

Lección 2.3

Diagramas de interacción entre objetos

Objetivos de aprendizaje



- Concebir el envío de un mensaje como una petición de colaboración de un objeto a otro.
- Identificar los distintos tipos de canales de comunicación a través de los cuales los objetos pueden enviarse mensajes.
- Comprender el significado de los distintos elementos de un diagrama de interacción (comunicación y/o secuencia) de UML.
- Entender la correspondencia entre los distintos tipos de diagramas de interacción.
- Conocer la implementación de los diagramas de interacción en un lenguaje de programación orientado a objetos.
- Aprender a modelar ejemplos de operaciones simples.

Contenidos

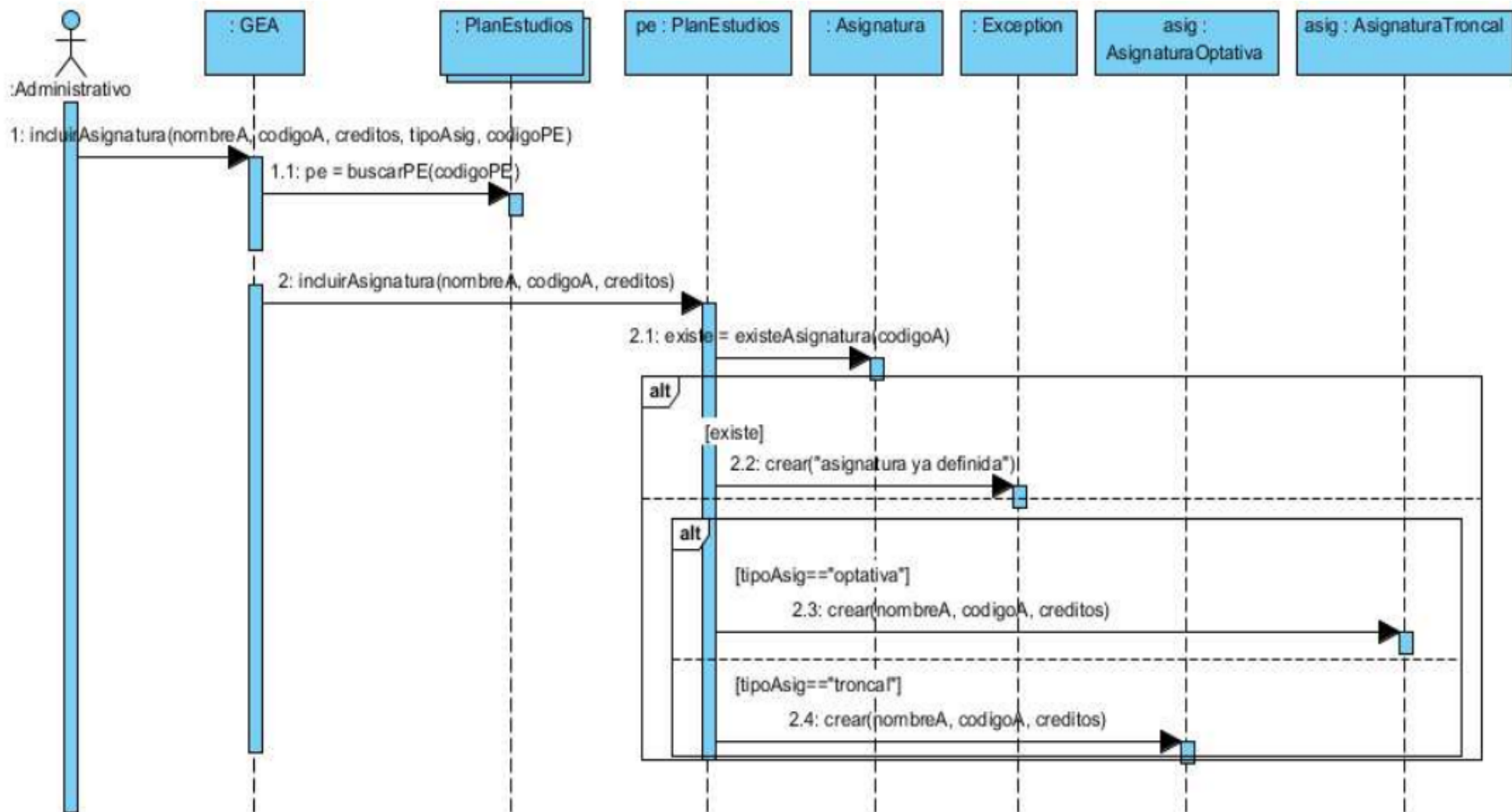


1. Diagramas de interacción.
2. Diagramas de secuencia.
3. Implementación de diagramas de secuencia.
4. Diagramas de comunicación.
5. Implementación de diagramas de comunicación.
6. Equivalencia entre diagramas.
7. Modelando interacción entre objetos.

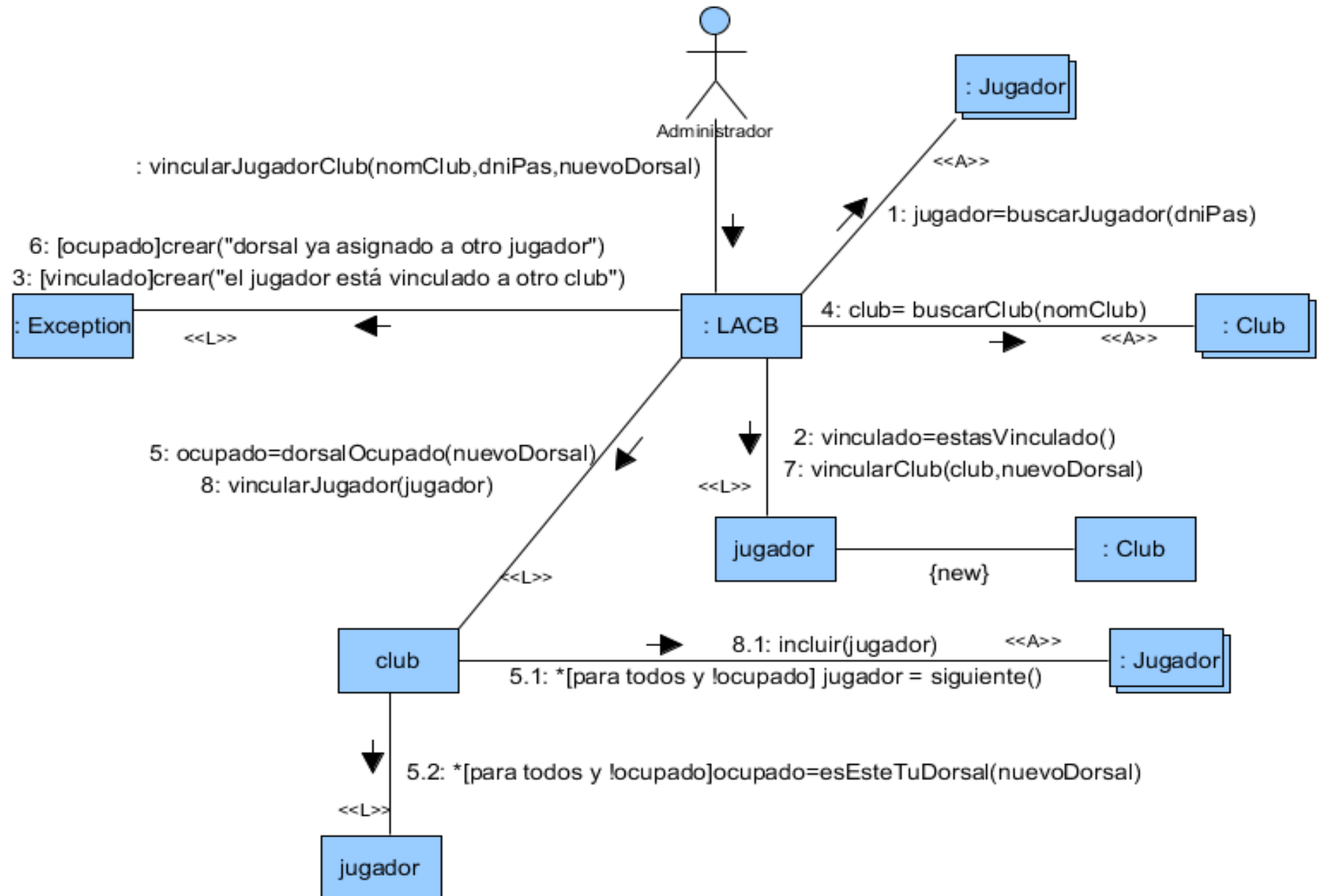
1. Diagramas de interacción

- Muestran la **colaboración entre objetos** para llevar a cabo una **funcionalidad general** asociada a un sistema o subsistema.
- Representan la comunicación entre objetos a través del **envío de mensajes**.
- Diagramas de interacción de UML: **Diagrama de secuencia y diagrama de comunicación**. Son semánticamente equivalentes, su diferencia esencial es:
 - Los de **secuencia** tienen una **componente temporal** fuerte, se centran en la secuencia temporal de los mensajes.
 - Los de **comunicación** tienen una **componente estructural** fuerte, se centran en las relaciones entre objetos que intercambian los mensajes.

1. Diagramas de interacción: Ejemplo



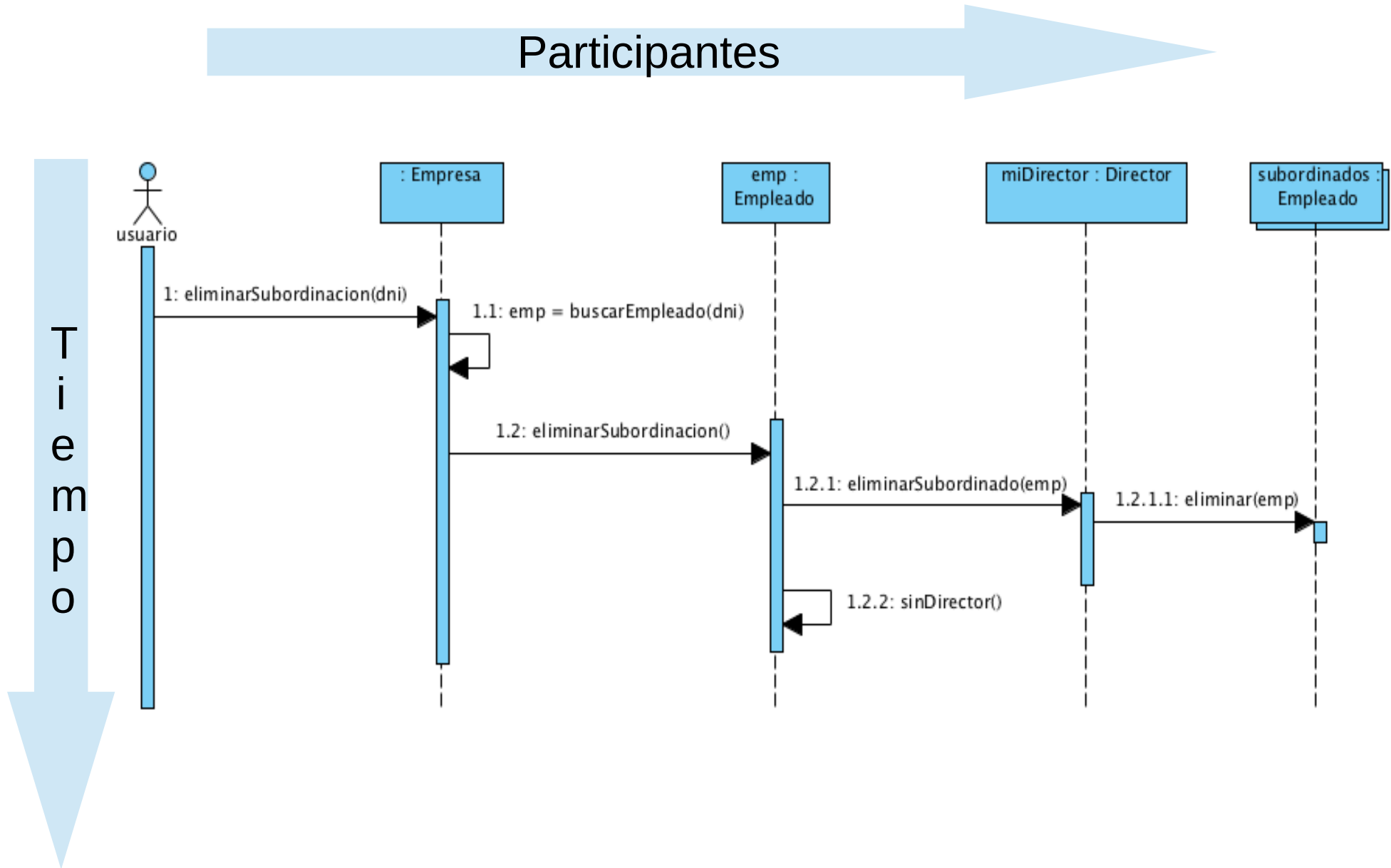
1. Diagramas de interacción: Ejemplo



2. Diagramas de secuencia

- Los diagramas de secuencia muestran de un forma visual muy clara el **orden** en el que ocurren los envíos de mensaje dentro de una interacción entre los distintos elementos que componen el sistema.
- Componentes:
 - **Participantes**: elementos del sistema que interactúan en la secuencia, **lo usual es que sean objetos**, pero también pueden ser otros componentes con capacidad de procesamiento.
 - **Envío de mensajes** entre los participantes.

2. Diagramas de secuencia

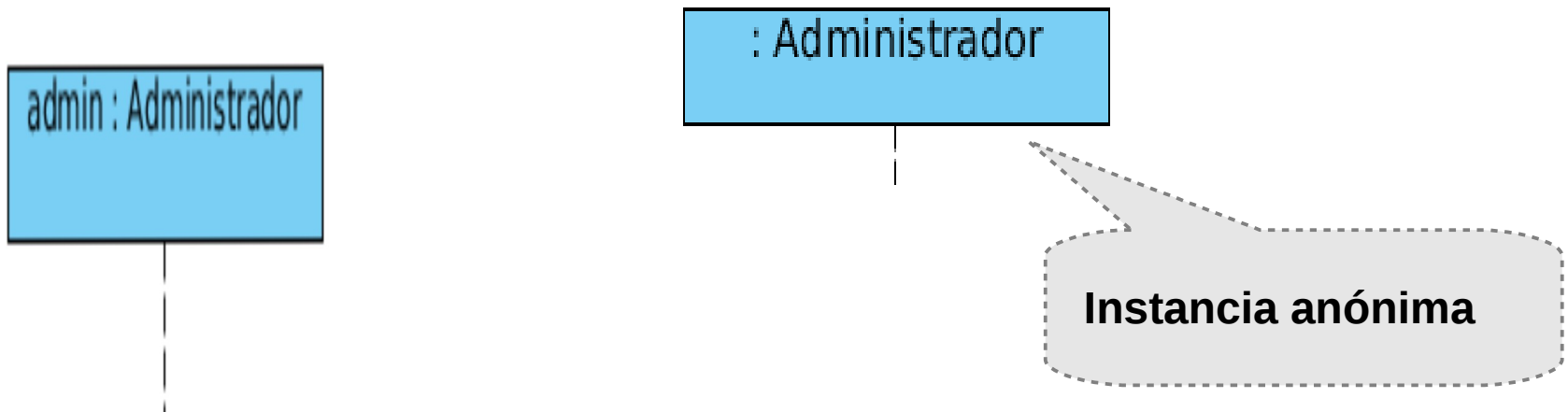


2. Diagramas de secuencia: Participante

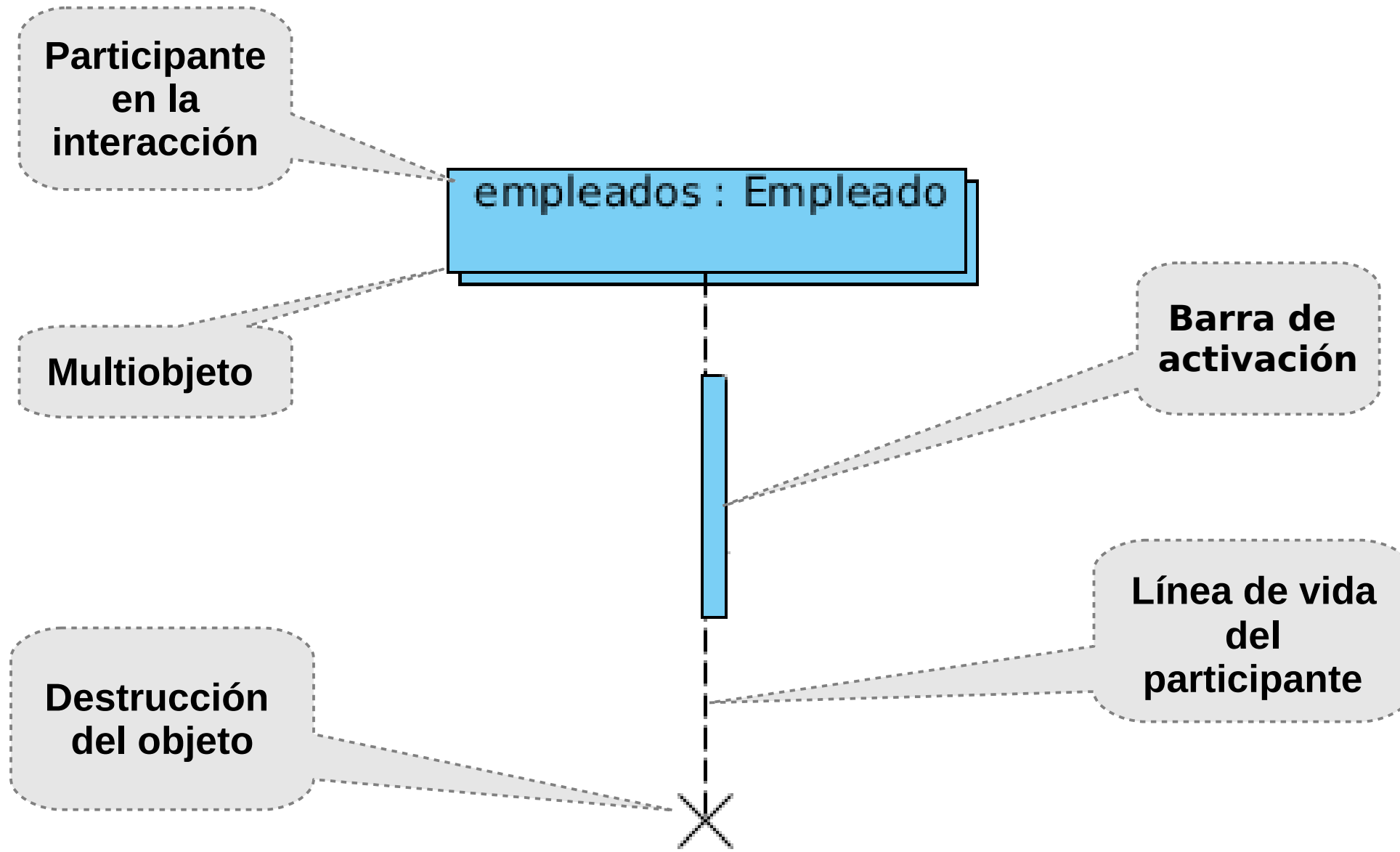
Sintaxis: `nombre : NombreClase`

- **nombre** → nombre de la instancia que participa en la interacción, debe indicarse en minúscula.
- **NombreClase** → nombre de la clase a la que pertenece ese participante, debe indicarse en mayúscula.

Representación:

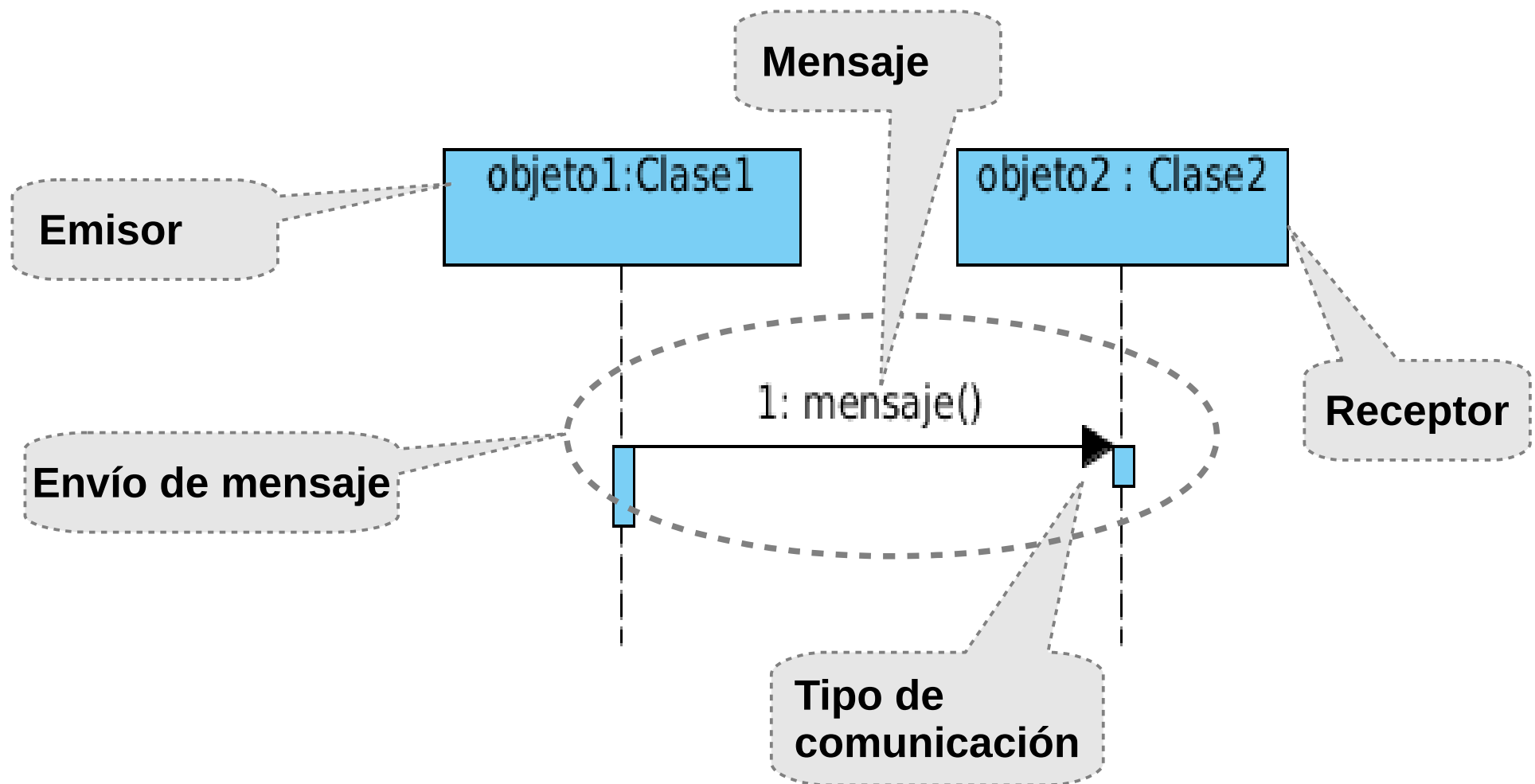


2. Diagramas de secuencia: Participante



2. Diagramas de secuencia: Envío de Mensaje

Un **envío de mensaje** es la comunicación entre dos participantes y en la que se especifica el tipo de comunicación, el mensaje en sí, el emisor y el receptor del mensaje.



2. Diagramas de secuencia: Envío de Mensaje

Sintaxis del mensaje

variable = mensaje(argumentos) : tipoDevuelto

Donde:

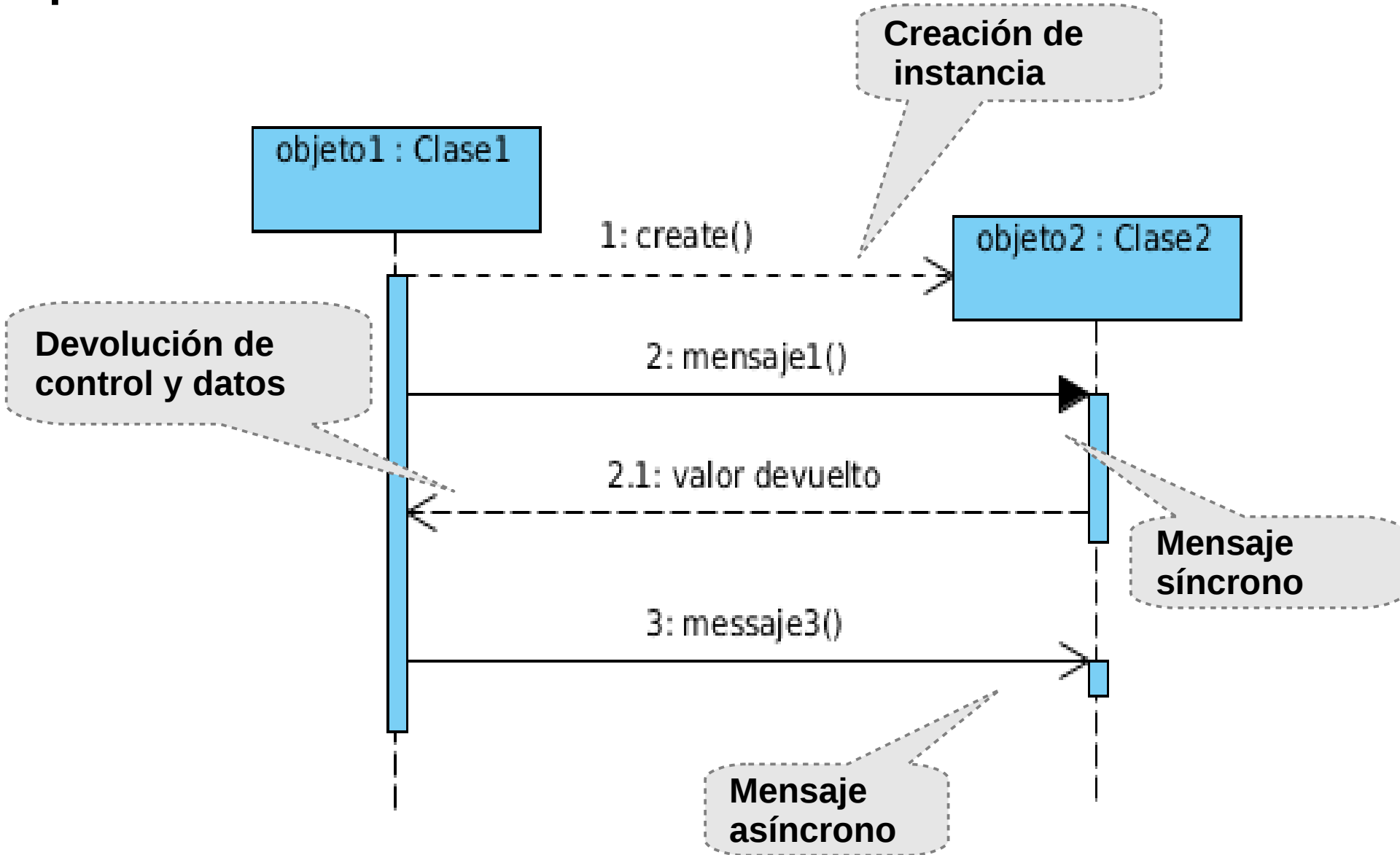
- **variable:** variable en la que se almacena el valor devuelto.
- **mensaje:** nombre del método invocado en el envío de mensaje.
- **argumentos:** argumentos pasados en el envío de mensaje, pueden ser valores o parámetros.
- **tipoDevuelto:** tipo o clase del valor devuelto.

Ejemplos:

- **hacerAlgo()**
- **hacerAlgo(argumento1,argumento2)**
- **hacerAlgo(argumento1:Clase1)**
- **haceralgo():ClaseR**
- **miVariable = hacerAlgo("abc"):ClaseR**

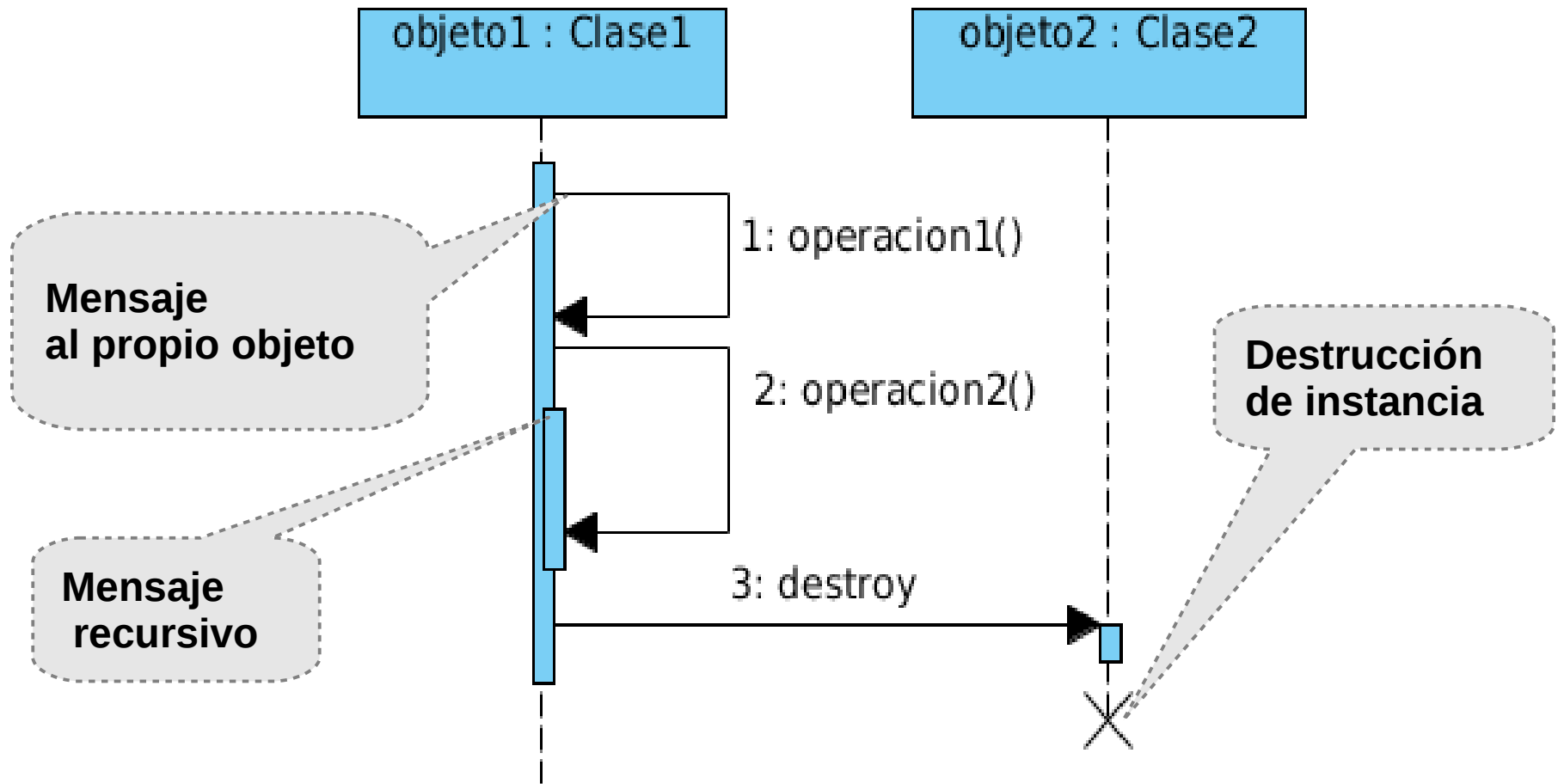
2. Diagramas de secuencia: Envío de Mensaje

Tipos de comunicación:



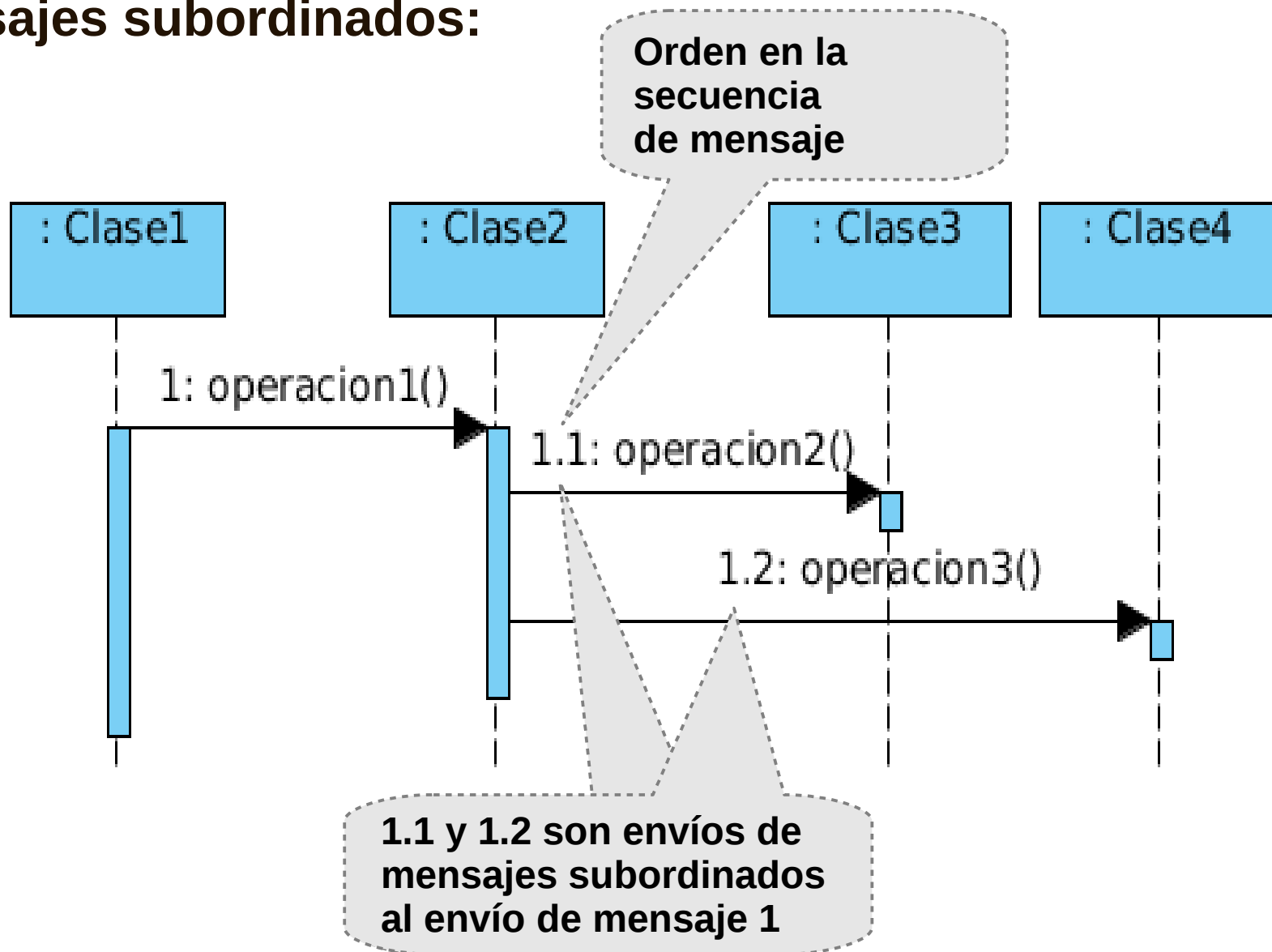
2. Diagramas de secuencia: Envío de Mensaje

Formas de comunicación:

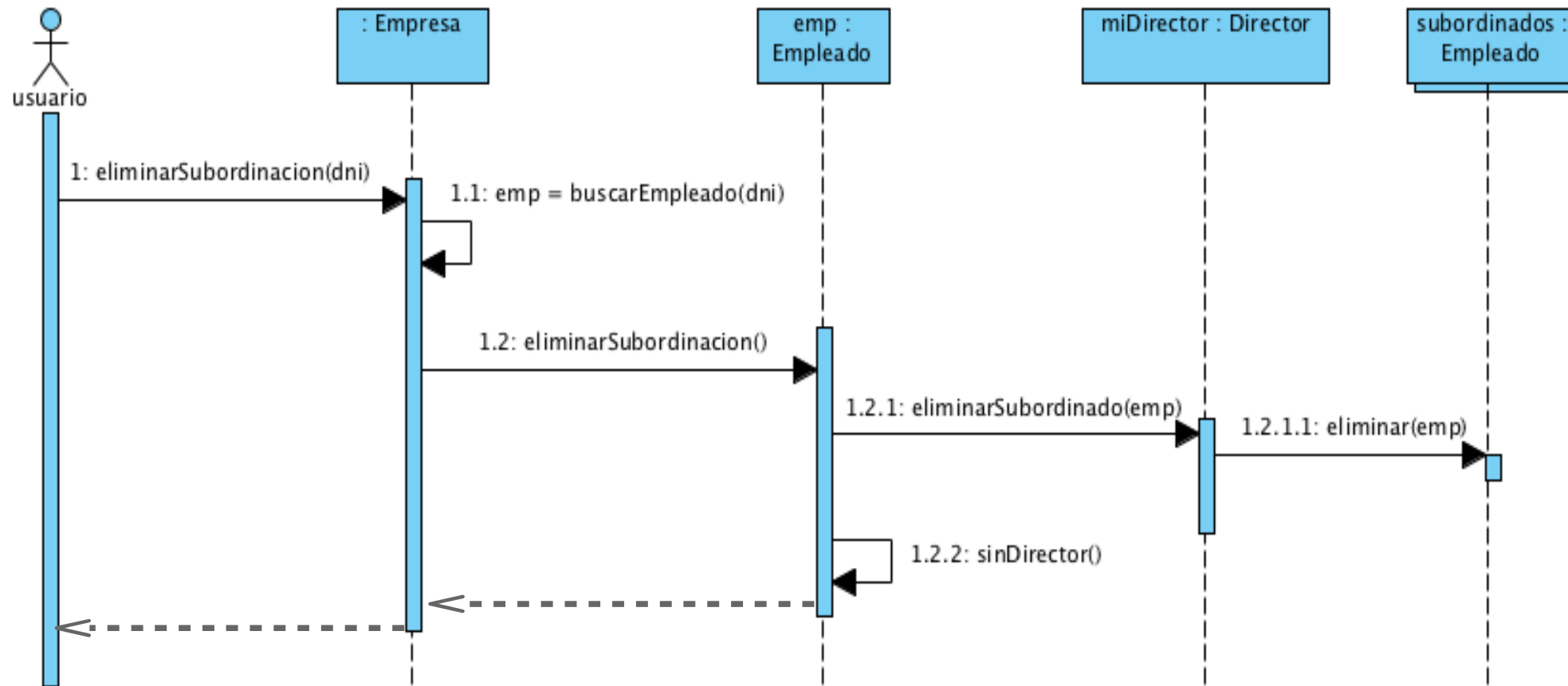


2. Diagramas de secuencia: Mensajes

Mensajes subordinados:



2. Diagramas de secuencia: Mensajes



Trabajar sobre este diagrama, viendo sus elementos sintácticos y semánticos



2. Diagramas de secuencia: Fragmentos

- **Fragmento de interacción:**

Secuencia de mensajes que ocurre bajo determinadas condiciones o propiedades. Puede tener una **condición de guarda** (expresión booleana con valor *true* por defecto).

- **Fragmento combinado:**

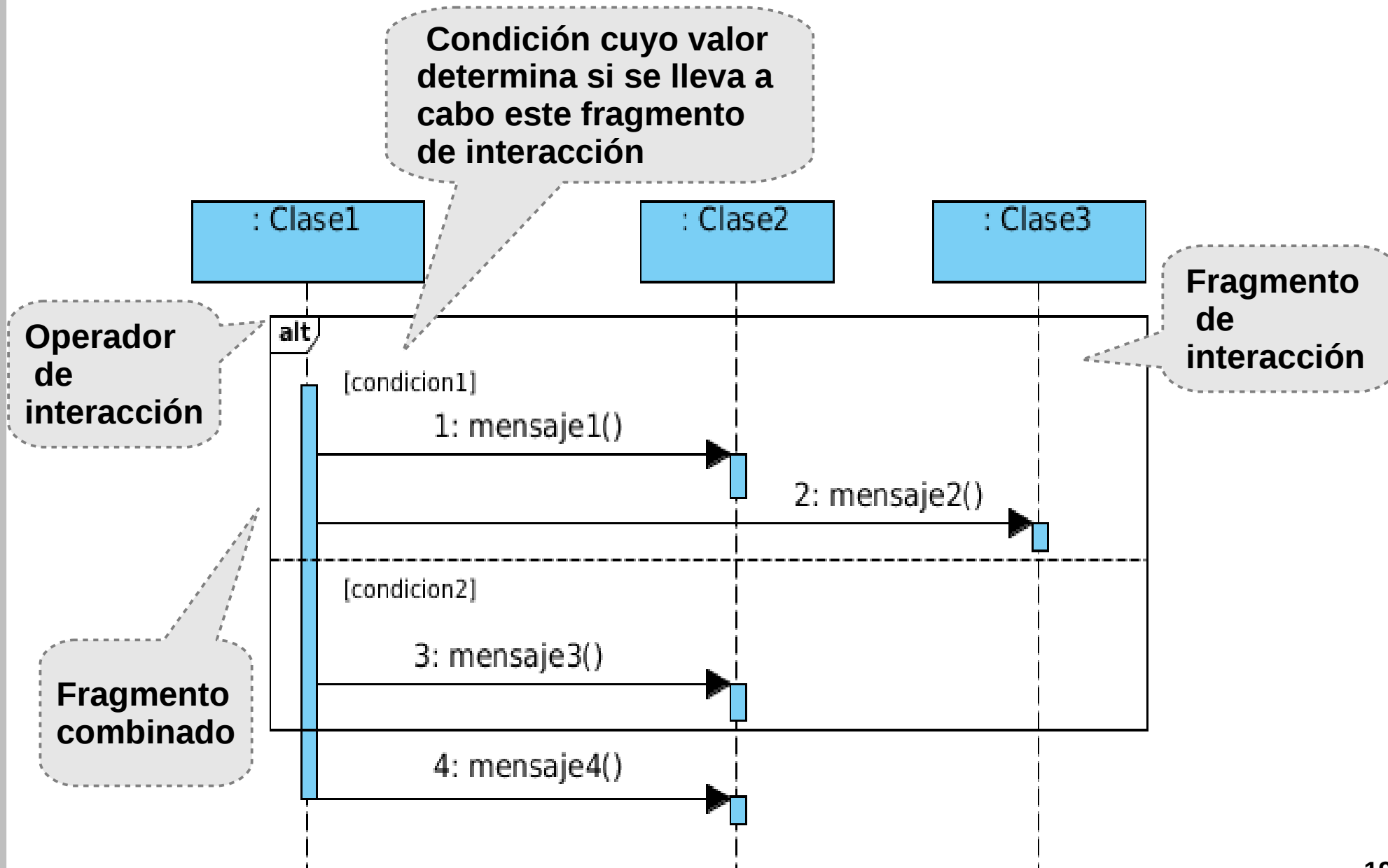
Unión de un **operador de interacción con uno o más fragmentos de interacción**, el número de éstos varía según el operador de interacción: alt, break, loop, option.

2. Diagramas de secuencia: Fragmentos

Tipo operador	Argumentos	Descripción
alt	[condicion1] [condicion2] .. [else]	Especifica que un fragmento de interacción puede ser ejecutado cuando se cumple determinada condición. Puede tener tantos fragmentos de interacción como se quiera.
break	[condicion1]	Cuando se cumple la condición se lleva a cabo el fragmento de interacción asociado y no se continua con lo que quede de ejecución.
loop	Min,max 0 [condicion]	Especifica las veces que puede realizarse un fragmento de interacción.
opt	[condicion]	El fragmento de interacción se realiza si se cumple la condición. Similar a alt, pero con un único fragmento de interacción.

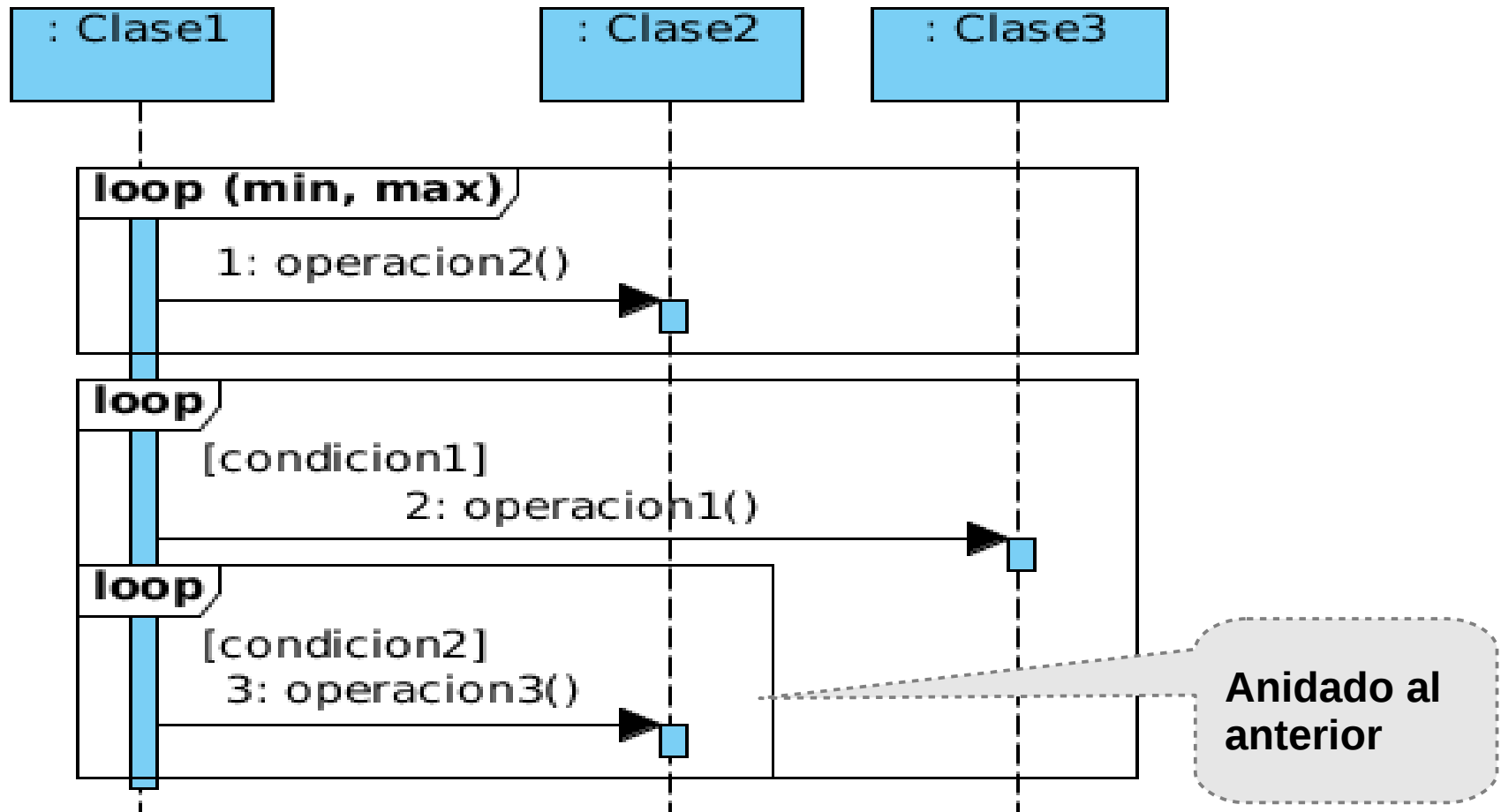
2. Diagramas de secuencia: Fragmentos

Ejemplo de **alt**



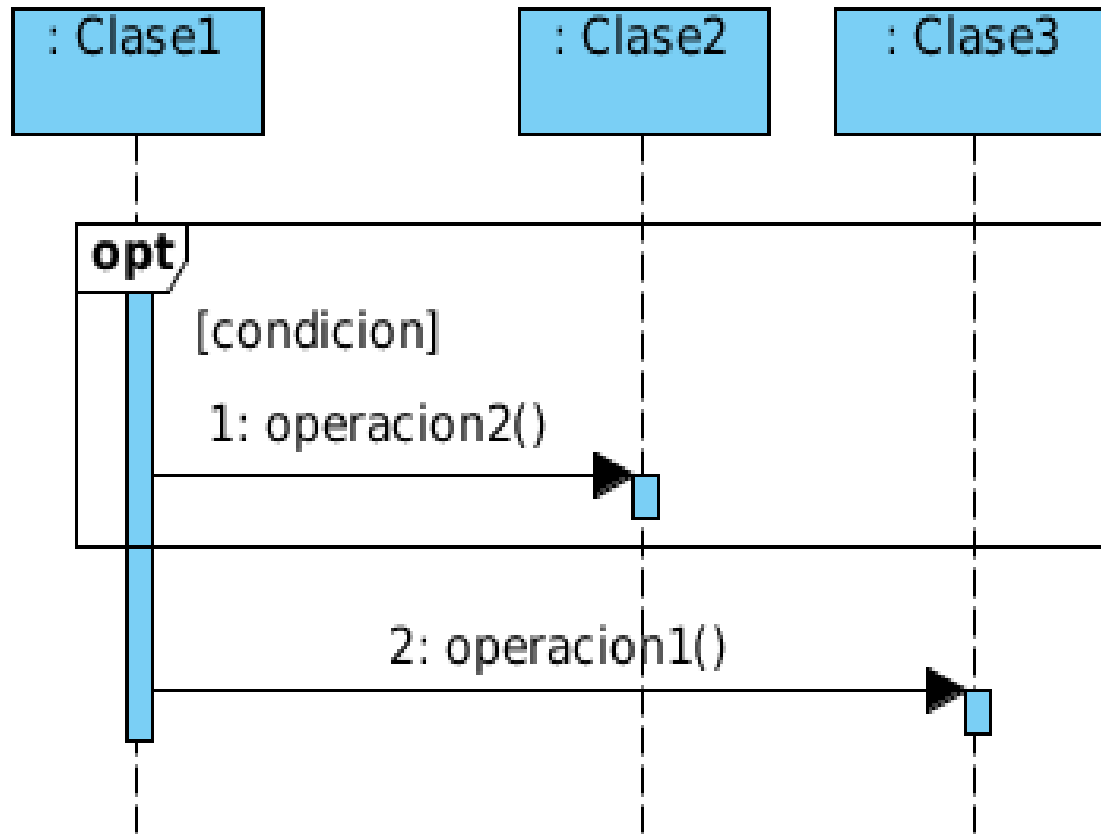
2. Diagramas de secuencia: Fragmentos

Ejemplo de **loop**

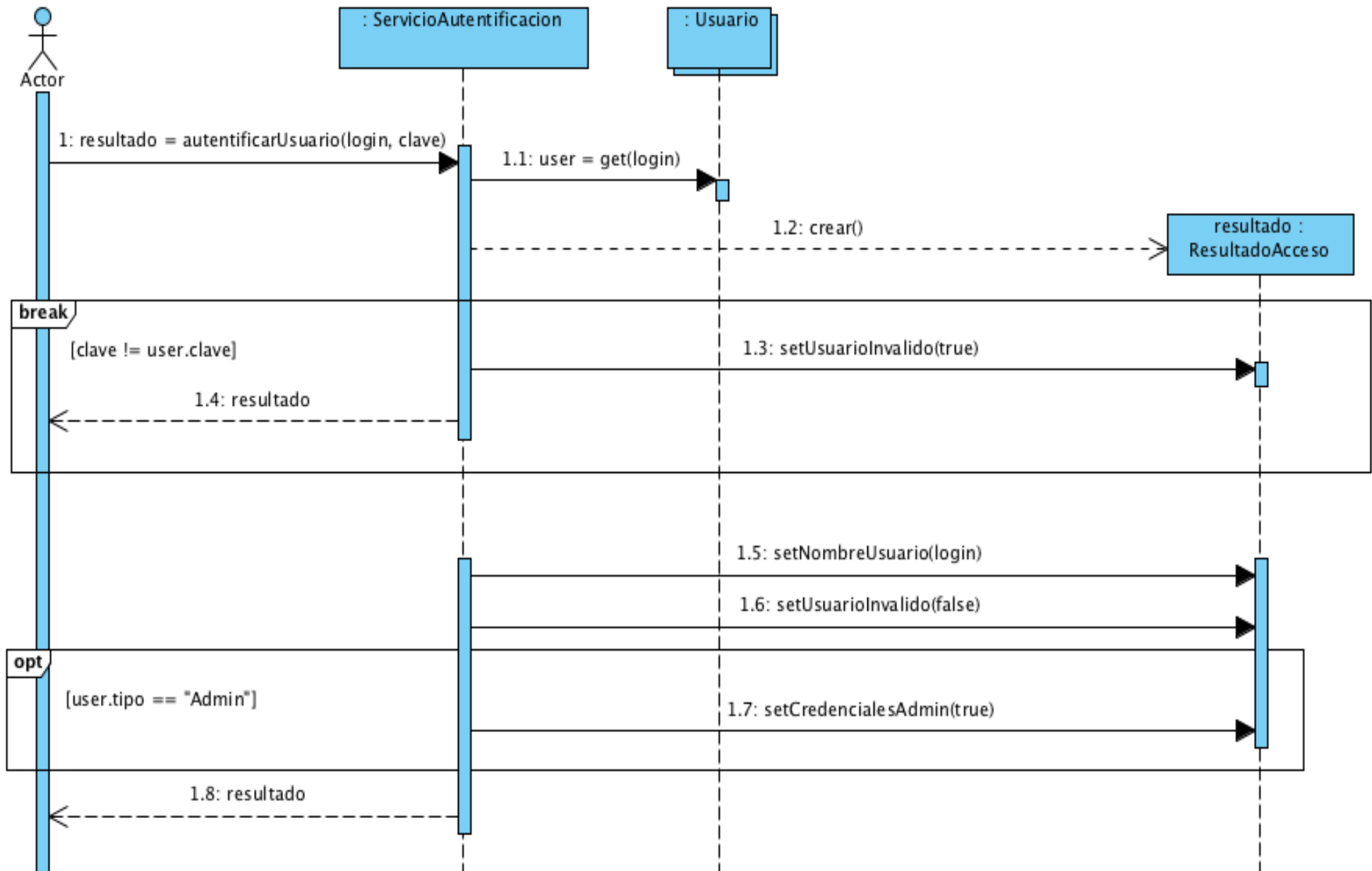


2. Diagramas de secuencia: Fragmentos

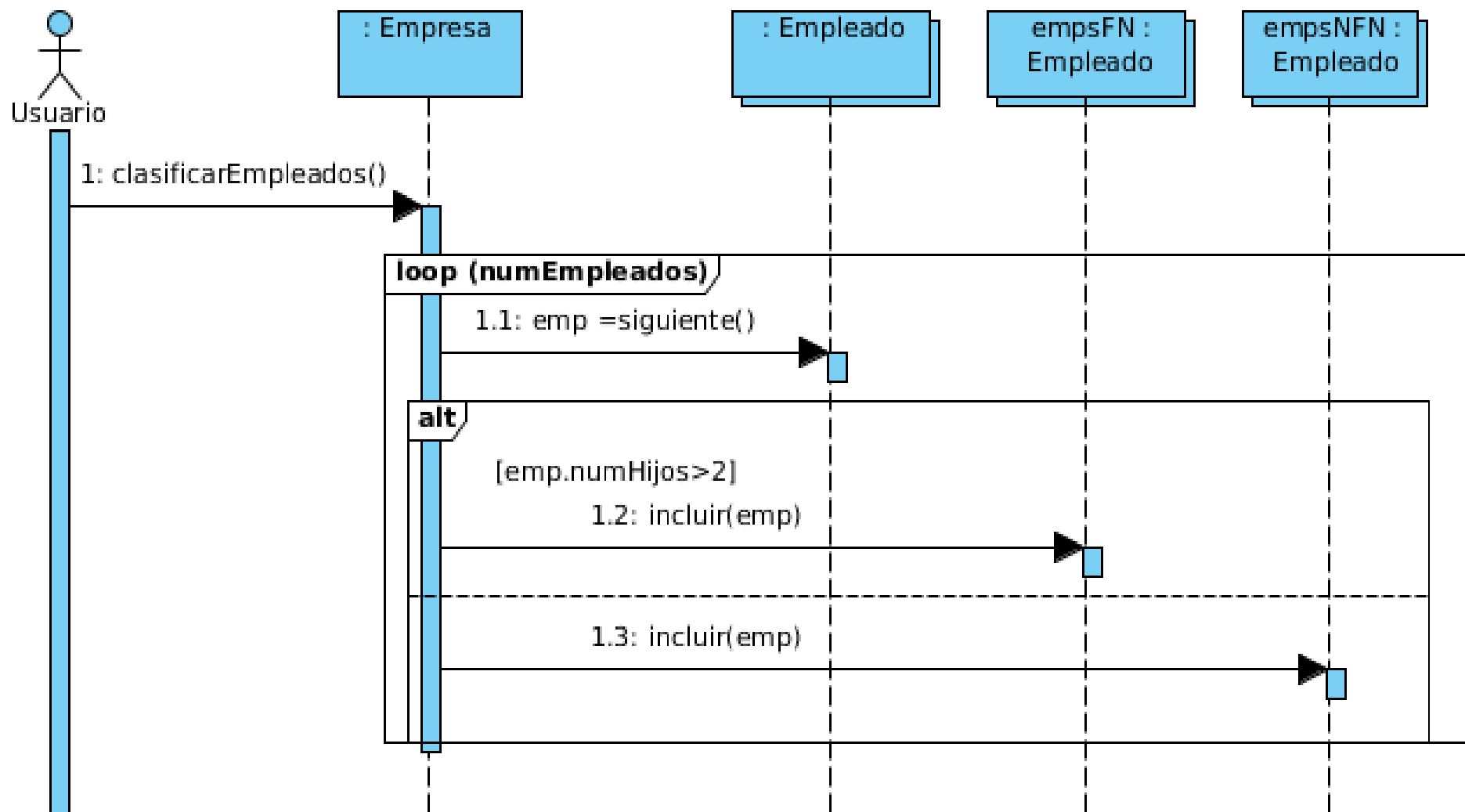
Ejemplo de **opt**



2. Diagramas de secuencia: Fragmentos

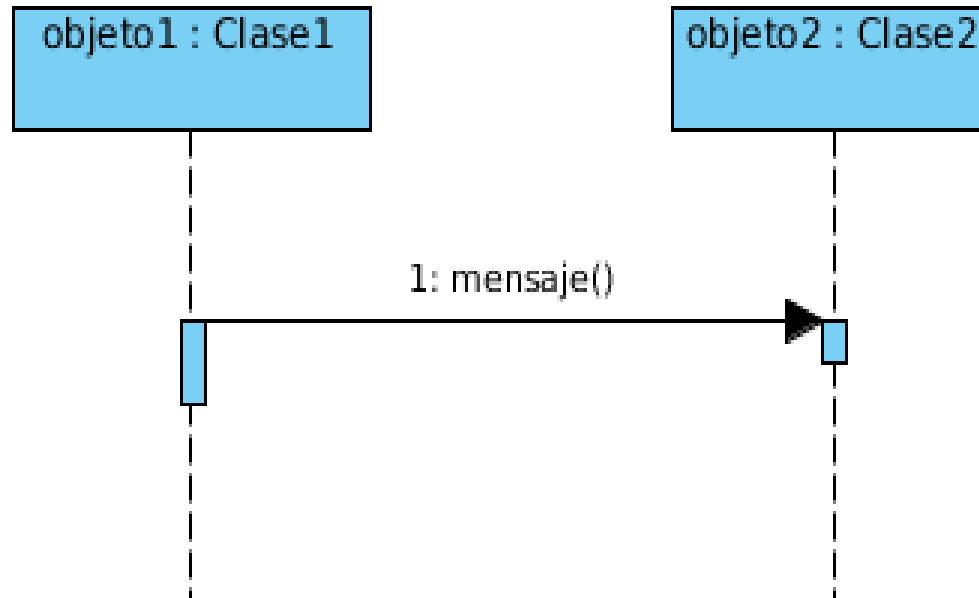


2. Diagramas de secuencia: Fragmentos



3. Implementación de DS

A partir de los diagramas de interacción se obtiene el código de los métodos de las clases.



Por ejemplo, el diagrama anterior se traduce en que en la clase Clase2 se define un método que responda a este envío de mensaje:

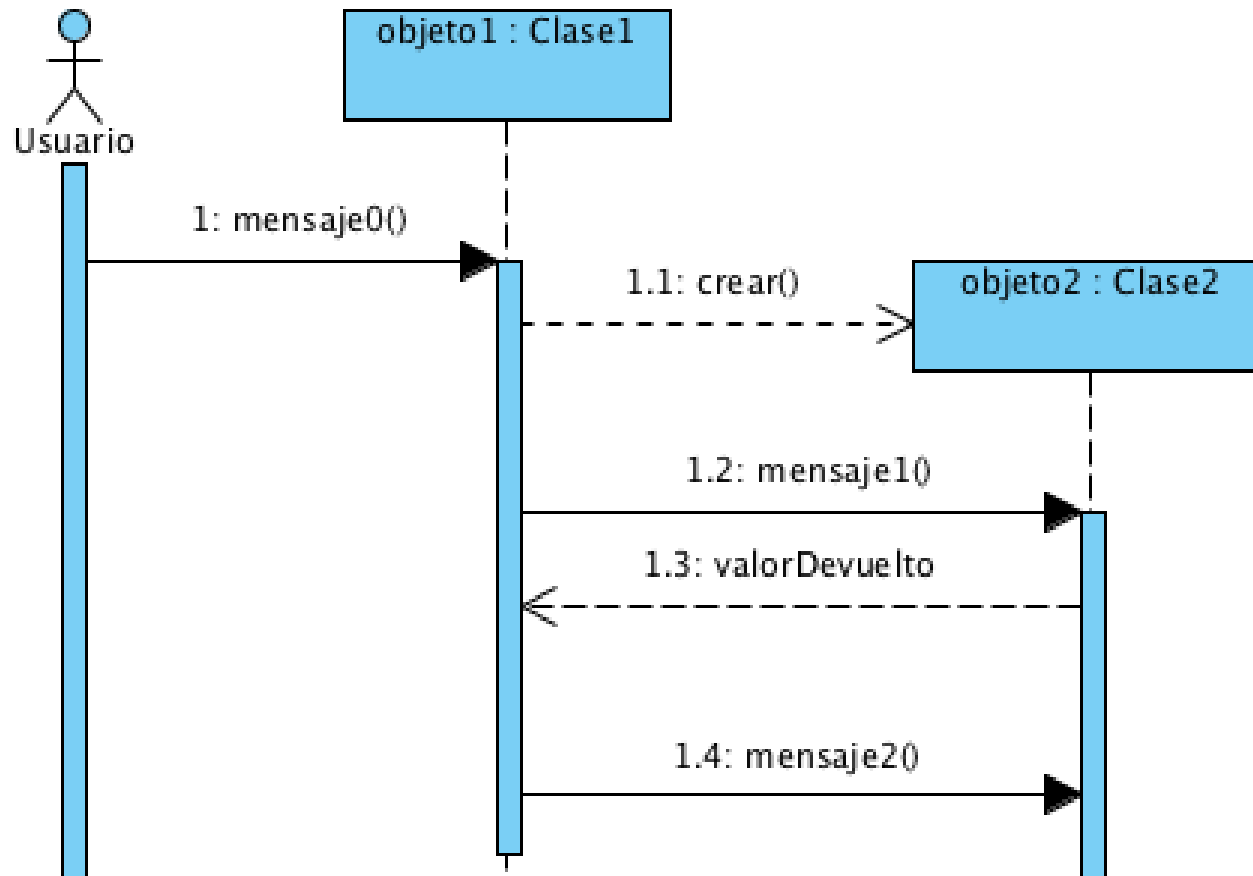
```
class Clase2{  
    void mensaje(){ }  
}
```



```
class Clase2  
    def mensaje  
    end  
end
```



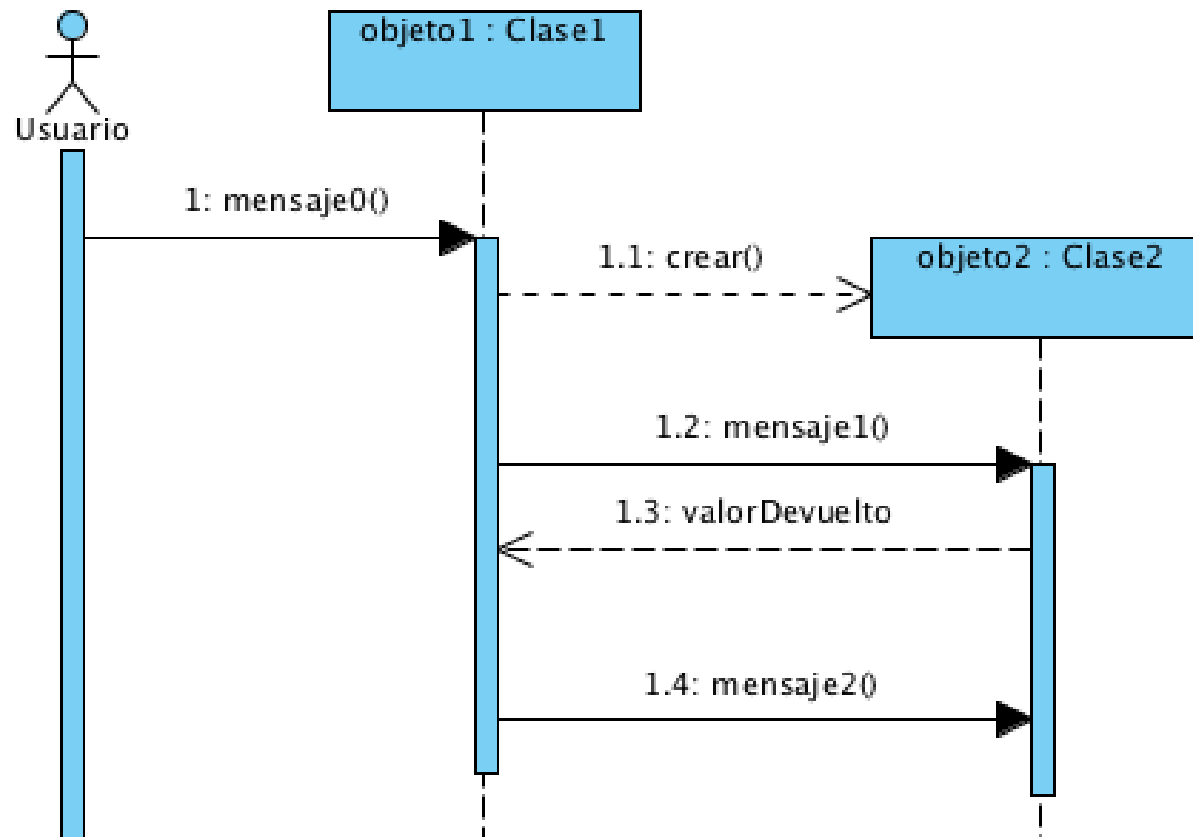
3. Implementación de DS



```
class Clase1{
    public void mensaje0() {
        Clase2 objeto2 = new Clase2();
        Clase3 valorDevuelto=objeto2.mensaje1();
        objeto2.mensaje2();
    }
}
```



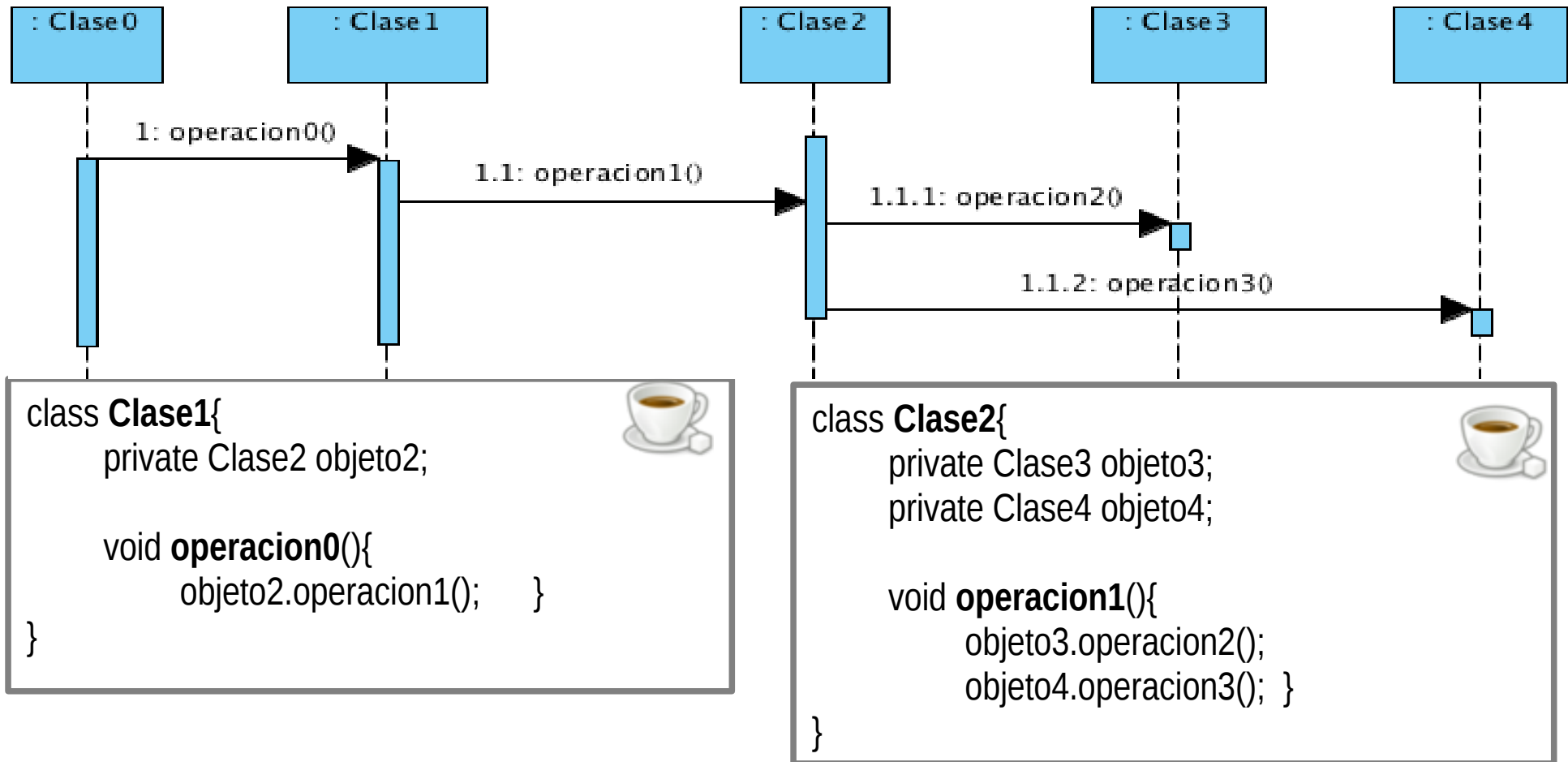
3. Implementación de DS



```
class Clase1
  def mensaje0
    objeto2 = Clase2.new
    valorDevuelto=objeto2.mensaje1
    objeto2.mensaje2
  end
end
```



3. Implementación de DS

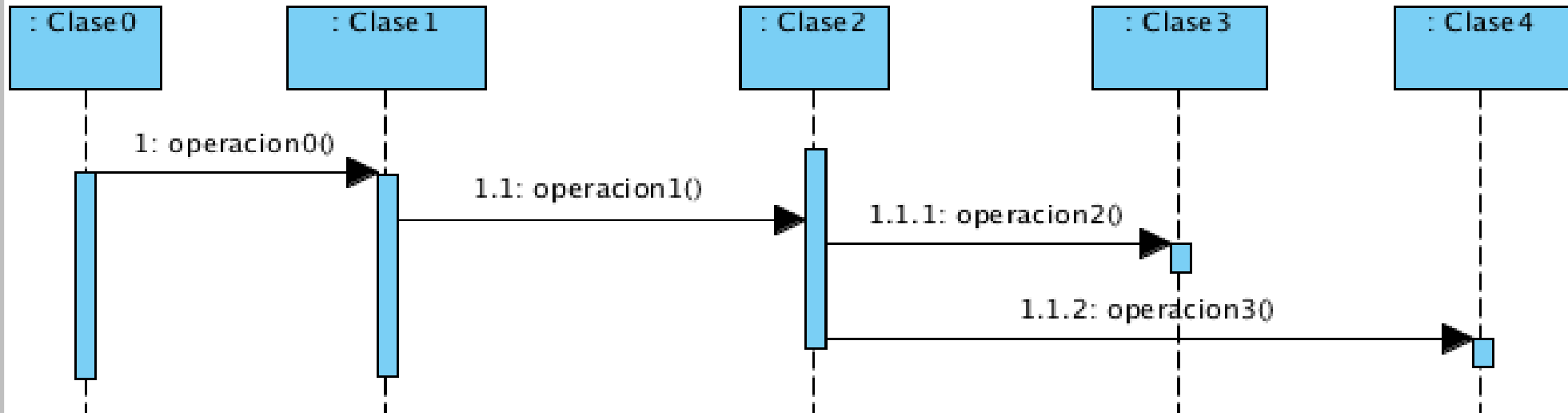


Alternativas

Dependiendo del diagrama de clases asociado, objeto3 y objeto4 podrían ser también:

variables locales en *operacion1* o **argumentos** de *operacion1*

3. Implementación de DS



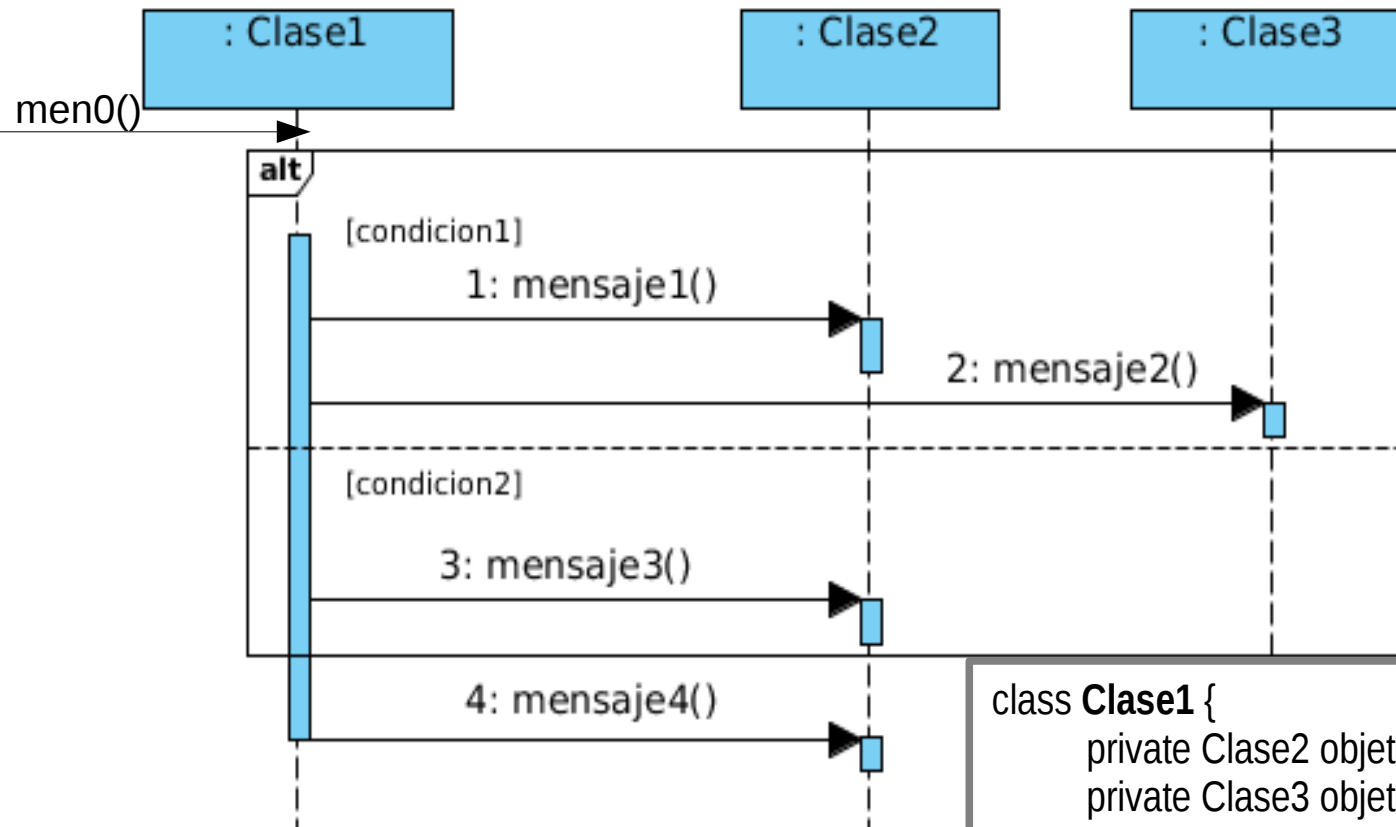
```
class Clase1
  ...
  def operacion0
    @objeto2.operacion1
  end
end
```

```
class Clase2
  ...
  def operacion1
    @objeto3.operacion2
    @objeto4.operacion3
  end
end
```


Alternativa:

```
def operacion1
  objeto3=Clase3.new
  objeto4=Clase4.new
  objeto3.operacion2
  objeto4.operacion3
end
```

3. Implementación de DS



De forma alternativa, y dependiendo del diagrama de clases asociado, objeto2 y objeto3 podrían ser variables locales

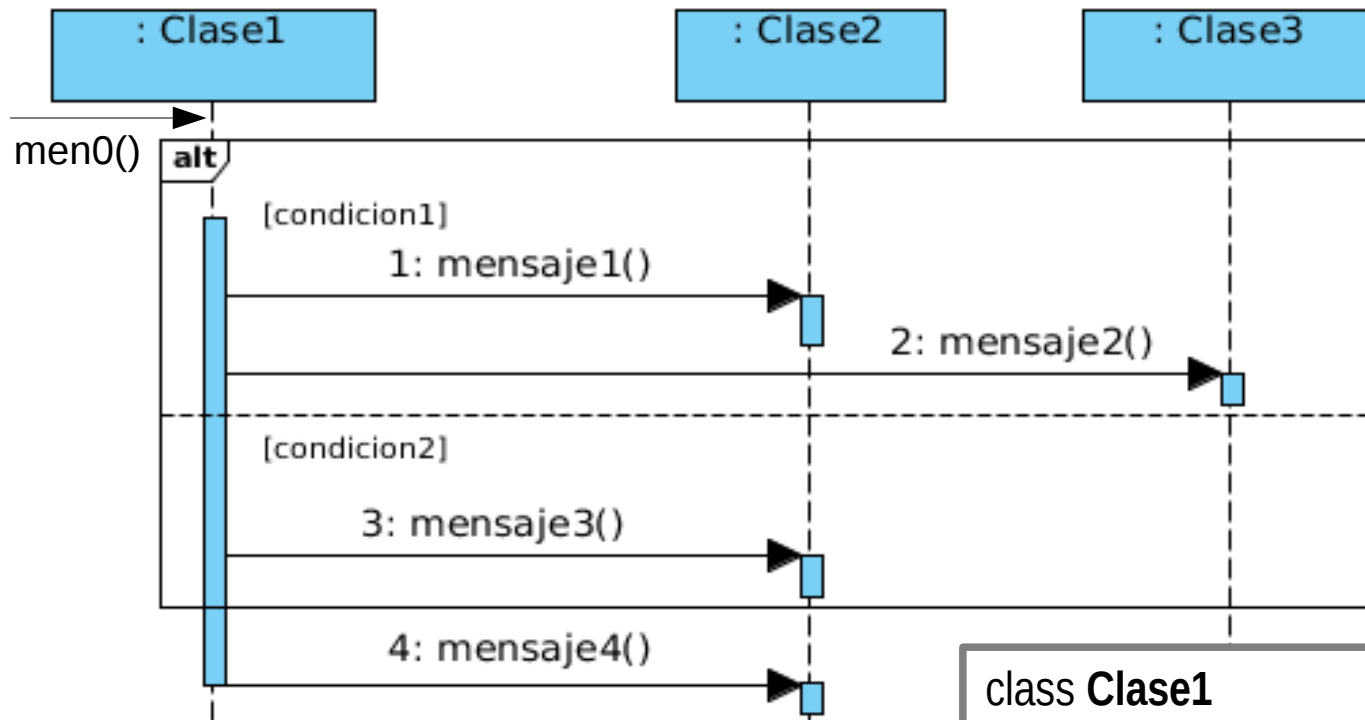


```
class Clase1 {
    private Clase2 objeto2;
    private Clase3 objeto3;

    void men0(){
        if(condicion1){
            objeto2.mensaje1();
            objeto3.mensaje2();
        } else {
            if(condicion2)
                objeto2.mensaje3();
        }
        objeto2.mensaje4();
    }
}
```

}}

3. Implementación de DS




De forma alternativa, y dependiendo del diagrama de clases asociado, objeto2 y objeto3 podrían ser variables locales

```
class Clase1
  def men0
    if(condicion1)
      @objeto2.mensaje1
      @objeto3.mensaje2
    elsif(condicion2)
      @objeto2.mensaje3
    end
    @objeto2.mensaje4
  end
end
```



3. Implementación de DS

Cuando las condiciones se correspondan con comprobación de valores (`var == valorx`), entonces el paso a código se puede hacer con un switch:




```
class Clase1{
  private Clase2 objeto2;
  private Clase3 objeto3;

  void men0(){
    switch(var) {

      case valor1:
        objeto2.mensaje1();
        objeto3.mensaje2();
        break;

      case valor2:
        objeto2.mensaje3();
        break;

    }
    objeto2.operacion4()
  }
}
```

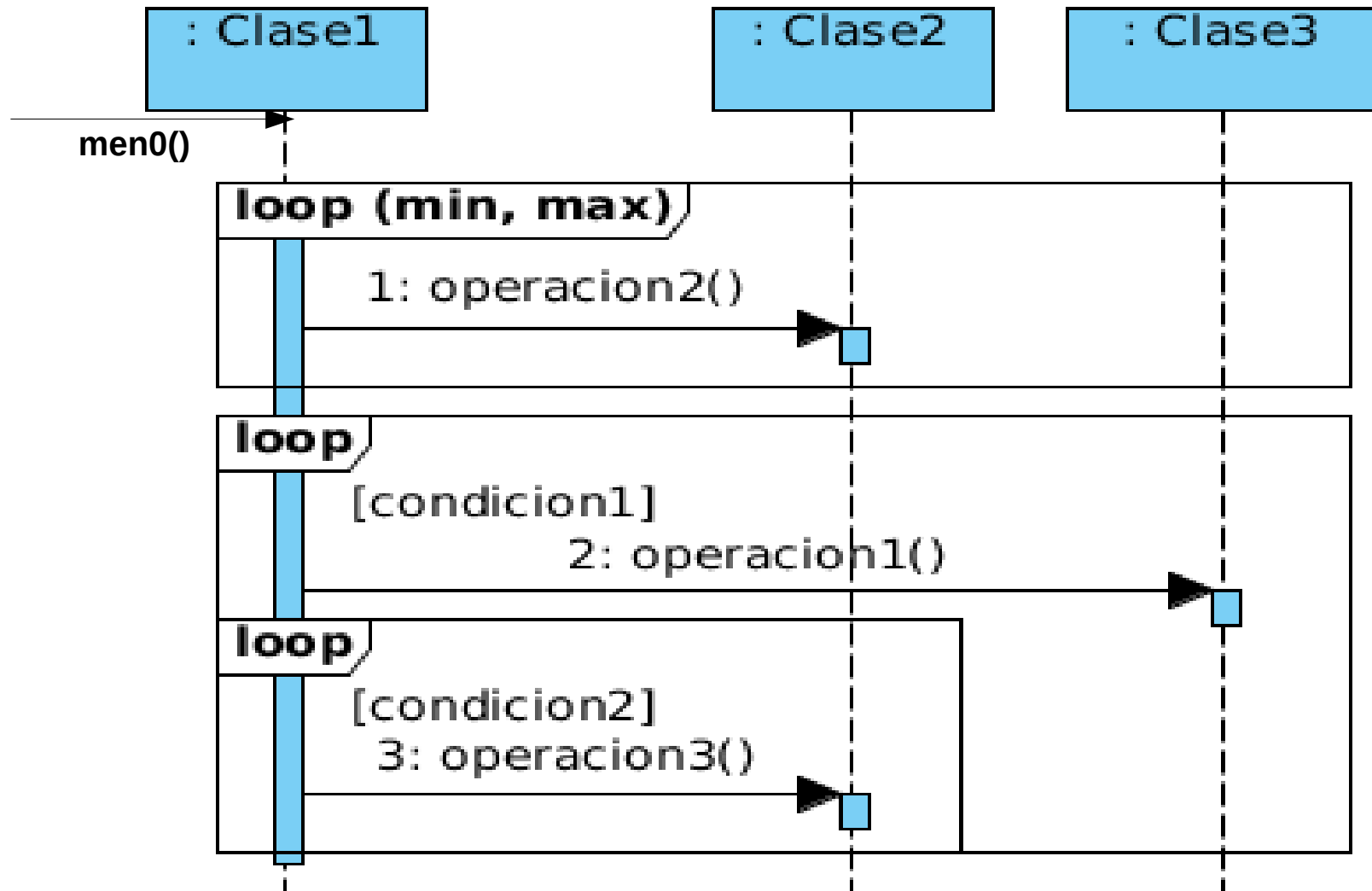


```
class Clase1

  def men0
    case var

      when valor1
        @objeto2.mensaje1
        @objeto3.mensaje2
      when valor2
        @objeto2.mensaje3
      end
      @objeto2.operacion4
    end
  end
end
```

3. Implementación de DS



3. Implementación de DS

Código del diagrama anterior

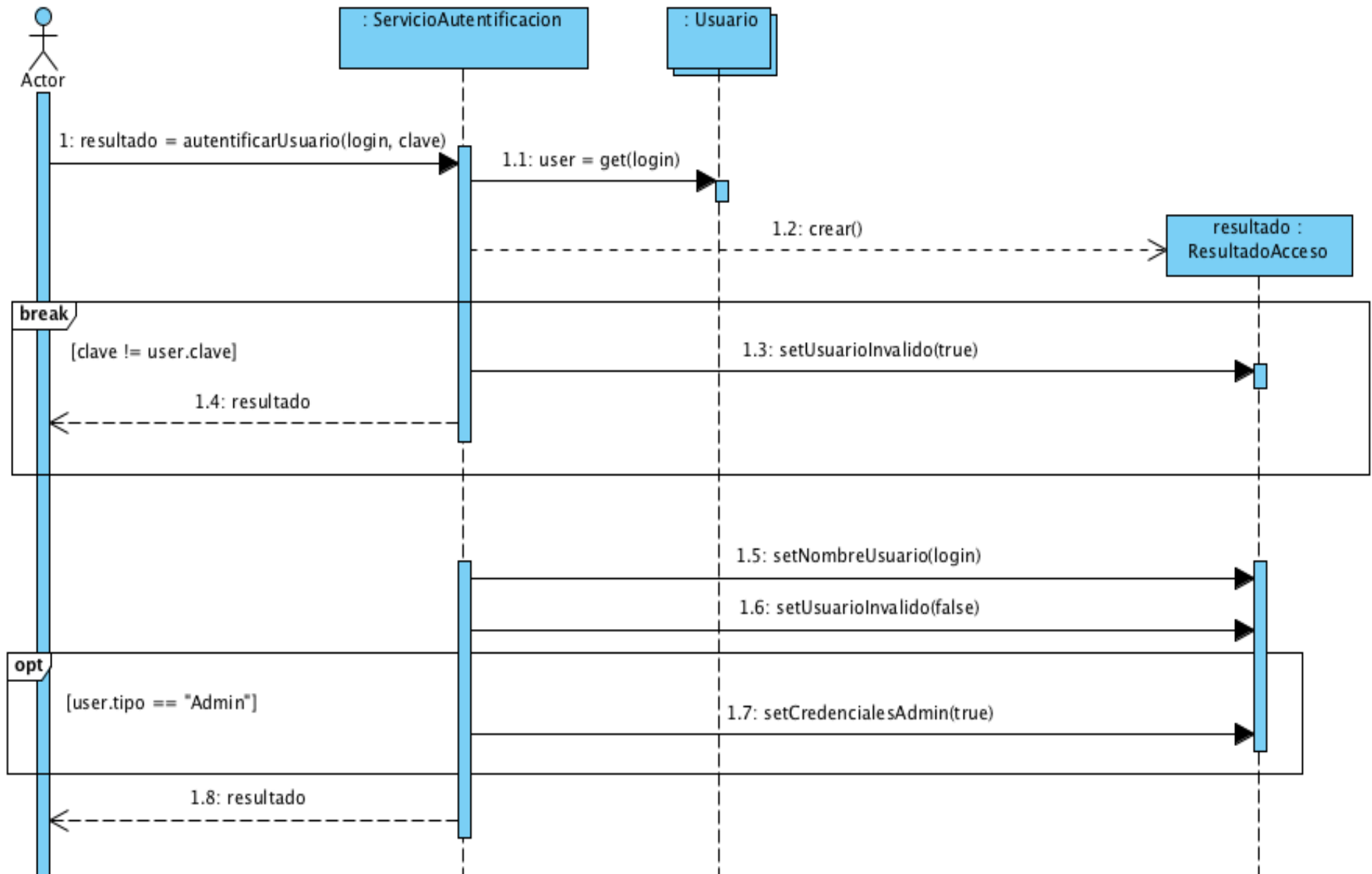


```
class Clase1{  
    private Clase2 objeto2;  
    private Clase3 objeto3;  
    void men0(){  
        for(int i = min; i<=max; i++)  
            objeto2.operacion2();  
        while(condicion1){  
            objeto3.operacion1();  
            while(condicion2)  
                objeto2.operacion3();  
        }  
    }  
}
```



```
class clase1  
    men0()  
        for i in min..max  
            @objeto2.operacion2  
        end  
        while(condicion1)  
            @objeto3.operacion1  
            while(condicion2)  
                @objeto2.operacion3  
            end  
        end  
    end  
end
```

3. Implementación de DS



3. Implementación de DS

Código del diagrama anterior

```
class ServicioAutenticacion{  
    public ResultadoAcceso autenticarUsuario(String login, String clave){  
        Usuario user = listaUsuarios.get(login);  
        ResultadoAcceso resultado = new ResultadoAcceso();  
        if(!clave.equals(user.getClave())){  
            resultado.setUserInvalido(true);  
            return resultado;  
        }  
        resultado.setNombreUsuario(login);  
        resultado.setUserInvalido(false);  
        if(user.getTipo().equals("Admin"))  
            resultado.setCredencialesAdmin(true);  
        return resultado;  
    }  
}
```



3. Implementación de DS

Código del diagrama anterior

```
class ServicioAutenticacion

  def autenticarUsuario(login, clave)
    user = listaUsuarios[login]
    resultado = ResultadoUsuario.new

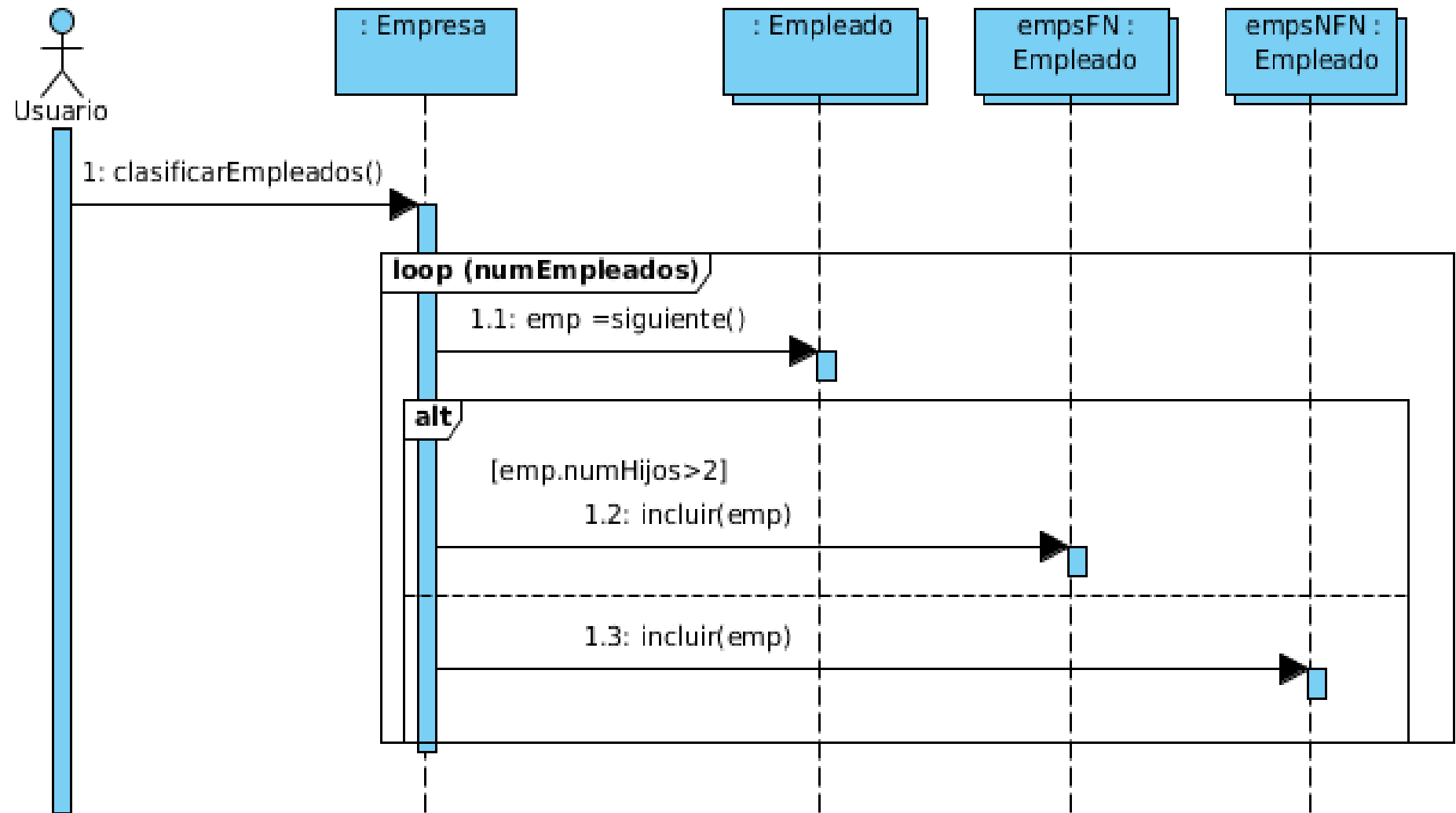
    if(!clave.eql?(user.clave))
      resultado.userInvalido=true
      return resultado
    end

    resultado.nombreUsuario=login
    resultado.userInvalido=false
    if(user.tipo.eql?("Admin"))
      resultado.credencialesAdmin=true
    end

    return resultado
  end
end
```



3. Implementación de DS



3. Implementación de DS

Código del diagrama anterior

```
class Empresa{
    private ArrayList<Empleado> empleados = new ArrayList();
    private ArrayList<Empleado> empsFN = new ArrayList();
    private ArrayList<Empleado> empsFNN = new ArrayList();
    public void clasificarEmpleados() {
        for(Empleado emp: empleados) {
            if(emp.numHijos>2)
                empsFN.add(emp);
            else
                empsFNN.add(emp);
        }
    }
}
```



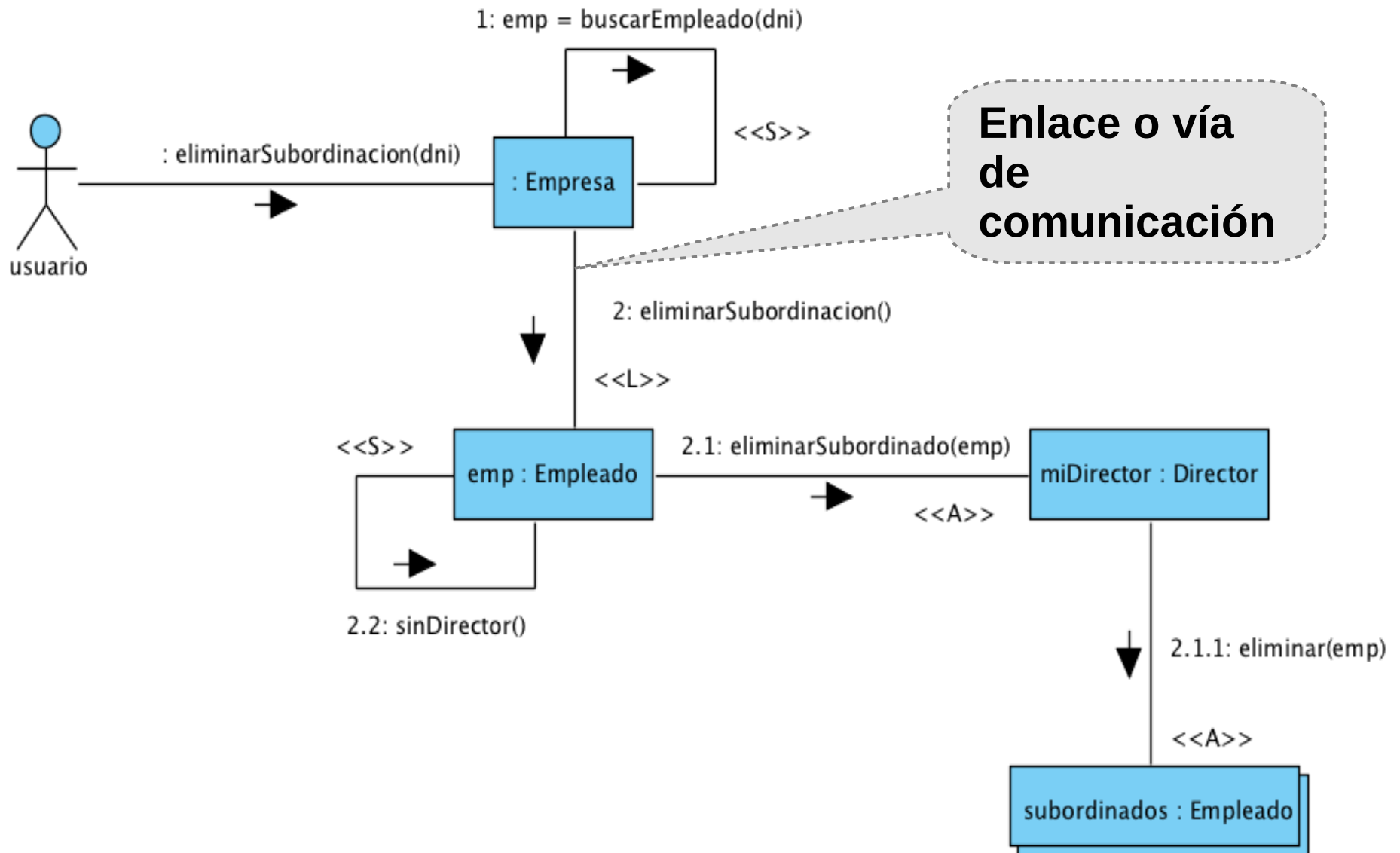
```
class Empresa
  def initialize
    @empleados = Array.new
    @empsFN = Array.new
    @empsFNN = Array.new
  end
  def clasificarEmpleados
    @empleados.each do |emp|
      if(emp.numHijos>2)
        @empsFN.push(emp)
      else
        @empsFNN.push(emp)
      end
    end
  end
end
```



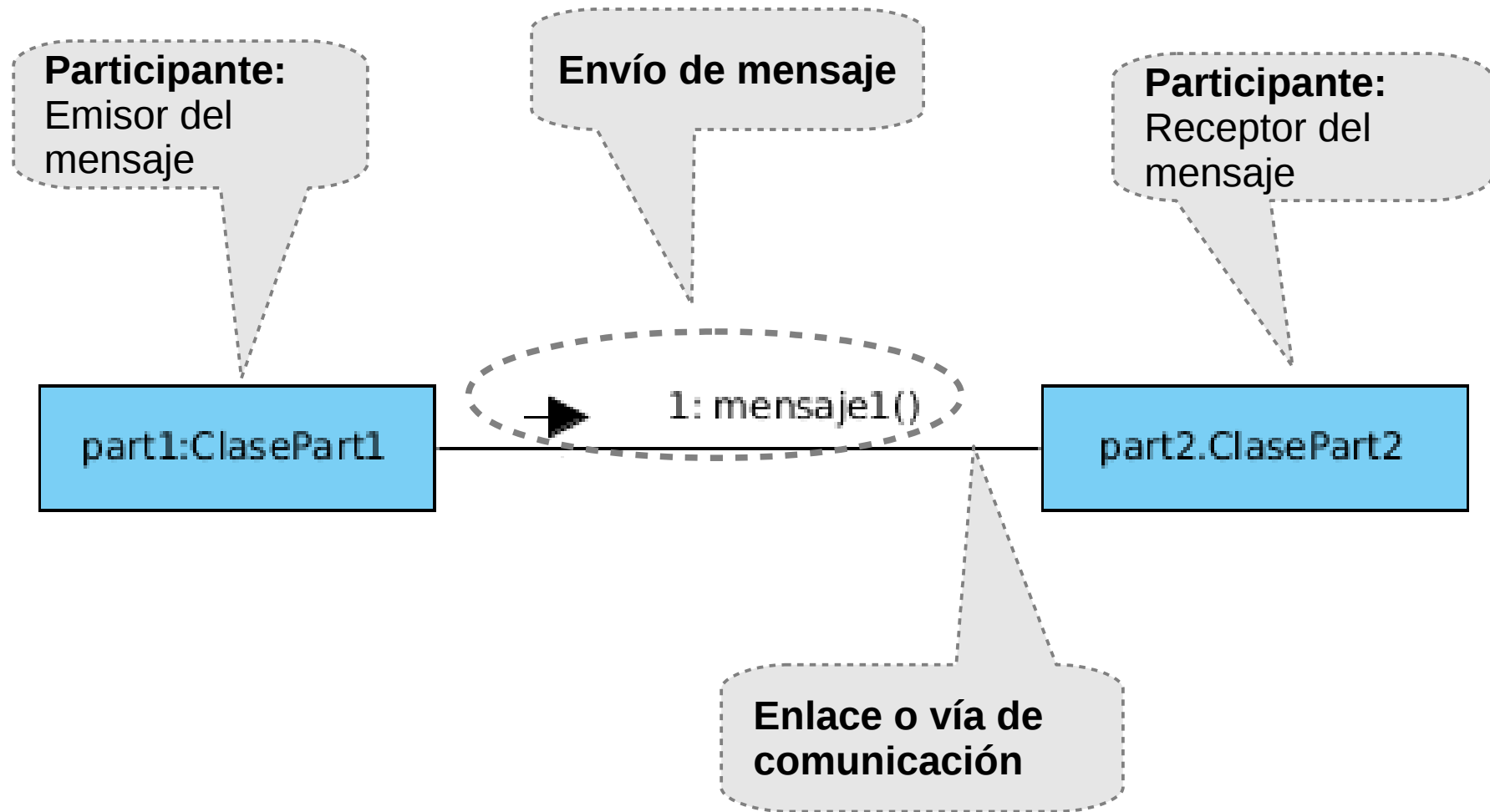
4. Diagramas de comunicación

- Muestran de forma **visual muy clara los vías de comunicación** que deben darse entre participantes para que pueda llevarse a cabo el envío de mensaje entre ellos.
- El orden temporal en el que ocurren los envíos de mensaje es un elemento secundario.
- El uso de diagramas de secuencia o comunicación depende de las preferencias del diseñador.
- Componentes de un diagrama de comunicación:
 - **Participantes.**
 - **Enlaces** o vías de comunicación entre participantes.
 - Envío de **mensajes.**

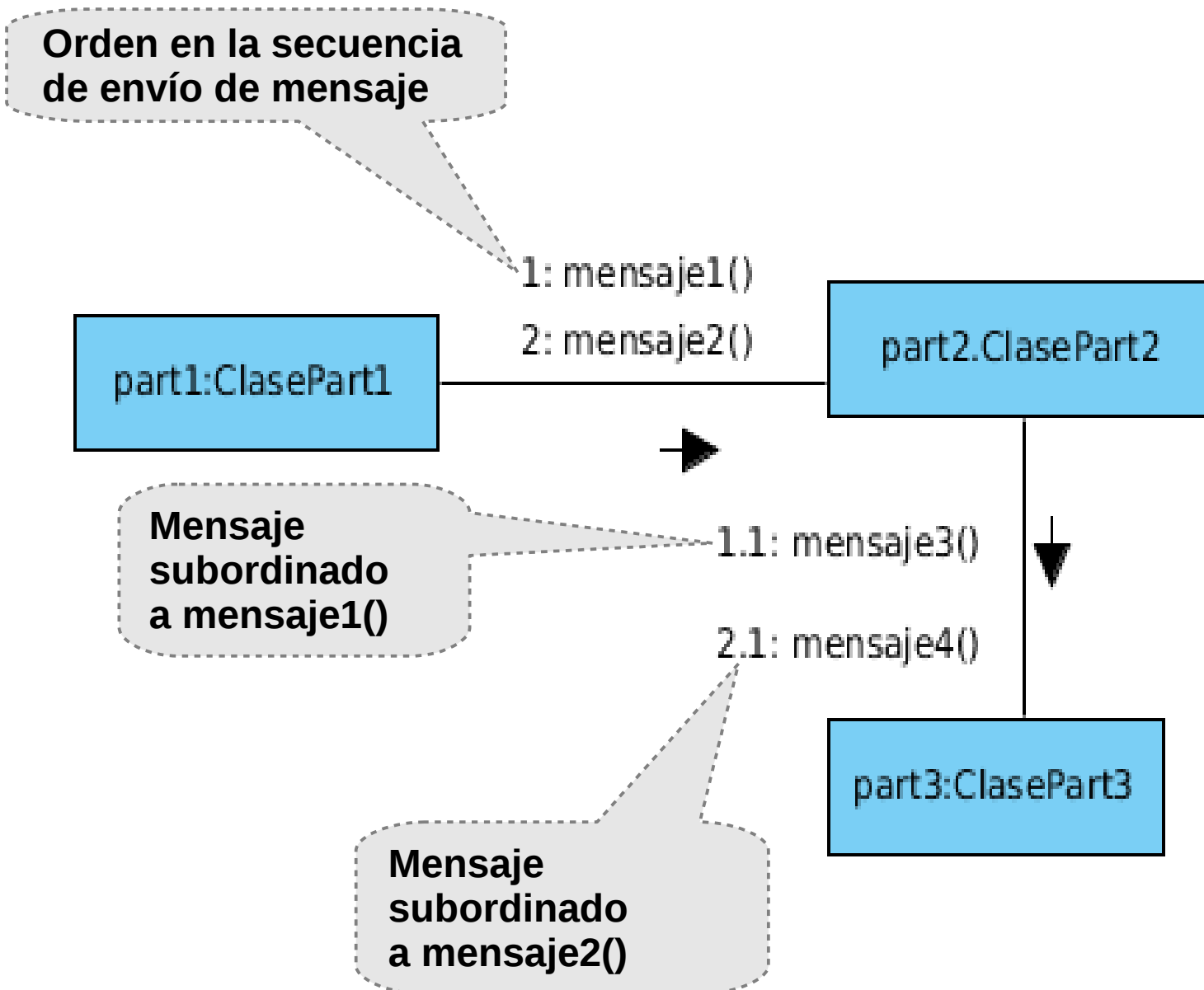
4. Diagramas de comunicación



4. Diagramas de comunicación: Envío de mensaje



4. Diagramas de comunicación: Envío de mensaje



4. Diagramas de comunicación: Tipos de enlace

Para que un objeto o participante (objetoX) pueda enviar un mensaje a otro (objetoY) deben conocerse, es decir, entre ellos debe haber una vía de comunicación o enlace.

Un enlace puede ser:

- **Global (G)**: el ámbito de objetoY es superior al del objetoX.
- **Asociación (A)**: entre los dos objetos existe una relación fuerte y duradera en el tiempo.
- **Parámetro (P)**: el objetoY es pasado como parámetro a un método del objetoX.
- **Local (L)**: el objetoY es referenciado dentro de un método del objetoX.
- **Self (S)**: el objetoX siempre se conoce a sí mismo.

4. Diagramas de comunicación: Tipos de enlace

Tipo de conocimiento o vías de comunicación de los objetos de una clase (Ejemplo) hacia otros objetos que se definen en ella (A modo de recordatorio del tema 2.1)

```
public class Ejemplo {  
    private static ClaseA variableGlobal;  
    private ClaseB variableAsociacion;
```

```
    public void metodo(ClaseC variableParametro) {
```

```
        ClaseF variableLocal;
```

```
        variableGlobal.operacion1();
```

```
        variableAsociacion.operacion2();
```

```
        variableParametro.operacion3();
```

```
        variableLocal.operacion4();
```

```
        this.operacion5();
```

```
    }
```

```
}
```

**Conocimiento
global**

**Conocimiento
asociación**

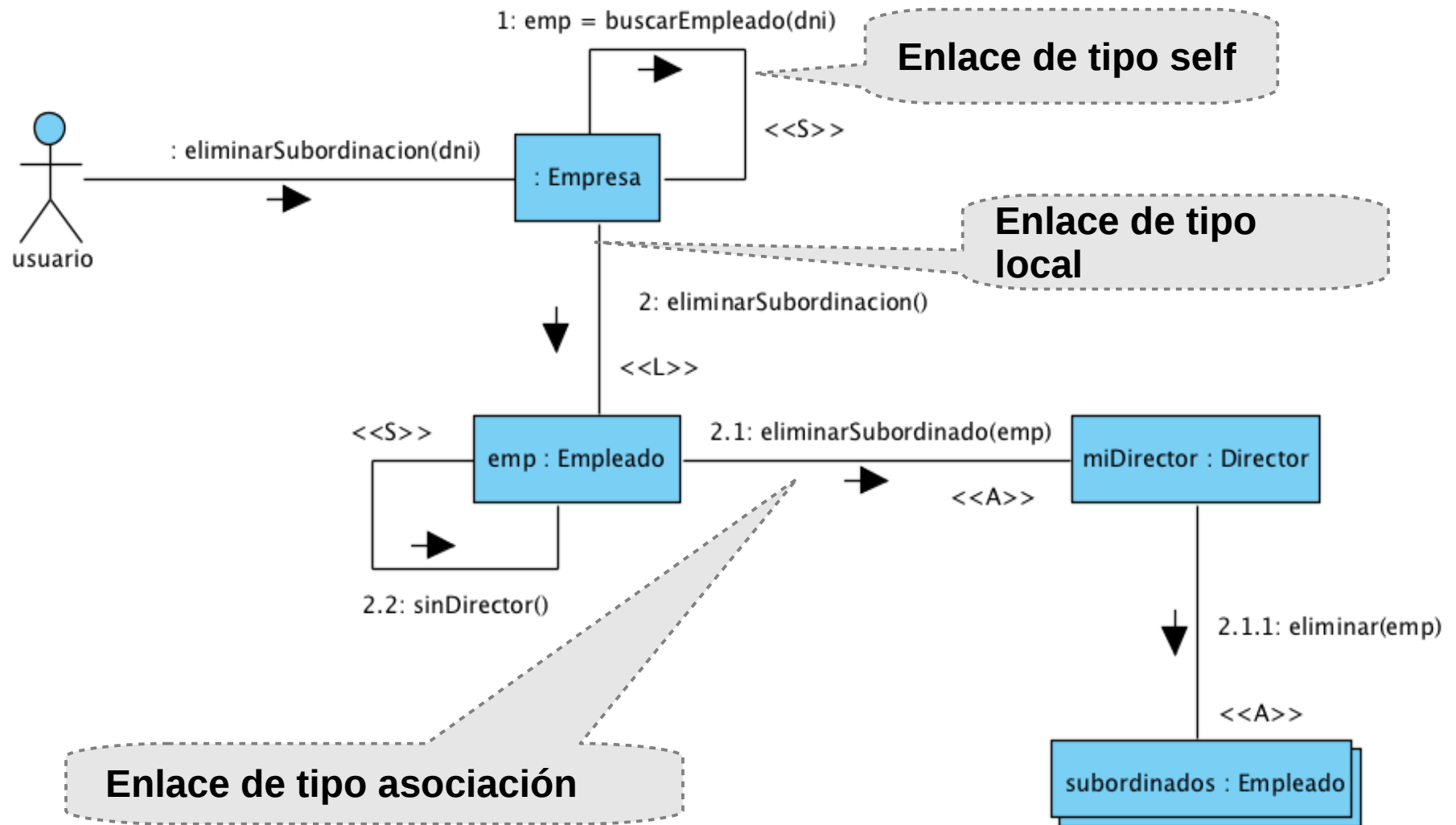
**Conocimiento
local**

**Conocimiento
parámetro**

**Conocimiento
self**

4. Diagramas de comunicación: Tipos de enlace

La forma de indicar el tipo de comunicación o enlace es a través de **estereotipos de visibilidad**, **<< >>**, sobre los enlaces en los diagramas.



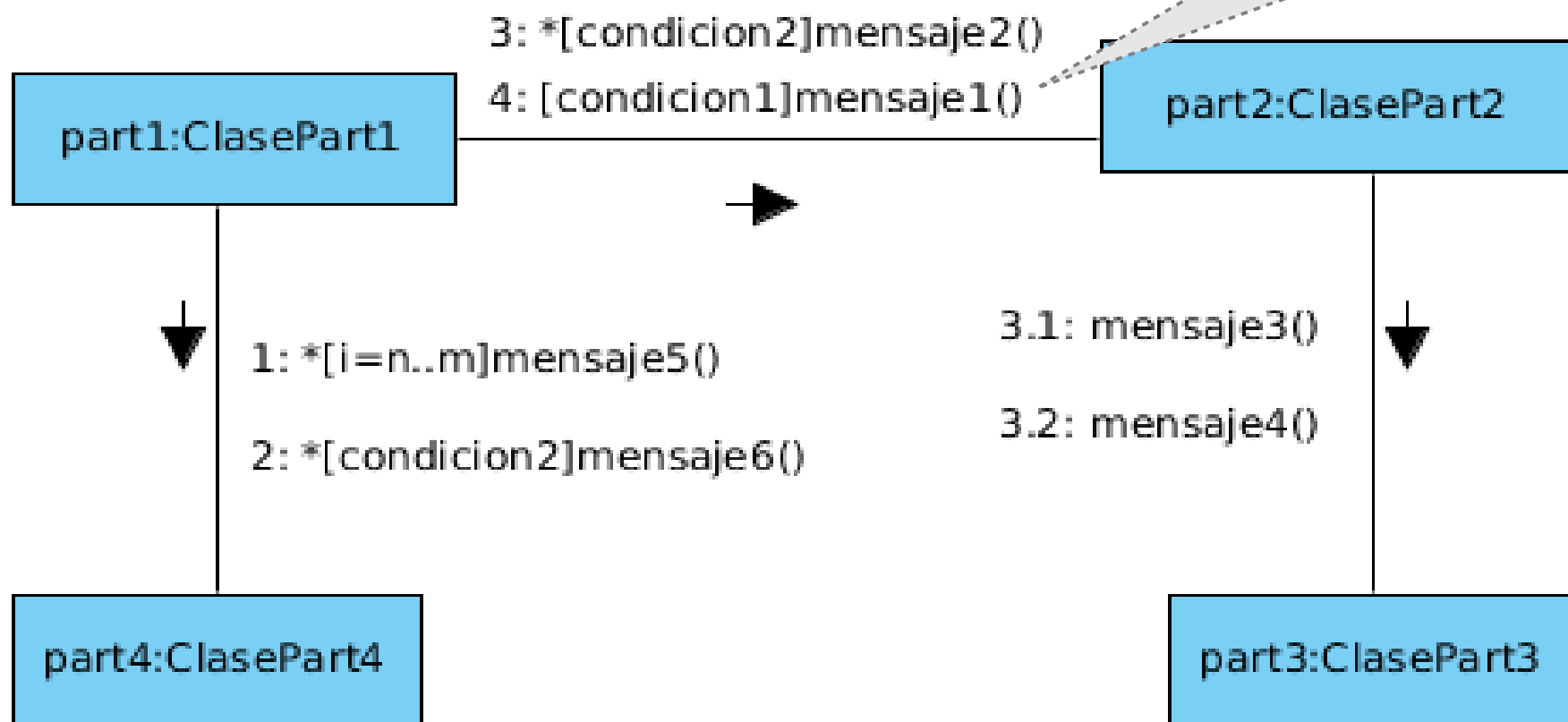
4. Diagramas de comunicación: Estructuras de control

Representación de las estructuras de control:

- **Selectivas**: anteponiendo al envío de mensaje [condición]
- **Iterativas**: anteponiendo al envío de mensaje *[condición]

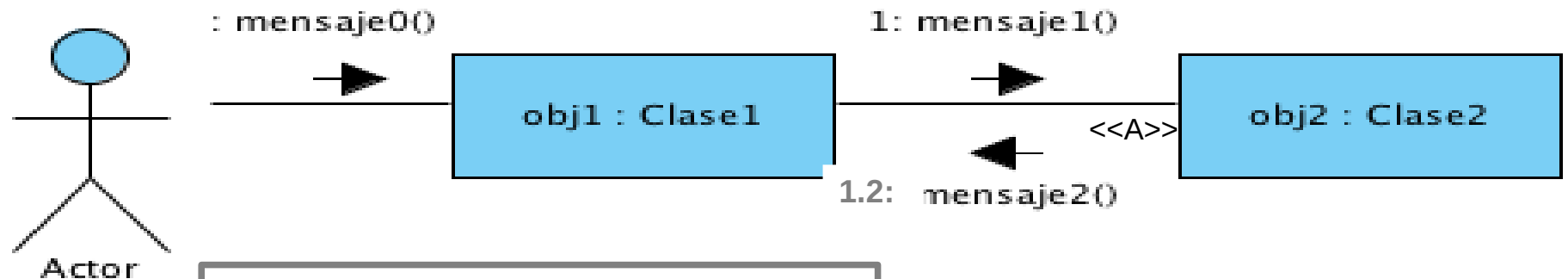
Iterativa: Mientras que se cumpla condicion2, part1 envía el mensaje mensaje2() a part2

Selectiva: Si se cumple condicion1, part1 envía el mensaje mensaje1() a part2



5. Implementación de DC

Implementación diagramas de comunicación



```
class Clase1{
    private Clase2 obj2;
    void mensaje0(){
        obj2.mensaje1();
    }
    void mensaje2(){ }
}
```

```
class Clase2{
    private Clase1 obj1;

    void mensaje1(){
        obj1.mensaje2();
    }
}
```


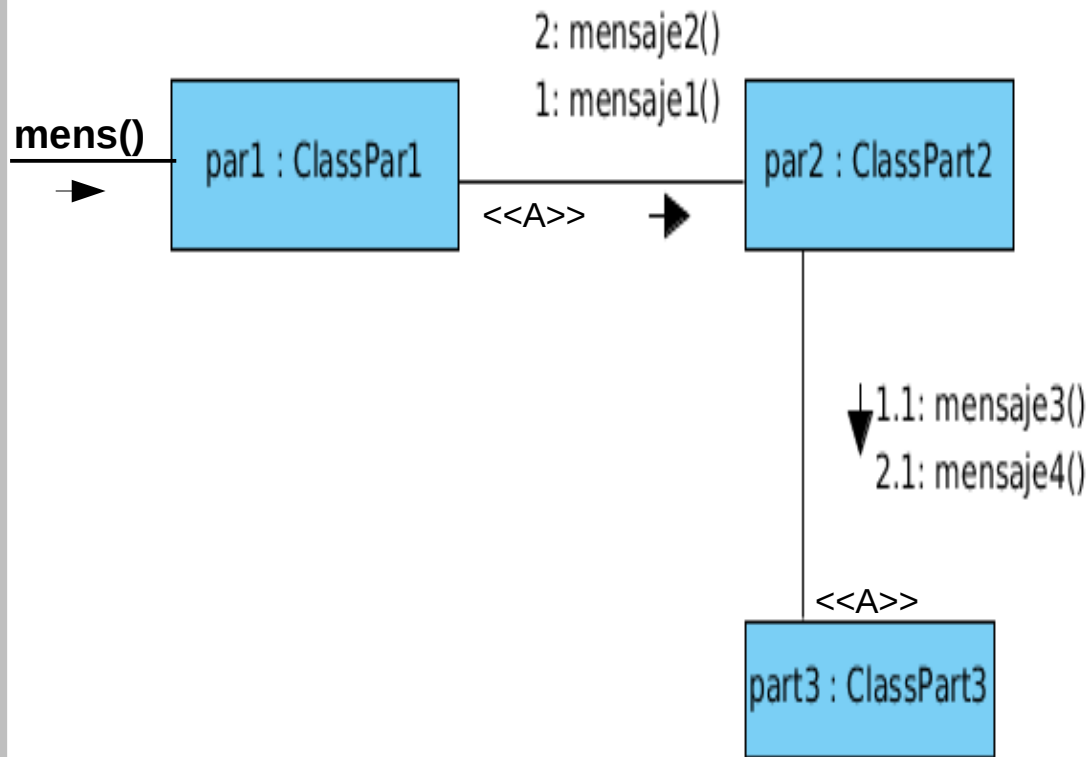
```
class Clase1
  def mensaje0
    @obj2.mensaje1
  end
  def mensaje2
  end
end
```

```
class Clase2
  def mensaje1
    @obj1.mensaje2
  end
end
```



5. Implementación de DC

Implementación diagramas de comunicación



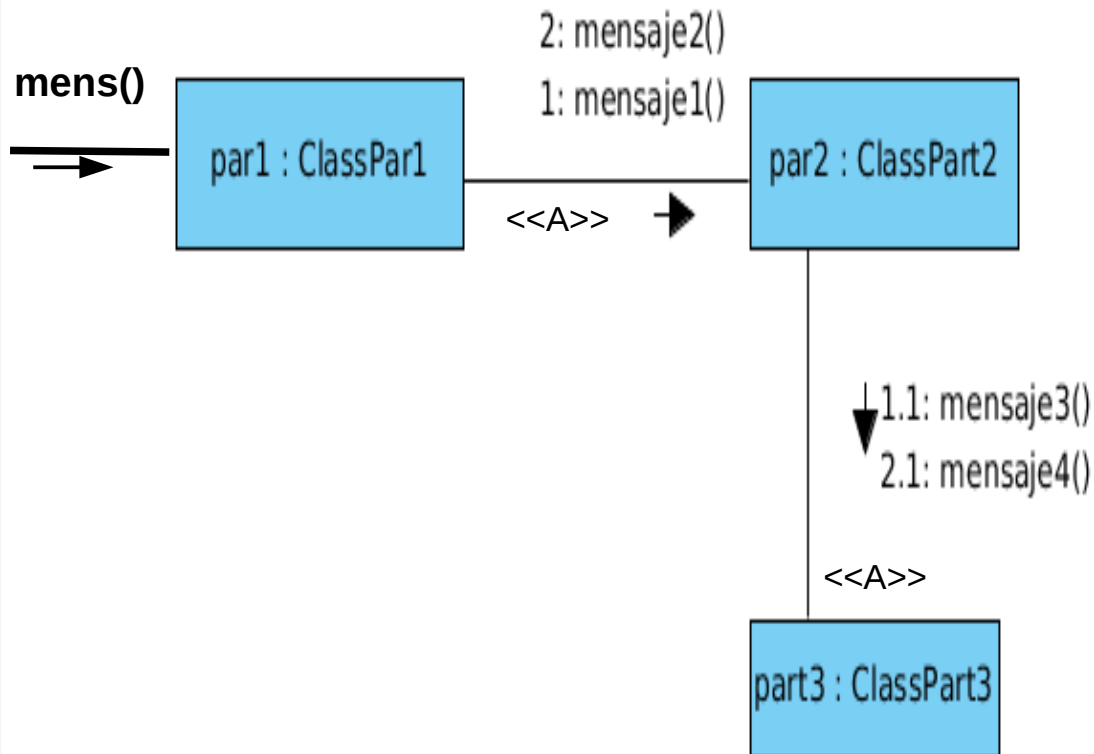
```
class ClassPar1{

    ClassPart2 par2;
    void mens(){
        par2.mensaje1();
        par2.mensaje2();
    }
}
class ClassPart2{

    ClassPart3 part3;
    void mensaje1(){
        part3.mensaje3();
    }
    void mensaje2(){
        part3.mensaje4();
    }
}
```


5. Implementación de DC

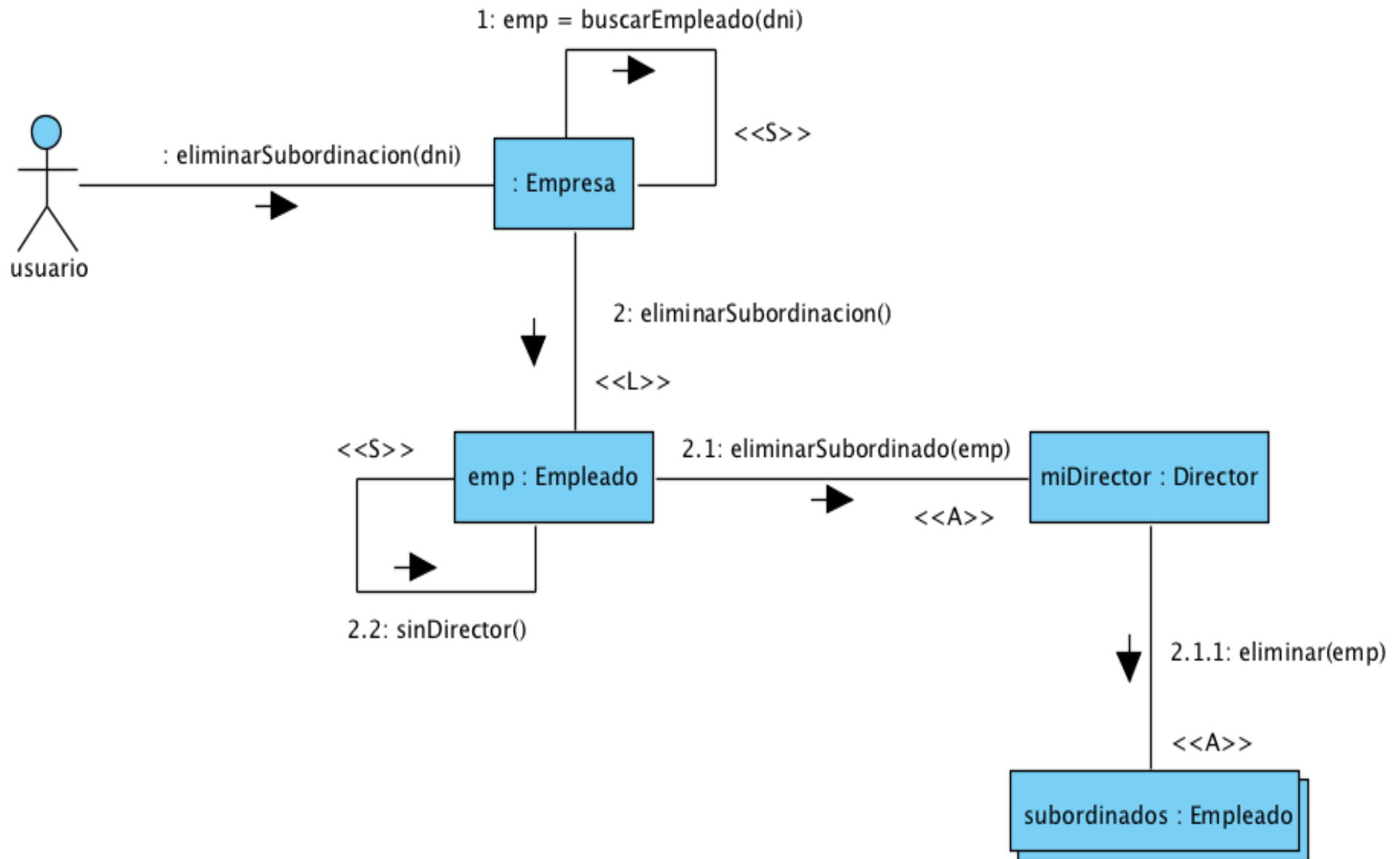
Implementación diagramas de comunicación



```
class ClassPar1
  def mens
    @par2.mensaje1
    @par2.mensaje2
  end
end
class ClassPart2
  def mensaje1
    @part3.mensaje3
  end
  def mensaje2
    @part3.mensaje4
  end
end
```



5. Implementación de DC



5. Implementación de DC

Implementación del diagrama de comunicación anterior

```
class Empresa{
    public void eliminarSubordinacion(String dni) {
        Empleado emp = buscarEmpleado(dni);
        emp.eliminarSubordinacion();    }}

class Empleado{
    private Director miDirector;
    public void eliminarSubordinacion(){
        miDirector.eliminarSubordinado(this);
        this.sinDirector();    }    }

class Director{
    private ArrayList<Empleado> subordinados;
    public void eliminarSubordinado(Empleado emp)
    {
        subordinados.remove(emp);    }}
```



5. Implementación de DC

Implementación del diagrama de comunicación anterior

```
class Empresa
  def eliminar_subordinacion(dni)
    emp = buscarEmpleado(dni)
    emp.eliminar_subordinacion
  end
end

class Empleado
  def eliminar_subordinacion
    @miDirector.eliminar_subordinado(self)
    self.sinDirector
  end
end

class Director
  def eliminar_subordinado(emp)
    @subordinados.delete(emp)
  end
end
```



6. Equivalencia entre diagramas

Diagrama de secuencia ↔ Diagrama de comunicación

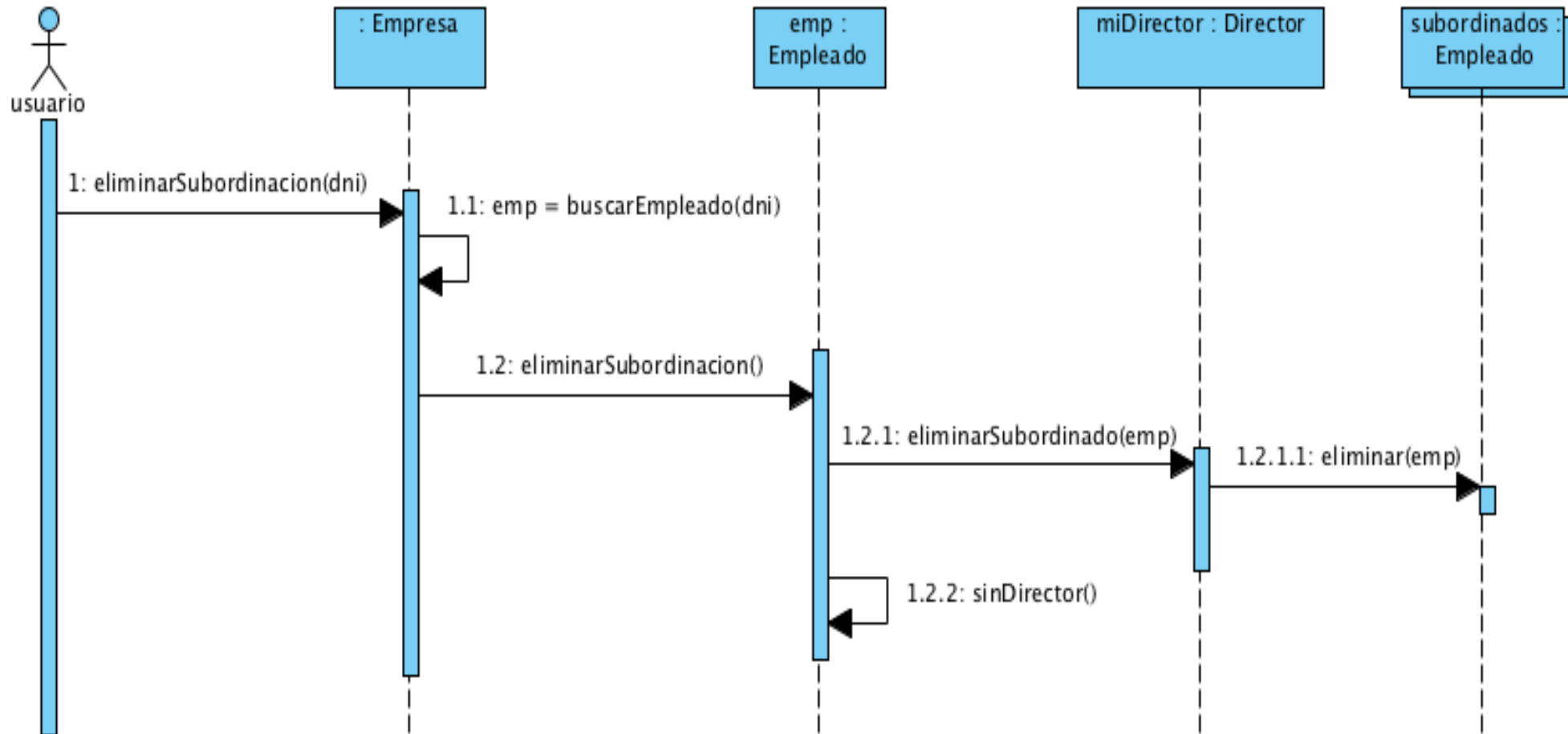


Diagrama de secuencia equivalente al diagrama de comunicación anterior

7. Modelando interacción entre objetos

Cuando el sistema es complejo y los objetos que intervienen en una operación son numerosos, hay que recurrir a una **metodología de diseño** que nos ayude. Para sistemas simples y con poco objetos que intervengan en la interacción, se recomienda:

Partir de:

- Diagrama de clases obtenido previamente, sobre todo a nivel de clases y sus atributos.
- De la descripción realizada de la operación.

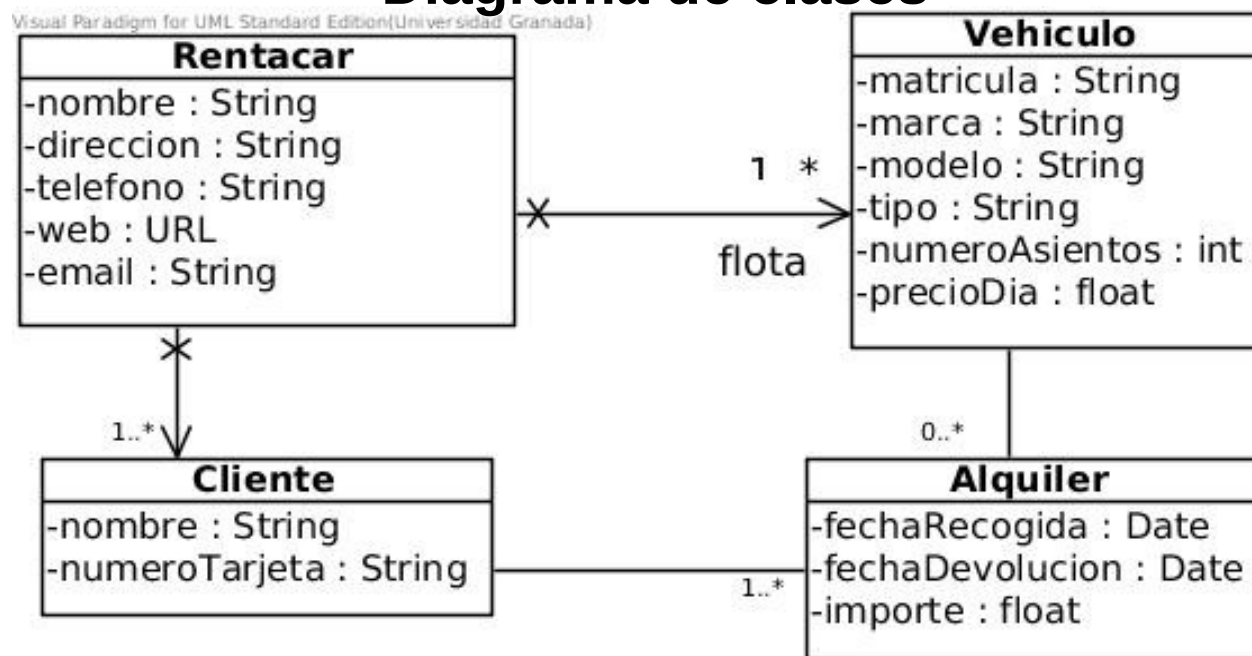
Y seguir el siguiente **Procedimiento**:

1. Identificar nombre y parámetros de la operación.
2. Identificar quién es el objeto responsable de realizar la operación y representar el primer nivel de envío de mensaje.
3. Identificar las responsabilidades de este objeto.
4. En base a estas responsabilidades, representar el primer nivel de subordinación.
5. Seguir con los siguientes niveles de subordinación, buscando las responsabilidades de los objetos que intervienen en este nivel.
6. Refinar el resultado obtenido.

7. Modelando interacción entre objetos: Ejemplo

Para ver el modelado de la interacción de objetos vamos a seguir con el ejemplo desarrollado en el tema 2.2: Agencia de alquiler de vehículos.

Diagrama de clases



Descripción de la operación

AlquilarVehiculo: Registrar el alquiler de un vehículo para un cliente.

El vehículo será el que previamente se ha localizado (matrícula) como libre para el día de recogida y el día de entrega indicados por el cliente.

El cliente proporcionará su nombre y su número de tarjeta.

Finalizada la operación debemos tener un objeto cliente nuevo y un objeto alquiler nuevo enlazado con el cliente y con el vehículo.

7. Modelando interacción entre objetos: Ejemplo

1. Parámetros de la operación:

Registrar el alquiler de un vehículo para un cliente. El vehículo será el que previamente se ha localizado (matrícula) como libre para el día de recogida y el día de entrega indicados por el cliente. El cliente proporcionará su nombre y su número de tarjeta. Finalizada la operación debemos tener un objeto cliente nuevo y un objeto alquiler nuevo enlazado con el cliente y con el vehículo.

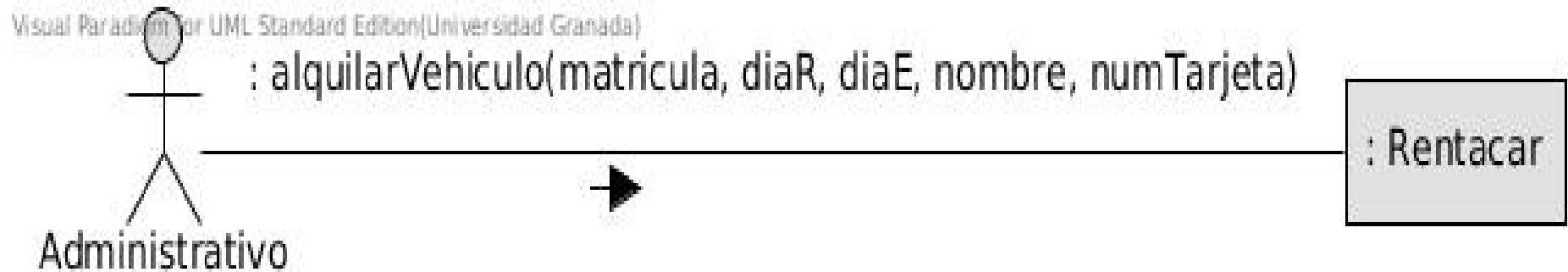
- Matrícula del vehículo para identificarlo (matricula)
- Día de recogida (diaR)
- Día de entrega (diaE)
- Nombre del cliente (nombre)
- Número de tarjeta (numTarjeta)

Resultado:

**alquilarVehiculo(matricula:String, diaR:Date, diaE:Date, nombre:String,
numTarjeta:String)**

7. Modelando interacción entre objetos: Ejemplo

2. Máximo responsable: Rentacar (es el que debe iniciar la operación).
Primer envío de mensaje



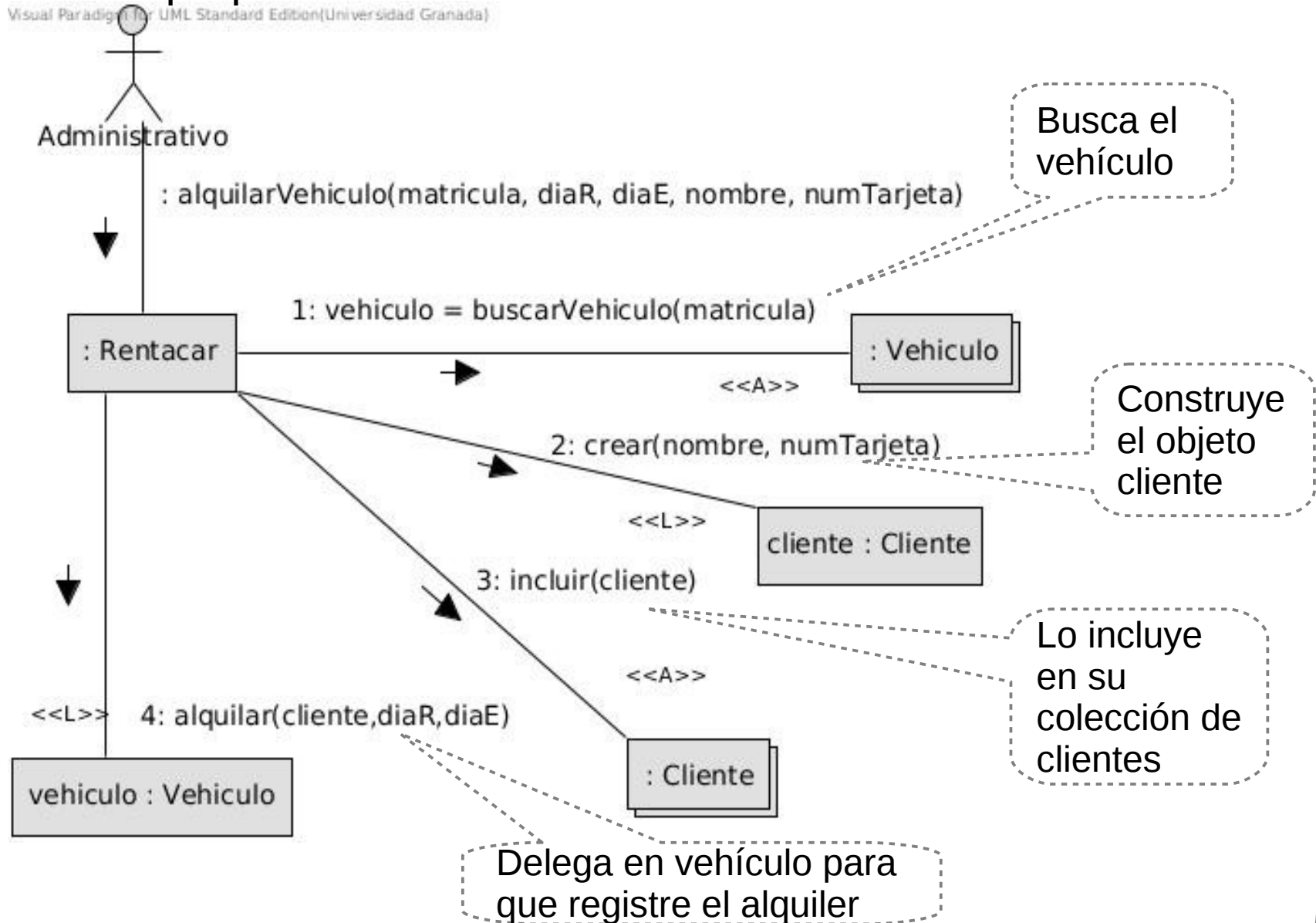
3. Identificar responsabilidades del objeto Rentacar:

- Buscar al vehículo cuya matrícula es la proporcionada.
- Crearse el objeto Cliente con los valores de inicialización dados e incluirlo en su lista de clientes con vehículos alquilados.
- Delegar en el vehículo encontrado su alquiler.

7. Modelando interacción entre objetos: Ejemplo

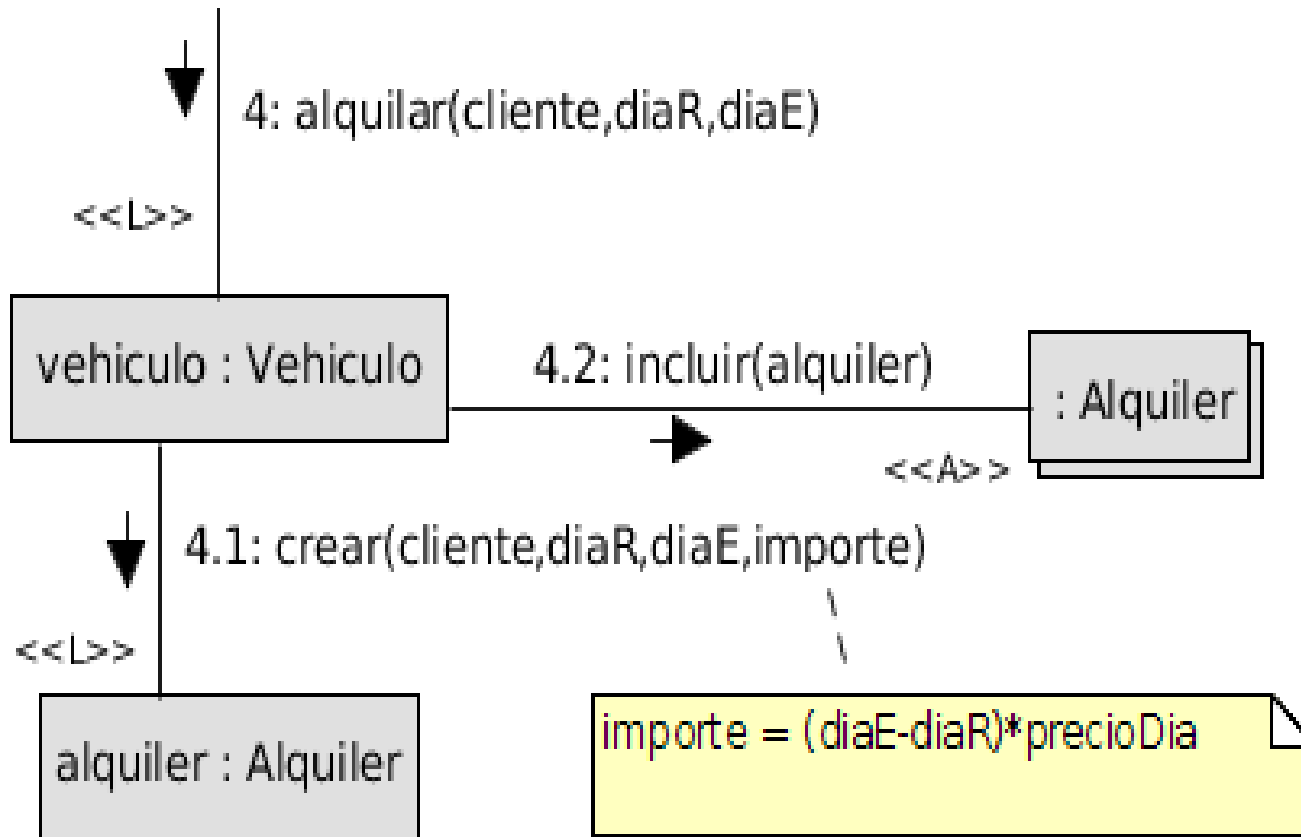
4. Representar primer nivel de subordinación que surge de la responsabilidad proporcionada a Rentacar

Visual Paradigm for UML Standard Edition (Universidad Granada)



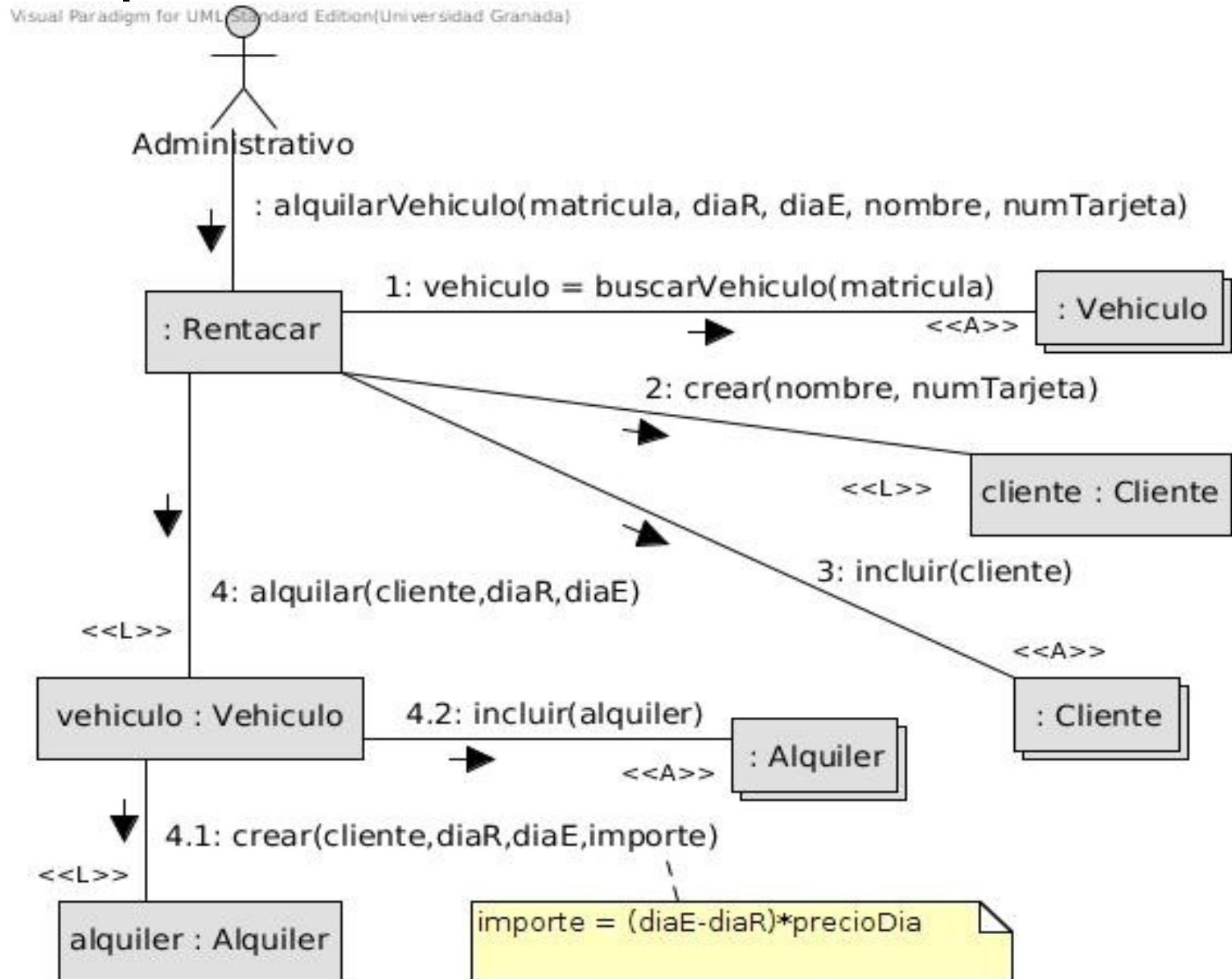
7. Modelando interacción entre objetos: Ejemplo

5. Sigüientes niveles de subordinación. Responsabilidad de vehículo: crear el objeto alquiler e incluirlo en su colección de alquileres



7. Modelando interacción entre objetos: Ejemplo

Diagrama completo



7. Ejercicios



1. Refinar el diagrama anterior teniendo en cuenta criterios de visibilidad entre los objetos.
2. Obtener otro diagrama de comunicación distinto a éste para esta misma operación.
3. Obtener el diagrama de secuencia equivalente.
4. Implementar en Java y en Ruby en diagrama de comunicación anterior.
5. Siguiendo con el mismo ejemplo, obtener el diagrama de comunicación de las operaciones correspondientes a:
 - Devolver un vehículo
 - Consultar la disponibilidad de un determinado tipo de vehículo para unas fechas.
 - Ver qué devoluciones de vehículos se van a dar en un día concreto.