DSI UTN FRC

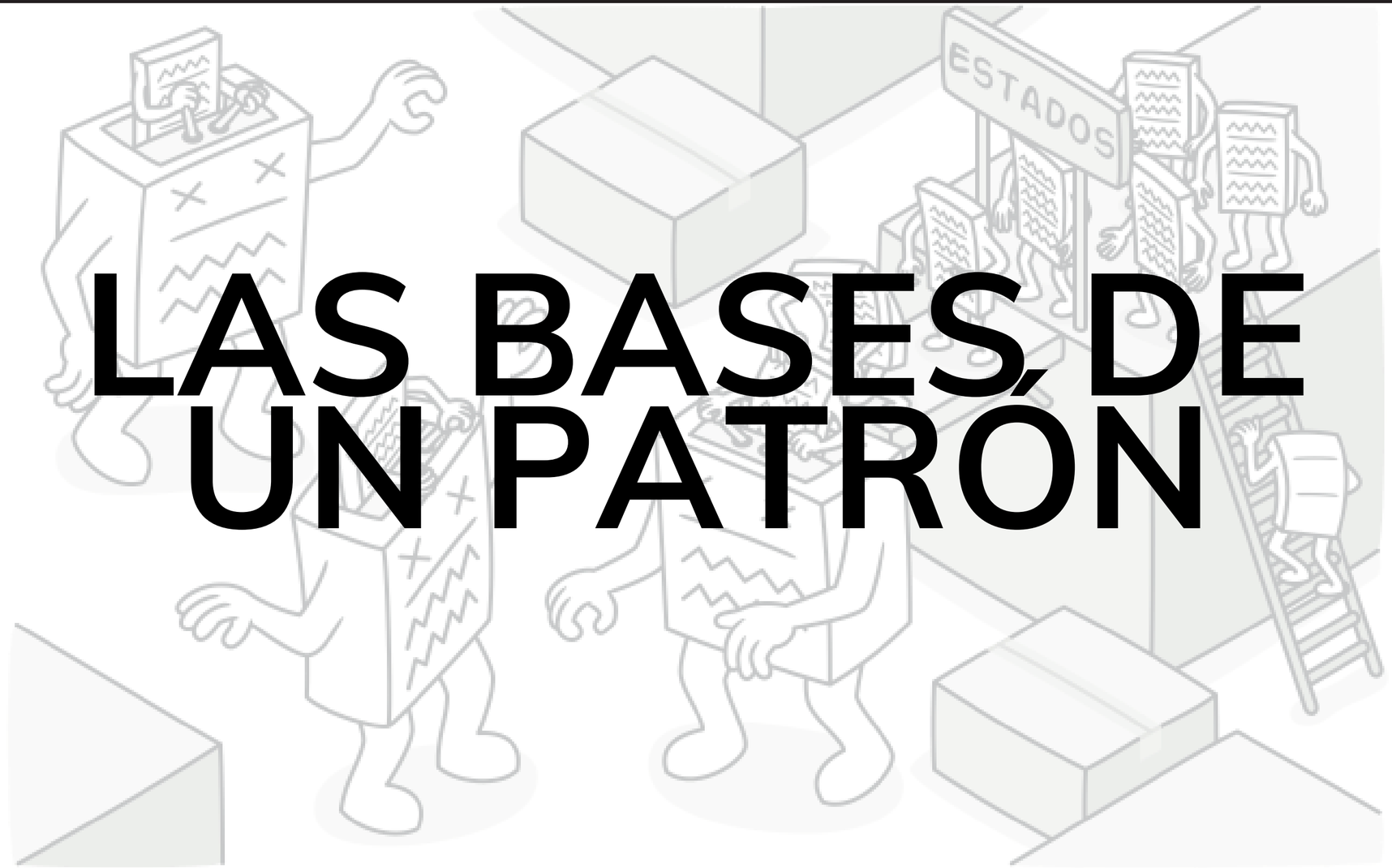
FRIGORÍFICO

Patrones de Diseño

Patrón State

Agenda

1. Patrón State: las bases
2. Caso de análisis
3. ¿Y el código?

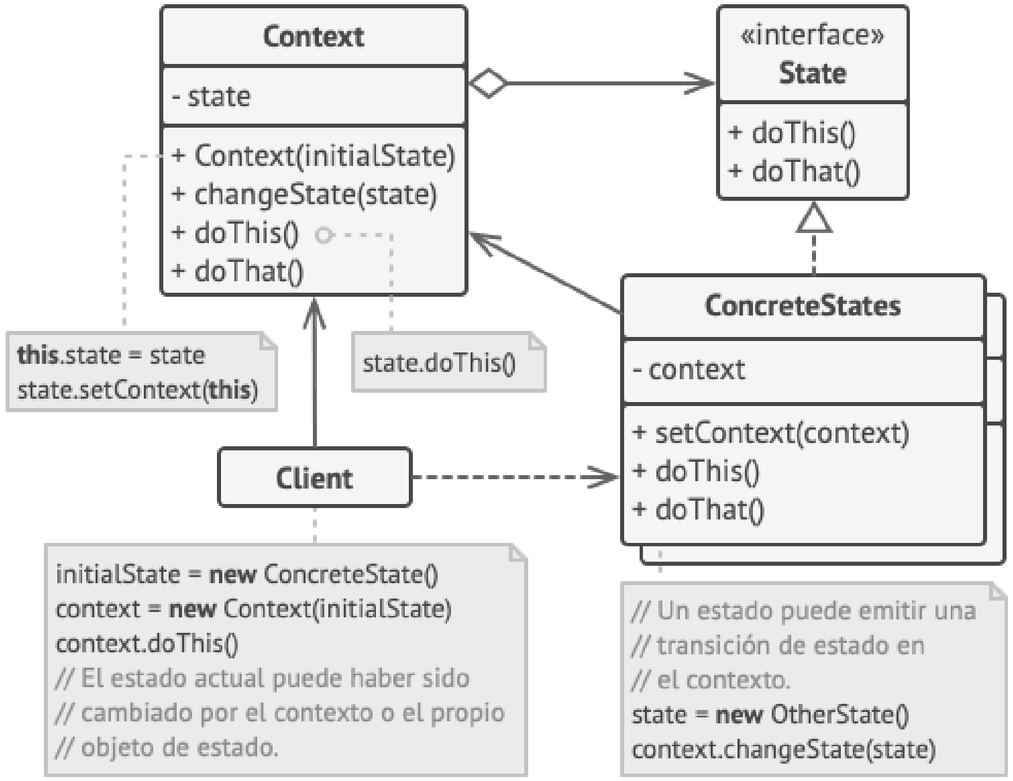


LAS BASES DE UN PATRÓN

Estructura

1 La clase **Contexto** almacena una referencia a uno de los objetos de estado concreto y le delega todo el trabajo específico del estado. El contexto se comunica con el objeto de estado a través de la interfaz de estado. El contexto expone un modificador (*setter*) para pasarle un nuevo objeto de estado.

2 La interfaz **Estado** declara los métodos específicos del estado. Estos métodos deben tener sentido para todos los estados concretos, porque no querrás que uno de tus estados tenga métodos inútiles que nunca son invocados.



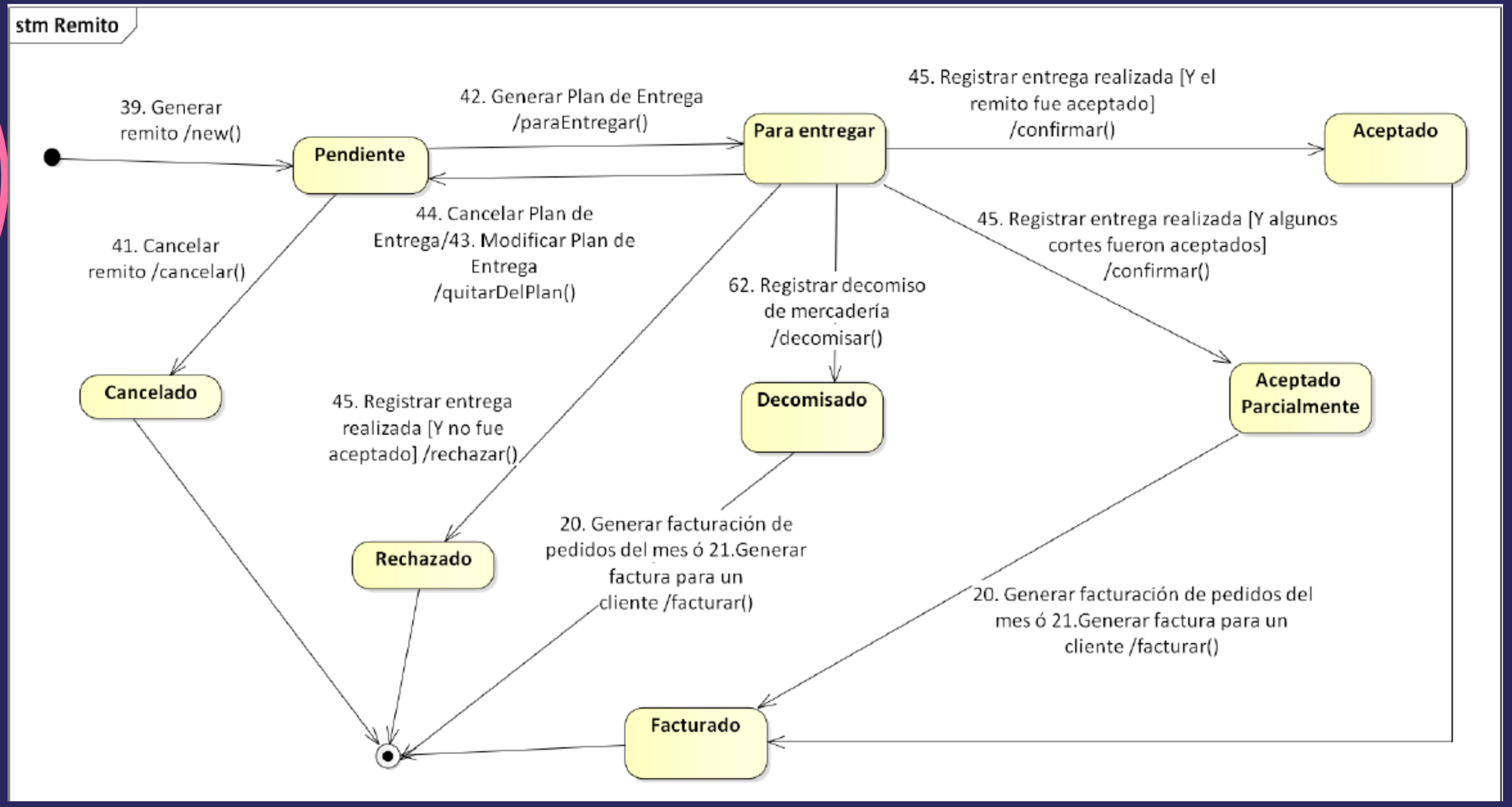
3 Los **Estados Concretos** proporcionan sus propias implementaciones para los métodos específicos del estado. Para evitar la duplicación de código similar a través de varios estados, puedes incluir clases abstractas intermedias que encapsulen algún comportamiento común.

Los objetos de estado pueden almacenar una referencia inversa al objeto de contexto. A través de esta referencia, el estado puede extraer cualquier información requerida del objeto de contexto, así como iniciar transiciones de estado.

4 Tanto el estado de contexto como el concreto pueden establecer el nuevo estado del contexto y realizar la transición de estado sustituyendo el objeto de estado vinculado al contexto.

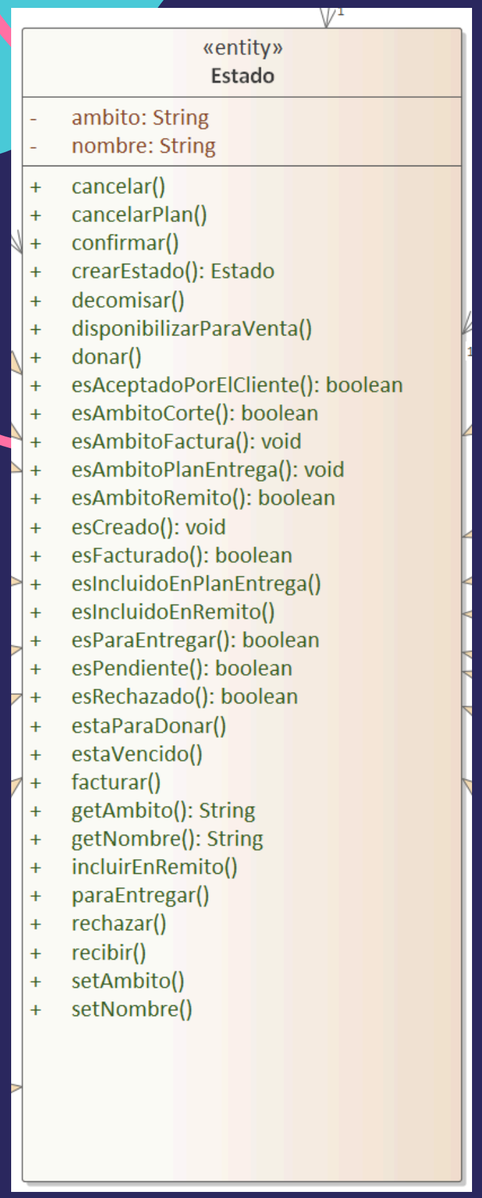
CU 42 - GENERAR PLAN DE ENTREGA

Máquina de Estados Remito



¿Cómo se traduce esto en la estructura?

¿Cómo se traduce esto en la estructura?



Clase Estado: definición

```
public abstract class Estado
{
    0 references
    private string Ambito {get; set};
    0 references
    private string Nombre {get; set};
}
```

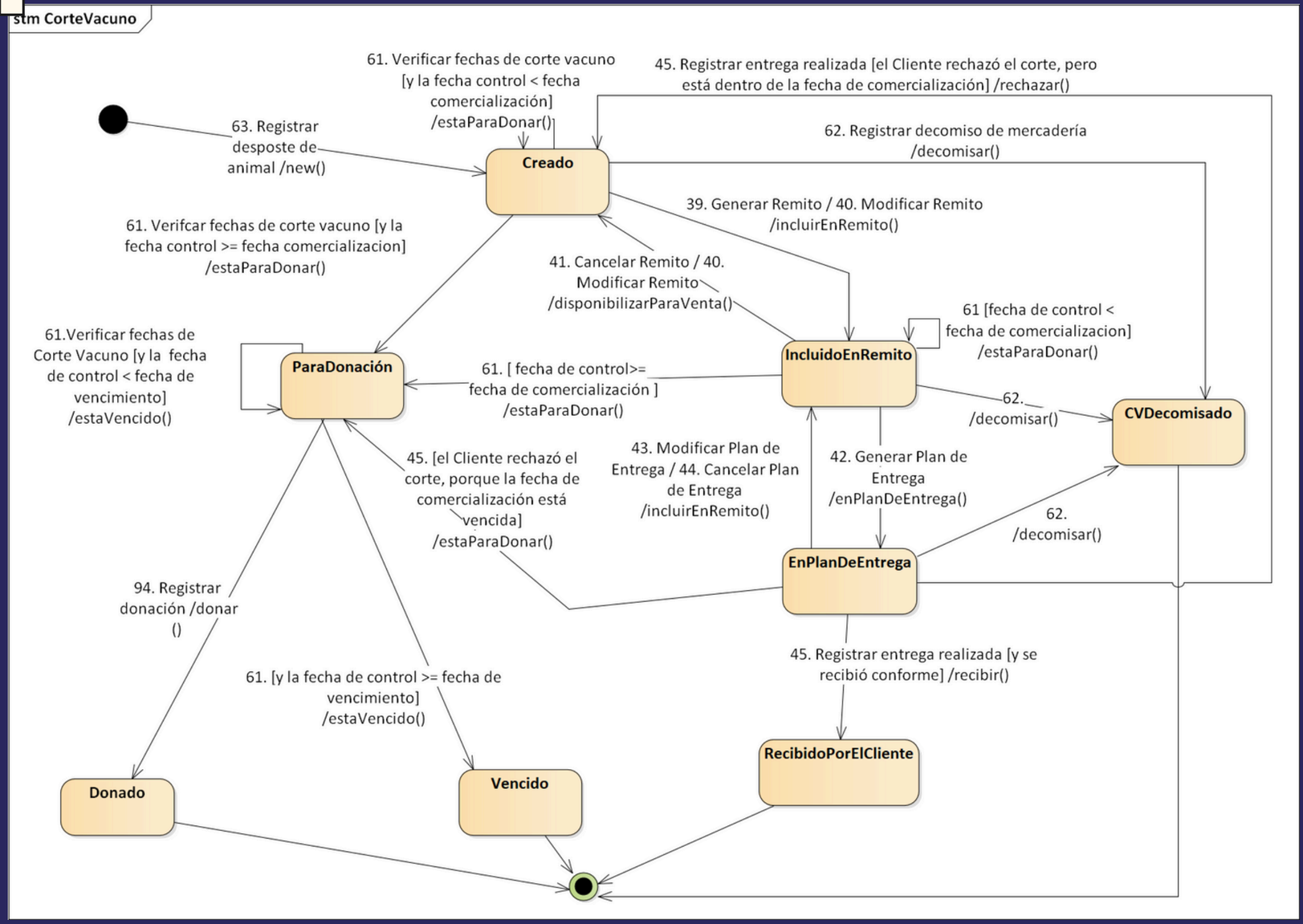
Clase Estado: métodos

```
public bool esAmbitoCorte()
{
    return this.ambito.Equals('corte');
}

0 references
public bool esAmbitoFactura()
{
    return this.ambito.Equals('factura');
}
```

Clase Estado: métodos

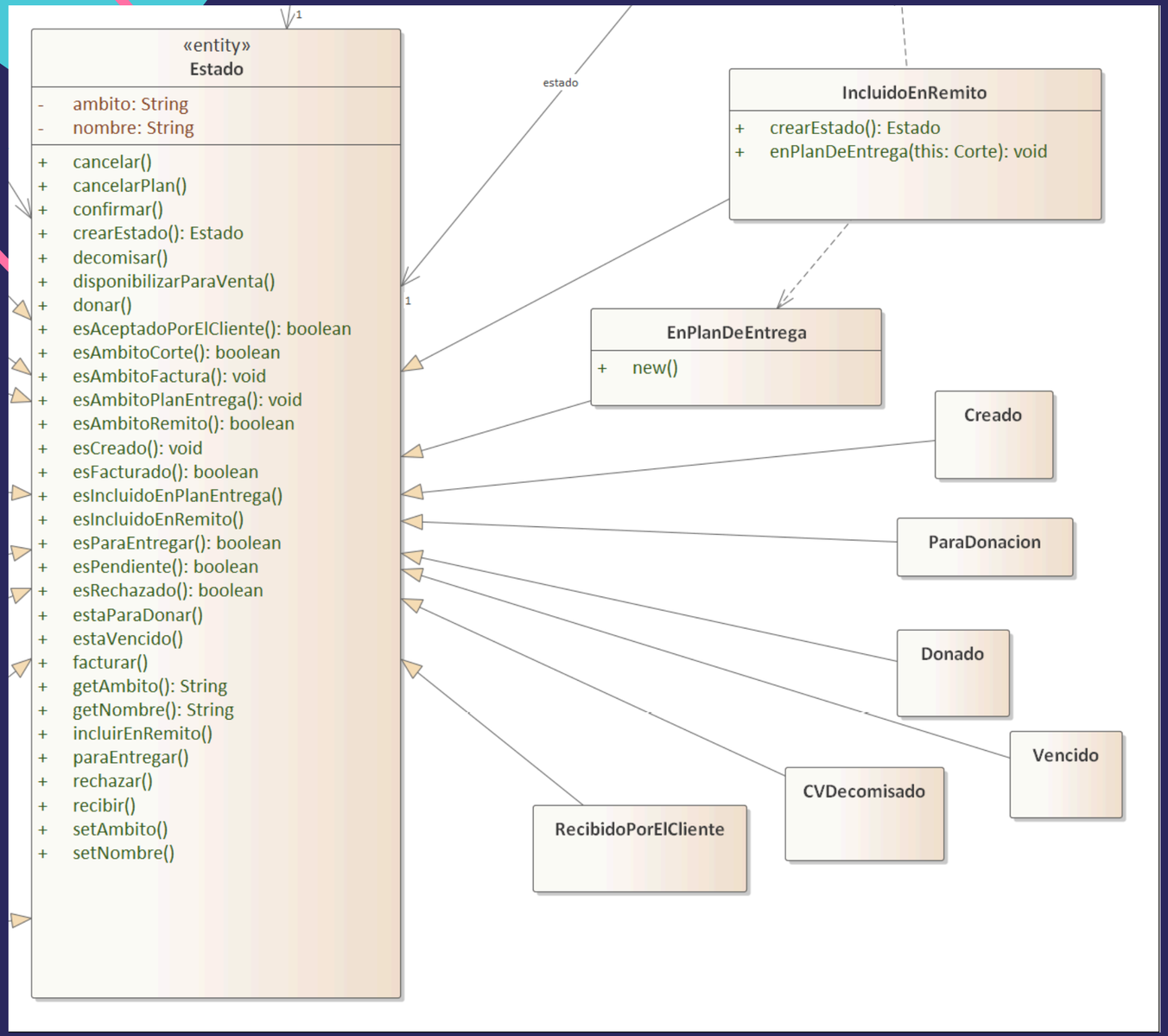
```
0 references
public abstract bool estaParaDonar();
0 references
public abstract bool estaVencido();
0 references
public abstract void facturar();
0 references
public abstract void incluirEnRemito();
0 references
public abstract void paraEntregar();
0 references
public abstract void rechazar();
0 references
public abstract void recibir ();
```

Máquina de Estados CorteV



¿Cómo se traduce esto en la estructura?

¿Cómo se traduce esto en la estructura?



Estados hijos: definición

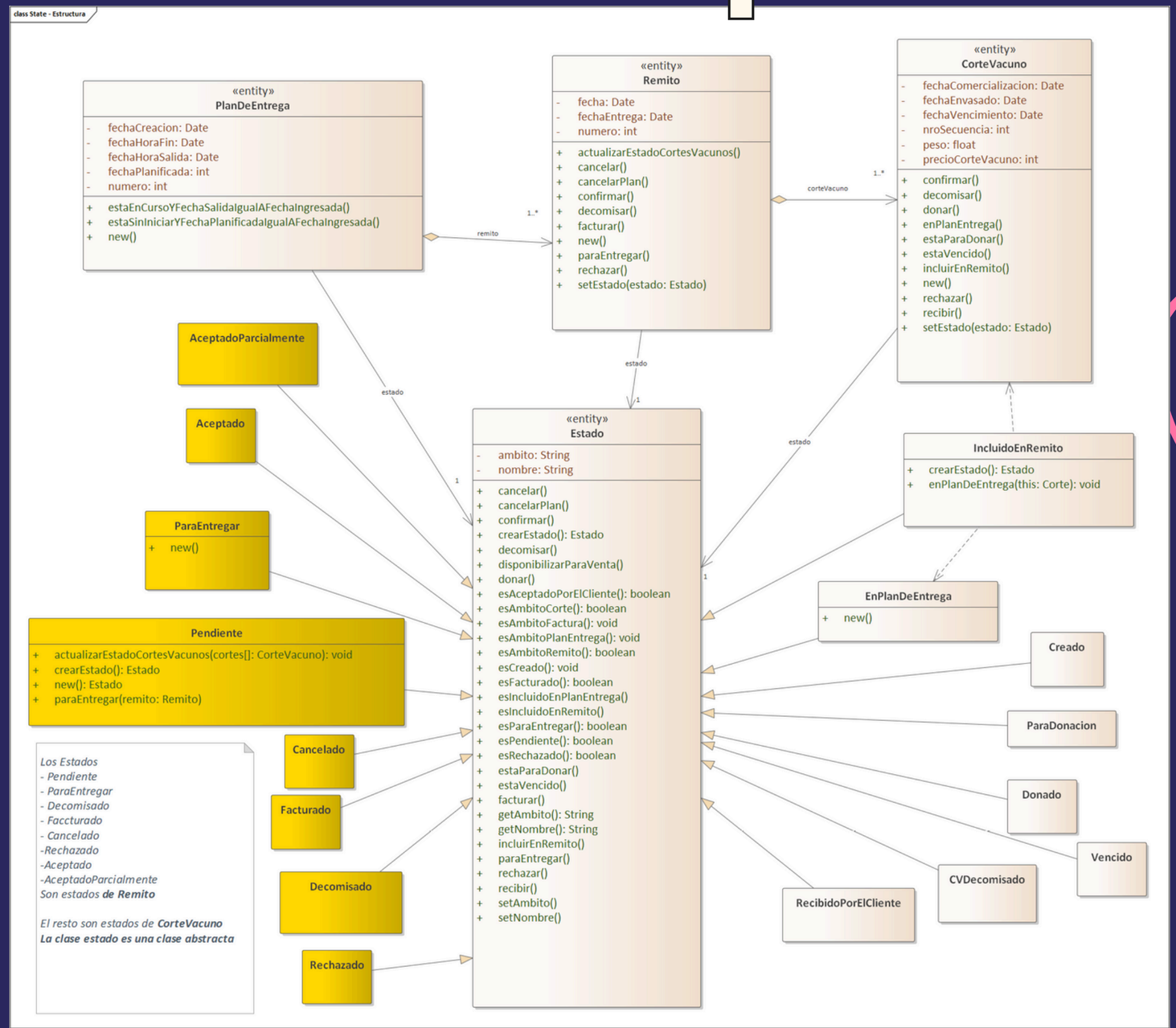
```
public class IncluidoEnRemito : Estado
{
    0 references
    public IncluidoEnRemito(string ambito, string nombre) : base(ambito, nombre)
    {
    }

    0 references
    public IncluidoEnRemito() : base('corteVacuno', 'incluidoEnRemito')
    {

    }
```

Estados hijos: métodos

```
public override void recibir()
{
    throw new NotImplementedException();
}

    2 references
public override void enPlanEntrega(Corte corte)
{
    enPlanEntrega estadoEnPlanEntrega = this.crearEstado();
    corte.setEstado(enPlanEntrega);
}
```

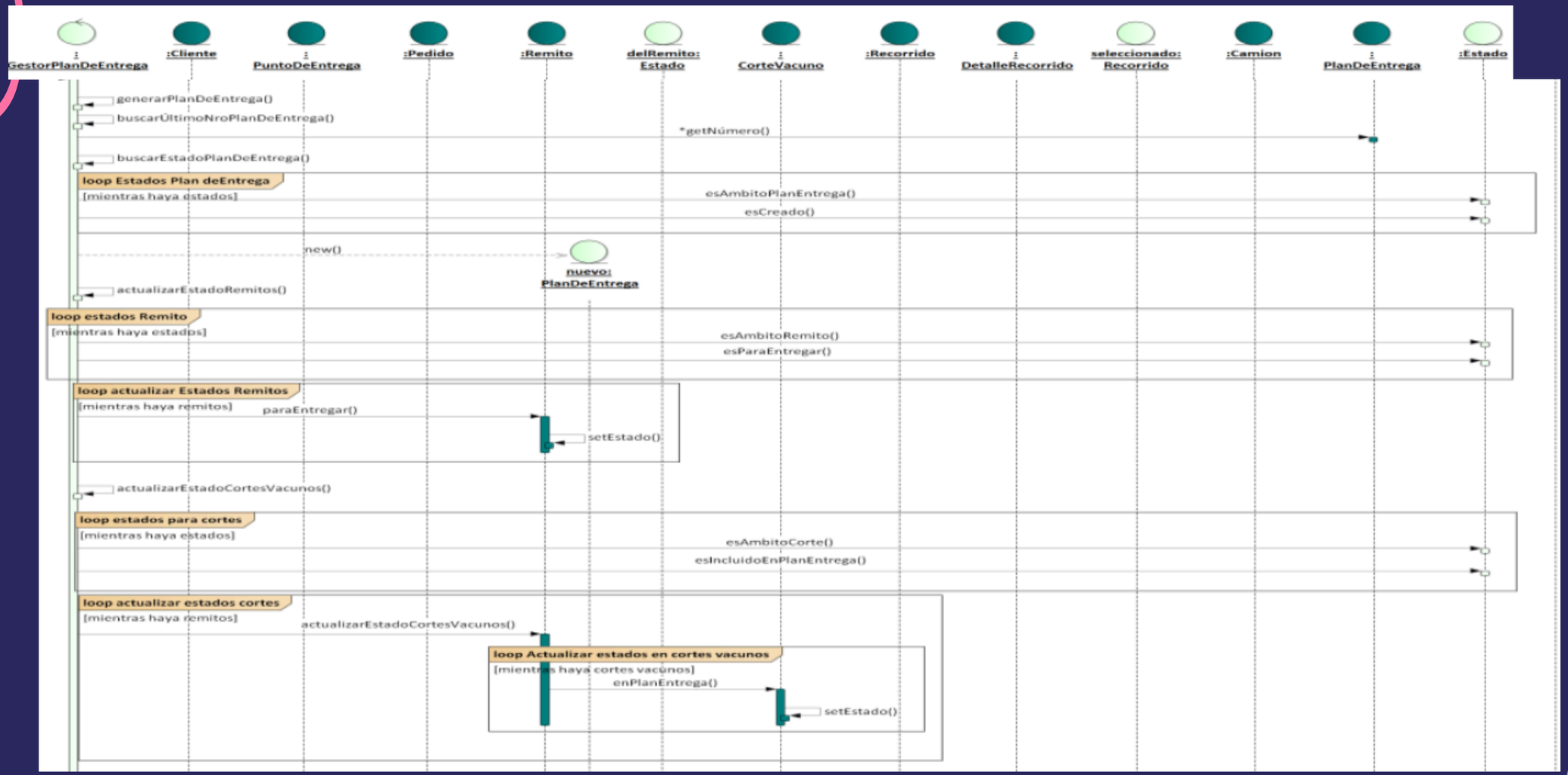
Vista Estática completa

- FRIGOSTATE
  - Controladores
    - GestorPlanDeEntrega.cs
  - Interfaces
    - PantallaPlanDeEntrega.cs
  - Modelo
    - Estados
      - Aceptado.cs
      - AceptadoParcialmente.cs
      - Cancelado.cs
      - Creado.cs
      - CVDecomisado.cs
      - Decomisado.cs
      - Donado.cs
      - EnPlanEntrega.cs
      - Estado.cs
      - Facturado.cs
      - IncluidoEnRemito.cs
      - ParaDonacion.cs
      - ParaEntregar.cs
      - Pendiente.cs
      - Rechazado.cs
      - RecibidoPorElCliente.cs
      - Vencido.cs
    - Barrio.cs
    - Camion.cs
    - Cliente.cs
    - CorteVacuno.cs
    - Estado.cs
    - Localidad.cs
    - Pedido.cs
    - PlanDeEntrega.cs
    - PuntoDeEntrega.cs
    - Remito.cs

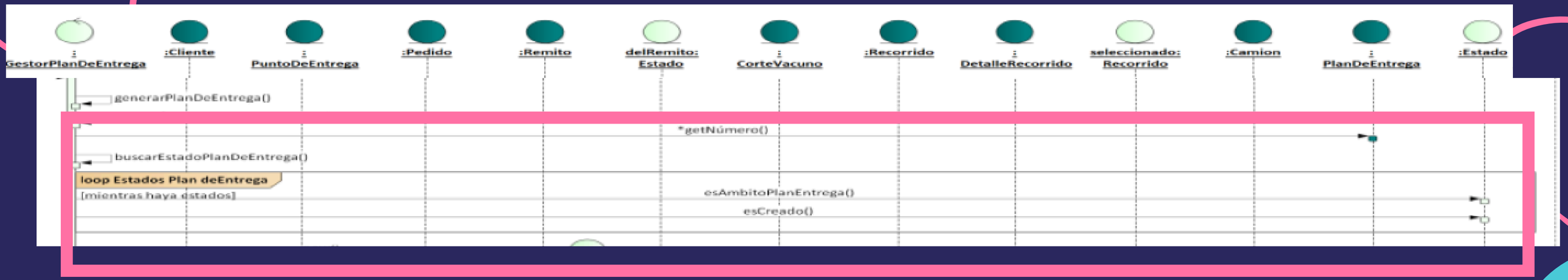


Implementación

iii. Rediseñar la estructura de forma de optimizar el manejo del comportamiento variable del **remito** y los **cortes vacunos** que lo componen, según la situación en la que se encuentre.

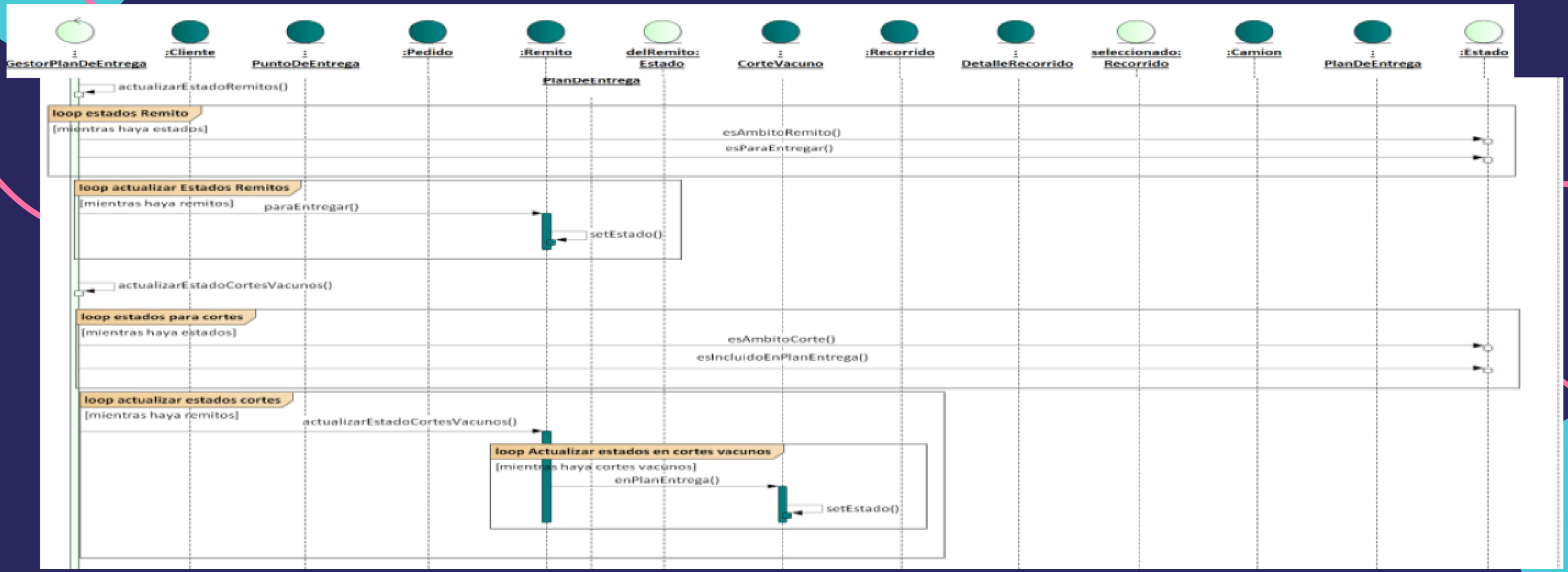


Implementación



¿Tenemos la búsqueda de los estados en la clase Estado?

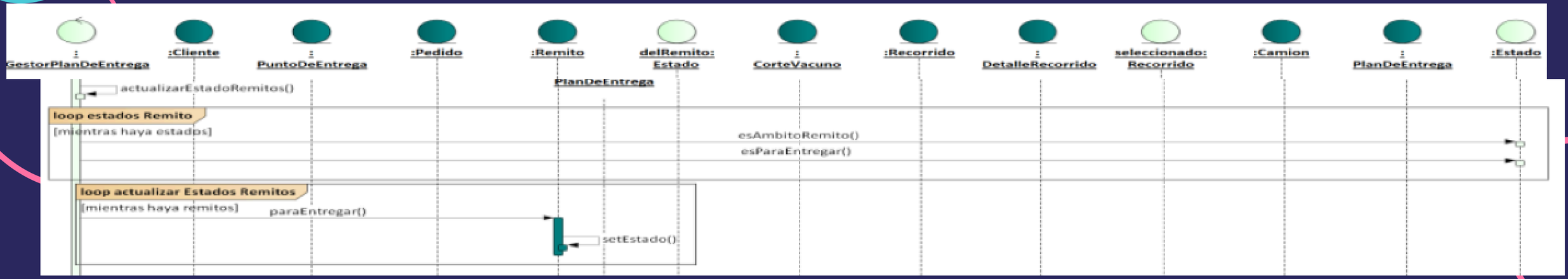
Implementación



¿Cómo cambia la secuencia para los remitos y los cortes?

Implementación

Antes




```
public class GestorPlanDeEntrega
    public void actualizarEstadoRemitos()
    {
        Estado estadoParaEntregar = this.estados.Find(estado =>
        {
            if (estado.esAmbitoRemito() && estado.esParaEntregar())
            {
                return estado;
            }
        })

        this.remitosSeleccionados.ForEach(remito =>
        {
            remito.paraEntregar(estadoParaEntregar);
        })
    }
```

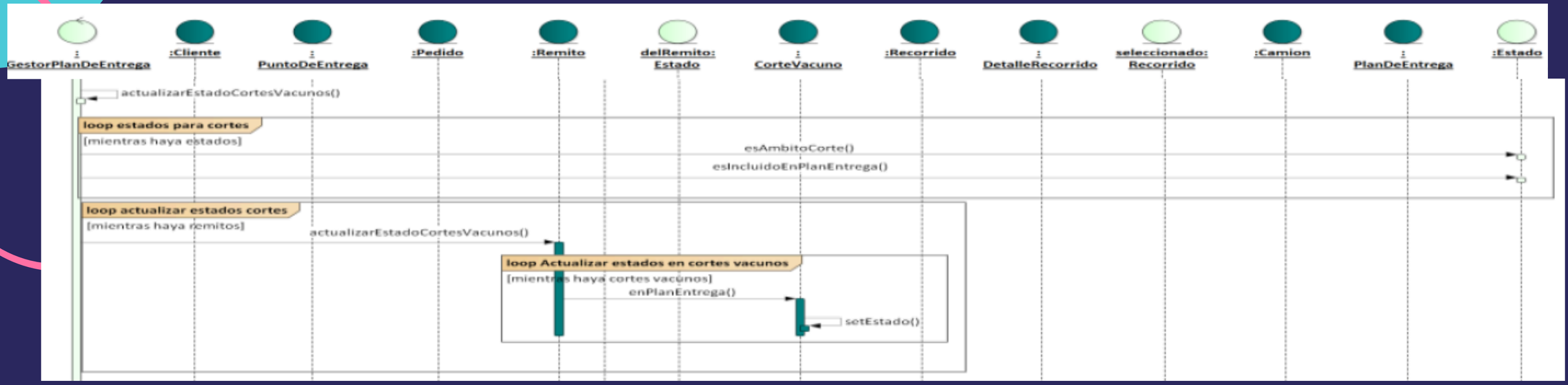
```
public class Remito
    public void paraEntregar(Estado estadoParaEntregar)
    {
        this.setEstado(estadoParaEntregar);
    }
```

¿Cómo delegamos la lógica?

¿Quién actualiza el estado de los CV?

Implementación

Antes



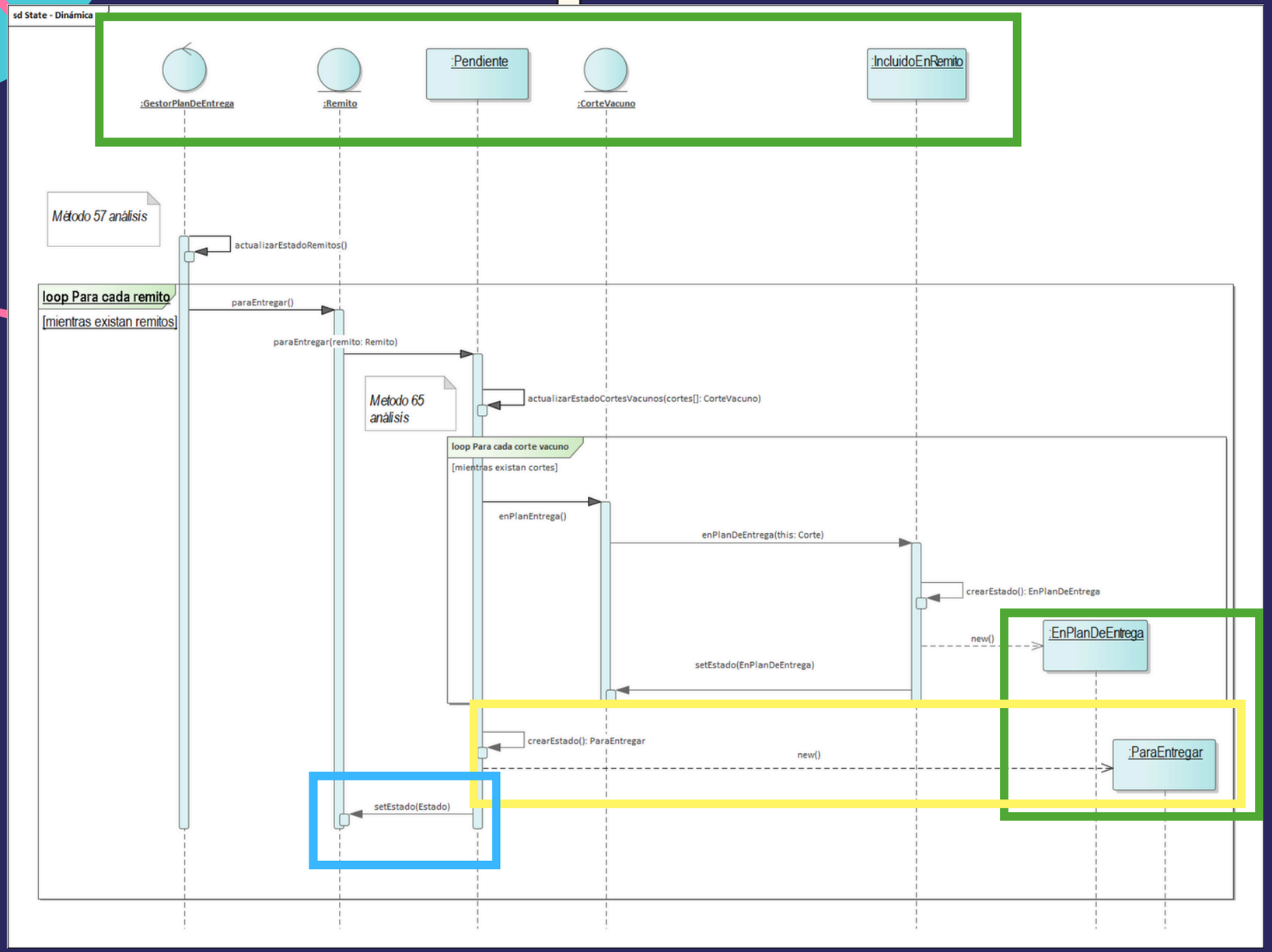

```
public class GestorPlanDeEntrega
    public void actualizarEstadoCortesVacunos()
    {
        Estado estadoIncluidoEnPlanEntrega = this.estados.Find(estado =>
        {
            if (estado.esAmbitoCorteVacuno() && estado.esIncluidoEnPlanEntrega())
            {
                return estado;
            }
        })

        this.remitosSeleccionados.ForEach(remito =>
        {
            remito.actualizarEstadoCortesVacunos(estadoIncluidoEnPlanEntrega);
        })
    }
```

```
public class Remito
    public void actualizarEstadoCortesVacunos(Estado estadoIncluidoEnPlanEntrega)
    {
        this.cortes.ForEach(corte =>
        {
            corte.EnPlanEntrega(estadoIncluidoEnPlanEntrega);
        })
    }
```

```
public class CorteVacuno
    public void enPlanEntrega(Estado estadoIncluidoEnPlanEntrega)
    {
        this.setEstado(estadoIncluidoEnPlanEntrega);
    }
```

Después

Implementación



Ahora, nos manejamos con los estados concretos

Los estados concretos son capaces de crear otros estados

El cambio de estado es responsabilidad del estado concreto

Implementación

Antes

```
public class GestorPlanDeEntrega
    public void actualizarEstadoRemitos()
    {
        Estado estadoParaEntregar = this.estados.Find(estado =>
        {
            if (estado.esAmbitoRemito() && estado.esParaEntregar())
            {
                return estado;
            }
        })

        this.remitosSeleccionados.ForEach(remito =>
        {
            remito.paraEntregar(estadoParaEntregar);
        })
    }
```

```
public class Remito
    public void paraEntregar(Estado estadoParaEntregar)
    {
        this.setEstado(estadoParaEntregar);
    }
```

Después

```
public class GestorPlanDeEntrega
    public void actualizarEstadoRemitos()
    {
        this.remitosSeleccionados.ForEach(remito =>
        {
            remito.paraEntregar();
        })
    }
```

```
public class Remito
    0 references
    public void paraEntregar()
    {
        this.estado.paraEntregar(this, this.cortes);
    }
```

```
public class Pendiente : Estado
    public override void paraEntregar(Remito remito, List<CorteVacuno> cortes)
    {
        actualizarEstadoCortesVacunos(cortes);
        ParaEntregar estadoParaEntregar = new ParaEntregar();
        remito.setEstado(estadoParaEntregar);
    }
```

Implementación

```
public class Pendiente : Estado
    public override void paraEntregar(Remito remito, List<CorteVacuno> cortes)
    {
        actualizarEstadoCortesVacunos(cortes);
        ParaEntregar estadoParaEntregar = new ParaEntregar();
        remito.setEstado(estadoParaEntregar);
    }
```

```
public class Pendiente : Estado
    public void actualizarEstadoCortesVacunos(List<CorteVacuno> cortes){
        cortes.ForEach(corte => {
            corte.enPlanEntrega()
        });
    }
```

```
public class CorteVacuno
    public void enPlanEntrega()
    {
        this.estado.enPlanEntrega(this);
    }
```

```
public class IncluidoEnRemito : Estado
    public override void enPlanEntrega(Corte corte)
    {
        enPlanEntrega estadoEnPlanEntrega = this.crearEstado();
        corte.setEstado(enPlanEntrega);
    }

    1 reference
    public override Estado crearEstado()
    {
        return new EnPlanEntrega(); ;
    }
```

Procedimiento

1. Definir en qué parte del CU se va a aplicar el patrón
2. Definir la clase Estado como abstracta con todos los métodos
3. Definir los estados concretos
4. Delegar al estado concreto la responsabilidad identificada
5. Revisar consistencias entre estática y dinámica

Gracias! :)