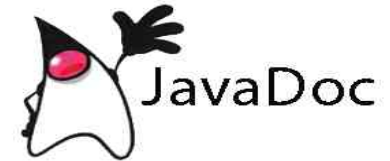
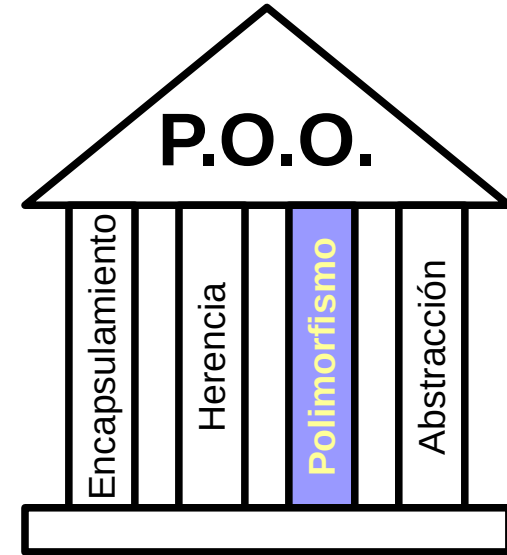


CLASE 4 Polimorfismo. Principio de sustitución de Liskov. Contratos. Requerimientos.

1. Principio de sustitución de Liskov.
2. Concepto de Polimorfismo.
3. Variables y objetos.
4. Concepto de enlace tardío (Late Binding).
5. Implementación de polimorfismo.
6. Herencia y modificador de acceso de miembros sobreescritos.
7. Acceso a miembros sobreescritos
8. El verdadero significado de protected.
9. Concepto de contrato. Definición.
10. Precondiciones y postcondiciones.
11. Documentación de los contratos con javadoc.
12. Como definir el contrato de un método.
13. Caso de estudio. Comprensión de requerimientos, comprensión del mundo del problema, paquetes, declaración de clases, asignación de responsabilidades, contrato de un método.



Principio de sustitución de Liskov



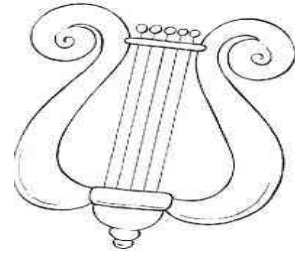
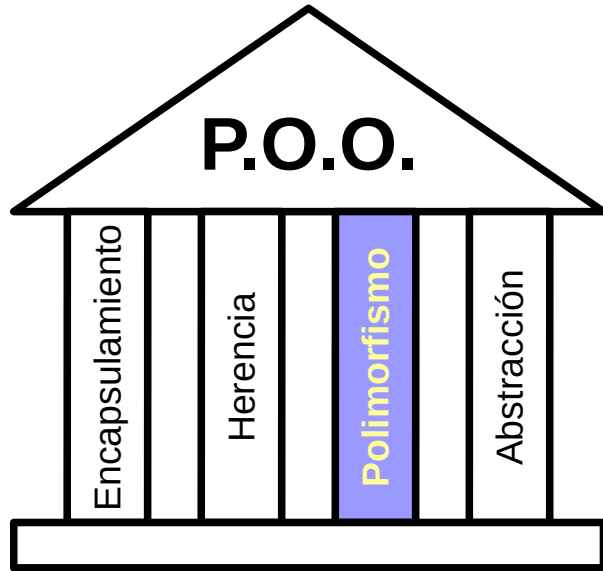
"Debe ser posible utilizar cualquier objeto instancia de una subclase en lugar de cualquier objeto instancia de su superclase sin que la semántica del programa escrito en los términos de la superclase se vea afectado"

Barbara Liskov

(donde va un padre, puede ir un hijo)

Concepto de Polimorfismo

La palabra polimorfismo proviene del griego y significa que *posee varias formas diferentes*



- Polimorfismo de sobrecarga
- Polimorfismo paramétrico
- Polimorfismo de inclusión
 - también llamado redefinición o subtipado

Concepto de Polimorfismo

Polimorfismo de Sobrecarga:

Permite definir **operadores** cuyos comportamientos varían de acuerdo a los parámetros que se les aplican. Así es posible, por ejemplo, agregar el **operador +** y hacer que se comporte de manera distinta cuando está haciendo referencia a una operación entre dos números enteros (suma) o bien cuando se encuentra entre dos cadenas de caracteres (concatenación).

Polimorfismo paramétrico:

Es la capacidad para definir varios métodos utilizando el **mismo nombre**, pero usando **parámetros diferentes (nombre y/o tipo)**. El polimorfismo paramétrico selecciona automáticamente el método correcto a aplicar en función del tipo de datos pasados en el parámetro.

- **int addition(int, int)** devolvería la suma de dos números enteros.
- **float addition(float, float)** devolvería la suma de dos flotantes.
- **char addition(char, char)** daría por resultado la suma de dos caracteres

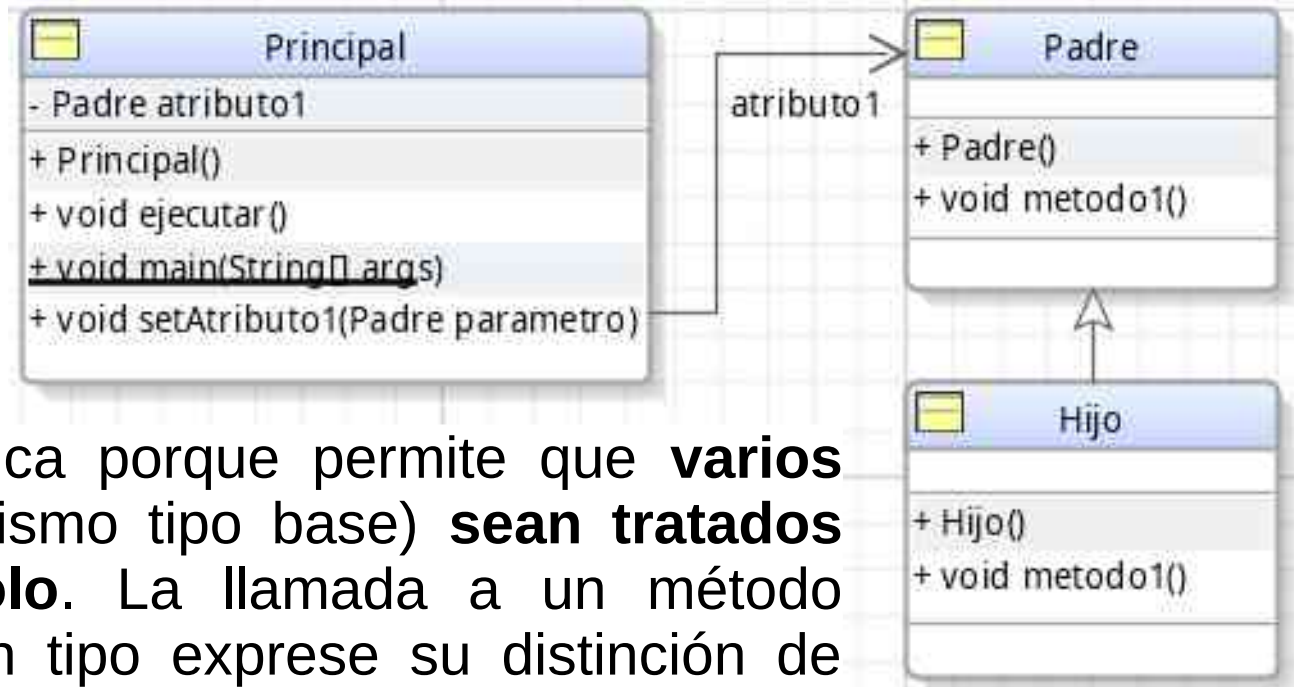
Concepto de Polimorfismo

Polimorfismo de inclusión, también llamado redefinición o subtipado:

La habilidad para redefinir un método en clases que se hereda de una clase base se llama especialización. Por lo tanto, ***se puede llamar un método de objeto sin tener que conocer su tipo intrínseco: esto es polimorfismo de subtipado***. Permite no tomar en cuenta detalles de las clases especializadas de una familia de objetos, enmascarándolos con una interfaz común (siendo esta la clase básica).

Concepto de Enlace Tardío (Late Binding)

La herencia permite el tratamiento de un objeto como si fuera de su propio tipo o del tipo base.



Esta característica es crítica porque permite que **varios tipos** (derivados de un mismo tipo base) **sean tratados como si fueran uno sólo**. La llamada a un método polimórfico permite que un tipo exprese su distinción de otro tipo similar, puesto que ambos se derivan del mismo tipo base. Esta distinción se expresa a través de diferencias en comportamiento de los métodos.

Concepto de Enlace Tardío (Late Binding)

```
public class Padre
{
    public Padre()
    {
        super();
    }

    public void metodo1()
    {
        System.out.println("ejecución" +
            "del Padre->metodo1");
    }
}
```

```
public class Hijo extends Padre
{
    public Hijo()
    {
        super();
    }

    @Override
    public void metodo1()
    {
        System.out.println("ejecución" +
            "del Hijo->metodo1");
    }
}
```

Concepto de Enlace Tardío (Late Binding)

```
public class Principal
{
    private Padre atributo1;

    public Principal()
    {
        super();
    }

    public void setAtributo1(Padre parametro)
    {
        this.atributo1 = parametro;
    }

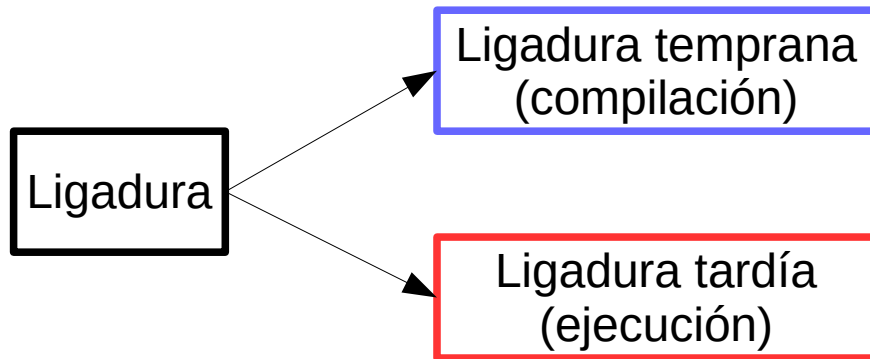
    public void ejecutar()
    {
        atributo1.metodo1();
    }
}
```

```
public static void main(String[] args)
{
    Principal p = new Principal();
    Padre padre = new Padre();
    Hijo hijo = new Hijo();
    int contador = 0;
    for (contador = 0; contador < 20; contador++)
    {
        if (Math.random() > 0.5)
            p.setAtributo1(padre);
        else
            p.setAtributo1(hijo);
        p.ejecutar();
    }
}
```


Concepto de Enlace Tardío (Late Binding)

En **tiempo de compilación** no se conoce cuál será el objeto verdaderamente usado al ejecutar el método `setAtributo(...)`

Por lo tanto se tiene que resolver en **tiempo de ejecución**.



```
public static void main(String[] args)
{
    Principal p = new Principal();
    Padre padre = new Padre();
    Hijo hijo = new Hijo();
    int contador = 0;
    for (contador = 0; contador < 20; contador++)
    {
        if (Math.random() > 0.5)
            p.setAtributo1(padre);
        else
            p.setAtributo1(hijo);
        p.ejecutar();
    }
}
```

Ligadura: conexión de una llamada a un método

Herencia y modificador de acceso de miembros sobreescritos

Repasando conceptos



Cuando se extiende una clase, se pueden añadir miembros y también se pueden redefinir los existentes. El efecto exacto de redefinir un miembro heredado depende de la clase de miembro que sea.

- **Sobrecargar un método:** proporcionar más de un método con el mismo nombre pero con diferente signatura para distinguirlos
- **Redefinir un método:** sustituir la implementación de un método de la superclase con uno propio. Las signaturas deben ser idénticas.

Sobrecargar un método heredado es agregar un método con el mismo nombre que en la superclase, pero con distinta signatura

Herencia y modificador de acceso de miembros sobreescritos

Los métodos que redefinen a otros poseen sus propios **especificadores de acceso**. Una subclase puede modificar el acceso de los métodos de la superclase, pero sólo para hacerlos **más accesibles**.

Un método de una instancia no puede tener la misma signatura que un *método estático heredado*, ni viceversa. Sin embargo, un método heredado se puede hacer *abstract*, incluso aunque no lo fuera el método de la superclase.

Acceso a miembros sobreescritos

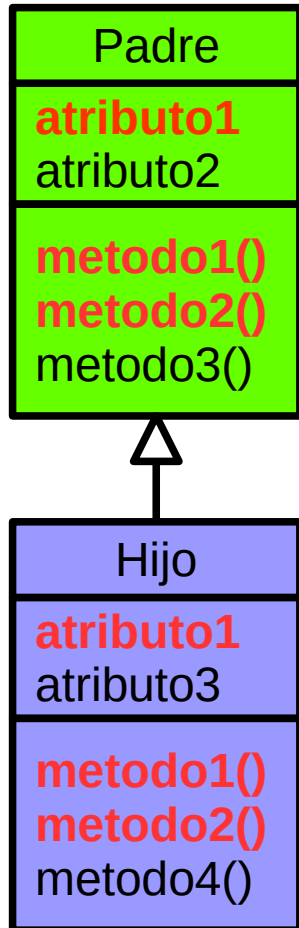
Al invocar a un método mediante una referencia de objeto, la clase real del objeto gobierna la implementación que se utiliza. Cuando se accede a un campo se utiliza el tipo *declarado* de la referencia (el tipo de la variable)...[Gosling pág. 70]

Ejemplo genérico:

```
Padre p = new Hijo();
```

Acceso a miembros sobreescritos

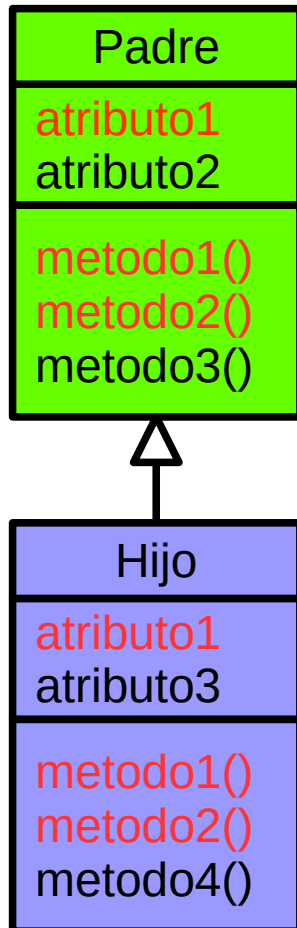
Supongamos una jerarquía de clases que contemple la redefinición de algunos métodos en la clase hijo:



- **atributo1** de Padre es ocultado por atributo1 de Hijo
- **metodo1()** y **metodo2()** de Padre son sobreescritos en Hijo
- **atributo2** es propio de Padre
- **atributo3** es propio de Hijo
- **metodo3()** es heredado por Hijo
- **metodo4()** es propio de Hijo

Acceso a miembros heredados

Consideremos la siguiente situación:



Padre p = new Hijo();

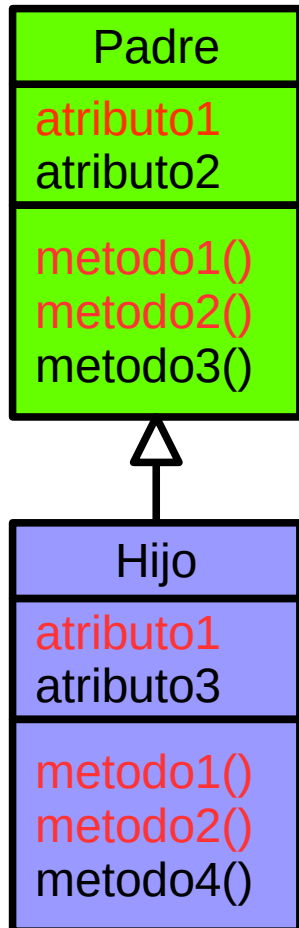
Qué método se activa?

p.metodo1() ?
p.metodo2() ?
p.metodo3() ?
p.metodo4() ?

Qué atributo responde?

p.atributo1 ?
p.atributo2 ?
p.atributo3 ?

Acceso a miembros sobreescritos



Padre p = new Hijo();

- Los *atributos* corresponden a los del tipo de la variable (clase *Padre*).
p.atributo3 es inaccesible.
- Los *métodos* corresponden a los del verdadero objeto (clase *Hijo*). Sólo pueden ser referenciados los sobreescritos.
p.metodo4() es inaccesible.

El verdadero significado de ***protected***

Un miembro **protected** puede ser accedido por las clases que extiendan de la clase actual, por clases del mismo paquete, y además puede ser accedido desde otra clase a través de referencias a objetos que sean al menos del mismo tipo que la clase (es decir, referencias del tipo de la clase o de uno de sus subtipos).

Ejemplo

Consideremos la implementación como ***lista enlazada*** de una fila. Los nodos de la fila están formados por objetos de tipo **Nodo**

EJEMPLO

```
public class Nodo
{
    private Nodo next;
    private Object element;

    public Nodo(Object element)
    {    this.element = element;
    }
    public Nodo(Object element,
                Nodo next)
    {    this.element = element;
        this.next = next;
    }
}
```

```
    public Object getElement()
    {    return element;
    }
    public void setElement(Object
                           element)
    {    this.element = element;
    }
    public Nodo getNext()
    {    return next;
    }
    public void setNext(Nodo next)
    {    this.next = next;
    }
}
```

EJEMPLO

```
public class Fila
{
    protected Nodo head;
    protected Nodo tail;

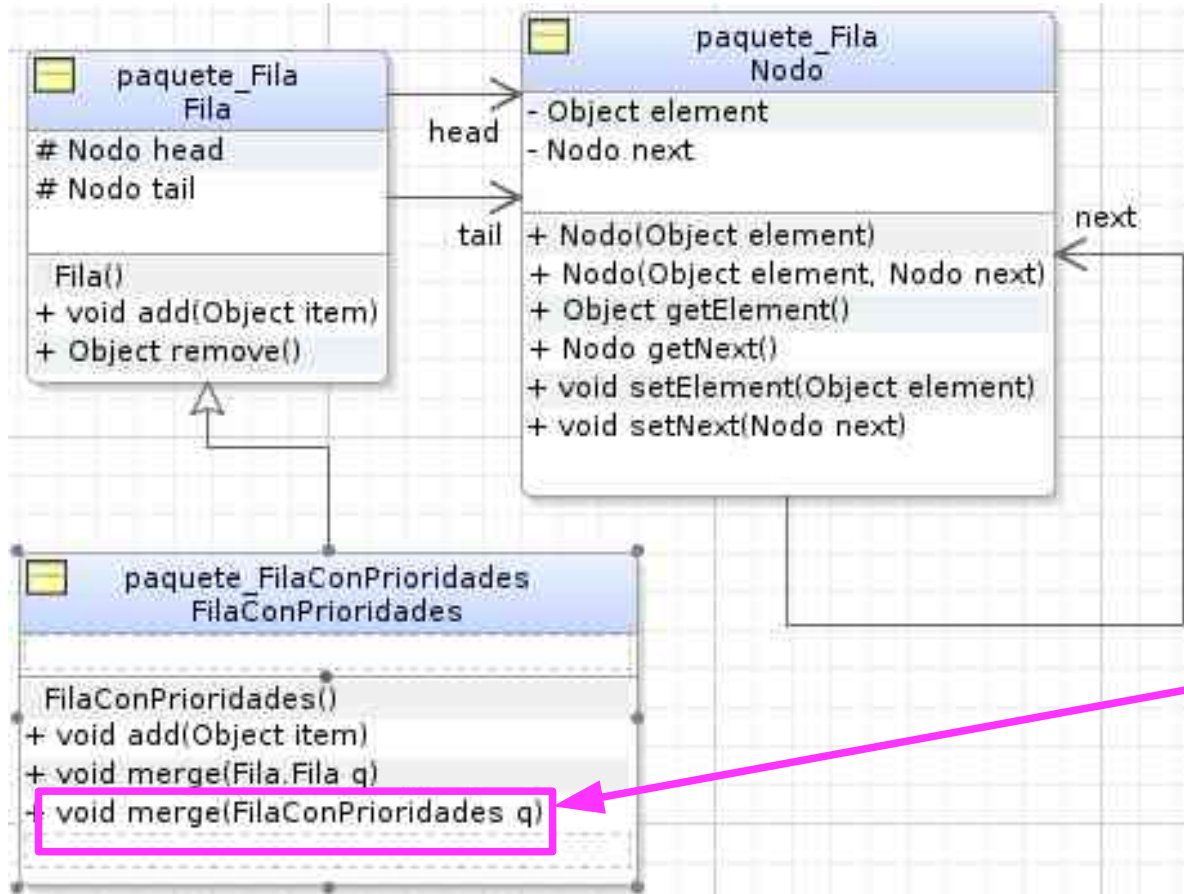
    public void add(Object item)
    { /* ... */
    }

    public Object remove()
    { /* ... */
    }
}
```

Una Fila consiste entonces en una referencia a las celdas de la cabecera y del final, y la implementación de add y remove.

Hacemos que las referencias *head* y *tail* sean **protected**, de forma que sean accesibles desde sus clases extendidas.

EJEMPLO



Ahora extendemos nuestra clase a otra: **FilaConPrioridades**. Y se guarda en otro paquete. De forma que los objetos se agreguen según un orden de prioridad y no al final.

Se redefine el método **add** desde donde podemos acceder a los atributos de la clase padre (`head` y `tail`).

Se agrega un método (**merge**) para fusionar dos **filas con prioridades** en una sola.

EJEMPLO

Dentro de la Class FilaConPrioridades:

```
public void merge(FilaConPriorides q)
{
    Nodo first = q.head;
    // ...
}
```

No estamos accediendo al miembro protected del objeto actual, sino al miembro protected de un objeto pasado como parámetro. Esto está permitido ya que la clase del parámetro es la misma que la que contiene a merge. También sería válido si **q** fuera una subclase de FilaConPrioridades

EJEMPLO

Se agrega otro método para fusionar una **Fila** con una **FilaConPrioridades**. Se redefine del método *merge* (dentro de la clase **FilaConPrioridades**)

```
public void merge(FilaConPrioridades q)
{
    Nodo first = q.head;
}
```

```
public void merge(Fila q)
{
    Nodo first = q.head;
}
```

**Este código
no compila**

Description

- Error(9,5): Fila() is not public in paquete_Fila.Fila; cannot be accessed from outside package
- Error(25,23): head has protected access in paquete_Fila.Fila

EJEMPLO

Problema: la clase FilaConPrioridad intenta acceder al miembro protected de Fila

Como Fila no es la misma clase ni es una subclase de FilaConPrioridad, el acceso no está permitido. Aunque cada FilaConPrioridad es una Fila, no todas las Filas son FilasConPrioridades.

Ja!



Concepto de contrato

Cómo garantizar que la aplicación del principio de sustitución de Liskov no cause problemas?

- No alcanza con respetar la firma de un método para evitar problemas en el código
- Se debe documentar un contrato de forma que simplemente con respetarlo se garantice que se pueda aplicar el principio.
- Se utilizan las precondiciones y postcondiciones.
 - En los métodos sobreescritos se deben respetar pre y post.
 - Se pueden relajar las pre y fortalecer las pc



Concepto de contrato

El **contrato** de un método establece bajo qué **CONDICIONES** el método tendrá éxito y cuál será el resultado una vez que se termine su ejecución. Por ejemplo, para el método.



Concepto de contrato – para qué sirve?

Supongamos un sistema para administrar una librería:

Definimos el método **adicionarLibroCatalogo** de la clase **TiendaLibro**, que nos permitía agregar un libro nuevo al catálogo.

Dicho método recibía como parámetro el libro que se quería añadir, y tenía la siguiente signatura:

```
void adicionarLibroCatalogo(Libro nuevoLibro)
```

Cómo programo esto?



Ejemplo



- 1) ¿El nuevoLibro es un libro válido, es decir, su valor es distinto de null y sus atributos (título, ISBN y precio) tienen un valor definido y correcto?
- 2) ¿Debemos verificar que el precio sea un número positivo antes de adicionarlo al catálogo? ¿Ya alguien verificó eso y es una pérdida de tiempo volverlo a hacer?
- 3) ¿Ya se verificó que el nuevoLibro no esté incluido en el catálogo? ¿Debemos verificar si existe ya un libro con ese ISBN antes de agregarlo al catálogo?
- 4) ¿Ya está creado el arreglo que representa el catálogo de la librería? Tal vez en el constructor de la clase se les olvidó crear un objeto de la clase ArrayList para almacenar los libros del catálogo. ¿Debo hacer esta verificación al comienzo del método?

Concepto de contrato

```
void adicionarLibroCatalogo(Libro nuevoLibro)
```

Aunque la signature de un método y su descripción informal pueden dar una idea general del servicio que un método debe prestar, esto **no es suficiente en la mayoría de los casos para definir con precisión el código que se debe escribir.**

Opción 1

```
public void adicionarLibroCatalogo( Libro nuevoLibro )  
{  
    catalogo.add( nuevoLibro );  
}
```

En esta versión, simplemente agregamos el nuevo libro al catálogo. Estamos suponiendo que alguien ya verificó que no hay otro libro con el mismo ISBN.

Opción 2

```
public void adicionarLibroCatalogo( Libro nuevoLibro )
{
    if ( nuevoLibro != null &&
        nuevoLibro.darTitulo( ) != null &&
        !nuevoLibro.darTitulo( ).equals( " " ) &&
        nuevoLibro.darISBN( ) != null &&
        !nuevoLibro.darISBN( ).equals( " " ) &&
        nuevoLibro.darPrecio( ) > 0 )
    {
        Libro libro = buscarLibro( nuevoLibro.darISBN( ) );
        if( libro == null )
            catalogo.add( nuevoLibro );
    }
}
```

En esta versión verificamos primero que el libro incluya información correcta.

Luego buscamos en el catálogo otro libro con el mismo ISBN.

Si no lo encontramos, entonces sí lo agregamos al catálogo.

Lo único que no verificamos es que el vector de libros ya esté creado.

Solución

¿Cómo tomar entonces la decisión de qué validar y qué suponer?

La respuesta es que **lo importante no es cuál de las soluciones tomemos**. Lo importante es que aquello que decidamos sea **claro para todos** y que exista un **acuerdo explícito** entre quien utiliza el método y quien lo desarrolla.

Solución

```
public void adicionarLibroCatalogo( Libro nuevoLibro )
{
    if ( nuevoLibro != null &&
        nuevoLibro.darTitulo( ) != null &&
        !nuevoLibro.darTitulo( ).equals( " " ) &&
        nuevoLibro.darISBN( ) != null &&
        !nuevoLibro.darISBN( ).equals( " " ) &&
        nuevoLibro.darPrecio( ) > 0 )
    {
        Libro libro = buscarLibro( nuevoLibro.darISBN( ) );
        if( libro == null )
            catalogo.add( nuevoLibro );
    }
}
```

Quien escribe el cuerpo de un método puede hacer ciertas suposiciones sobre los parámetros o sobre los atributos.

void adicionarLibroCatalogo(Libro nuevoLibro)

Quien llama un método necesita saber cuáles son las suposiciones que hizo quien lo construyó, sin necesidad de entrar a estudiar la implementación.

Contrato

La solución a este problema es establecer claramente un **contrato** en cada método, en el que sean claros sus compromisos y sus suposiciones



Definición del contrato de un método



Este tema está estrechamente relacionado con el proceso de asignar **RESPONSABILIDADES** a las clases. Esta suma de suposiciones y **COMPROMISOS** son las que se integran en los contratos, de manera que **debemos aprender a documentarlas, a leerlas y a manejar los errores** que se pueden producir cuando estos contratos no se cumplen.

Precondiciones y Postcondiciones

El **contrato** de un método establece bajo qué **CONDICIONES** el método tendrá éxito y cuál será el resultado una vez que se termine su ejecución. Por ejemplo, para el método:

```
public void afiliarResulto(String cedula, String nombre)  
throws Exception
```

Podemos establecer que las suposiciones **ANTES** (condiciones previas) de ejecutar el método son:

- La lista de socios ya fue creada.
- La cédula no es null ni vacía.
- No se ha verificado si ya existe un socio con esa cédula.
- El nombre no es null ni vacío.

DESPUÉS (condiciones posteriores) de ejecutar el método, el resultado debe ser uno de los siguientes:

- Todo funcionó bien y el socio se afilió al club.
- Se produjo un error y se informó del problema con una **excepción**. El socio no quedó afiliado al club.

Precondiciones

precondición

Conjunto de suposiciones, expresadas como condiciones que deben ser verdaderas para que el método se ejecute con éxito.

Estas precondiciones pueden referirse a:

- El estado del objeto que va a ejecutar el método (el valor de sus atributos).
- El estado de algún elemento del mundo con el cual el objeto tenga una asociación.
- Condiciones sobre los parámetros de entrada entregados al método.

Tarea 4



Objetivo: Identificar la precondición de un método.

Identifique la precondición del método de la clase `Socio` que permite registrar un consumo, el cual tiene la siguiente signatura:

```
void registrarConsumo( String nombreA, String concepto, double valor )
```

Suposiciones sobre el parámetro `nombreA`.

Suposiciones sobre el parámetro `concepto`.

Suposiciones sobre el parámetro `valor`.

Suposiciones sobre el estado del objeto que va a ejecutar este método.

Suposiciones sobre el estado de alguno de los objetos con los cuales existe una asociación.

Postcondiciones

postcondición

La descripción del resultado obtenido después de ejecutar un método la llamamos su postcondición. Esta se expresa en términos de un **conjunto de condiciones que deben ser verdaderas después de que el método ha sido ejecutado**, siempre y cuando no se haya lanzado una excepción.

Estas postcondiciones hacen referencia a:

- Una descripción del valor de retorno.
- Una descripción del estado del objeto después de haber ejecutado el método.

Tarea 5



Objetivo: Identificar las postcondiciones de algunos métodos.

Describe en términos de condiciones la situación del objeto y el resultado, después de haber ejecutado los siguientes métodos de la clase `Socio`.

```
void registrarConsumo( String nombreA, String concepto, double valor )
```

Descripción del estado del objeto que ejecutó el método, expresada como una lista de condiciones que deben ser verdaderas.

```
boolean existeAutorizado( String nombreAutorizado )
```

Descripción del retorno del método, expresada como una lista de condiciones que deben ser verdaderas.

Preguntas típicas

Preguntas típicas que surgen en el momento de definir un contrato y de implementar un método que lo cumpla

¿Un método debe verificar en algún punto las condiciones que hacen parte de la precondition?

La respuesta es **no**. Lo que aparece en la precondition **se debe suponer como cierto** y se debe utilizar como si lo fuera. Si algo falla en la ejecución por culpa de eso, es el problema de aquél que hizo la llamada sin cumplir el contrato.

¿Qué lugar ocupan las excepciones en los contratos?

Un contrato sólo debe decir que lanza una excepción cuando, aún cumpliéndose todo lo pedido en la precondition, es imposible llegar a cumplir la postcondición. Eso quiere decir que **ninguna excepción** puede asociarse con el incumplimiento de una precondition.

Preguntas típicas

Preguntas típicas que surgen en el momento de definir un contrato y de implementar un método que lo cumpla

¿Qué incluir entonces en la precondition?

En la precondition sólo se deben incluir condiciones que resulten fáciles de garantizar por parte de aquél que utiliza el método. Si le impongo verificaciones cuya verificación previa a la invocación del método le demandará un gran costo en tiempo, terminaremos construyendo programas ineficientes. Si quiero asegurarme de algo así en la ejecución del método, pues basta con eliminarlo de la precondition y lanzar una excepción si no se cumple.

¿Por qué es inconveniente verificar todo dentro del método invocado?

Por ineficiencia. Es mucho mejor repartir las responsabilidades de verificar las cosas entre el que hace el llamado y el que hace el método. Si en el contrato queda claro quién se encarga de qué, es más fácil y eficiente resolver los problemas.

Documentación de los Contratos con Javadoc



Objetivo: Mostrar un contrato completo y la página html generada por la herramienta Javadoc.

En este ejemplo se presenta el contrato del método de la clase `Club` que afilia un nuevo socio. En la parte de abajo aparece la visualización del archivo html generado automáticamente por la herramienta Javadoc.

```
/**
 * Este método afilia un nuevo socio al club.<br>
 *
 * <b>pre:</b> La lista de socios está inicializada (no es null).<br>
 * <b>post:</b> Se ha afiliado un nuevo socio en el club con los datos dados.<br>
 *
 * @param cedula Es la cédula del nuevo socio. cedula != null, cedula != " "
 * @param nombre Es el nombre del nuevo socio. nombre != null, nombre != " "
 *
 * @throws Exception si un socio con la misma cédula ya estaba afiliado al club,
 *                   dispara una excepción indicando que la nueva afiliación no se
 *                   pudo llevar a cabo.
 */
public void afiliarSocio( String cedula, String nombre ) throws Exception
{
}
```


Caso de Estudio: Un Club Social

Se quiere construir una aplicación para manejar la información de socios de un club. Además de los socios, al club pueden ingresar personas autorizadas por éstos, que hayan sido registradas con anterioridad. Tanto los socios como las personas autorizadas pueden realizar consumos en los restaurantes del club. Cada socio está identificado con su nombre y su cédula. Una persona autorizada por un socio se identifica únicamente por su nombre. Cuando un socio (o una persona autorizada por él) realiza un consumo en el club, se crea una factura que será cargada a la cuenta del socio. El club guarda las facturas y permite que en cualquier momento el socio vaya y cancele cualquiera de ellas. Cada factura tiene un concepto que describe el consumo, el valor de lo consumido y el nombre de quien lo hizo.

Interfaz de Usuario

The screenshot shows a window titled 'Club' with three tabs: 'Consumos', 'Facturas', and 'Contabilidad'. The 'Afiliar' tab is selected. Below the tabs, there are two sub-tabs: 'Afiliar' and 'Autorizados'. The 'Afiliar' sub-tab is active, showing a form titled 'Datos Personales' with two input fields labeled 'Nombre' and 'Cédula'. At the bottom of the form is a button labeled 'Afiliar'.

- Esta primera ficha permite afiliar a un nuevo socio al club.
- Hay dos campos obligatorios: uno para el nombre y otro para la cédula del nuevo socio.
- Con el botón **Afiliar** se agrega el socio al club.

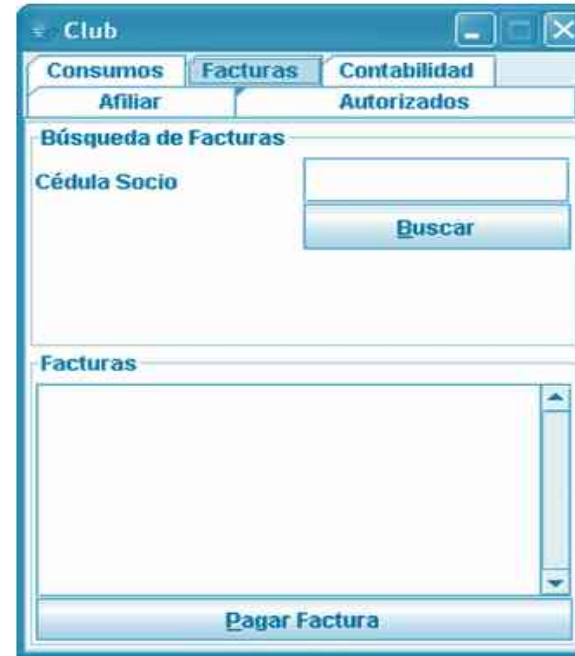
The screenshot shows the same 'Club' window, but now the 'Autorizados' sub-tab is active. It features a section titled 'Búsqueda de Socio' with an input field for 'Cédula Socio' and a button labeled 'Buscar'. Below this is a section titled 'Autorizados' with an input field for 'Nombre' and a large empty list area. At the bottom of the window is a button labeled 'Agregar Autorizado'.

- Esta segunda ficha permite registrar las personas autorizadas por un socio. Para hacerlo se debe localizar primero el socio, para lo cual hay que ingresar su cédula y utilizar el botón **Buscar**.
- Después de seleccionado el socio, es posible registrar personas autorizadas utilizando para esto el botón **Agregar Autorizado**.

Interfaz de Usuario



- Esta tercera ficha del programa permite crear una nueva factura para un socio. Al igual que en el caso anterior, primero se debe localizar el socio, para lo cual hay que indicar su cédula y oprimir el botón **Buscar**.
- Después de localizado el socio, se debe dar el nombre de quien hizo el consumo (sólo puede ser el socio o uno de sus autorizados), el concepto y el monto.
- Con el botón **Registrar** se crea la factura para el socio.



- Desde esta ficha se administran y pagan las facturas de un socio. Primero se debe seleccionar el socio, dando su cédula.
- En la parte de abajo de la ventana aparece la lista de todas las facturas pendientes que tiene el socio.
- Seleccionando una de las facturas de la lista y oprimiendo el botón **Pagar Factura**, ésta se da por cancelada.

Comprensión de Requerimientos

Nombre	R1 - Registrar una persona autorizada por un socio
Resumen	Los socios tienen un conjunto de personas autorizadas que pueden ingresar al club y hacer consumos en sus restaurantes. Este requerimiento permite a un socio registrar una de estas personas.
Entradas	
Nombre	Descripción
nombre	Nombre de la persona autorizada por el socio
cedula	Cédula del socio al que se registrará el autorizado
Resultados	
Nombre	Descripción
Registro de autorizado	El socio tiene una nueva persona autorizada

Comprensión de Requerimientos

Nombre	R2 - Pagar una factura
Resumen	El socio puede pagar una factura, de su lista de facturas pendientes.
Entradas	
Nombre	Descripción
socio	Cédula del socio que pagará la factura
factura	La factura que quiere pagar el socio, de su lista de facturas pendientes
Resultados	
Nombre	Descripción
La factura ha sido pagada	La factura cancelada ya no está pendiente para el socio.

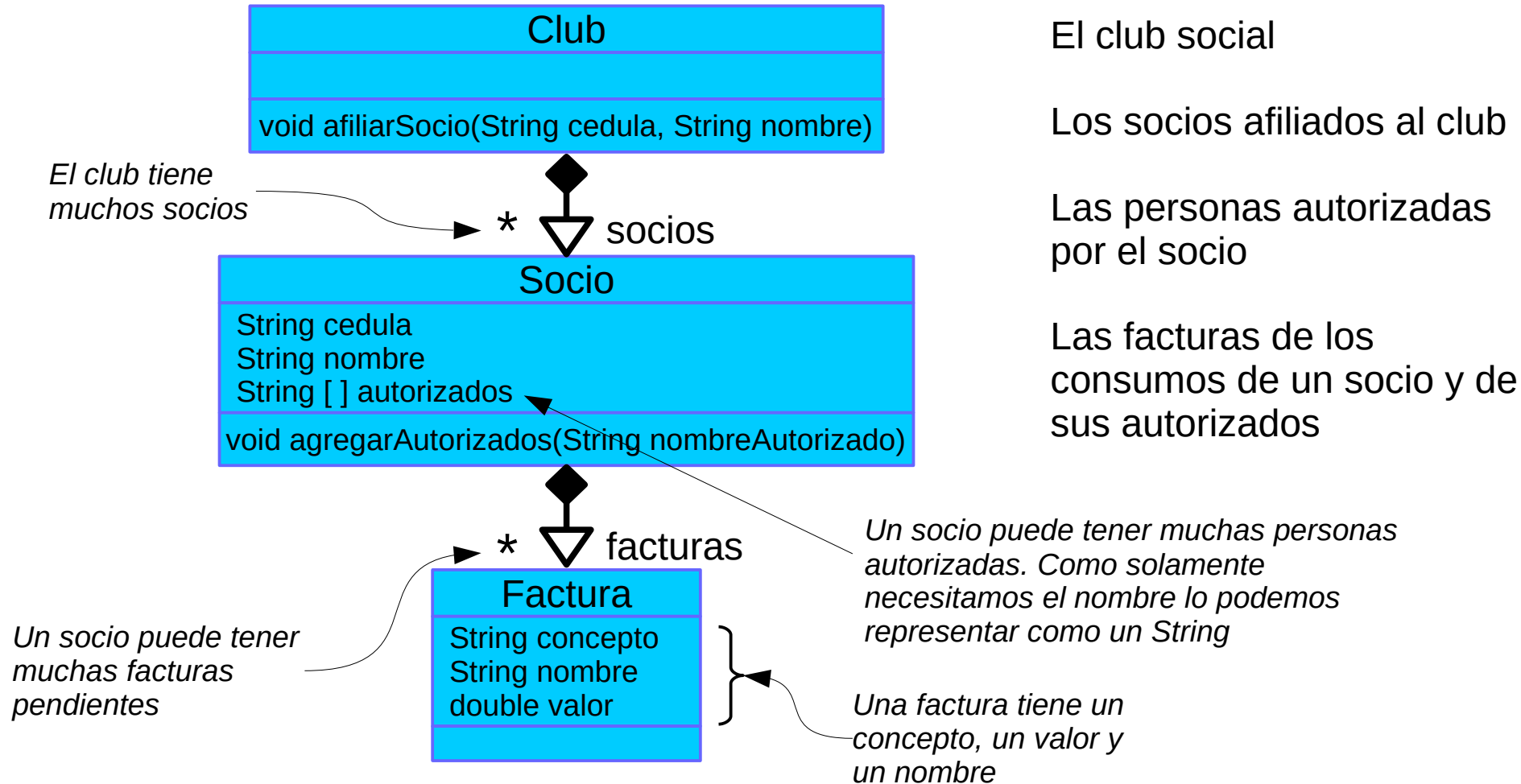
Comprensión de Requerimientos

Nombre	R3 - Afiliar un socio al club
Resumen	Se agrega un nuevo socio al club.
Entradas	
Nombre	Descripción
nombre	Nombre del socio
cedula	Cédula del socio
Resultados	
Nombre	Descripción
Afiliación del socio	Se adicionó el nuevo socio al club

Comprensión de Requerimientos

Nombre	R4 - Registrar un consumo en la cuenta de un socio
Resumen	El socio o una de sus personas autorizadas, puede agregar un consumo a la cuenta del socio.
Entradas	
Nombre	Descripción
concepto	Concepto del consumo
valor	Valor por el cual fue el consumo
nombre	Persona que realizó el consumo
Resultados	
Nombre	Descripción
Registro de consumo	Se crea una nueva factura por el valor del consumo y se registra al socio

Comprensión del Mundo del Problema



Arquitectura de paquetes para el caso del club

interfaz

InterfazClub
PanelAutorizados
PanelBotonesAgregar
PanelBusquedaFacturas
PanelBusquedaPersonas
PanelBusquedaSocio
PanelDatosPersonales
PanelFacturas
PanelRegistroConsumos

test

ClubTest
SocioTest

mundo

Club
Socio
Factura

Declaración de Clases

```
public class Club
{
    //-----
    // Atributos
    //-----
    private ArrayList socios;
}
```



A partir del diagrama de clases, vemos que hay una asociación de cardinalidad variable entre la clase Club y la clase Socio.



Esta asociación representa el grupo de socios afiliados al club, que modelaremos como un vector (una contenedora de tamaño variable).

```
public class Socio
{
    //-----
    // Atributos
    //-----
    private String cedula;
    private String nombre;
    private ArrayList facturas;
    private ArrayList autorizados;
}
```



Un socio tiene una cédula y un nombre, los cuales se declaran como atributos de la clase String.



Para representar las personas autorizadas por el socio, utilizaremos un vector de cadenas de caracteres (autorizados), en donde almacenaremos únicamente sus nombres.



Para guardar las facturas pendientes del socio, tendremos un segundo vector (facturas), cuyos elementos serán objetos de la clase Factura.

Declaración de Clases

```
public class Factura
{
    //-----
    // Atributos
    //-----
    private String concepto;
    private String autorizado;
    private double valor;
}
```



La clase Factura es la más sencilla de las tres. Sólo tiene tres atributos para representar la información que necesitamos: el concepto de la factura, el nombre de la persona y el monto.

Asignación de Responsabilidades – Técnica del Experto



La técnica del experto define quién es responsable de hacer algo, pero son las reglas del mundo las que nos dicen cómo cumplir con dicha responsabilidad.

R2:
afiliar a un socio



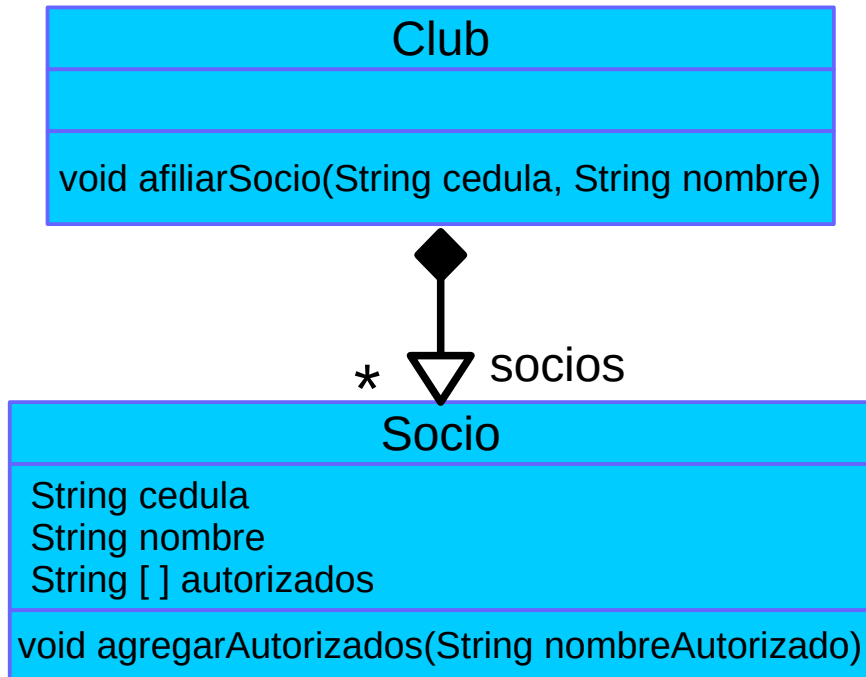
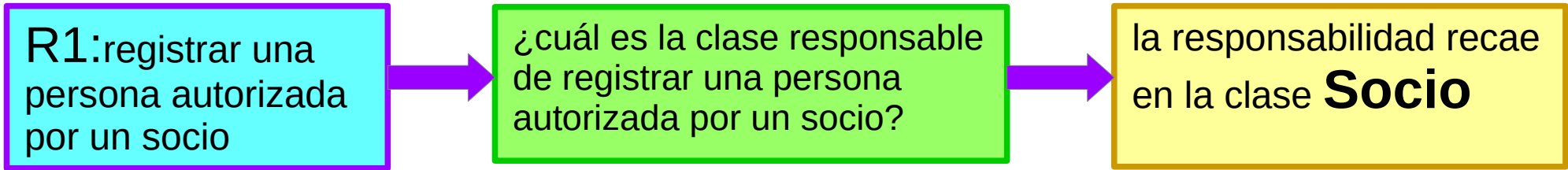
¿quién es el responsable de agregar un nuevo socio al club?



la responsabilidad debe recaer en la clase dueña de la lista de socios.

Hablando en términos de métodos, esa decisión nos dice que no debemos tener un método que retorne el arreglo de socios para que otro pueda agregar allí al nuevo, sino que debemos tener un método para afiliar un socio, en la clase Club, que se encargue de esta tarea.

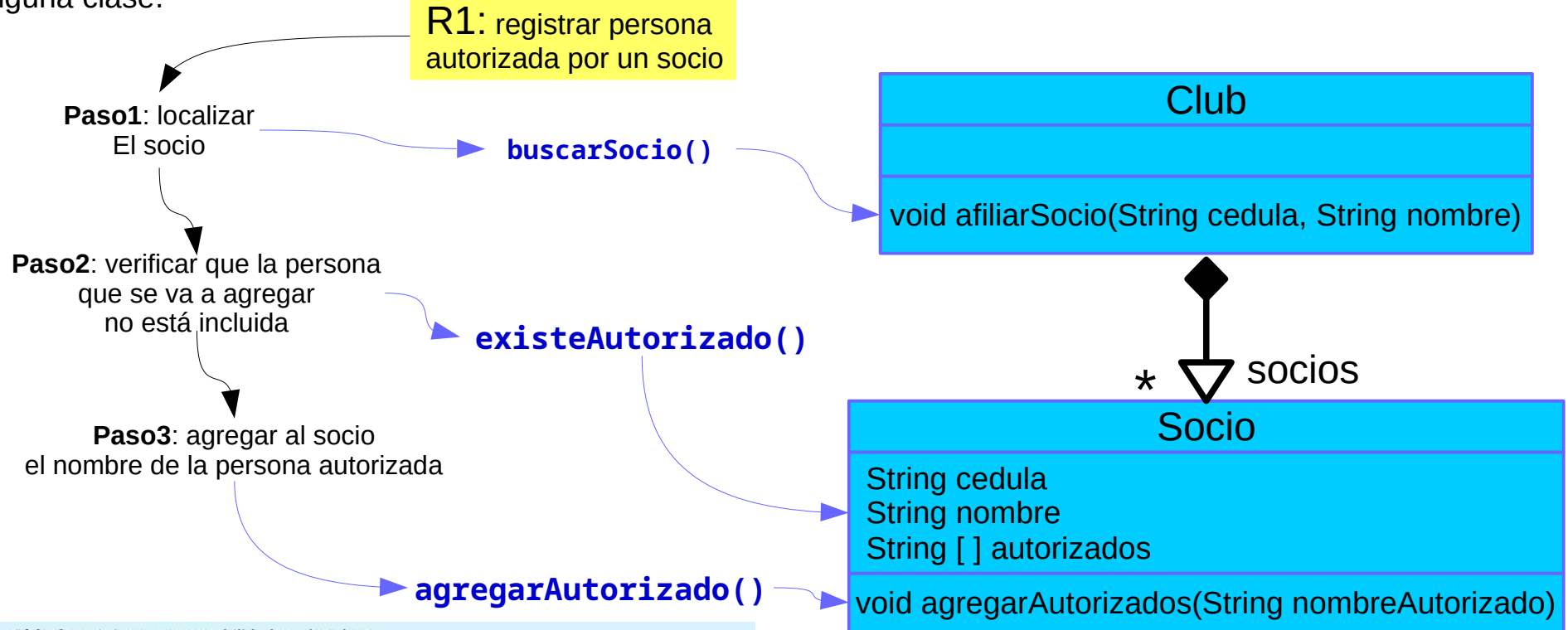
Asignación de Responsabilidades – Técnica del Experto



Para usar la técnica del experto debemos recorrer todos los atributos y asociaciones del diagrama de clases y definir los métodos con los cuales vamos a manejar dicha información.

Asignación de Responsabilidades – Técnica de Descomposición de Requerimientos

Muchos de los requerimientos funcionales requieren realizar más de un paso para satisfacerlos. Puesto que cada paso corresponde a una invocación de un método sobre algún objeto existente del programa, podemos utilizar esta secuencia de pasos como guía para definir los métodos necesarios y, luego, asignar esa responsabilidad a alguna clase.



Objetivo: Asignar responsabilidades a las clases.

Decida a qué clase corresponde la responsabilidad de cada uno de los pasos definidos en la tarea anterior y justifique su decisión.

Manejo de Excepciones – anunciar que se puede producir la excepción

```
public void afiliarResulto(String c, String n) throws Exception  
{ ...  
}
```



Al informar que un método lanza una excepción, estamos agrupando dos casos posibles:

- Caso 1: la excepción va a ser creada y lanzada por el mismo método que la declara. Esto quiere decir que es el mismo método el que se encarga de detectar el problema, de crear la instancia de la clase `Exception` y de lanzarla.
- Caso 2: la excepción fue producida por alguna instrucción en el cuerpo del método que hace la declaración, el cual decide no atraparla sino dejarla seguir. Este "dejarla seguir" se informa también con la misma cláusula `throws`.

Manejo de Excepciones

```
public void afiliarSocio(String cd, String nm) throws Exception
{
    // Revisa que no haya ya un socio con la misma cédula
    Socio s = buscarSocio( cd );
    if( s == null )
    {
        // Se crea el objeto para representar el nuevo socio
        Socio nuevoSocio = new Socio( cd, nm );
        // Se agrega el nuevo socio al club
        socios.add( nuevoSocio );
    }
    else
    {
        // Si el socio ya estaba, lanza esta excepción
        throw new Exception( " El socio ya existe " );
    }
}
```

Contrato de un Método

Objetivo: Identificar la precondition de un método.

Identifique la precondition del método de la clase Socio que permite registrar un consumo, el cual tiene la siguiente signatura:

void registrarConsumo(String nombreA, String concepto, double valor)

Suposiciones sobre el parámetro nombreA	
Suposiciones sobre el parámetro concepto	
Suposiciones sobre el parámetro valor	
Suposiciones sobre el estado del objeto que va a ejecutar este método	
Suposiciones sobre el estado de alguno de los objetos con los cuales existe una asociación	

Contrato de un Método

Objetivo: Identificar las postcondiciones de algunos métodos.

Describe en términos de condiciones la situación del objeto y el resultado, después de haber ejecutado los siguientes métodos de la clase Socio.

```
void registrarConsumo( String nombreA, String concepto, double valor )
```

Descripción del estado del objeto que ejecutó el método, expresada como una lista de condiciones que deben ser verdaderas.

```
boolean existeAutorizado( String nombreAutorizado )
```

Descripción del retorno del método, expresada como una lista de condiciones que deben ser verdaderas.

Ejercicio

Estudiar la definición de los contratos de los métodos de las clases Club, Socio y Factura, y contestar las siguientes preguntas:

- 1) ¿Qué pasa si el método buscarSocio de la clase Club no encuentra el socio cuya cédula recibió como parámetro?
- 2) ¿Qué precondition exige el método buscarSocio de la clase Club respecto del atributo que representa la cédula?
- 3) ¿Qué retorna el método darConcepto de la clase Factura? ¿Qué condiciones cumple dicho valor? ¿Qué nombre se usó en el contrato para representar el valor de retorno?
- 4) ¿Cuál es la postcondición del método pagarFactura de la clase Socio?
- 5) ¿Cuál es la precondition sobre el parámetro valor en el método registrarConsumo de la clase Socio?
- 6) ¿En cuántos casos lanza una excepción el método agregarAutorizado de la clase Socio?
- 7) ¿Qué sucede si en el método agregarAutorizado de la clase Socio, el parámetro de entrada corresponde al nombre del socio?