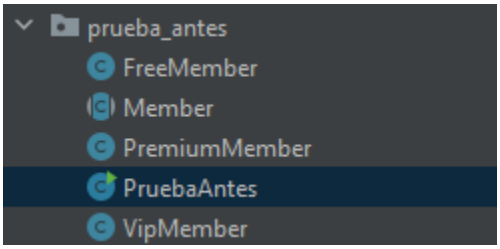
SOLID y refactorización

En un principio tenenos las siguientes clases en el paquete prueba_antes




```java
package prueba_antes;


3 inheritors
public abstract class Member {
    1 usage
    private final String name;
    public Member(String nombre) {
        this.name = nombre;
    }
    no usages  3 implementations
    public abstract void joinTournament();
    1 usage  3 implementations
    public abstract void organizeTournament();
}
```

Al ser Member una clase abstracta las subclases deben implementar los métodos abstractos de dicha clase, por ello subclase Member esta obligada a implementar tanto el método joinTournament como organizeTournament.

```java
package prueba_antes;


1 usage
public class PremiumMember extends Member {
    1 usage
    public PremiumMember(String nombre) {
        super(nombre);
    }

    no usages
    @Override
    public void joinTournament(){
        System.out.println("Se une al toreno");

    }
    1 usage
    @Override
    public void organizeTournament(){
        System.out.println("Organiza torneo");
    }
}
```

Dentro del constructor de la clase PremiumMember llamamos al constructor de la super clase que en este caso es Member y le pasamos como argumento el nombre del miembro que pertenece a esta clase PremiumMember y no habría problema implementar los métodos que hereda de Member.

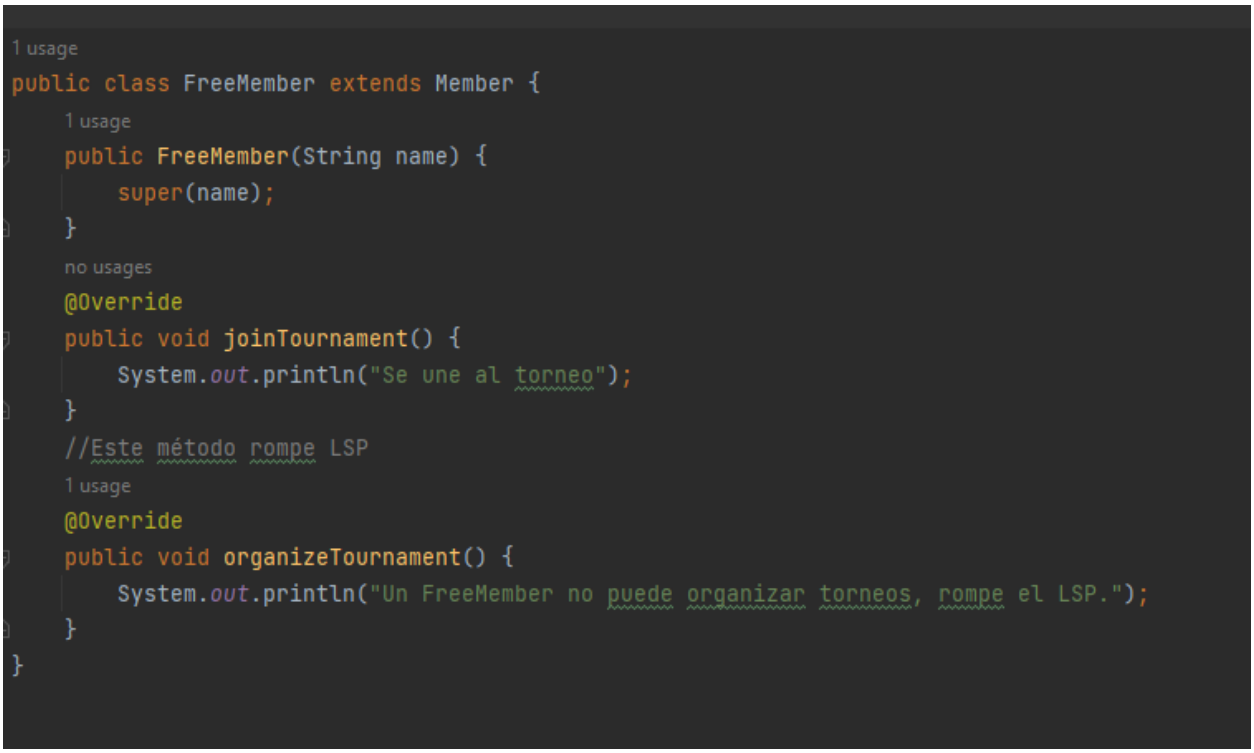
```java
package prueba_antes;


1 usage
public class VipMember  extends  Member{
    1 usage
    public VipMember(String nombre) {
        super(nombre);
    }

    no usages
    @Override
    public void joinTournament(){
        System.out.println("Se une al toreno");

    }
    1 usage
    @Override
    public void organizeTournament(){
        System.out.println("Organiza torneo");
    }
}
```

Analogamente ocurre con la clase VipMember



```
1 usage
public class FreeMember extends Member {
    1 usage
    public FreeMember(String name) {
        super(name);
    }
    no usages
    @Override
    public void joinTournament() {
        System.out.println("Se une al torneo");
    }
    //Este método rompe LSP
    1 usage
    @Override
    public void organizeTournament() {
        System.out.println("Un FreeMember no puede organizar torneos, rompe el LSP.");
    }
}
```

FreeMember. La clase FreeMember puede unirse a torneos, pero no puede organizar torneos. Este es un problema que debemos abordar en el método OrganizeTournament(). Podemos lanzar una excepción con un mensaje significativo o podemos mostrar un mensaje.

```java
package prueba_antes;


import java.util.List;

public class PruebaAntes {
    public static void main(String[] args) {
        List<Member> miembros = List.of(
                new PremiumMember( nombre: "Abejita Azul"),
                new VipMember( nombre: "Kaperucita Feliz"),
                new FreeMember( name: "Inspectora Motita")
        );
        for (Member member: miembros) {
            member.organizeTournament();
        }


    }
}
```

Ejecutando PruebaAntes.main() podemos ver lo siguiente:

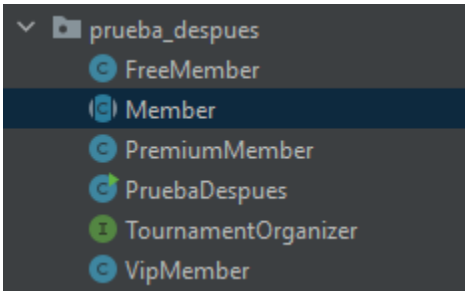
```
Organiza torneo
Organiza torneo
Un FreeMember no puede organizar torneos, rompe el LSP.

Process finished with exit code 0
```

Esta situación es un escándalo. No podemos continuar con la implementación de la aplicación.

Rediseñamos la solución para obtener un código compatible con LSP a través de un proceso de refactorización.

Veamos las siguientes clases en el paquete prueba_despues




```java
package prueba_despues;

3 inheritors
public abstract class Member {
    1 usage
    private final String name;
    public Member(String nombre) {
        this.name = nombre;
    }
    1 usage  3 implementations
    public abstract void joinTournament();

}
```

El método abstracto organizeTournament que formaba parte de la clase abstracta Member lo hemos trasladado a una interface, de tal modo que la subclase FreeMember no esté obligada a implementarla sino solo aquellas que implementen dicha interfaz.

```java
package prueba_despues;

2 usages  2 implementations
public interface TournamentOrganizer {
    no usages  2 implementations
    public abstract void organizeTournament();
}
```

```
1 usage
public class VipMember  extends Member implements TournamentOrganizer {
    1 usage
    public VipMember(String nombre) { super(nombre); }

    1 usage
    @Override
    public void joinTournament(){
        System.out.println("Se une al torneo");

    }
    no usages
    @Override
    public void organizeTournament() { System.out.println("Organiza torneo"); }
}
```

Al hacer esto la clase VipMember no cambia su funcionalidad

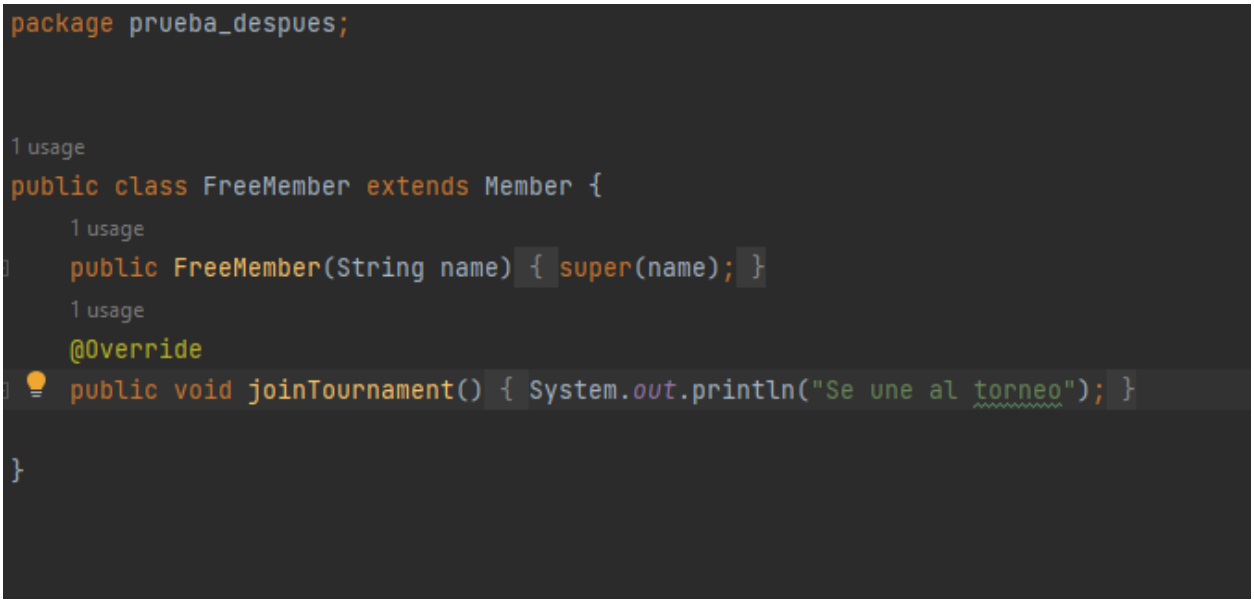
```
package prueba_despues;

1 usage
public class PremiumMember extends Member implements  TournamentOrganizer{
    1 usage
    public PremiumMember(String nombre) { super(nombre); }

    1 usage
    @Override
    public void joinTournament(){
        System.out.println("Se une al torneo");

    }
    no usages
    @Override
    public void organizeTournament() { System.out.println("Organiza torneo"); }
}
```

Al hacer esto la clase PremiumMember no cambia su funcionalidad



De esta manera solucionamos este problema que teníamos en el paquete prueba_antes para obtener un código compatible con LSP a través de un proceso de refactorización.

```java
package prueba_despues;


import java.util.List;

public class PruebaDespues {
    public static void main(String[] args) {

        List<Member> miembros = List.of(
                new PremiumMember( nombre: "Abejita Azul"),
                new VipMember( nombre: "Kaperucita Feliz"),
                new FreeMember( name: "Inspectora Motita")
        );
        for (Member member: miembros) {
            member.joinTournament();
        }

    }
}
```

Al ejecutar PruebaDespues.main vemos lo siguiente:

```
Se une al torneo
Se une al torneo
Se une al torneo

Process finished with exit code 0
```