# Proceso de implementación a partir de un diseño.

Semana 4

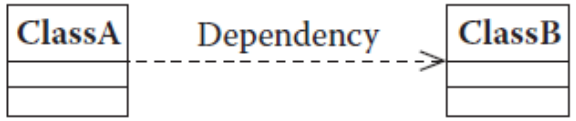
# Agenda

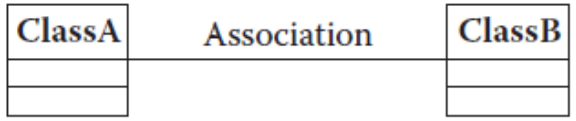
- Traduciendo un diagrama de clase en código fuente y viceversa
- Traduciendo un diagrama de secuencia en código fuente y viceversa

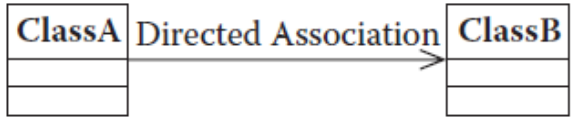
# Traduciendo un diagrama de clase en código fuente y viceversa

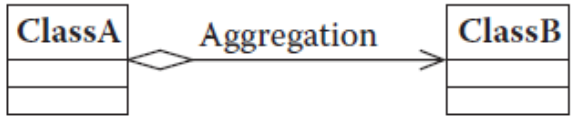
# Relaciones en Diagramas de Clases

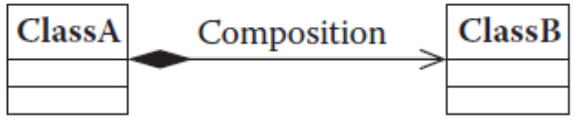
- Las clases deben estar relacionadas con otras.

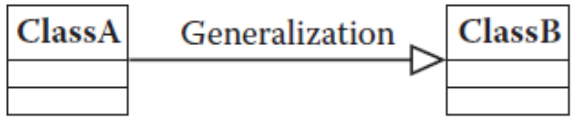
1. 

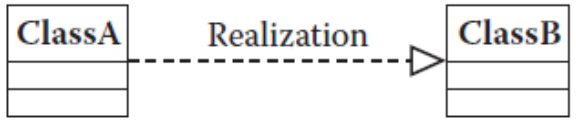

2. 


3. 

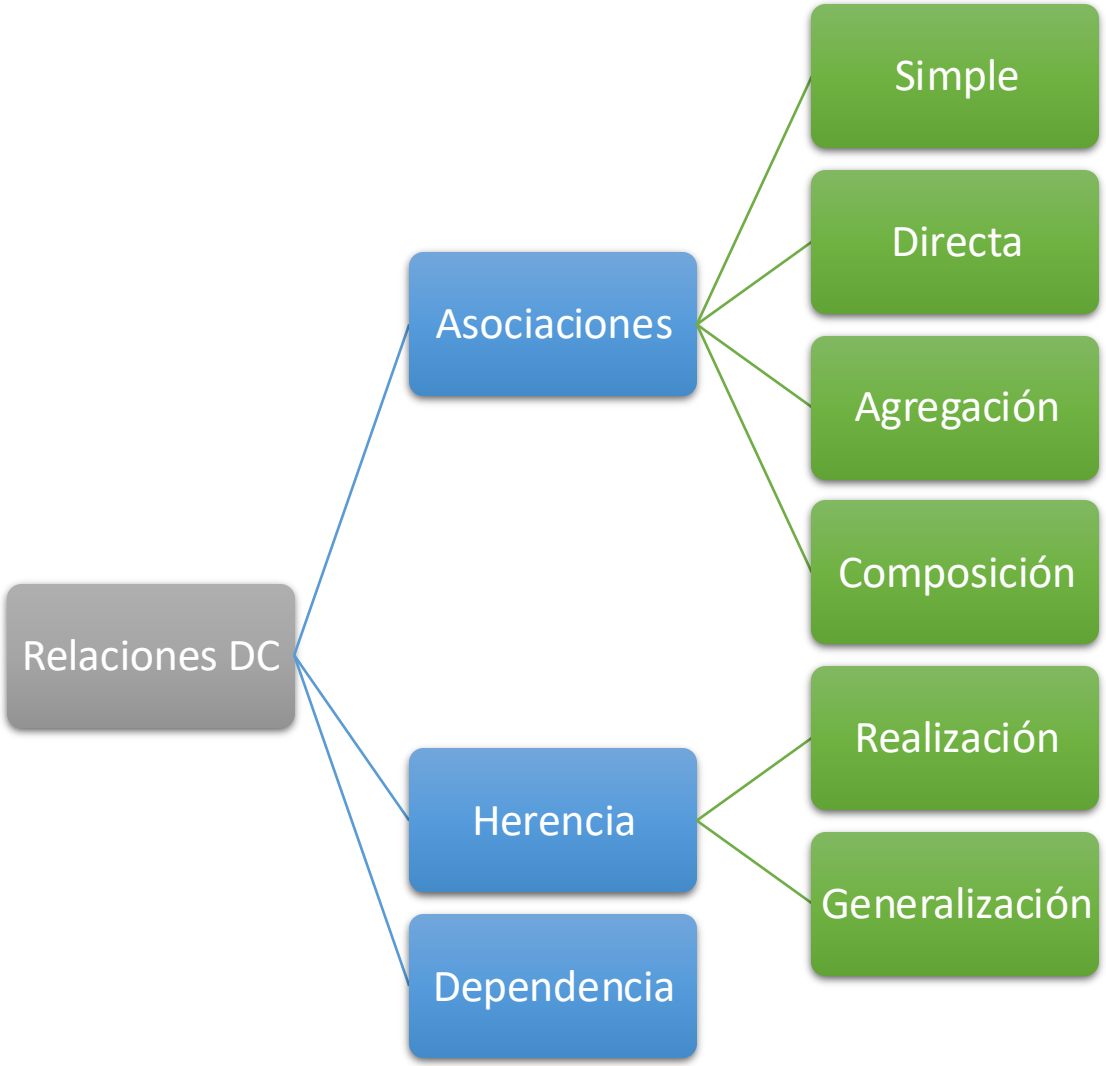

4. 


5. 

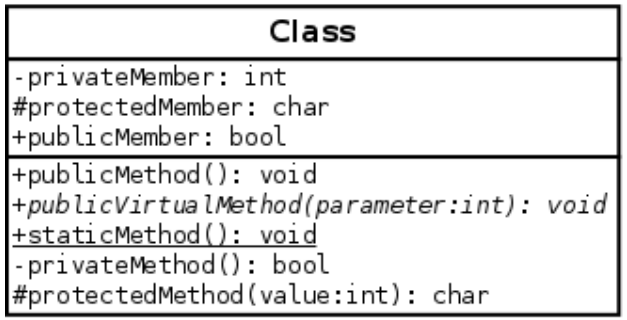

6. 


7. 

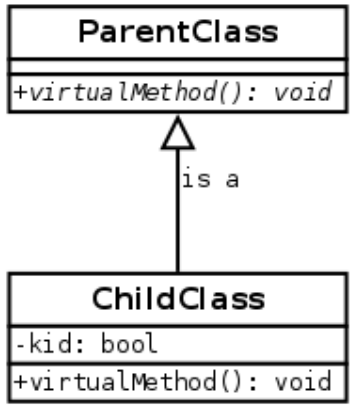
# Relaciones en DC



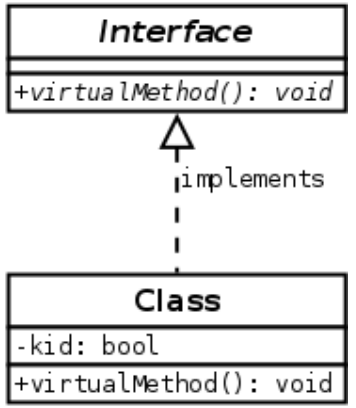
# UML Class Diagram Cheat Sheet



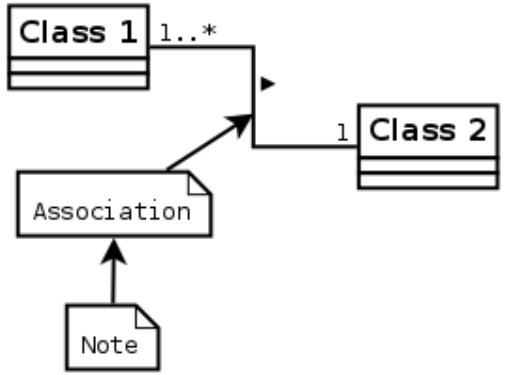

**Generalization**



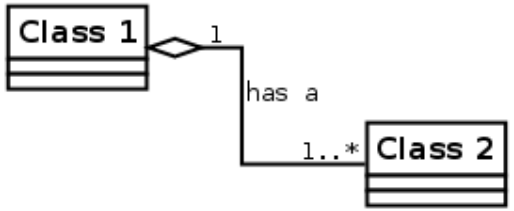

**Realization**



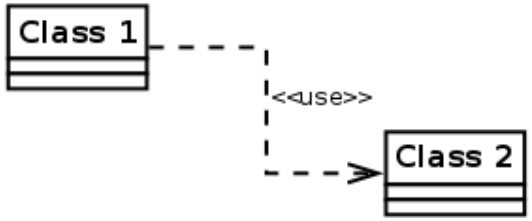

**Association**



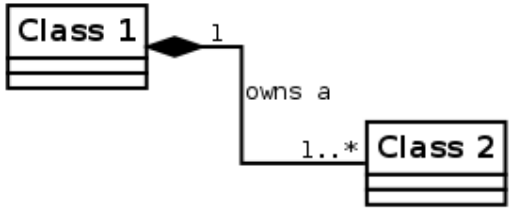

**Aggregation**



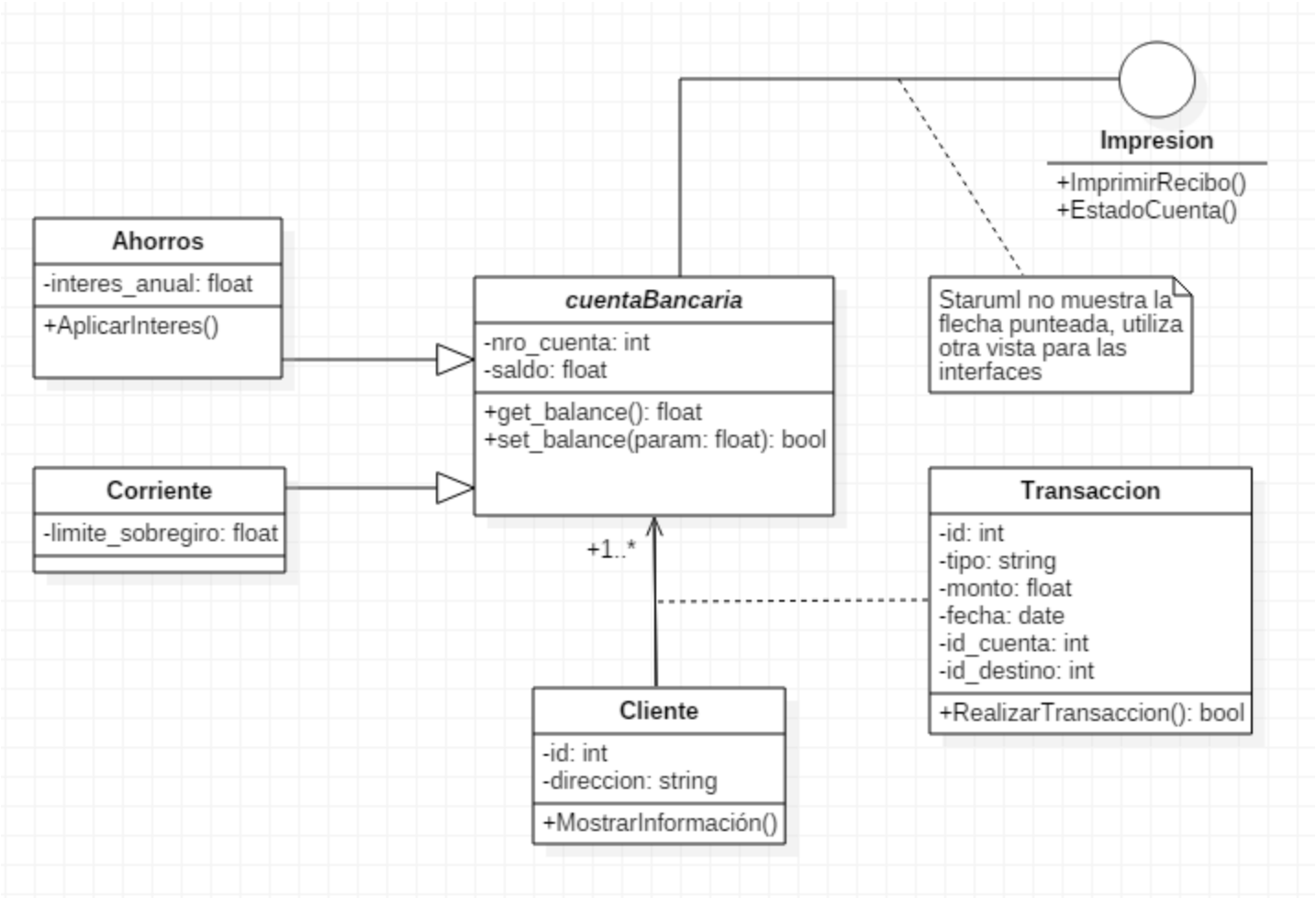

**Dependency**



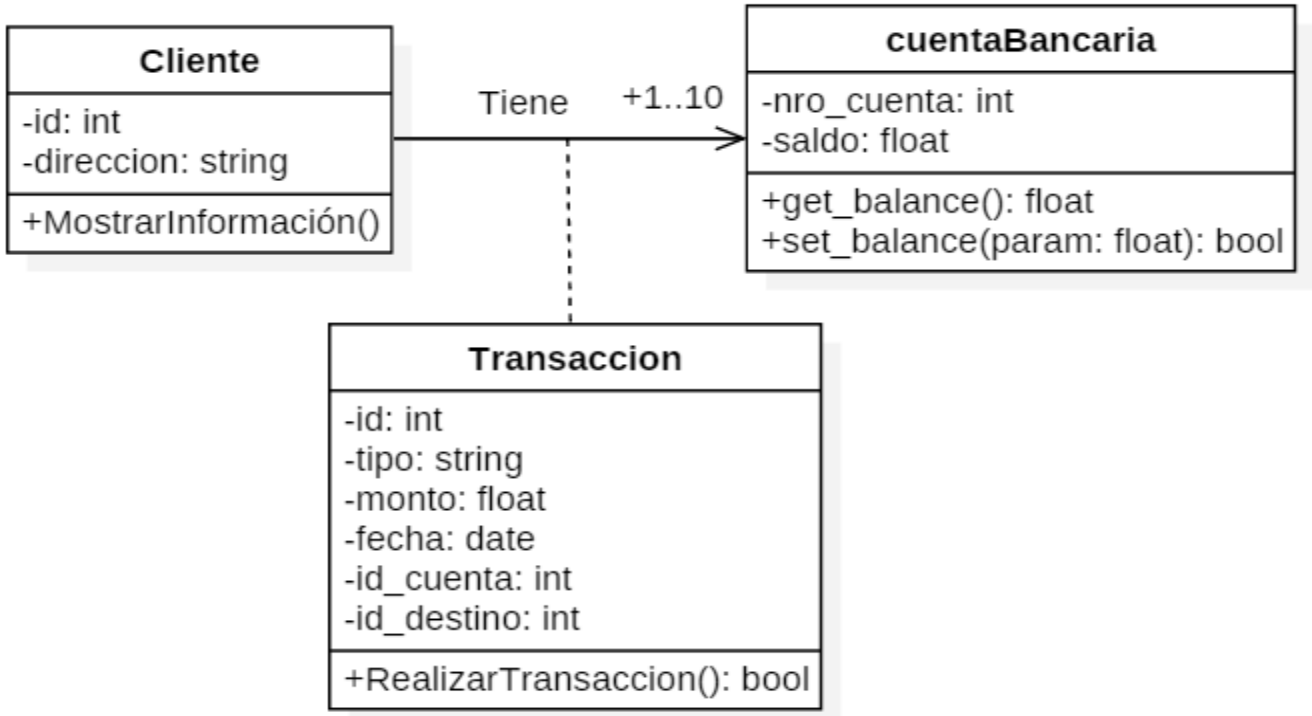

**Composition**



# Ejemplo de Diagrama de clases



# Asociación


Cliente
-id: int
-direccion: string
+MostrarInformación()
Tiene
+1..10
cuentaBancaria
-nro_cuenta: int
-saldo: float
+get_balance(): float
+set_balance(param: float): bool
Transaccion
-id: int
-tipo: string
-monto: float
-fecha: date
-id_cuenta: int
-id_destino: int
+RealizarTransaccion(): bool

# Código - Asociación

```java

import java.util.*;

/**
 *
 */
public abstract class CuentaBancaria implements Impresion {

    /**
     * Default constructor
     */
    public CuentaBancaria() {
    }

    private int nro_cuenta;

    private float saldo;


    public float get_balance() {
        // TODO implement here
        return 0.0f;
    }

    public boolean set_balance(float param) {
        // TODO implement here
        return null;
    }

    public void ImprimirRecibo() {
        // TODO implement here
    }

    public void EstadoCuenta() {
        // TODO implement here
    }


}
```

```java

import java.util.*;

/**
 *
 */
public class Cliente {

    /**
     * Default constructor
     */
    public Cliente() {
    }

    private int id;

    private String direccion;

    public ArrayList<CuentaBancaria> cuentas;

    public void MostrarInformación() {
        // TODO implement here
    }


}
```

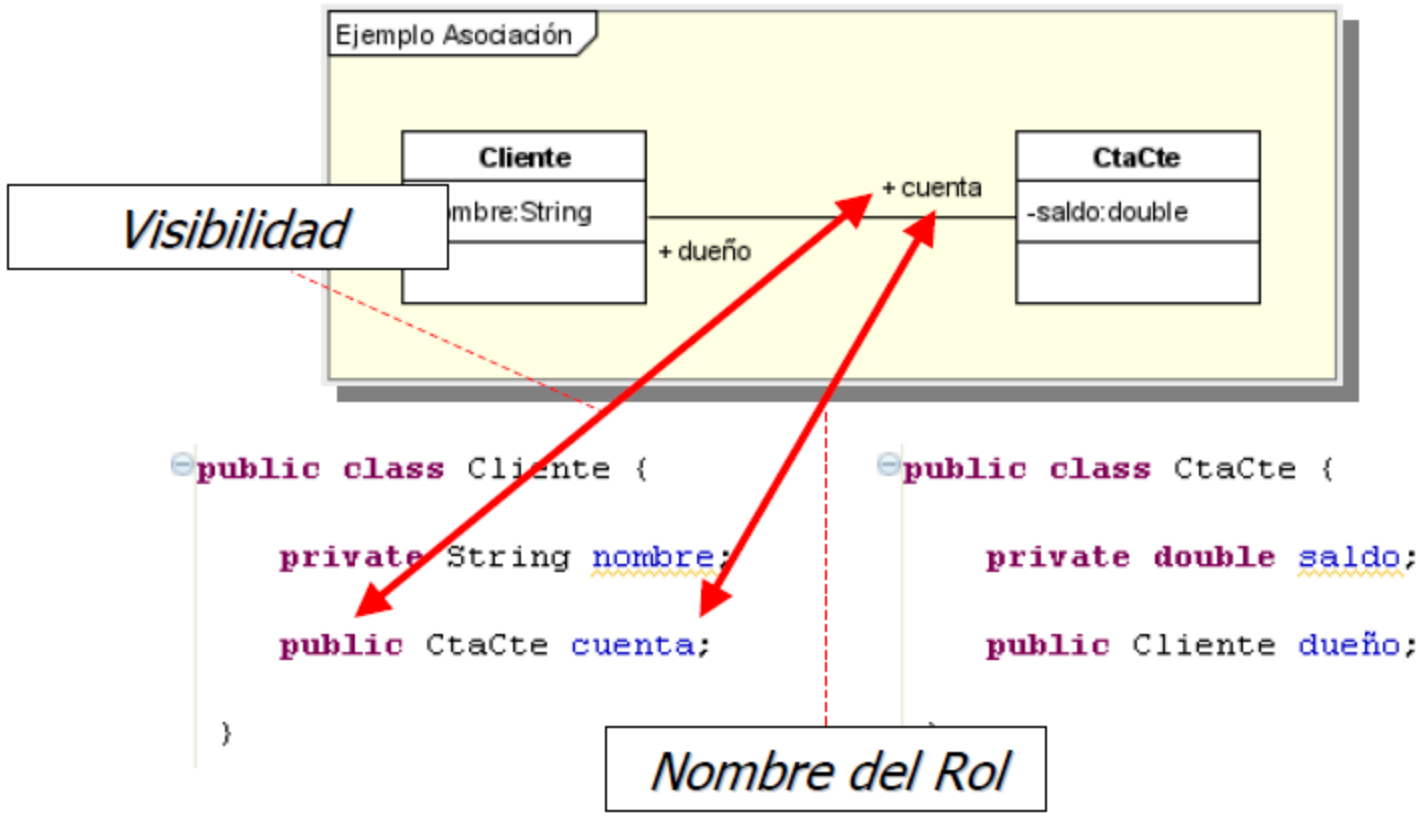
# Clase de asociación

- La clase de asociación se instancia al ejecutar la relación entre Cliente y CuentaBancaria.
- RealizarTransaccion () asocia los clientes y las cuentas.

```java

import java.util.*;

/**
 *
 */
public class Transaccion {

    /**
     * Default constructor
     */
    public Transaccion() {
    }

    private int id;

    private String tipo;

    private float monto;

    private Date fecha;

    private int id_cuenta;

    private int id_destino;

    /**
     * @return
     */
    public boolean RealizarTransaccion(int id_cuenta, int id_destino) {
        // TODO implement here
        return null;
    }

}
```

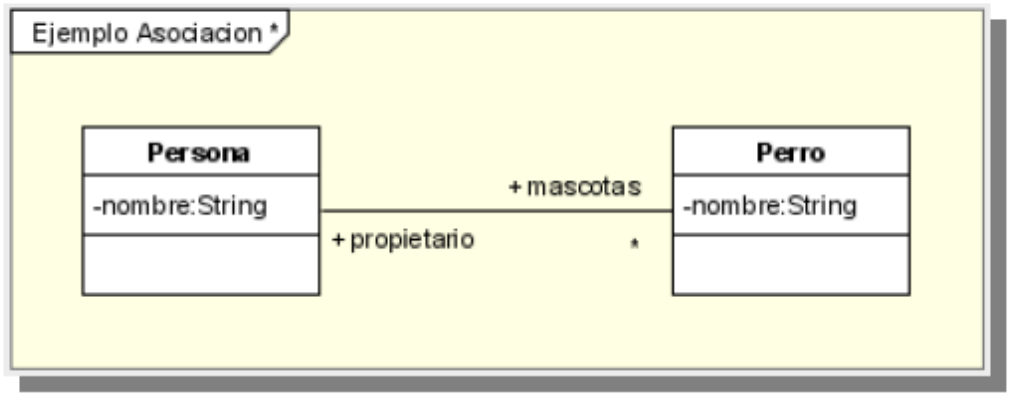
# Bidireccionalidad

- Bidireccional con multiplicidad 0..1 o 1



# Bidireccionalidad

- Bidireccional con multiplicidad *

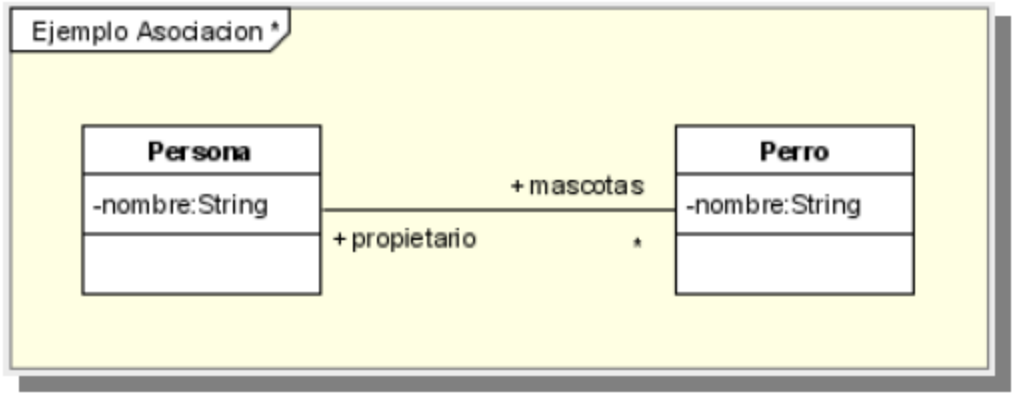



```java
public class Persona {

    private String nombre;

    public java.util.Collection mascotas = new java.util.TreeSet();

}
```

```java
public class Perro {

    private String nombre;

    public Persona propietario;

}
```

# Bidireccionalidad

- Bidireccional con multiplicidad *



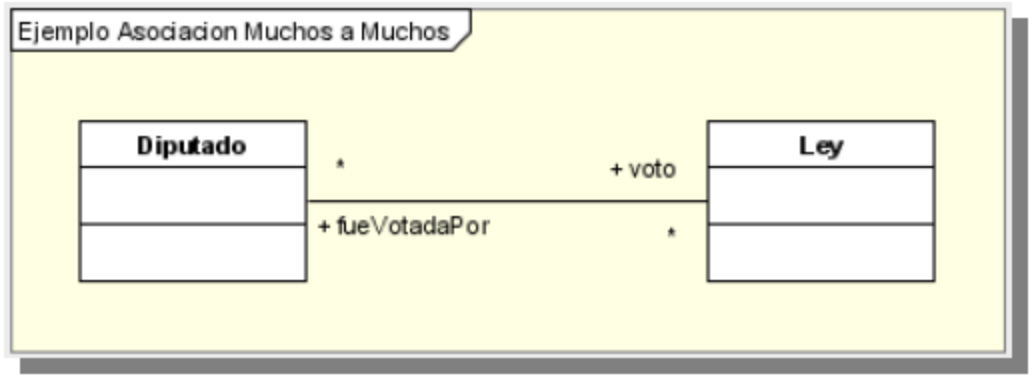

```java
public class Persona {

    private String nombre;

    public java.util.Collection mascotas = new java.util.TreeSet();

}
```

```java
public class Perro {

    private String nombre;

    public Persona propietario;

}
```

*Decisión de Implementación*

# Bidireccionalidad

- Bidireccional con multiplicidad *

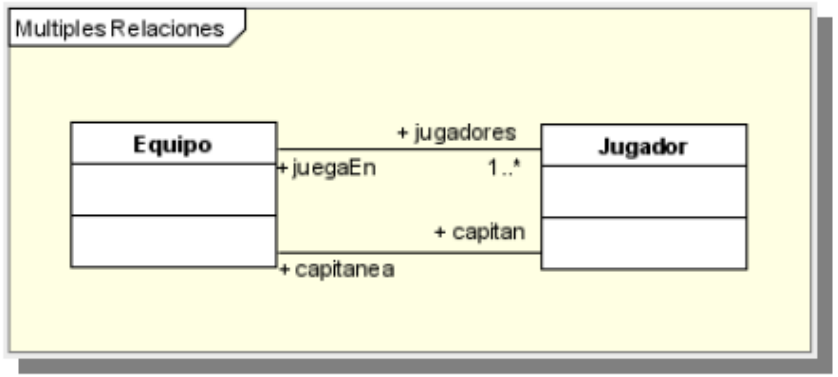



```
public class Diputado {

    public java.util.Collection voto = new java.util.TreeSet();

}
```

```
public class Ley {

    public java.util.Collection fueVotadaPor = new java.util.TreeSet();

}
```

# Múltiples relaciones

- ¿Con más de una relación?

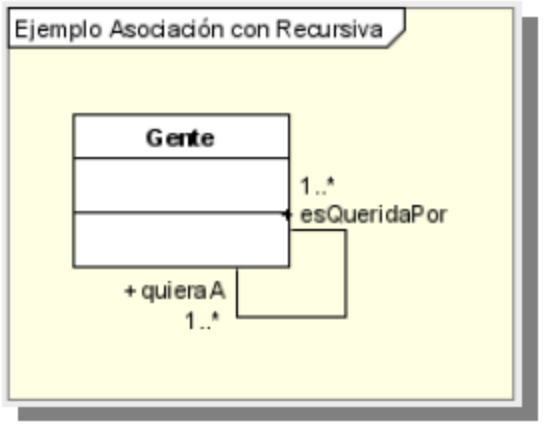



```java
public class Equipo {

    public Jugador capitan;

    public java.util.Collection jugadores = new java.util.TreeSet();

}
```

```java
public class Jugador {

    public Equipo capitanea;

    public Equipo juegaEn;

}
```

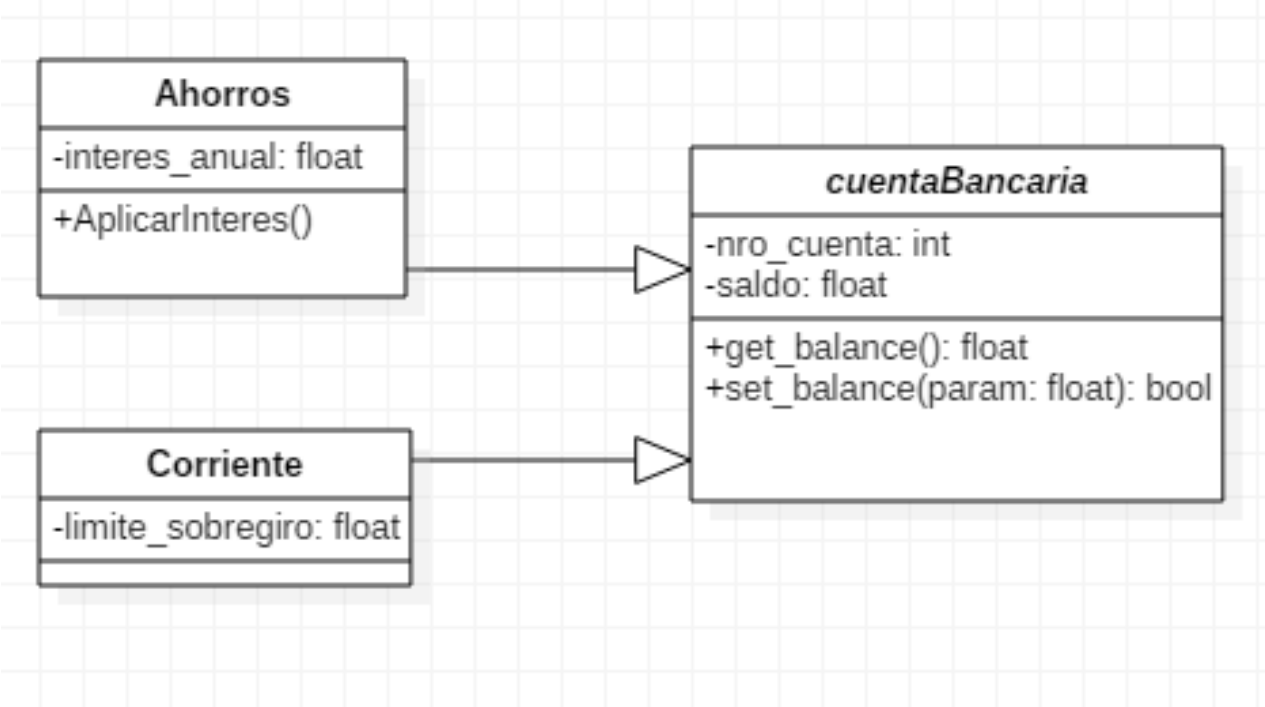
# Asociación con recursividad

- ¿Y con esto?




```
public class Gente {

    public java.util.Collection quieraA = new java.util.TreeSet();

    public java.util.Collection esQueridaPor = new java.util.TreeSet();

}
```

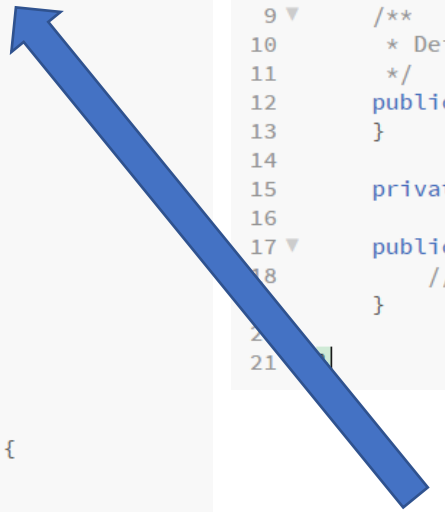
# Ejemplo - Generalización



# Generalización

```java
import java.util.*;

/**
 *
 */
public abstract class CuentaBancaria implements Impresion {

    /**
     * Default constructor
     */
    public CuentaBancaria() {
    }

    private int nro_cuenta;

    private float saldo;


    public float get_balance() {
        // TODO implement here
        return 0.0f;
    }

    public boolean set_balance(float param) {
        // TODO implement here
        return null;
    }

    public void ImprimirRecibo() {
        // TODO implement here
    }

    public void EstadoCuenta() {
        // TODO implement here
    }


}
```

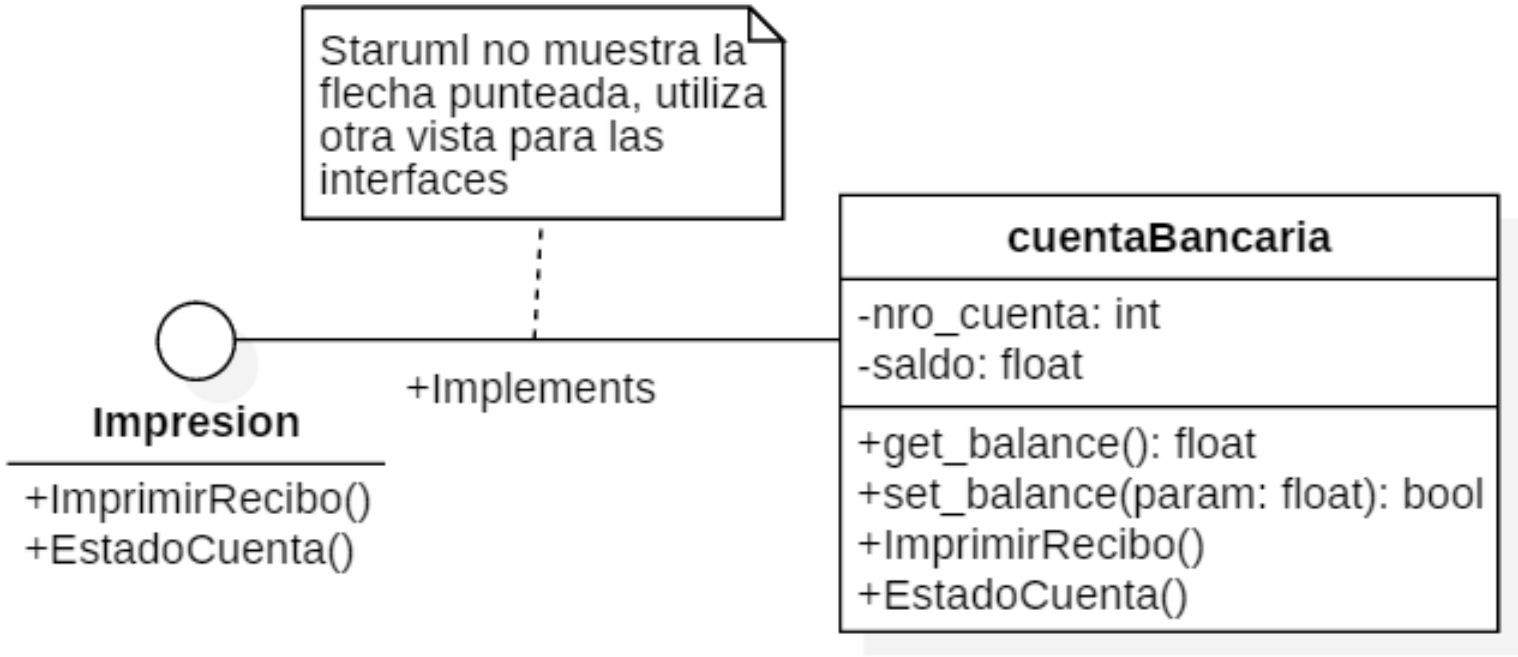
```java
import java.util.*;

/**
 *
 */
public final class Ahorros extends CuentaBancaria {

    /**
     * Default constructor
     */
    public Ahorros() {
    }

    private float interes_anual;

    public void AplicarInteres() {
        // TODO implement here
    }

}
```



En el DC el nombre de la clase está en cursivas
esto se traduce como clase abstracta

# Ejemplo - Realización

- Impresión es una **Interface**
- CuentaBancaria **implementa** esa interface y por lo tanto desarrolla los métodos **ImprimirRecibo**() y **EstadoCuenta**()



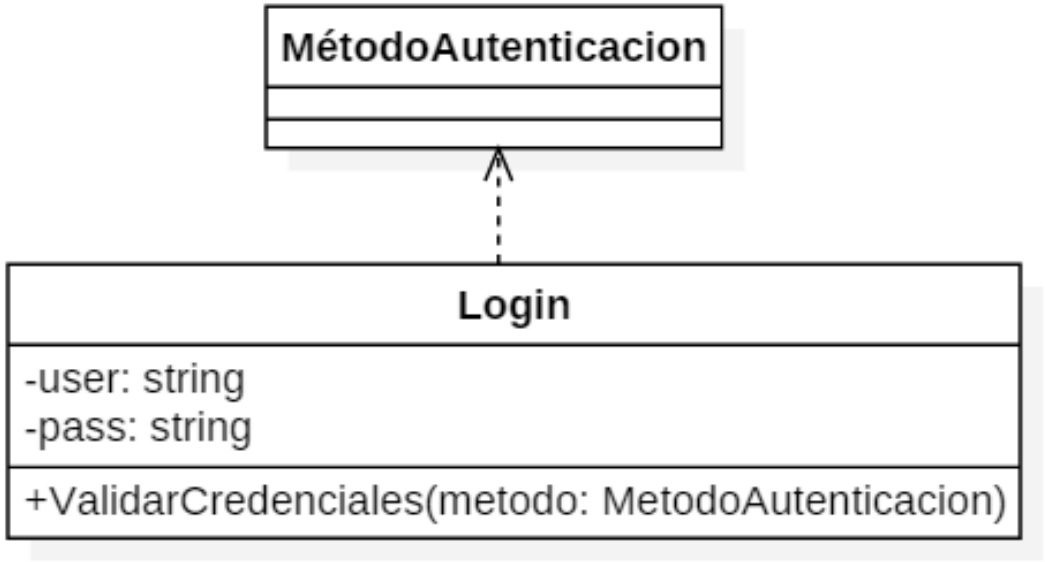
# Realización

```java

import java.util.*;

/**
 *
 */
public abstract class CuentaBancaria implements Impresion {

    /**
     * Default constructor
     */
    public CuentaBancaria() {
    }

    private int nro_cuenta;

    private float saldo;


    public float get_balance() {
        // TODO implement here
        return 0.0f;
    }

    public boolean set_balance(float param) {
        // TODO implement here
        return null;
    }

    public void ImprimirRecibo() {
        // TODO implement here
    }

    public void EstadoCuenta() {
        // TODO implement here
    }


}
```

```java

import java.util.*;

/**
 *
 */
public interface Impresion {

    public void ImprimirRecibo();

    public void EstadoCuenta();

}
```

# Dependencia

- Indica que los cambios sobre la clase B pueden afectar a la clase A.
- ClassA usa los servicios de  ClassB.
- En Java se podría representar con un import.

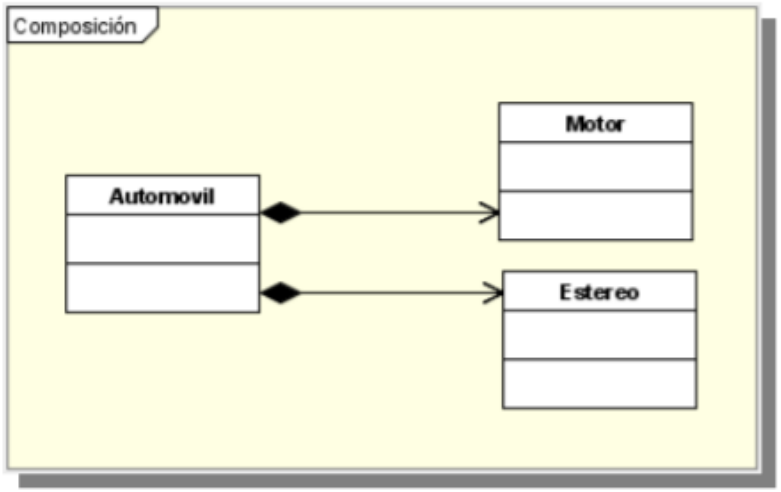


# Dependencia

```java

import java.util.*;
import authentication.MetodoAutenticacion;
/**
 *
 */
public class Login {

    /**
     * Default constructor
     */
    public Login() {
    }

    /**
     *
     */
    private string user;

    /**
     *
     */
    private string pass;

    /**
     * @param metodo
     */
    public void ValidarCredenciales(MetodoAutenticacion metodo) {
        // TODO implement here
    }

}
```

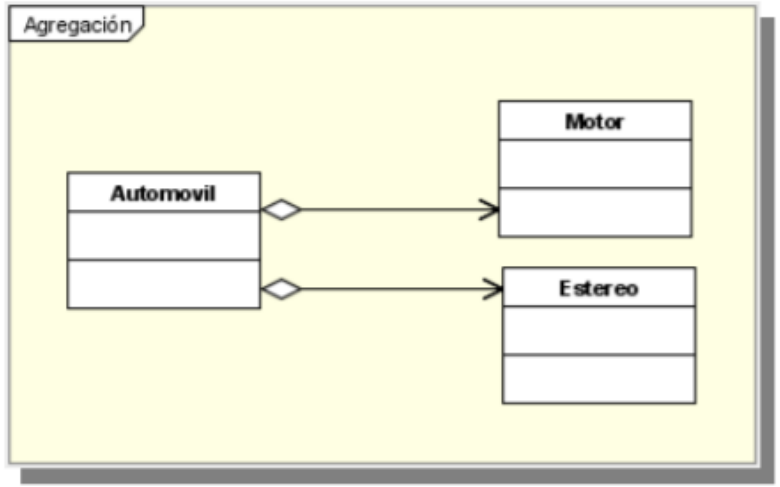
```java

import java.util.*;

package authentication;
/**
 *
 */
public class MetodoAutenticacion {

    /**
     * Default constructor
     */
    public MetodoAutenticacion() {
    }

}
```

# EjemplIFICACIÓN de composición



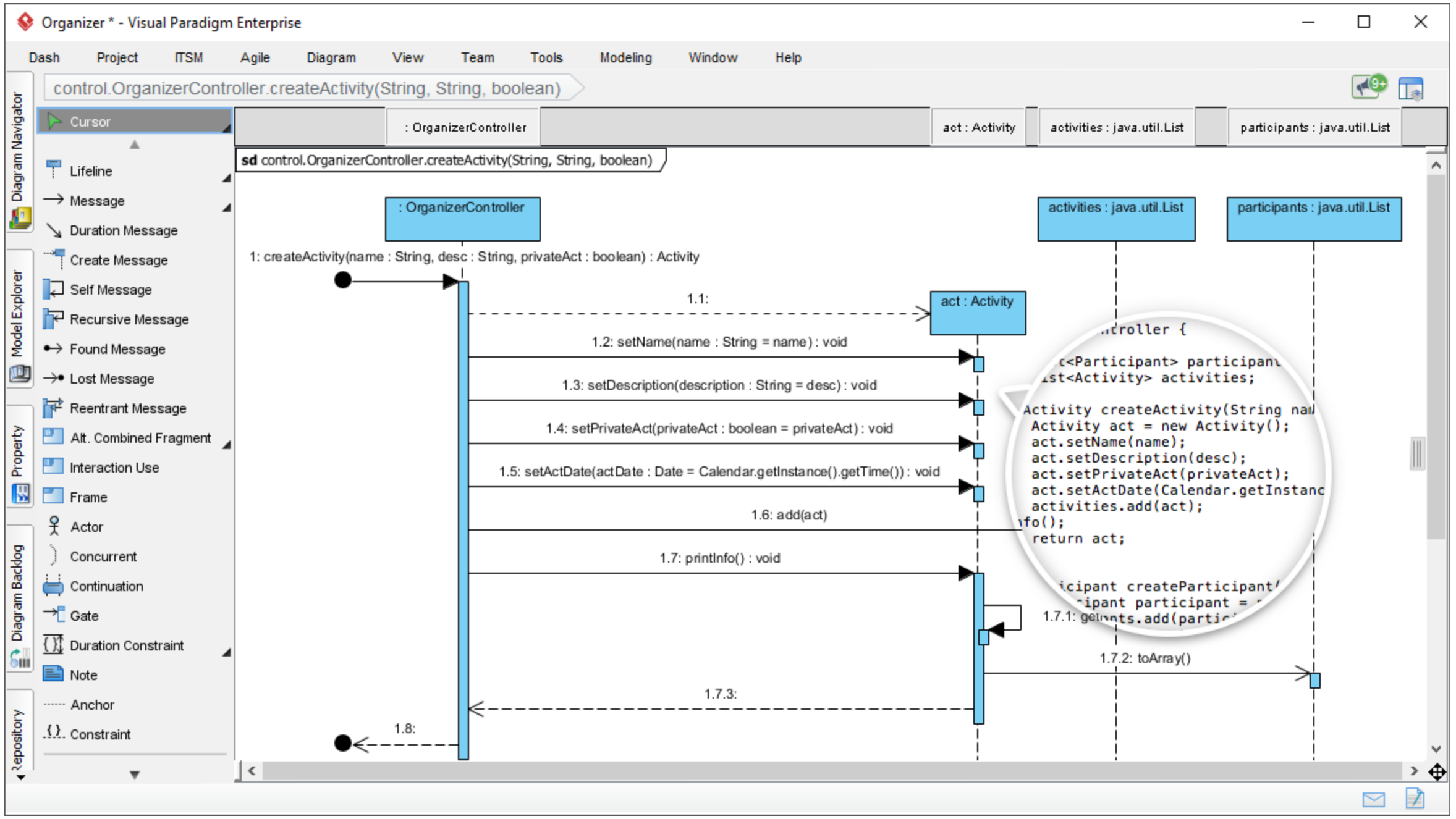

```
public class Automovil {

    public Estereo estereo;
    public Motor motor;

    public Automovil() {
        estereo = new Estereo();
        motor = new Motor();
    }

}
```

# Ejemplificación de Agregación




```
public class Automovil {

    public Estereo estereo;
    public Motor motor;

    public Automovil() {
    }

    public void ensamblar(Estereo e, Motor m) {
        estereo = e;
        motor = m;
    }

}
```

# Traduciendo un diagrama de secuencia en código fuente y viceversa


Organizer * - Visual Paradigm Enterprise
Dash
Project
ITSM
Agile
Diagram
View
Team
Tools
Modeling
Window
Help
control.OrganizerController.createActivity(String, String, boolean)
Diagram Navigator
Model Explorer
Property
Diagram Backlog
Repository
Cursor
Lifeline
Message
Duration Message
Create Message
Self Message
Recursive Message
Found Message
Lost Message
Reentrant Message
Alt. Combined Fragment
Interaction Use
Frame
Actor
Concurrent
Continuation
Gate
Duration Constraint
Note
Anchor
Constraint
: OrganizerController
act : Activity
activities : java.util.List
participants : java.util.List
sd control.OrganizerController.createActivity(String, String, boolean)
1: createActivity(name : String, desc : String, privateAct : boolean) : Activity
1.1:
1.2: setName(name : String = name) : void
1.3: setDescription(description : String = desc) : void
1.4: setPrivateAct(privateAct : boolean = privateAct) : void
1.5: setActDate(actDate : Date = Calendar.getInstance().getTime()) : void
1.6: add(act)
1.7: printInfo() : void
1.7.1:
1.7.2: toArray()
1.7.3:
1.8:
Activity act = new Activity();
act.setName(name);
act.setDescription(desc);
act.setPrivateAct(privateAct);
act.setActDate(Calendar.getInstanc
activities.add(act);
return act;

# Antes de finalizar

# Puntos para recordar

- Recordar cómo ir de un diagrama de clase a código fuente y viceversa.
- Recordar cómo ir de un diagrama de secuencia a código fuente y viceversa.

# Lectura adicional

- Rumbauhg, Jacobson and Booch, “The Unified Modeling Language Reference Manual”
  - Chapters 1, 2,3, 4, 5 y 8
- Perdita Stevens with Rob Pooley, “Using UML”
  - Chapters 1, 2,3, 5, 6, 7, 8 y 9
- Pressman and Maxin , “Software Engineering”
  - Appendix 1: An Introduction to UML
  - Appendix 2: Object-Oriented Concepts

# Próxima sesión

- Patrones de diseño