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

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, siguiendo los conceptos de diseño orientado a objetos presentados en el Tema 5.

El objetivo principal es analizar y comparar diferentes estrategias de diseño para implementar este modelo, asi como casos especiales donde un mismo socio pueda desempeñar múltiples roles simultáneamente. Se justifica por qué las clases descritas inicialmente no pueden ser implementadas directamente en Java debido a la limitación de herencia simple. Asimismo, se propone una solución basada en composición e interfaces discutiendo la decision tomada.

Finalmente, se presenta una solución acompañada de un nuevo diagrama de diseño que muestra la arquitectura propuesta, con las relaciones implementadas, validaciones con *assert* para comprobar restricciones en el nuevo código, asi como métodos actualizados a las necesidades presentadas para el caso especial de un Socio con multiples roles

1. Apartado A

1.1. Diseño del Código de Andamiaje

Introducción

El concepto de "código de andamiaje" en el diseño orientado a objetos, particularmente en Java, hace referencia al conjunto de estructuras y métodos necesarios para implementar asociaciones entre clases, asegurando la consistencia y la integridad del sistema.

Su propósito es proporcionar un marco inicial sobre el que los desarrolladores pueden construir las funcionalidades particulares de un proyecto. Sin embargo, la línea entre "andamiaje" e "implementación completa" puede ser fina ya que estamos añadiendo nuevas funcionalidades que luego se convierten en el nuevo marco inicial para futuros cambios que vayamos a realizar en los próximos apartados. A continuación se exponen las decisiones diseño que completan el andamiaje inicial.

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])

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).

[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.

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

La asociación es directa porque `Refugio` gestiona los `Animales` mediante un `set`, sin una clase intermedia que relacione ambas entidades.(ver Código 1.4.5)

TODO: CAMBIAR EL CODE

En nuestra implementación, hemos decidido la utilización de un `Set` para las estructuras de datos en vez de `List` por varios factores:

- **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`.
- **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.
- **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.

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 `fechaAdopcion` para capturar detalles específicos de la relación.

Como la reificación nos permiten 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 1.4.6)
- 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.

Elegimos este enfoque, ya que proporciona la flexibilidad necesaria para agregar atributos y gestionar reglas de negocio específicas. La asociación directa fue descartada porque no permitiría capturar detalles adicionales, como la fecha de adopción, ni manejar eficientemente las restricciones relacionadas con el proceso de adopción.

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 1.4.1, 1.4.3, 1.4.4)

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. Cada rol se modela como una clase independiente que puede ser asociada dinámicamente a un `Socio` mediante una colección de roles.

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 de manera aislada.
- Aunque la composición sería más flexible, introducirá complejidad adicional innecesaria para los requisitos actuales del sistema.

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 visibilidad privada. Para interactuar con los atributos, se proporcionan métodos públicos controlados (`getters` y `setters`) que incluyen validaciones (mediante `asserts`) para garantizar que los datos se mantengan en un estado consistente.

Debido a esto nos aseguramos que los datos sean modificados de manera controlada y consistente. Facilita la incorporación de validaciones o pruebas unitarias lo que completa el comportamiento del esperado del sistema.Dicho esto también se requiere de implementar más métodos, como `getters`, `setters` y validaciones necesarias, lo que aumenta la cantidad de código. Además, estas validaciones podrían hacer que el diseño sea más extenso y menos directo.

Ejemplo en el sistema: Uso de encapsulación estricta en la clase `Animal`:
En nuestro sistema, la clase `Animal` utiliza atributos privados y métodos públicos controlados para garantizar consistencia y validaciones en tiempo de ejecución.Este enfoque asegura que cualquier intento de modificar el estado de un `Animal` pase por validaciones definidas en los métodos públicos. (ver Código 1.4.7)

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 las listas no puedan modificarse desde fuera de la clase. Además, 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 1.4.5 aunque se implementa en varias clases)
Decisión Tomada: Encapsulación Controlada con Enumerations:
En nuestro diseño, optamos por una encapsulación controlada en lugar de colecciones completamente inmutables. Esto se debe a que:

- Proporciona flexibilidad para realizar cambios dinámicos en las colecciones a través de métodos controlados, lo que es necesario para operaciones como agregar o eliminar animales en un refugio.
- 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.

1.2.4. Estrategias para Manejo de Adopciones y Donaciones

a) Operaciones Independientes

Descripción: En este enfoque, cada operación (como adopción, registro de animales o donaciones) se implementa de forma independiente, sin interacciones entre ellas. Cada acción tiene su propio método o flujo lógico separado.

Este diseño asegura que las operaciones están bien definidas y separadas, lo que facilita su comprensión. Además, la simplicidad del diseño permite que sea directo y fácil de implementar.Sin Embargo, puede llevar a la duplicación de código si varias operaciones comparten lógica similar (por ejemplo, validar la existencia de un animal o donante). También puede ser menos flexible, ya que cualquier cambio en una operación podría requerir modificaciones en múltiples partes del sistema.

Ejemplo: Operaciones independientes en nuestra implementación En nuestro diseño, las adopciones y donaciones se gestionan mediante clases específicas (`Adopcion` y `Donacion`), cada una con su propia lógica y atributos.

b) Reutilización de Lógica Compartida entre Operaciones

Descripción: En lugar de mantener las operaciones completamente separadas, este enfoque identifica y reutiliza lógica común entre las operaciones (como validaciones o actualizaciones de estado). Aunque no implementamos este enfoque en nuestra solución actual, sería posible centralizar las validaciones comunes mediante una clase auxiliar, como se muestra en el siguiente ejemplo.

Esta opción reduce la duplicación de código, ya que la lógica compartida se implementa una sola vez y facilita la incorporación de nuevas funcionalidades relacionadas con las operaciones existentes. Pero puede introducir una dependencia más estrecha entre las clases, lo que podría aumentar la complejidad del sistema en caso de cambios importantes.

Ejemplo Propuesto: Centralización de Validaciones Aunque nuestra implementación actual gestiona las validaciones directamente en los métodos de las clases relevantes (`Adoptante`, `Donante`). Por ejemplo, podríamos considerar una clase auxiliar para centralizarlas en el futuro:

Listing 1: Clase Auxiliar para Validaciones

```
public class Validacion {
    public static void validarEstadoAnimal(Animal animal,
       EstadoAnimal estadoEsperado) {
        assert animal.getEstadoAnimal() == estadoEsperado :
            "El estado del animal no coincide con el esperado.";
    }
}
```

En nuestra implementación actual, la validación se realiza directamente dentro de las clases:

Listing 2: Manejo de validaciones dentro del método Adoptar

```
public void adoptar(Animal a, Voluntario v) {
    assert a.getEstadoAnimal() == EstadoAnimal.DISPONIBLE :
        "El animal no esta disponible.";
    Adopcion adopcion = new Adopcion(a, this, v, new Date());
    adopciones.add(adopcion);
}
```

Decisión Tomada: Mantener las Validaciones en las Clases Relevantes

En nuestra implementación actual, las validaciones se realizan directamente en las clases donde ocurren las operaciones. Esto se alinea con la claridad y simplicidad requeridas por el sistema. Sin embargo, reconocemos que la centralización de lógica compartida podría ser útil en sistemas más complejos. FIXME: EL GETTER ES MAL EJEMPLO DE ESTO (ver los getters en Código 1.4.6 por ejemplo)

1.2.5. 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.

En nuestro sistema, todas las relaciones unidireccionales con 1 a muchos, por ejemplo: TODO: EXPANDIR ESTA IDEA

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 1.4.6 como, por ejemplo, se actualiza el estado del animal tras ser adoptado)

Decisión Tomada: Relaciones Bidireccionales

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

Figura 1: UMA

FALTA HACER Y METER EL DIAGRAMA DE DISEÑO DE NUESTRO SISTEMA.RECOMENDABLE PONER UNA MINI EXPLICACIÓN.

1.4. Implementación del Modelo

1.4.1. Clase Socio

La clase `Socio` es abstracta y representa la base para las distintas subclases: `Adoptante`, `Voluntario`, y `Donante`. Esta clase asegura que cada socio tenga un `ID` único, una fecha de registro válida y un refugio asociado.

```
public abstract class Socio {
    private int ID;
    private Date fecha;
    private final Refugio refugioAsociado;

    public Socio(int ID, Date fecha, Refugio refugioAsociado) {
        assert ID > 0 : "El ID del socio debe ser valido.";
        assert fecha != null : "La fecha de registro no puede ser
            nula.";
        assert refugioAsociado != null : "El refugio asociado no
           puede ser nulo.";
        this.ID = ID;
        this.fecha = fecha;
        this.refugioAsociado = refugioAsociado;
    }
    public int getID() {
        return ID;
    }
    public Date getDate() {
        return this.fecha;
    }
    public Refugio getRefugio() {
        return this.refugioAsociado;
    }
}
```

1.4.2. Clase Donante

La clase `Donante` extiende de `Socio` y gestiona las donaciones realizadas por un socio. Las donaciones se almacenan en un `Set` para evitar duplicados. También consideramos necesario recalcar que las donaciones son un float ya que al contar que una menor cantidad de decimales lo consideramos más eficiente que un double

```
public class Donante extends Socio {
    private Set<Donacion> donaciones;

    public Donante(int ID, Date date, Refugio r, Double cantidad)
        {
        super(ID, date, r);
        assert cantidad > 0 : "La cantidad inicial donada debe
           ser mayor a cero.";
        donaciones = new HashSet<>();
```

```
        this.donar(cantidad);
    }

    public void donar(Double cantidad) {
        assert cantidad > 0 : "La cantidad donada debe ser mayor
            a cero.";
        LocalDate fechaDonacion = LocalDate.now();
        Donacion d = new Donacion(cantidad, Date.from(
            fechaDonacion.atStartOfDay(ZoneId.systemDefault()).
            toInstant()), this);
        donaciones.add(d);
        Refugio r = super.getRefugio();
        r.setLiquidez(r.getLiquidez() + cantidad);
        r.addSocio(this);
        assert donaciones.contains(d);
    }
}
```

1.4.3. Clase Adoptante

La clase `Adoptante` extiende de `Socio` y gestiona las adopciones realizadas por un adoptante. Las adopciones se almacenan en un `Set`.

```
public class Adoptante extends Socio {
    private Set<Adopcion> adopciones;

    public Adoptante(int ID, Date date, Refugio r) {
        super(ID, date, r);
        adopciones = new HashSet<>();
    }

    public void adoptar(Animal a, Voluntario v) {
        assert !adopciones.stream().anyMatch(ad -> ad.getAnimal()
            .equals(a)) : "El adoptante ya tiene registrado este
            animal";
        v.tramitarAdopcion(a, this);
    }

    public void addAdopcion(Adopcion a) {
        adopciones.add(a);
    }
}
```

1.4.4. Clase Voluntario

La clase `Voluntario` extiende de `Socio` y gestiona los trámites de adopción realizados por un voluntario.

```
public class Voluntario extends Socio {
    Set<Adopcion> tramites;
```

```
    public Voluntario(int ID, Date date, Refugio r) {
        super(ID, date, r);
        tramites = new HashSet<>();
    }

    public void tramitarAdopcion(Animal a, Adoptante ad) {
        assert a.getEstadoAnimal() == EstadoAnimal.DISPONIBLE : "
            El animal ya esta adoptado.";
        LocalDate fechaAdopcion = LocalDate.now();
        Adopcion adopcion = new Adopcion(a, ad, this, Date.from(
            fechaAdopcion.atStartOfDay(ZoneId.systemDefault()).
            toInstant()));
        tramites.add(adopcion);
    }
}
```

1.4.5. Clase Refugio

La clase `Refugio` gestiona el conjunto de `Socios` y `Animales`. Las operaciones están centralizadas para simplificar la gestión. Igual que en Donante en esta clase el atributo liquidez también es un float

```
public class Refugio {
    private double liquidez;
    private Set<Animal> animalesRegistrados;
    private Set<Socio> socios;

    public Refugio(double liquidez) {
        assert liquidez >= 0 : "La liquidez debe ser no negativa.
            ";
        this.liquidez = liquidez;
        animalesRegistrados = new HashSet<>();
        socios = new HashSet<>();
    }

    public void addSocio(Socio s) {
        assert s != null : "El socio no puede ser nulo.";
        socios.add(s);
    }
}
```

1.4.6. Clase Donacion

La clase `Donacion` representa una donación realizada por un `Donante`. Incluye la cantidad que como anteriormente mencionamos por temas de eficiencia es un float, la fecha de la donación y el donante asociado. Las validaciones aseguran que los valores sean válidos en el momento de la creación de la instancia.

```
public class Donacion {
    private Double cantidad;
    private Date date;
    private final Donante donante;

    public Donacion(Double cantidad, Date date, Donante donante)
       {
        assert cantidad != null && cantidad > 0 : "La cantidad
           debe ser positiva.";
        assert date != null && !date.after(new Date()) : "La
           fecha no puede ser nula ni estar en el futuro.";
        assert donante != null : "El donante no puede ser nulo.";
        this.cantidad = cantidad;
        this.date = date;
        this.donante = donante;
    }

    public Double getCantidad() {
        assert cantidad != null && cantidad > 0 : "La cantidad no
            puede ser nula.";
        return cantidad;
    }

    public void setCantidad(Double cantidad) {
        this.cantidad = cantidad;
    }

    public Date getDate() {
        assert date != null : "La fecha no puede ser nula.";
        return date;
    }

    public void setDate(Date date) {
        this.date = date;
    }

    public Donante getDonante() {
        return this.donante;
    }

    @Override
    public String toString() {
        return String.format("Donacion: %.2f, %tY-%tB-%td",
           cantidad, date, date, date);
    }
}
```

1.4.7. Clase Adopcion

La clase `Adopcion` modela una adopción de un `Animal` realizada por un `Adoptante`, gestionada por un `Voluntario`. Implementa la bidireccionalidad entre estas entidades para mantener consistencia en las asociaciones.

```
public class Adopcion {
    private Date fecha;
    final private Animal animal;
    final private Adoptante adoptante;
    final private Voluntario voluntario;

    public Adopcion(Animal a, Adoptante ad, Voluntario v, Date
       fecha) {
        assert a != null : "El animal no puede ser nulo.";
        assert ad != null : "El adoptante no puede ser nulo.";
        assert v != null : "El voluntario no puede ser nulo.";
        assert a.getEstadoAnimal() == EstadoAnimal.DISPONIBLE : "
           El animal debe estar disponible para adopcion.";
        assert fecha != null && !fecha.after(new Date()) : "La
           fecha no puede ser nula ni estar en el futuro.";

        this.animal = a;
        this.adoptante = ad;
        this.voluntario = v;
        this.fecha = fecha;

        a.setEstadoAnimal(EstadoAnimal.ADOPTADO);
        ad.addAdopcion(this);
        assert Collections.list(ad.getAdopciones()).contains(this
           ) :
        "La adopcion no fue anadida correctamente al adoptante.";
        v.addTramite(this);
        assert Collections.list(v.getTramites()).contains(this) :
        "La adopcion no fue anadida correctamente al voluntario."
           ;
    }

    public Date getFecha() {
        return this.fecha;
    }

    public void setFecha(Date fecha) {
        assert fecha != null && !fecha.after(new Date()) : "La
           fecha no puede ser nula ni estar en el futuro";
        this.fecha = fecha;
    }

    public Animal getAnimal() {
        return this.animal;
```

```
    }

    public Voluntario getVoluntario() {
        return this.voluntario;
    }

    public Adoptante getAdoptante() {
        return this.adoptante;
    }

    @Override
    public String toString() {
        return String.format("Adopcion: %tY-%tB-%td, %s, %s",
           fecha, fecha, fecha, animal, adoptante);
    }
}
```

1.4.8. Clase Animal

La clase `Animal` modela a un animal registrado en el sistema. Cada animal tiene un ID único, una fecha de nacimiento, un estado actual y está asociado a un `Refugio`.

```
public class Animal {
    private int ID;
    private Date nacimiento;
    private EstadoAnimal estadoAnimal;
    final private Refugio refugio;
    private Adopcion adopcion;

    public Animal(int ID, Date nacimiento, EstadoAnimal
       estadoAnimal, Refugio refugio, Adopcion adopcion) {
        assert ID > 0 : "El ID del animal debe ser valido.";
        assert nacimiento != null : "La fecha de nacimiento no
           puede ser nula.";
        assert estadoAnimal != null : "El estado del animal debe
           estar definido.";
        assert refugio != null : "El refugio debe existir.";

        this.ID = ID;
        this.nacimiento = nacimiento;
        this.estadoAnimal = estadoAnimal;
        this.refugio = refugio;
        this.adopcion = adopcion;
    }

    public EstadoAnimal getEstadoAnimal() {
        return estadoAnimal;
    }

    public void setEstadoAnimal(EstadoAnimal estadoAnimal) {
```

```
        assert estadoAnimal != null : "El estado del animal debe
            estar definido.";
        this.estadoAnimal = estadoAnimal;
    }

    public Date getNacimiento() {
        return nacimiento;
    }

    public void setNacimiento(Date nacimiento) {
        assert nacimiento != null : "La fecha de nacimiento no
            puede ser nula";
        this.nacimiento = nacimiento;
    }

    public Refugio getRefugio() {
        return refugio;
    }

    public Adopcion getAdopcion() {
        return this.adopcion;
    }

    public void setAdopcion(Adopcion adopcion) {
        assert adopcion != null;
        this.adopcion = adopcion;
    }

    @Override
    public String toString() {
        return String.format("Animal: ID=%d, nacimiento=%tF,
            estado=%s", ID, nacimiento, estadoAnimal);
    }
}
```

1.5. 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 restricciones con `assert`, proporcionando un modelo consistente y flexible.

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 `get` con 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 entidades relacionadas.

TODO: AÑADIR COMO RESUMEN LAS DESICIONES QUE NO SE HAYANN MENCIONADO DE LO QUE DIEGO A RECOPILADO EN LLAMADA

Apartado B

Se nos pide considerar el caso en el que un mismo socio puede desempeñar múltiples roles. Por ejemplo:

- Un *Voluntario* podría también haber adoptado un animal.
- Un *Adoptante* podría decidir realizar donaciones.

Esto supone un caso que, dado nuestro modelo donde cada rol se implementa como una subclase de la clase base `Socio` y el lenguaje de programación usado, es imposible de cumplir.

Herencia simple en Java

En la Programación Orientada a Objetos con Java, no es posible establecer una herencia múltiple ya que una clase en Java solo puede heredar de una única clase base. Esto significa que una vez *Socio* se cree como instancia de una subclase específica (por ejemplo, *Voluntario*), no puede ser también instancia de otra (como *Adoptante* o *Donante*).

Dado que las clases *Voluntario*, *Adoptante* y *Donante* heredan todas de la misma clase base *Socio*, no es posible que un mismo objeto *Socio* asuma múltiples roles simultáneamente.

Impacto en la implementación

El modelo dado se basa en subclases que encapsulan el comportamiento específico de cada rol. Si intentamos que un socio tuviera múltiples roles, tendríamos que duplicar la información del mismo socio en varias instancias lo que rompería la unicidad del objeto. Por ejemplo:

- Dos objetos diferentes representarían al mismo socio, pero con roles distintos, lo que podría llevar a datos contradictorios.
- La relación entre *Socio* y *Refugio* perdería sentido al no poder asociarse al objeto ambiguo.

Próximo pasos

Implementar el caso descrito utilizando el modelo actual de subclases es técnicamente imposible debido por las razones explicadas previamente. Este problema sugiere que el modelo debe ser replanteado utilizando una estructura más flexible. Opciones disponibles serían:

- **Composición en lugar de herencia:** Donde se puede asociar roles como atributos o relaciones del objeto *Socio*.
- **Interfaces:** Permitiría que un socio implemente múltiples roles sin la necesidad de una jerarquía rígida.

A continuación, se discute el enfoque que hemos visto más adecuado para resolver el caso propuesto justificando las decisiones de diseño que conlleva.