Universidad de Málaga

ETSI Informática

Diseño orientado a objetos de un refugio de animales

Modelado y Diseño del Software (2024–25)

Daniil Gumeniuk
Angel Bayon Pazos
Diego Sicre Cortizo
Pablo Ortega Serapio
Angel Nicolás Escaño López
Francisco Javier Jordá Garay
Janine Bernadeth Olegario Laguit

Grupo 1.1

Diciembre 2024

Índice

1 Apartado 2 **4**
1.1 Enunciado 4
1.2 Análisis de opciones de Diseño 4
1.2.1 Manejo de las Asociaciones 4
1.2.2 Manejo de Roles de los Socios 6
1.2.3 Consistencia y Gestión de Datos 7
1.2.4 Representación de Relaciones en el Sistema 8
1.3 Diagrama de Diseño 10
1.4 Consideraciones 10
1.5 Implementación del Modelo 12
1.6 Testing 12
1.7 Conclusión 12

Índice de figuras

Resumen

Esta práctica aborda el diseño e implementación de un sistema orientado a objetos para gestionar un refugio de animales utilizando Java y conceptos de diseño orientado a objetos vistos en el Tema 5.

El objetivo principal es analizar las posibles estrategias de diseño que permitan implementar este modelo, abordando desafíos como la necesidad de que un mismo socio pueda desempeñar múltiples roles simultáneamente.

A lo largo del documento se discute por qué las clases descritas inicialmente no pueden ser implementadas directamente en Java y se propone una posible solución mediante técnicas como composición, interfaces, y herencia múltiple simulada para garantizar la consistencia del sistema.

Finalmente, la solución propuesta acompañada de un diagrama de diseño que ilustra la arquitectura del sistema, muestra la reutilización de métodos, la integridad de los datos y la flexibilidad necesaria para adaptarse a los requerimientos del modelo conceptual.

1. Apartado 2

1.1. Enunciado

A menudo, los coches que la empresa de alquiler de coches pone a disposición de sus clientes se tienen que poner fuera de servicio (por reparaciones, para pasar la ITV, etc.). Queremos representar esta situación en nuestro sistema para que cuando los coches estén fuera de servicio no puedan ser alquilados aunque registraremos, si hay, un coche que lo sustituye. Es por eso que nuestro sistema tendrá que proporcionar dos funcionalidades. La funcionalidad (1) poner un coche fuera de servicio (si ya está fuera de servicio o si está en servicio pero es sustituto de algún coche que está fuera de servicio, la funcionalidad no tendrá ningún efecto). Esta funcionalidad pondrá el coche fuera de servicio y registrará la fecha hasta la cual estará fuera de servicio y, si hay, buscará y registrará también un coche sustituto (será cualquier coche del mismo modelo del coche que se pone fuera de servicio que esté asignado a la misma oficina y que esté en servicio). La funcionalidad (2) pone un coche que estaba fuera de servicio en servicio. En concreto, nos centraremos en implementar la funcionalidad (1) añadiendo la operación *takeOutOfService(backToService:date)* de la clase Car. No hace falta implementar la funcionalidad (2).

- ¿Qué patrón de diseño recomendarías para representar la información descrita (si es que recomiendas alguno)?
- Muestra el diagrama de clases resultante de añadir estas funcionalidades.
- Muestra el código Java de la operación takeOutOfService de la clase Car.

A continuación se proporcionan las soluciones para cada uno de los apartados.

1.2. Análisis de opciones de Diseño

En esta sección se exponen las diferentes opciones de diseño para la implementación del sistema de gestión de un refugio de animales, conforme al modelo de clases y operaciones proporcionado.

El sistema requiere gestionar los socios del refugio (quienes pueden desempeñar diferentes roles como voluntarios, donantes y adoptantes), así como el registro, adopción y donación de animales. Además, se deben considerar las relaciones entre las entidades (socios, animales, refugio, donaciones) y las restricciones del sistema, como la consistencia en los datos y las operaciones.

Importante mencionar que en nuestro diseño, **no hemos aplicado un único enfoque de manera exclusiva**. Hemos adoptado una combinación de estrategias dependiendo de las necesidades de cada relación dentro del sistema justificándolo de forma adecuada.

1.2.1. Manejo de las Asociaciones

a) Asociación Directa (Sin Reificación[1])

[1]La reificación es una técnica en programación orientada a objetos que se basa en convertir un concepto abstracto, como una relación, en una entidad concreta o clase.

Descripción: En este enfoque, las asociaciones entre clases se implementan directamente como atributos en las clases relacionadas.

Esta práctica es fácil de implementar porque la cantidad de clases a gestionar y el número de clases necesarias para representar las relaciones es menor. No obstante, añadir atributos adicionales a las asociaciones (como fechas en el proceso de adopción) puede traer problemas de consistencia al manejar relaciones complejas como la de un socio con múltiples roles (lo exploraremos en futuras secciones).

Ejemplo: Implementación de `Refugio` con asociación directa a `Animal`

La asociación es directa porque `Refugio` gestiona los `Animales` mediante varios `Set`, sin una clase intermedia que relacione ambas entidades. (ver Código **??**)

En nuestra implementación, hemos decidido la utilización de un `Set` para las estructuras de datos en vez de `List` en algunos casos por las razones que se detallan en 1.4

Por ejemplo, podríamos añadir la siguiente condición para evitar que un socio pertenezca varias veces a uno o varios refugios:

El uso de `Set` nos ahorra esa comprobación añadiendo que hemos asumido que **solo puede haber un refugio**, entonces esta comprobación deja de ser necesaria. Sin embargo, en colecciones donde no necesitamos hacer esta clase de comprobación, se usan `ArrayList`.

b) Reificación de la Asociación (Clase de Asociación)

Descripción: En este enfoque, las asociaciones complejas entre clases se modelan mediante clases intermedias. Por ejemplo, la clase `Adopcion` representa la relación entre un `Animal`, un `Adoptante`, y un `Voluntario`, incluyendo atributos como `fecha` de adopción para capturar detalles específicos de la relación.

Como la reificación nos permite agregar atributos y métodos específicos a las relaciones, facilita la implementación de restricciones complejas relacionadas con la asociación. Sin embargo, aumenta el número de clases y relaciones a gestionar, lo que hace el diseño más denso ya que debemos implementar y gestionar las clases de asociación, así como los métodos para acceder a las relaciones.

Nuestra Implementación: Uso de `Adopcion` como clase de asociación:

- Se representa la relación entre `Animal`, `Adoptante` y `Voluntario` mediante una clase intermedia. (ver Código **??**)
- Atributos como `fecha` añaden flexibilidad al modelo, permitiendo capturar detalles adicionales de la relación.
- Se gestionan las relaciones bidireccionales entre las entidades involucradas, asegurando consistencia en los datos.

1.2.2. Manejo de Roles de los Socios

a) Subclases Específicas para cada Rol

Descripción: Cada rol (`Voluntario`, `Donante`, `Adoptante`) se implementa como una subclase de la clase `Socio`. Esto permite encapsular los atributos y métodos específicos de cada rol dentro de su respectiva subclase.

Esto proporciona claridad al diseño, ya que cada rol está claramente representado con métodos específicos para su comportamiento. Además, permite encapsular los atributos y métodos particulares de cada tipo de socio, lo que mejora la organización y legibilidad del código. Aunque tiene una limitación significativa para manejar roles múltiples, ya que no permite que un socio asuma más de un rol sin duplicar instancias de las subclases. Esto

hace que el diseño sea rígido y menos flexible en casos donde los roles pueden cambiar dinámicamente o coexistir (volveremos a hablar de esto en los siguientes apartados).

Ejemplo: Subclases específicas para los roles

El diseño tiene implementadas las subclases `Donante`, `Adoptante`, y `Voluntario` como extensiones de la clase `Socio`.

(ver Código ??, ??, ??)

b) Uso de Composición de Roles

Descripción: En lugar de modelar cada rol como una subclase de `Socio`, este enfoque utiliza la composición para permitir que un socio tenga múltiples roles simultáneamente.

Este enfoque es mucho más flexible, ya que permite asignar múltiples roles a un socio sin necesidad de crear combinaciones de subclases. También simplifica el manejo de roles dinámicos y permite cambios en tiempo de ejecución. Puede reducir la claridad del diseño, ya que no existe una distinción explícita entre los diferentes tipos de socios. Además, requiere implementar lógica adicional para validar qué operaciones son aplicables para los roles asignados a cada socio.

Nuestra implementación: Uso de Subclases Específicas para cada Rol:

Hemos decidido no cambiar como están implementados los roles mediante subclases específicas (para este apartado) en lugar de composición. Esto se debe a que en nuestro modelo actual, los roles están claramente definidos y no se requiere que un socio tenga múltiples roles de manera simultánea. Además:

- La claridad y encapsulación que proporciona la herencia permiten manejar las responsabilidades y comportamientos específicos de cada tipo de socio.
- Aunque la composición sería más flexible, introducirá complejidad adicional innecesaria para la implementación actual.

No obstante, los requisitos del sistema cambiarán en futuros apartados pidiendo que un socio tenga múltiples roles simultáneamente. En el apartado correspondiente, se discute porque la composición sería una solución más adecuada y como se ha implementado.

1.2.3. Consistencia y Gestión de Datos

a) Encapsulación Estricta

Descripción: Este enfoque restringe el acceso directo a los atributos y métodos de las clases mediante el uso de distintas visibilidades. Para interactuar con los atributos, se proporcionan métodos controlados (`getters` y `setters`) que incluyen validaciones (mediante `asserts`) para garantizar que los datos se mantengan en un estado consistente. Facilita la incorporación de validaciones o pruebas unitarias lo que completa el comportamiento esperado del sistema.

Ejemplo en el sistema: Uso de encapsulación estricta en la clase `Refugio`:

En nuestro sistema, la clase `Refugio` utiliza atributos privados y distintas visibilidades de métodos garantizar consistencia y control en tiempo de ejecución. ver en Código **??** como distintos métodos tienen visibilidades especificas para el rol que deben cumplir. En 1.4 se explican a detalle la función de cada una de las visibilidades presentes.

b) Uso de Colecciones Inmutables

Descripción: En este enfoque, las colecciones utilizadas para representar relaciones (por ejemplo, listas o conjuntos de `Animal` en `Refugio`) son inmutables. Esto garantiza que las relaciones no puedan ser modificadas accidentalmente fuera de las clases que las gestionan.

Mejora la integridad del sistema al garantizar que las relaciones no se modifiquen de manera no controlada. Por otro lado introduce rigidez ya que no permite realizar cambios dinámicos en las relaciones sin reemplazar completamente la colección. Esto puede dificultar la gestión de operaciones como agregar o eliminar elementos.

Ejemplo de Uso de Colecciones Inmutables en la Clase `Refugio`:

En nuestro sistema, el método `getAnimalesRegistrados` de la clase `Refugio` devuelve una vista inmutable de los animales registrados. Esto asegura que los `Set` no puedan modificarse ya que el uso de `Collections.enumeration` garantiza que la colección de animales no pueda ser alterada fuera de la clase `Refugio`, manteniendo la consistencia de los datos. (ver Código **??** como ejemplo aunque se implementa esta práctica en varias clases del sistema)

Decisión Tomada: Encapsulación Controlada con Enumerations:

En nuestro diseño, optamos por una encapsulación controlada en lugar de colecciones completamente inmutables. Utilizar enumeraciones en los métodos `get` garantiza que las colecciones no se modifiquen desde fuera de las clases, preservando la integridad de los datos.

Este enfoque combina lo mejor de ambos mundos: flexibilidad para realizar cambios controlados y protección contra modificaciones accidentales en las estructuras de datos.

1.2.4. Representación de Relaciones en el Sistema

a) Relaciones Unidireccionales

Descripción: En una relación unidireccional, solo una entidad tiene conocimiento de la relación. Por ejemplo, un `Adoptante` puede conocer al `Animal` que adopta, pero el `Animal` no necesita saber nada sobre el `Adoptante`.

Estas relaciones gestionan la relación con una clase, lo que reduce la complejidad del sistema. El problema, es que limita las consultas entre clases relacionadas y puede volverse más complejo añadir funcionalidades.

b) Relaciones Bidireccionales

Descripción: En una relación bidireccional, ambas entidades conocen y mantienen referencias mutuas. Por ejemplo, cuando un `Adoptante` adopta un `Animal`, ambos se actualizan mutuamente para reflejar la relación.

Las relaciones de este estilo garantizan la consistencia de los datos, ya que ambas partes relacionadas están sincronizadas al mantener referencias mutuas explícitas. Sin embargo, como hay que tener una sincronización constante entre las todas las clases relacionadas, si tuviéramos muchas relaciones bidireccionales puede dificultar el mantenimiento por que genera un alto nivel de acoplamiento.

Ejemplo en el Sistema: Relaciones Bidireccionales en `Adopcion`

En nuestro diseño, la relación entre `Animal`, `Adoptante`, y `Voluntario` es bidireccional y se asegurando consistencia en ambas direcciones reflejando los cambios realizados en una clase en las demás involucradas. (ver Código **??** como, por ejemplo, se actualiza el estado del animal tras ser adoptado)

Decisión Tomada:

Hemos implementado relaciones bidireccionales para las asociaciones complejas del sistema, como las adopciones, ya que garantizan consistencia y sincronización entre las entidades relacionadas. Sin embargo, para relaciones más simples, como la lista de animales en un refugio, usamos relaciones unidireccionales para mantener la simplicidad.

1.3. Diagrama de Diseño

Este es un diagrama enriquecido en el que se muestran las relaciones, productos, operaciones y funciones de todas las clases.

1.4. Consideraciones

Aquí tienes el párrafo reestructurado y enumerado correctamente, manteniendo los itemize correspondientes:

1. Gestión de relaciones:

Las relaciones (1..*) que tenemos se han gestionado usando un `Set` en cada una por las siguientes razones:

- **Control de elementos repetidos:** Los `Set` usan los métodos de `hashCode` e `equals` para hacer la comprobación de la existencia de elementos en la colección. Si un nuevo elemento coincide con uno existente, no se inserta, evitando comprobaciones adicionales que haríamos con el uso de `List` mediante `asserts`. En nuestro caso, `equals` hace comprobación de la existencia de un animal mediante el tipo e ID de la instancia.
- **Complejidad algorítmica:** En los `HashSet`, la búsqueda y la inserción tienen una complejidad de $O(1)$, ya que se basan en tablas hash. Por otro lado, en una `List` (como `ArrayList`), las operaciones de búsqueda tienen una complejidad de $O(n)$, porque requiere iterar sobre los elementos que es la misma en el peor de los casos para el `HashSet` si hay muchas colisiones en la tabla hash.
- **Orden de los elementos:** En nuestra implementación, no es necesario mantener un orden específico en las colecciones. Por esta razón, el uso `Set` es más adecuado que `List` teniendo en cuenta los puntos anteriores.

2. Encapsulación y visibilidad de métodos:

En distintas clases del sistema, se puede apreciar el uso de métodos públicos, privados y protegidos siguiendo los lineamientos de la Encapsulación Directa 1.2.3. Las funcionalidad que cada visibilidad implementa en el código:

- **public** permite el acceso desde cualquier clase del programa.
- **private** solo hace accesible el método dentro de la misma clase.
- **protected** hace el método accesible a clases dentro del mismo paquete.

En nuestra implementación decidimos utilizar visibilidades `protected` con métodos que modifican las colecciones de datos mediante `add` o `remove` para evitar modificaciones indeseadas.

3. Representación de atributos de precio:

En distintas clases hacemos referencia a **liquidez** como un `float` porque al estar tratando con dinero, vamos a tener como máximo de dos decimales. Usar un `double` es malgastar memoria que no es necesaria.

4. Control de restricciones:

Controlamos las restricciones mediante el uso de `assert` los cuales permiten validar que ciertas condiciones se cumplan durante la ejecución del programa. Si una restricción no se cumple, se lanza un error en tiempo de ejecución, lo que ayuda a detectar inconsistencias.

1.5. Implementación del Modelo

1.6. Testing

Con la siguiente clase hemos comprobado el funcionamiento del sistema:

1.7. Conclusión

El diseño e implementación del código de andamiaje para el sistema se realizó siguiendo los principios fundamentales del diseño orientado a objetos, adaptados a los requerimientos específicos de este apartado. Se tomaron la decisiones de diseño adecuadas, como la gestión de asociaciones entre clases, la encapsulación de datos y la validación de restricciones con `assert`, proporcionando un modelo consistente y completo.

Una de las decisiones clave fue el uso combinado de asociaciones directas para relaciones simples y la reificación de asociaciones para relaciones más complejas junto con métodos `get` que devuelven conjuntos inmutables. Esto permitió mantener un equilibrio entre la simplicidad de las implementaciones directas, como la gestión de animales en el refugio, y la flexibilidad de las relaciones complejas, como las adopciones, donde se requieren atributos adicionales y validaciones específicas mientras protegíamos las listas de cada objeto en el sistema.

Además, la bidireccionalidad en relaciones como las adopciones, garantizó la consistencia del modelo al sincronizar automáticamente los datos entre las clases participantes.