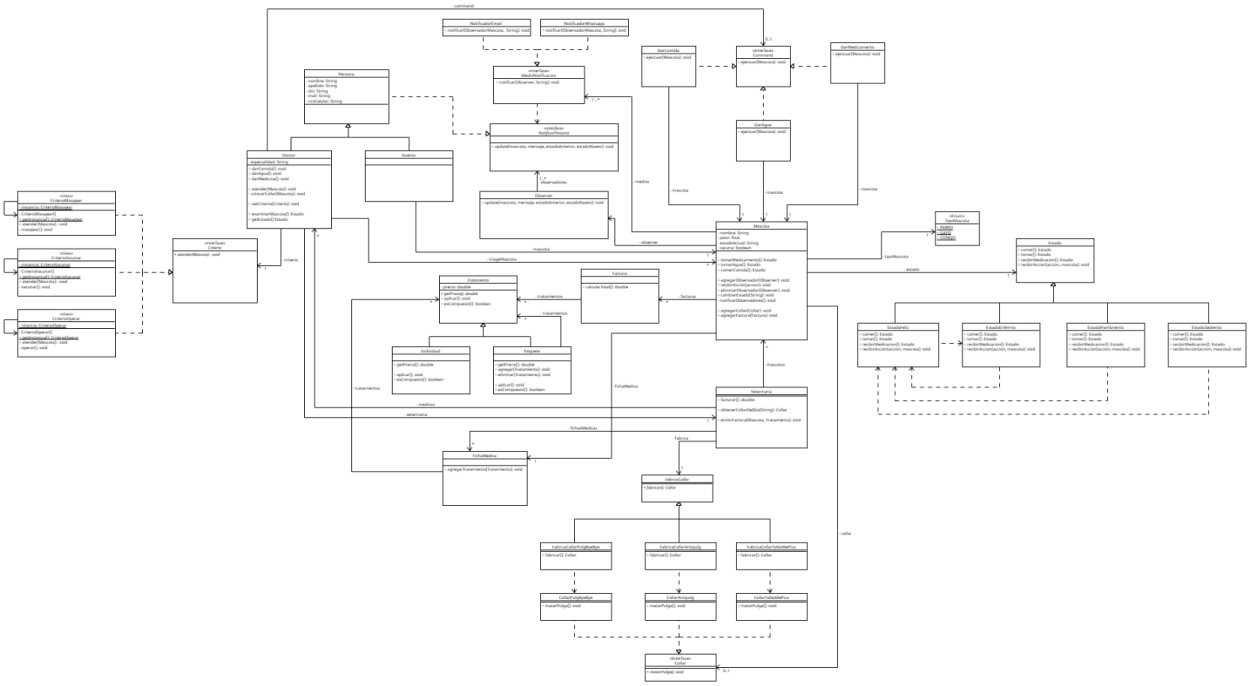
- **Clases Involucradas:** Tratamiento, Individual, Paquete, Factura, FichaMedica (como consumidora de tratamientos aplicados)

Patrón 5: Factory Method

- **Problema que Resuelve:** A todas las mascotas atendidas se les coloca un collar antipulgas, pero la marca puede variar ("Antipulg", "PulgByeBye", "YaNoMepica") y el doctor debe "colocar el collar que la veterinaria esté usando ese día sin que le importe de qué marca es". Se requiere desacoplar la creación del collar concreto de su uso, permitiendo que la clínica cambie de marca o agregue nuevas variantes sin modificar el código que aplica el collar a la mascota.
- **Solución Propuesta:** Se define la interfaz Collar con la operación matarPulga(). Para cada marca se modela una clase concreta (CollarAntipulg, CollarPulgByeBye, CollarYaNoMePica). La creación de estos objetos se delega a una jerarquía de fábricas basada en FabricaCollar, que expone el método fabricar(): Collar. Cada subclase (FabricaCollarAntipulg, FabricaCollarPulgByeBye, FabricaCollarYaNoMePica) implementa fabricar() retornando la variante de collar correspondiente. La clase Veterinaria conoce qué fábrica usar "ese día" y ofrece el método obtenerCollarDelDia(String): Collar, mientras que el Doctor únicamente invoca obtenerCollarDelDia(...) y luego colocarCollar(mascota), sin acoplarse a ninguna marca concreta. Esto facilita cambiar o extender las marcas disponibles sin impactar el resto del sistema.
- **Clases Involucradas:** Collar (interfaz), CollarAntipulg, CollarPulgByeBye, CollarYaNoMePica, FabricaCollar (clase base), FabricaCollarAntipulg, FabricaCollarPulgByeBye, FabricaCollarYaNoMePica, Veterinaria, Doctor, Mascota

Diagrama de Clases (UML)

Diagramas de clases en formato PDF para mayor legibilidad:
DIAGRAMA PDF.pdf

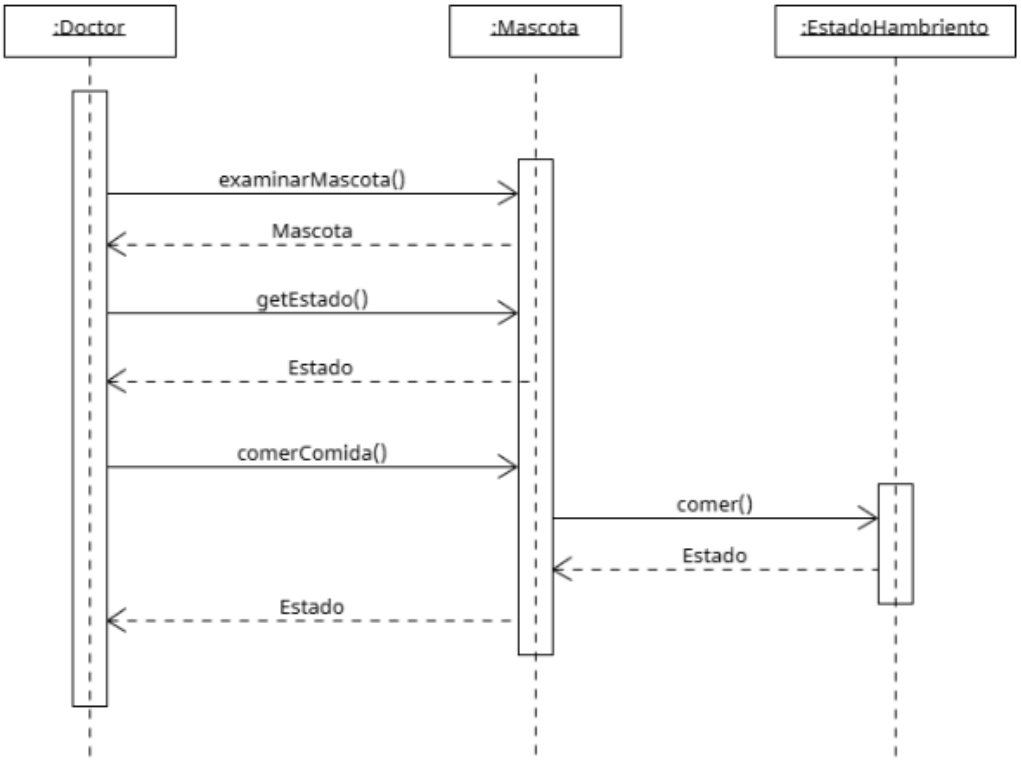


Diagramas de Secuencia

Diagramas de secuencia en formato PDF para mayor legibilidad:
SECUENCIAS PDF.pdf

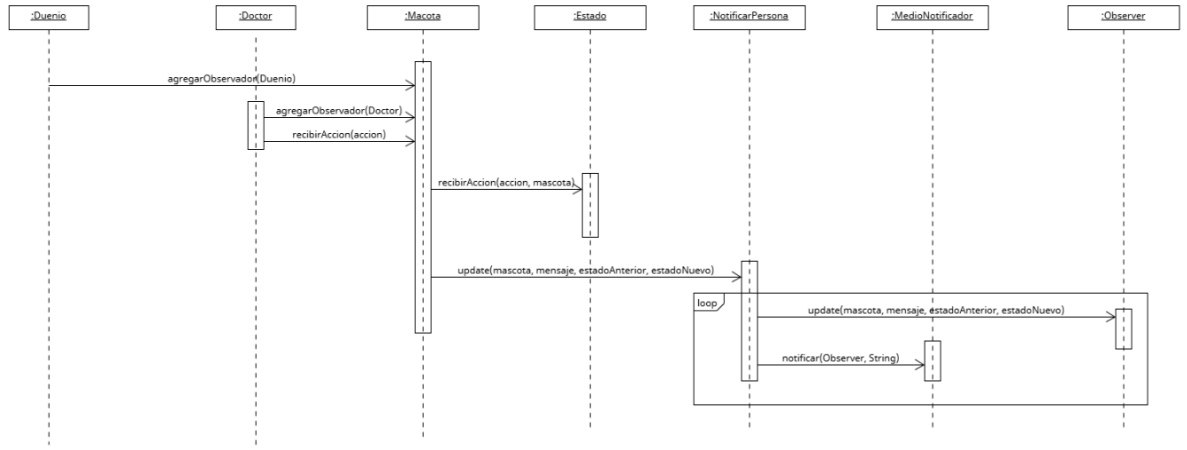
Secuencia 1: Ingreso de mascota y Cambio de Estado (Patrón State)

- **Descripción del Flujo:** Modela el escenario donde un Doctor atiende a una mascota internada. Primero la examina (examinarMascota()), consulta su estado actual a través de la mascota (getEstado()) y, por ejemplo, detecta que está hambrienta. Luego ejecuta la acción correspondiente (comerComida()), que la mascota delega en su objeto de estado (comer() en EstadoHambriento). Este estado concreto decide la transición (de Hambriento a Feliz) y actualiza el estado interno de la mascota.



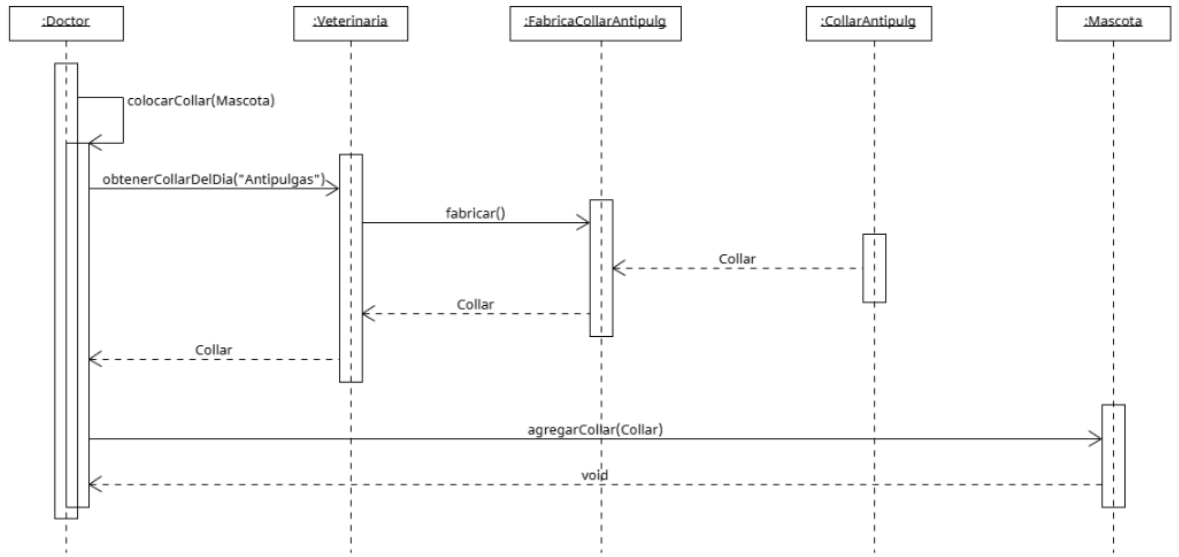
Secuencia 2: Notificación de Cambio de Estado a Observadores (Patrón Observer)

- **Descripción del Flujo:** Representa cómo se notifica automáticamente a dueños y doctores cuando cambia el estado de una mascota. Primero, el sistema registra a los interesados: la mascota recibe solicitudes para agregarObservador(Duenio) y agregarObservador(Doctor), quedando ambos suscriptos. Más adelante, cuando la mascota recibe una acción (recibirAccion(accion)) que implica un cambio de estado, el nuevo estado invoca recibirAccion(accion, mascota) y se dispara el mecanismo de notificación. La mascota recorre su lista de observadores y, dentro de un bucle, llama a update(mascota, mensaje, estadoAnterior, estadoNuevo) sobre cada uno, utilizando internamente un MedioNotificador para enviar el mensaje (por ejemplo, por mail). De esta forma, tanto el dueño como el doctor reciben en tiempo real la información del cambio de estado.



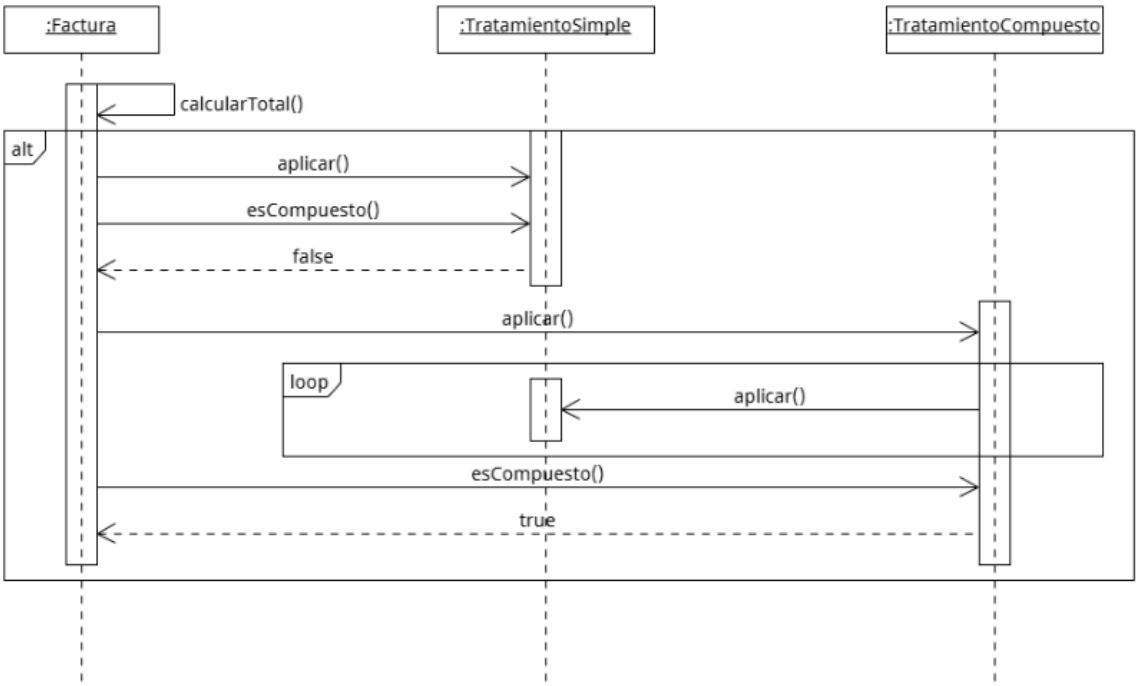
Secuencia 3: Creación y Asignación de Collar Antipulgas (Patrón Factory Method)

- **Descripción del Flujo:** Describe cómo se obtiene y coloca el collar antipulgas adecuado sin que el doctor dependa de una marca concreta. El flujo comienza cuando el Doctor solicita colocar un collar a la mascota (colocarCollar(Mascota)) y le pide a la Veterinaria el collar que se está usando ese día (obtenerCollarDelDia("Antipulgas")). La Veterinaria delega la creación en la fábrica concreta correspondiente (por ejemplo, FabricaCollarAntipulg), que ejecuta el método fabricar() y devuelve una instancia de CollarAntipulg. La Veterinaria retorna este Collar al Doctor, quien finalmente lo asocia a la mascota mediante agregarCollar(Collar). Así, la lógica de creación de los distintos collares queda encapsulada en las fábricas, mientras que Doctor y Mascota sólo trabajan con la abstracción Collar.



Secuencia 4: Facturación de Tratamientos Simples y Compuestos (Patrón Composite)

- **Descripción del Flujo:** Modela cómo la veterinaria calcula y aplica los tratamientos facturados a una mascota, ya sean individuales o paquetes. La Factura inicia el proceso invocando calcularTotal(). Para cada tratamiento asociado, llama a aplicar() y consulta esCompuesto(). Si el tratamiento es simple (esCompuesto() -> false), se aplica directamente una única vez. Si, en cambio, es un tratamiento compuesto (esCompuesto() -> true), dentro de un bucle el tratamiento compuesto llama a aplicar() sobre cada tratamiento hijo que contiene internamente (pueden ser simples o, a su vez, compuestos), acumulando el precio total. De esta forma, la factura puede tratar de manera uniforme a tratamientos simples y paquetes, recorriendo la estructura compuesta sin conocer sus detalles internos.



Justificación de Decisiones de Diseño

En esta sección, los alumnos pueden explicar decisiones adicionales que tomaron y que no están cubiertas por los patrones. Por ejemplo:

- **Relación Doctor-Veterinaria:** Se decidió modelar las relaciones de esta manera ya que el enunciado plantea que el Doctor es quien coloca el Collar a la Mascota, por

lo tanto el doctor guarda como atributo la Veterinaria donde trabaja para poder obtener el collar del día.

- **Relación Doctor-Mascota:** Se decidió que el doctor tenga una lista de Mascota para justificar el enunciado donde aclara que el Doctor hace seguimiento y puede recibir notificaciones así no esté trabajando en la Veterinaria.
- **FichaMedica - Factura:** Tomamos la decisión de diseñar las relaciones Mascota-Factura, Veterinaria-FichaMedica y Mascota-FichaMedica ya que si bien a través de la ficha médica se pueden obtener los costos, tiene sentido que la Mascota guarde las facturas con los tratamientos que se realizó. Lo mismo del lado de la Veterinaria, tiene sentido que tenga un listado de FichaMedica en su totalidad a la par de el listado de Mascota que son atendidas de guardia (que no es el historial de atenciones)

Conclusión

El desarrollo del sistema para la clínica veterinaria “Cuidamos tu mascota” nos permitió comprender la importancia de una arquitectura de software flexible, donde la correcta aplicación de patrones de diseño fue fundamental para resolver la complejidad del enunciado planteado. Al integrar soluciones como el patrón State y Observer, conseguimos modelar con naturalidad el comportamiento cambiante de las mascotas y asegurar que tanto dueños como doctores reciban notificaciones en tiempo real, todo ello sin generar un acoplamiento rígido entre las clases. De igual manera, la gestión de los tratamientos médicos y la facturación se optimizó mediante el uso de Composite, Strategy y Singleton, lo cual nos permitió administrar de forma uniforme tanto servicios individuales como paquetes completos, respetando a su vez la unicidad de los criterios clínicos compartidos por el personal médico.

Esta estrategia de diseño se vio reforzada por la implementación de Factory Method y Command, herramientas que facilitaron el desacople en la creación de insumos, como los collares antipulgas, y la encapsulación de las acciones médicas, garantizando que el sistema pueda adaptarse a nuevos productos o procedimientos sin alterar su funcionamiento base. En definitiva, este trabajo práctico no solo cumple los requerimientos operativos de la clínica, sino que establece una estructura robusta, extensible y fácil de mantener, demostrando cómo un buen diseño orientado a objetos prepara al software para evolucionar ante futuros cambios.