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 A **4**

1.1 Diseño del Código de Andamiaje 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 5

1.2.3 Consistencia y Gestión de Datos 6

1.2.4 Representación de Relaciones en el Sistema 7

1.3 Diagrama de Diseño 9

1.4 Consideraciones 9

1.5 Implementación del Modelo 11

1.5.1 Clase Socio 11

1.5.2 Clase Donante 12

1.5.3 Clase Adoptante 14

1.5.4 Clase Voluntario 15

1.5.5 Clase Refugio 16

1.5.6 Clase Donacion 19

1.5.7 Clase Adopcion 21

1.5.8 Clase Animal 23

1.6 Conclusión 24

2 Apartado B **25**

2.1 Herencia simple en Java 25

2.2 Impacto en la implementación 25

2.3 Próximo pasos 25

3 Apartado C **26**

Índice de figuras

1 Diagrama 9

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 varios `Set`, sin una clase intermedia que relacione ambas entidades. (ver Código 1.5.5)

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:

```
    if(!s.getRefugio().equals(this)){
    System.out.println("El socio ya esta asociado a otro refugio.
       ");
    return;
}
```

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 1.5.7)
- 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 1.5.2, 1.5.3, 1.5.4)

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.

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 1.5.5 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 1.5.5 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 1.5.7 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

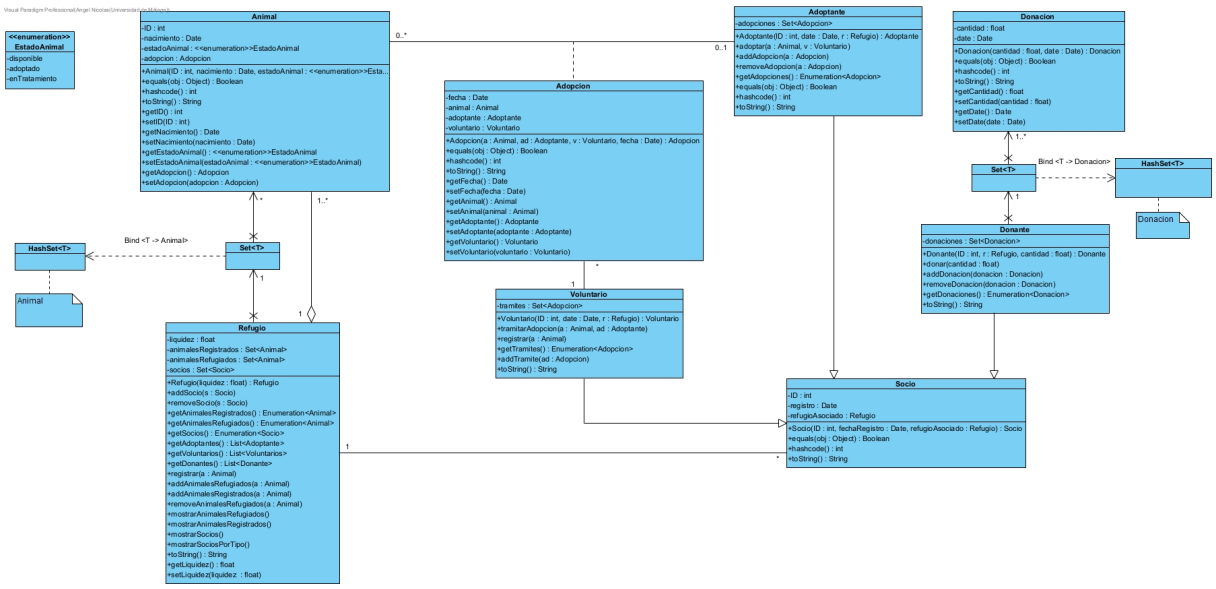

Figura 1: Diagrama

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

1.5.1. 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.

```
package sistema;

import java.util.Collections;
import java.util.Date;

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

    public Socio(int ID, Date fechaRegistro, Refugio
       refugioAsociado) {
        assert ID > 0 : "El ID del socio debe ser valido.";

        assert fechaRegistro != null : "La fecha de registro
           no puede ser nula.";

        assert refugioAsociado != null : "El refugio asociado
            no puede ser nulo.";

        this.ID = ID;
        this.registro = fechaRegistro;
        this.refugioAsociado = refugioAsociado;
        refugioAsociado.addSocio(this);

        assert refugioAsociado.getSocios().hasMoreElements()
           && Collections.list(refugioAsociado.getSocios()).
           contains(this);
    }

    public int getID() {
        return this.ID;
    }

    private void setID(int ID) {
        assert ID > 0;

        assert Collections.list(this.getRefugio().getSocios()
           ).stream().noneMatch((s) -> {
            return s.getID() == ID;
        }) : "El ID ya existe en el refugio";
```

```
        this.ID = ID;
    }

    public Date getRegistro() {
        return this.registro;
    }

    public void setRegistro(Date fechaRegistro) {
        assert fechaRegistro != null;

        this.registro = fechaRegistro;
    }

    public Refugio getRefugio() {
        return this.refugioAsociado;
    }

    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        } else if (obj instanceof Socio) {
            Socio socio = (Socio)obj;
            return this.ID == socio.ID;
        } else {
            return false;
        }
    }

    public int hashCode() {
        return Integer.hashCode(this.ID);
    }
}
```

1.5.2. Clase Donante

La clase `Donante` extiende de `Socio` y gestiona las donaciones realizadas por un socio. Las donaciones se almacenan en un `HashSet` para asegurar que no hayan elementos repetidos y por eficiencia. (cómo se explicó en 1.4) Por otro lado, hemos decidido implementar al crear un nuevo objeto Donante en el sistema, el constructor llamará directamente al método donar ya que es una condición necesaria para ser donante. Las donaciones que sean modificadas son tratadas desde la clase Donación y el `HashSet donaciones` se actualizará.

```
package sistema;

import java.time.LocalDate;
import java.time.ZoneId;
import java.util.*;
```

```
public class Donante extends Socio{
    private Set<Donacion> donaciones;
    public Donante(int ID, Date date,Refugio r, float
       cantidad) {
        super(ID,date,r);
        assert cantidad > 0 : "La cantidad inicial donada
           debe ser mayor a cero.";
        donaciones = new HashSet<>();
        donar(cantidad);
    }

    public void donar(float cantidad){
        assert cantidad > 0 : "La cantidad donada debe ser
           mayor a cero.";
        assert Collections.list(this.getRefugio().getSocios()
           ).contains(this): "El socio debe ser donante antes
           de poder donar";
        Donacion d = new Donacion(cantidad, Date.from(
           fechaDonacion.atStartOfDay(ZoneId.systemDefault()).
           toInstant()));
        addDonacion(d);
        Refugio r = super.getRefugio();
        r.setLiquidez(r.getLiquidez() + cantidad);
        assert donaciones.contains(d);
    }

    protected void addDonacion(Donacion donacion){
        assert donacion != null: "La donacion no puede ser
           nula";
        donaciones.add(donacion);
    }

    protected void removeDonacion(Donacion donacion){
        assert donacion != null : "La donacion no puede ser
           nula.";
        if (donaciones.contains(donacion) && donaciones.size
           () > 1) {
            donaciones.remove(donacion);
        } else if (donaciones.contains(donacion) &&
           donaciones.size() == 1) {
            System.out.println("Todo donante debe tener al
               menos una donacion, estas intentando eliminar
               la unica donacion asociada a este socio donante
               ");
        } else {
            System.out.println("Este socio no ha realizado la
                donacion que intentas eliminar");
        }
    }
```

```
    public Enumeration<Donacion> getDonaciones(){
        return Collections.enumeration(this.donaciones);
    }

    @Override
    public String toString() {
        return "Donante " + super.getID();
    }
}
```

1.5.3. Clase Adoptante

La clase `Adoptante` extiende de `Socio` y simula las adopciones realizadas por un adoptante. Las adopciones se almacenan en un `HashSet` para evitar adopciones duplicadas.

```
package sistema;

import java.util.*;

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 a.getEstadoAnimal() == EstadoAnimal.DISPONIBLE
           : "El animal ya ha sido adoptado";
        Refugio refugioDelVoluntario = v.getRefugio();
        a.setEstadoAnimal(EstadoAnimal.ADOPTADO);
        refugioDelVoluntario.removeAnimalesRefugiados(a);
        v.tramitarAdopcion(a, this);
    }

    protected void addAdopcion(Adopcion a){
        this.adopciones.add(a);
    }

    protected void removeAdopcion(Adopcion a){
        if (adopciones.contains(a)) adopciones.remove(a);
        else System.out.println("Este animal ya no esta
           asociado al adoptante");
    }
    public Enumeration<Adopcion> getAdopciones(){
        return Collections.enumeration(adopciones);
    }
    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
```

```
        if(obj instanceof Adoptante ){
            Adoptante adoptante = (Adoptante) obj;
            return adoptante.getID() == this.getID();
        }
        return false;
    }
    @Override
    public int hashCode() {
        return Integer.hashCode(this.getID());
    }

    @Override
    public String toString() {
        return "Adoptante " + super.getID();
    }
}
```

1.5.4. Clase Voluntario

La clase `Voluntario` extiende de `Socio` y gestiona los trámites de adopción realizados por un voluntario. Los voluntarios se almacenan en un `HashSet`, evitando así voluntarios duplicados.

```
package sistema;

import java.time.LocalDate;
import java.time.ZoneId;
import java.util.*;

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 ad != null : "El adoptante no puede ser nulo."
            ;
        LocalDate fechaAdopcion = LocalDate.now();
        Adopcion adopcion = new Adopcion(a, ad, this, Date.
            from(fechaAdopcion.atStartOfDay(ZoneId.
            systemDefault()).toInstant()));
        addTramite(adopcion);
        ad.addAdopcion(adopcion);
    }
    public void registrar(Animal a){
        Refugio r = super.getRefugio();
        assert r != null : "El refugio asociado no puede ser
            nulo.";
```

```
            assert a != null : "El animal no puede ser nulo.";
            a.setEstadoAnimal(EstadoAnimal.DISPONIBLE);
            r.addAnimalesRefugiados(a);

        }
        public Enumeration<Adopcion> getTramites(){
            return Collections.enumeration(tramites);
        }
        protected void addTramite(Adopcion ad){
            assert ad != null : "El tramite de adopcion no puede
               ser nulo.";
            tramites.add(ad);

        }
        protected void removeTramite(Adopcion ad){
            assert  ad != null: "El tramite de adopcion no puede
               ser nulo.";
            tramites.remove(ad);
        }


        @Override
        public String toString() {
            return "Voluntario " + super.getID();
        }
    }
```

1.5.5. Clase Refugio

La clase `Refugio` gestiona el conjunto de `Socios` y `Animales`.
liquidez está declarado como un `float` por las razones que se exponen en 1.4.

```
package sistema;
import java.util.*;

public class Refugio {
    private float liquidez;
    private Set<Animal> animalesRegistrados;
    private Set<Animal> animalesRefugiados;
    private Set<Socio> socios;

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

```
public float getLiquidez()
    return liquidez;
}
public void setLiquidez(float liquidez) {
    assert liquidez >= 0 : "La liquidez debe ser no negativa"
       ;
    this.liquidez = liquidez;
}
protected void addSocio(Socio s) {
    assert s != null : "El socio no puede ser nulo.";
    if(socios.contains(s)) {
        System.out.println("El socio ya esta registrado.");
        return;
    }
    socios.add(s);
}
protected void removeSocio(Socio s) {
    assert s != null : "El socio no puede ser nulo.";
    if (socios.contains(s)) {
        socios.remove(s);
    } else {
        System.out.println("Este socio no esta registrado en
           el refugio.");
    }
}

public Enumeration<Animal> getAnimalesRegistrados() {
    return Collections.enumeration(animalesRegistrados);
}
public Enumeration<Animal> getAnimalesRefugiados() {
    return Collections.enumeration(animalesRefugiados);
}
public Enumeration<Socio> getSocios() {
    return Collections.enumeration(socios);
}

public List<Adoptante> getAdoptantes() {
    List<Adoptante> adoptantes = new ArrayList<>();
    for (Socio s : socios) {
        if (s instanceof Adoptante) {
            adoptantes.add((Adoptante) s);
        }
    }
    return adoptantes;
}
public List<Voluntario> getVoluntarios() {
    List<Voluntario> voluntarios = new ArrayList<>();
    for (Socio s : socios) {
        if (s instanceof Voluntario) {
```

```
                voluntarios.add((Voluntario) s);
            }
        }
        return voluntarios;
    }
    public List<Donante> getDonantes() {
        List<Donante> donantes = new ArrayList<>();
        for (Socio s : socios) {
            if (s instanceof Donante) {
                donantes.add((Donante) s);
            }
        }
        return donantes;
    }
    public void registrar(Animal a){
        this.addAnimalesRegistrados(a);
    }
    protected void addAnimalesRefugiados(Animal a){
        assert a != null : "El animal no puede ser nulo.";
        if(!animalesRefugiados.contains(a)){
            animalesRefugiados.add(a);
            this.addAnimalesRegistrados(a);
        } else System.out.println("Este animal ya esta en el
           refugio.");
    }
    private void addAnimalesRegistrados(Animal a){
        assert a != null : "El animal no puede ser nulo.";
        if (!animalesRegistrados.contains(a)) {
            animalesRegistrados.add(a);
        } else {
            System.out.println("El animal ya esta registrado.");
        }
    }

    protected void removeAnimalesRefugiados(Animal a){
        assert a != null : "El animal no puede ser nulo.";
        if (animalesRefugiados.contains(a)) {
            animalesRefugiados.remove(a);
        } else {
            System.out.println("El animal no se encuentra en este
                Refugio.");
        }
    }
    protected void removeAnimalesRegistrados(Animal a){
        assert a != null : "El animal no puede ser nulo.";
        if (animalesRegistrados.contains(a) &&
           animalesRegistrados.size() > 1) {
            animalesRegistrados.remove(a);
        } else if (animalesRegistrados.contains(a) &&
```

```
            animalesRegistrados.size() == 1) {
             System.out.println("Todo refugio debe tener al menos
                un animal registrado, estas intentando eliminar el
                unico animal existente.");
        } else {
             System.out.println("El animal no se encuentra en este
                 Refugio.");
        }
    }

    public void mostrarAnimalesRefugiados(){
        System.out.println(animalesRefugiados.toString());
    }
    public void mostrarAnimalesRegistrados(){
        System.out.println(animalesRegistrados.toString());
    }
    public void mostrarSocios() {
        for (Socio s : socios) {
            System.out.println(s);
        }
    }
    public void mostrarSociosPorTipo() {
        System.out.println("Adoptantes: " + getAdoptantes());
        System.out.println("Voluntarios: " + getVoluntarios());
        System.out.println("Donantes: " + getDonantes());
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append("Animales Registrados: ").append(
           animalesRegistrados).append("\n");
        sb.append("Animales Refugiados: ").append(
           animalesRefugiados).append("\n");
        sb.append("Socios: ").append(socios).append("\n");
        sb.append("Liquidez: ").append(liquidez);
        return sb.toString();
    }
}
```

1.5.6. Clase Donacion

La clase `Donacion` representa una donación realizada por un `Donante`. Incluye la cantidad que como anteriormente mencionamos en 1.4 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.

```
package sistema;
```

```java
import java.util.Date;
import java.util.Objects;

public class Donacion {
    private float cantidad;
    private Date date;

    public Donacion(float cantidad, Date date) {
        assert cantidad > 0.0F : "La cantidad debe ser
           positiva.";

        assert date != null && !date.after(new Date()) : "La
           fecha no puede ser nula ni estar en el futuro.";

        this.cantidad = cantidad;
        this.date = date;
    }

    public float getCantidad() {
        assert this.cantidad > 0.0F : "La cantidad no puede
           ser nula.";

        return this.cantidad;
    }

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

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

        return this.date;
    }

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

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

    public boolean equals(Object o) {
        if (this == o) {
            return true;
```

```
        } else if (o != null && this.getClass() == o.getClass
            ()) {
            Donacion donacion = (Donacion)o;
            return Float.compare(this.cantidad, donacion.
                cantidad) == 0 && Objects.equals(this.date,
                donacion.date);
        } else {
            return false;
        }
    }

    public int hashCode() {
        return Objects.hash(new Object[]{this.cantidad, this.
            date});
    }
}
```

1.5.7. 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.

```
package sistema;

import java.util.Date;

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

```
}

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;
}

public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    } else if (!(obj instanceof Adopcion)) {
        return false;
    } else {
        Adopcion adopcion = (Adopcion)obj;
        boolean ok = this.adoptante.equals(adopcion.
           adoptante) && this.animal.equals(adopcion.
           animal);
        return ok;
    }
}

public int hashCode() {
    return this.adoptante.hashCode() + this.animal.
       hashCode();
}

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

```
}
```

1.5.8. 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`.

```
package sistema;
import java.util.Date;

public class Animal {
    private int ID;
    private Date nacimiento;
    private EstadoAnimal estadoAnimal;
    private Adopcion adopcion;

    public Animal(int ID, Date nacimiento, EstadoAnimal
       estadoAnimal) {
        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.";

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

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

```
        this.adopcion = adopcion;
    }
    public int getID() {
        return ID;
    }

    @Override
    public boolean equals(Object obj) {
        if( this == obj ) return true;
        if(obj instanceof Animal ){
            Animal animal = (Animal) obj;
            return this.ID == animal.ID;
        }
        return false;
    }
    @Override
    public int hashCode() {
        return Integer.hashCode(ID);
    }
    @Override
    public String toString() {
        return String.format("Animal: ID=%d, nacimiento=%tF,
           estado=%s", ID, nacimiento, estadoAnimal);
    }
}
```

1.6. 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.

Apartado B

Hemos considerado diferentes opciones para poder implementar la opción de que un socio que ya tiene un rol pueda realizar las acciones de otro. Entre estas opciones tenemos:

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.

Apartado C

Se nos pide considerar el caso en el que un mismo socio puede desempeñar múltiples roles. Tras discutir diferentes opciones, hemos llegado a las siguientes ideas:

- **Composición en lugar de herencia:** Facilita la reutilización al delegar la lógica común a clases auxiliares, pero puede aumentar el código necesario para configurar relaciones entre las clases.
- **Interfaces:** Permite a las clases hijas implementar acciones específicas de forma flexible, pero puede generar complejidad si hay demasiadas interfaces, ya que cada acción requiere una definición separada.
- **Rediseño del modelo con una clase Socio más rica:** Centraliza el comportamiento en la clase base, simplificando el diseño y evitando la duplicación de lógica, pero puede hacer que la clase base sea demasiado compleja.
- **Uso de patrones de diseño como el Singleton:** Permite compartir una única instancia de una clase para gestionar lógica o estado común entre las clases hijas, asegurando acceso centralizado y consistente. Aunque reduce la duplicación de lógica, puede introducir un acoplamiento fuerte entre las clases.

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