

# 04

## DEFINICIÓN Y CUMPLIMIENTO DE RESPONSABILIDADES



# 1. Objetivos Pedagógicos

Al final de este nivel el lector será capaz de:

- Utilizar la definición de un [contrato](#) para construir un [método](#).
- Utilizar la definición del [contrato](#) de un [método](#) para invocarlo de manera correcta.
- Utilizar algunas técnicas simples para realizar la [asignación](#) de responsabilidades a las [clases](#).
- Utilizar la técnica metodológica de [dividir y conquistar](#) para resolver los requerimientos funcionales de un problema.
- Escribir una [clase](#) completa del modelo del mundo, siguiendo una [especificación](#) dada en términos de un conjunto de contratos.
- Documentar los contratos de los métodos utilizando la sintaxis definida por la herramienta [Javadoc](#).
- Utilizar la [clase](#) `Exception` de Java para manejar los problemas asociados con la violación de los contratos.
- Entender la documentación de un conjunto de clases escritas por otros y utilizar dicha documentación para poder incorporar y usar adecuadamente dichas clases en un programa que se está construyendo.

## 2. Motivación

En el nivel 3 presentamos un caso de estudio relacionado con una tienda de libros. En dicho ejemplo definimos el **método** registrarLibro de la **clase** TiendaDeLibros, que nos permitía agregar un libro nuevo al catálogo de la tienda. Dicho **método** recibía como **parámetro** las características del libro que se quería añadir, y tenía la siguiente **signatura**:

```
Libro registrarLibro(String pTitulo, String pIsbn, double pPrecioVenta, double pPrecioCompra, String pRutaImagen)
```

Si alguien nos pidiera que implementáramos dicho **método**, sería indispensable que nos contestara antes las siguientes preguntas:

- ¿Las características del libro que se va a registrar son válidas, es decir, su título, su ISBN, su precio de compra, su precio de venta y su imagen tienen un valor definido y correcto? ¿Debemos verificar que los precios sean un número positivo antes de adicionarlo al catálogo? ¿Ya alguien verificó eso y es una pérdida de tiempo volverlo a hacer?
- ¿Ya se verificó que el libro que se desea registrar no esté incluido en el catálogo? ¿Debemos verificar si existe ya un libro con ese ISBN antes de agregarlo al catálogo?
- ¿Ya está creado el **vector** que representa el catálogo de la tienda de libros? 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**?
- Fíjese que aunque la **signatura** 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. Debe ser claro que la **implementación** del **método** puede cambiar radicalmente, dependiendo de la respuesta que se dé a las preguntas que se plantearon anteriormente. Por ejemplo, si hacemos la suposición de que no hay en el catálogo otro libro con el mismo ISBN del libro que se va a añadir, el cuerpo del **método** sería el siguiente:

```
public Libro registrarLibro( String pTitulo, String pIsbn, double pPrecioVenta, double pPrecioCompra, String pRutaImagen )
{
    Libro nuevo = new Libro( pTitulo, pIsbn, pPrecioVenta, pPrecioCompra, pRutaImagen );
    catalogo.add( nuevo );
    return nuevo;
}
```

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

El asunto es que si nuestra suposición no es válida, vamos a crear dos libros en el catálogo con el mismo ISBN, lo cual introduce una inconsistencia en la información y puede generar problemas en el programa. Esta [clase](#) de errores son de extrema gravedad, puesto que permiten llegar a un estado en el modelo del mundo que no corresponde a una situación válida de la realidad. La solución más simple parecería, entonces, hacer siempre todas las verificaciones, como se muestra en la siguiente [implementación](#) del [método](#):

```
public Libro registrarLibro( String pTitulo, String pIsbn, double pPrecioVenta, double
    pPrecioCompra, String pRutaImagen )
{
    Libro nuevo = null;

    if( pTitulo != null && !pTitulo.equals( " " ) &&
        pIsbn != null && !pIsbn.equals( " " ) &&
        pPrecioVenta > 0 && pPrecioCompra > 0 &&
        pRutaImagen != null && !pRutaImagen.equals( " " ) )
    {

        Libro buscado = buscarLibroPorISBN( pIsbn );

        if( buscado == null )
        {
            nuevo = new Libro( pTitulo, pIsbn, pPrecioVenta, pPrecioCompra, pRutaImagen );
            catalogo.add( nuevo );
        }
    }
    return nuevo;
}
```

- En esta versión verificamos primero que la información del libro esté 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.

Este otro extremo parece un poco exagerado, puesto que algunas verificaciones pueden tomar mucho tiempo, ser costosas e inútiles. ¿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 decidimos sea claro para todos y que exista un acuerdo explícito entre quien utiliza el [método](#) y quien lo desarrolla. Si nosotros decidimos que dentro del [método](#) no vamos a verificar que el ISBN exista en el catálogo, aquél que llama el [método](#) deberá saber que es su obligación verificar esto antes de hacer la llamada.

De esta discusión podemos sacar dos conclusiones:

- Quien escribe el cuerpo de un **método** puede hacer ciertas suposiciones sobre los parámetros o sobre los atributos, y esto puede afectar en algunos casos el resultado. El problema es que dichas suposiciones sólo quedan expresadas como parte de las instrucciones del **método**, y no son necesariamente visibles por el programador que va a utilizarlo. Sería muy dispendioso para un programador tener que leer el código de todos los métodos que utiliza.
- Quien llama un **método** necesita saber cuáles son las suposiciones que hizo quien lo construyó, sin necesidad de entrar a estudiar la **implementación**. Si no tiene en cuenta estas suposiciones, puede obtener resultados inesperados (por ejemplo, dos libros con el mismo ISBN).

La solución a este problema es establecer claramente un **contrato** en cada **método**, en el que sean claros sus compromisos y sus suposiciones, tal como se ilustra en la [figura 4.1](#).

**Fig. 4.1 Contrato entre dos sujetos: el que lo implementa y el que lo usa**



Un **contrato** se establece entre dos sujetos: el que implementa un **método** y el que lo usa. El primero se compromete a escribir un **método** que permita conseguir un resultado si se cumplen ciertas condiciones o suposiciones, las cuales se hacen explícitas como parte del **contrato** (por ejemplo, adquiere el compromiso de añadir un libro, si no hay ningún otro libro con el mismo ISBN). El segundo sujeto puede usar el servicio que implementó el primero y

se compromete a cumplir las condiciones de uso. Esto puede implicar hacer verificaciones sobre la información que pasa como [parámetro](#) o garantizar algún aspecto del estado del mundo.

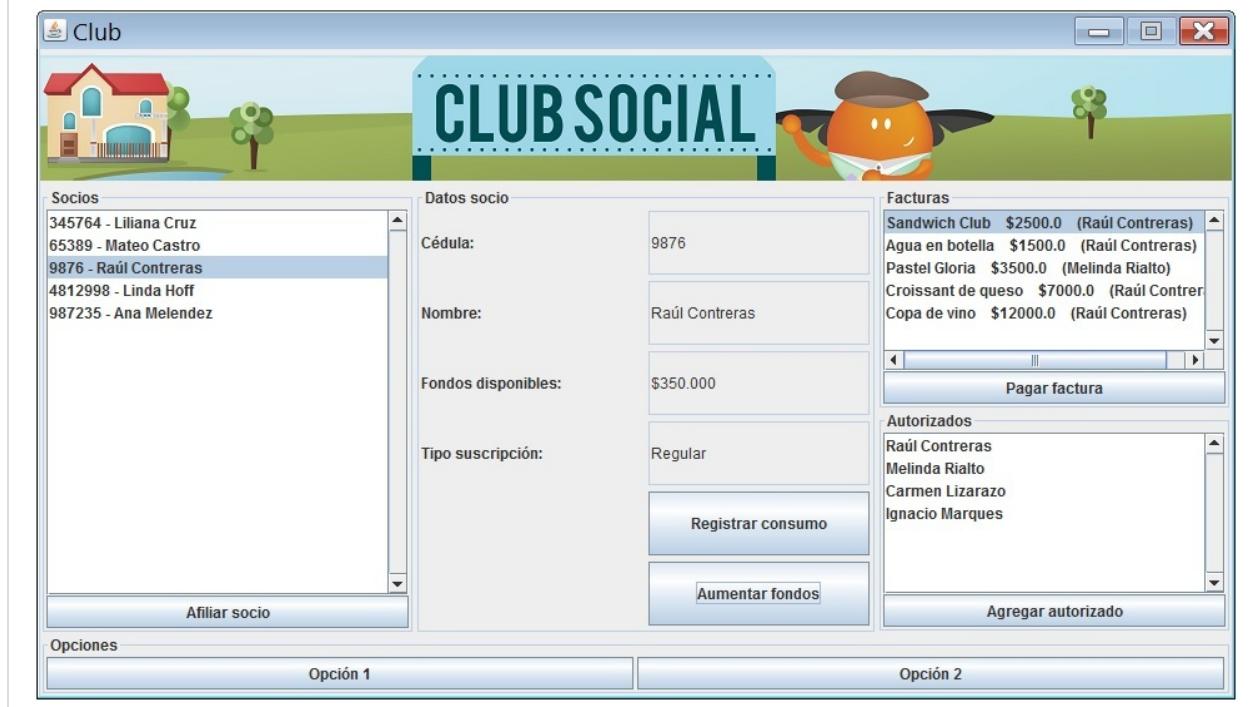
En este capítulo vamos a concentrarnos en la manera de definir los contratos de los métodos. Este tema está estrechamente relacionado con el proceso de asignar responsabilidades a las clases, algo crítico, puesto que es allí donde tomamos las decisiones de quién es el responsable de hacer qué. Esta suma 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.

### 3. Caso de Estudio N° 1: Un Club Social

Se quiere construir una aplicación para manejar la información de socios de un club. El club maneja dos tipos de suscripciones de socios: Regular o VIP. El número máximo de socios VIP que maneja el club es 3. 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. No puede haber dos socios con la misma cédula. Cuando un socio se afilia al club debe hacerlo con un fondo inicial (para pagar sus propios consumos y los de sus personas autorizadas) según el tipo de suscripción que tenga. Los socios regulares deben afiliarse con un fondo inicial de \$50.000 y los socios VIP con \$100.000. Los socios pueden aumentar sus fondos en cualquier momento, pero tienen una restricción máxima, que también depende de su tipo de suscripción, de la siguiente manera: regulares \$1'000.000 y VIP \$5'000.000. Para que un socio pueda añadir personas autorizadas a su lista, debe contar con fondos.

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 es cargada a la cuenta del socio. Cada factura tiene un concepto que describe el consumo, el valor de lo consumido y el nombre de quien lo hizo. Para hacer un consumo, el socio debe contar con fondos suficientes para pagarlo. El club guarda las facturas y permite que en cualquier momento el socio vaya y cancele cualquiera de ellas. Una factura sólo puede ser pagada si el socio cuenta con fondos suficientes para hacerlo. Al pagar la factura, esta es eliminada de la lista de facturas por pagar del socio y se descuenta el valor de los fondos del socio.

La [interfaz de usuario](#) que se diseñó para este ejemplo se muestra en la [figura 4.2](#). Esta interfaz tiene varios botones para que el usuario pueda seleccionar los distintos servicios de la aplicación.

**Fig. 4.2 Diseño de la interfaz de usuario para el caso de estudio del club**

- El botón "Afiliar socio" permite afiliar a un nuevo socio al club.
- El botón "Aregar autorizado" permite registrar las personas autorizadas por un socio.
- El botón "Registrar consumo" permite crear una nueva factura para un socio.
- A la derecha de la ventana aparece la lista de todas las facturas pendientes por pagar que tiene el socio. Para cada factura se indica el concepto del consumo, el valor y la persona que lo realizó.
- Seleccionando una de las facturas de la lista y oprimiendo el botón "Pagar factura", ésta se da por cancelada.

## 3.1. Comprensión de los Requerimientos

La primera tarea de este nivel consiste en la identificación y [especificación](#) de los requerimientos funcionales del problema.

### Tarea 1

**Objetivo:** Describir los requerimientos funcionales del caso de estudio.

Para el caso de estudio del club, complete la siguiente tabla con la [especificación](#) de los requerimientos funcionales.

#### Requerimiento funcional 1

Nombre	<b>R1 - Agregar una persona autorizada por un socio.</b>
Resumen	Agrega un autorizado a la lista de autorizados de un socio. Una persona autorizada puede ingresar al club y realizar consumos en sus restaurantes.
Entradas	(1) socio: cédula del socio al que se registrará el autorizado. (2)nombre: nombre de la persona autorizada por el socio.
Resultado	Se agrega el autorizado a la lista de autorizados del socio. Si el nombre del socio es igual al nombre del autorizado, no se agrega el autorizado y se muestra un mensaje al usuario indicándolo. Si el autorizado ya existe en la lista, no se agrega el autorizado y se muestra un mensaje al usuario indicándolo. Si el socio no tiene fondos para financiar un nuevo autorizado, se muestra un mensaje al usuario indicándolo.

### Requerimiento funcional 2

Nombre	<b>R2 - Pagar una factura.</b>
Resumen	Paga una factura de la lista de facturas pendientes de un socio.
Entradas	(1) socio: cédula del socio que pagará la factura. (2) factura: la factura que quiere pagar el socio, de su lista de facturas pendientes.
Resultado	Se elimina la factura de la lista de facturas pendientes de un socio y se disminuyen los fondos disponibles por el valor del consumo. Si el socio no tiene fondos suficientes para pagar la factura, no se elimina la factura y se muestra un mensaje al usuario indicándolo.

### Requerimiento funcional 3

Nombre	R3 - Afiliar un socio al club.
Resumen	
Entradas	
Resultado	

**Requerimiento funcional 4**

Nombre	R4 - Registrar un consumo.
Resumen	
Entradas	
Resultado	

**Requerimiento funcional 5**

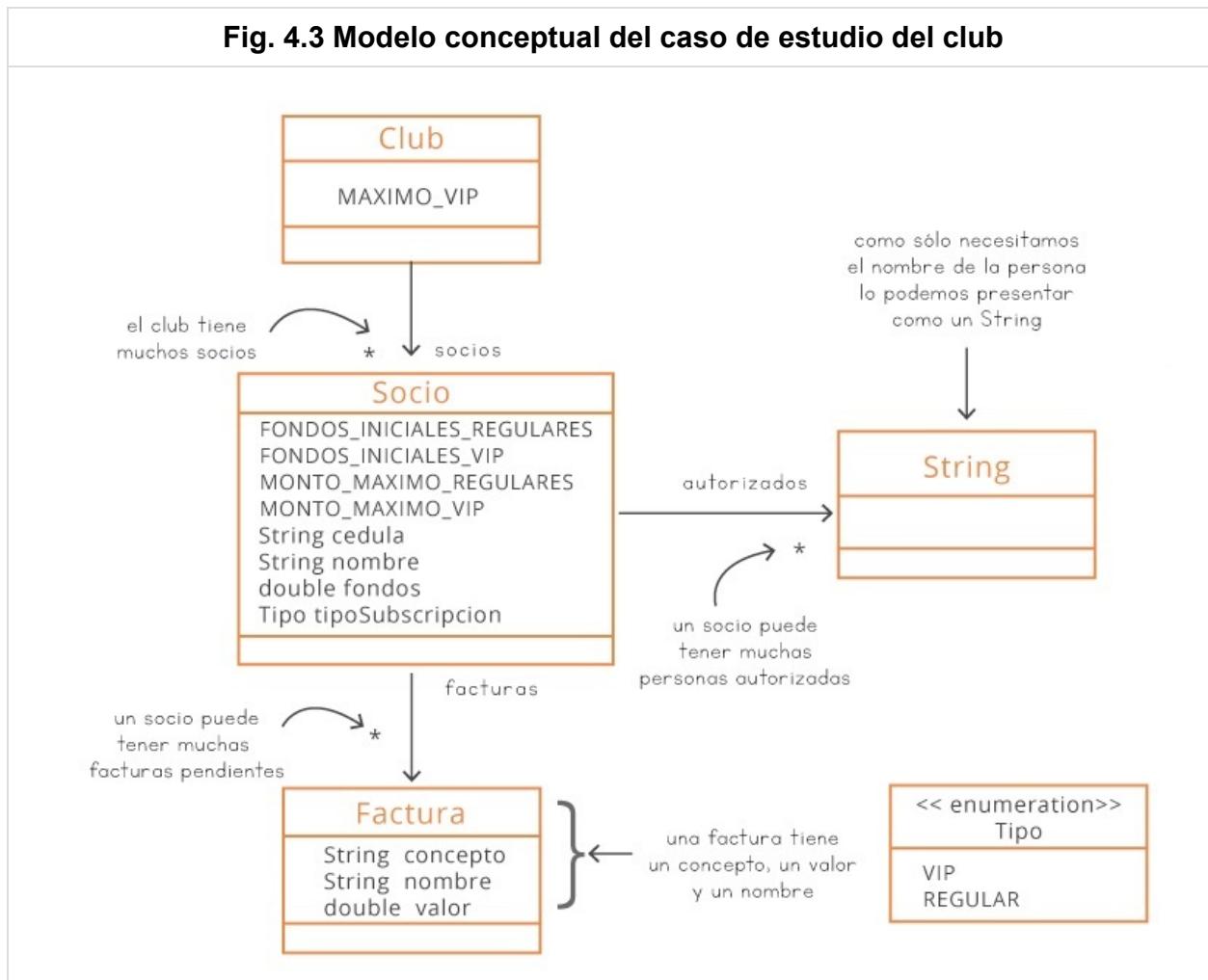
Nombre	R5 - Aumentar los fondos de la cuenta de un socio.
Resumen	
Entradas	
Resultado	

## 3.2. Comprensión del Mundo del Problema

En la [figura 4.3](#) aparece el modelo conceptual del caso de estudio. Allí podemos identificar las entidades del problema:

- El club social.
- Los socios afiliados al club.
- Las personas autorizadas por el socio.
- Las facturas de los consumos de un socio y de sus autorizados.

Fig. 4.3 Modelo conceptual del caso de estudio del club



### 3.3. Definición de la Arquitectura

La solución de este caso de estudio la dividimos en tres subproblemas, de acuerdo con la arquitectura presentada en el nivel 1. La solución de cada uno de los componentes del programa (modelo del mundo, [interfaz de usuario](#) y pruebas) va expresada como un conjunto de clases, en un [paquete](#) distinto, tal como se muestra en la [figura 4.4](#).

**Fig. 4.4 Arquitectura de paquetes para el caso del club**

En este nivel vamos a trabajar únicamente en las clases que corresponden al paquete que implementa el modelo del mundo. En el nivel 5, veremos la manera de construir las clases del paquete que implementa la [interfaz de usuario](#).

## 3.4. Declaración de las Clases

En esta sección presentamos las principales decisiones de modelado de los atributos y las asociaciones, mostrando las declaraciones en Java de las tres clases del modelo del mundo (`Club`, `Socio`, `Factura`). La definición de los métodos se hará a lo largo del nivel, ya que éste es el tema central de esta parte del libro.

```

public class Club
{
    // -----
    // Constantes
    // -----

    /**
     * Cantidad máxima de socios VIP que acepta el club.
     */
    public final static int MAXIMO_VIP = 3;

    // -----
    // Atributos
    // -----

    /**
     * Lista de socios del club.
     */
    private ArrayList<Socio> 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
{
    // -----
    // Enumeraciones
    // -----

    /**
     * Enumeraciones para los tipos de suscripción.
     */
    public enum Tipo
    {
        /**
         * Representa el socio VIP.
         */
        VIP,
        /**
         * Representa el socio regular.
         */
        REGULAR
    }
    // -----
    // Constantes
    // -----

```

```
/**  
 * Dinero base con el que empiezan todos los socios regulares.  
 */  
public final static double FONDOS_INICIALES_REGULARES = 50000;  
  
/**  
 * Dinero base con el que empiezan todos los socios VIP.  
 */  
public final static double FONDOS_INICIALES_VIP = 100000;  
  
/**  
 * Dinero máximo que puede tener un socio regular en sus fondos.  
 */  
public final static double MONTO_MAXIMO_REGULARES = 1000000;  
  
/**  
 * Dinero máximo que puede tener un socio VIP en sus fondos.  
 */  
public final static double MONTO_MAXIMO_VIP = 5000000;  
  
// -----  
// Atributos  
// -----  
  
/**  
 * Cédula del socio.  
 */  
private String cedula;  
  
/**  
 * Nombre del socio.  
 */  
private String nombre;  
  
/**  
 * Dinero que el socio tiene disponible.  
 */  
private double fondos;  
  
/**  
 * Tipo de subscripción del socio.  
 */  
private Tipo tipoSubscripcion;  
  
/**  
 * Facturas que tiene por pagar el socio.  
 */  
private ArrayList<Factura> facturas;  
  
/**  
 * Nombres de las personas autorizadas para este socio.  
 */  
private ArrayList<String> autorizados;
```

```
}
```

- Un socio tiene una cédula y un nombre, los cuales se declaran como atributos de la [clase String](#).
- El dinero disponible que tiene un socio para pagar sus consumos se declara mediante el [atributo fondos](#) de tipo double.
- Los posibles valores que puede tomar el tipo de suscripción se modela a través de una enumeración llamada [Tipo](#), cuyos posibles valores son VIP o REGULAR.
- 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](#).

```
public class Factura
{
    // -----
    // Atributos
    // -----
    /**
     * Es la descripción del consumo que generó esta factura.
     */
    private String concepto;

    /**
     * Es el valor del consumo que generó la factura.
     */
    private double valor;

    /**
     * Nombre de la persona que hizo el consumo que generó la factura.
     */
    private String nombre;
}
```

# 4. Asignación de Responsabilidades

## 4.1. La Técnica del Experto

La primera técnica de [asignación](#) de responsabilidades que vamos a utilizar se llama el [experto](#). Esta técnica establece que el dueño de la información es el responsable de ella, y que debe permitir que otros tengan acceso y puedan pedir que se cambie su valor. Esta técnica la hemos venido utilizando de manera intuitiva desde el nivel 1. Por ejemplo, en el caso de estudio del empleado, dado que la [clase](#) Empleado tiene un [atributo](#) llamado salario, esta técnica nos dice que debemos definir en esa [clase](#) algunos métodos para consultar y modificar esta información.

Esto no quiere decir que se deban definir siempre dos métodos por [atributo](#), uno para retornar el valor y el otro para modificarlo. Hay casos en los cuales la modificación debe seguir reglas distintas a la simple [asignación](#) de un valor. Siguiendo con el caso del empleado, en la empresa se puede establecer que los cambios de salario siempre se hacen como aumentos porcentuales. Al usar la [técnica del experto](#) se debe tener en cuenta que las modificaciones deben reflejar las reglas del mundo en donde se mueve la [clase](#), y que son estos dos criterios los que definen las responsabilidades y las signaturas de los métodos que se deben incluir. Para el ejemplo que venimos desarrollando, en lugar de un [método](#) con [signatura](#) cambiarSalario(nuevoSalario) deberíamos incluir un [método](#) que cambie los salarios por aumento aumentarSalario(porcentaje). Esta misma idea vale para los métodos que son responsables de dar información. Suponga por ejemplo que se guarda como parte de la información del empleado la palabra clave con la cual tiene acceso al sistema de información de la empresa. En ese caso, en lugar de un [método](#) que retorne dicha información (darPalabraClave()) deberíamos, por razones de seguridad, incluir un [método](#) que informe si la cadena que tecleó el usuario es su palabra clave (esValida(entrada)).

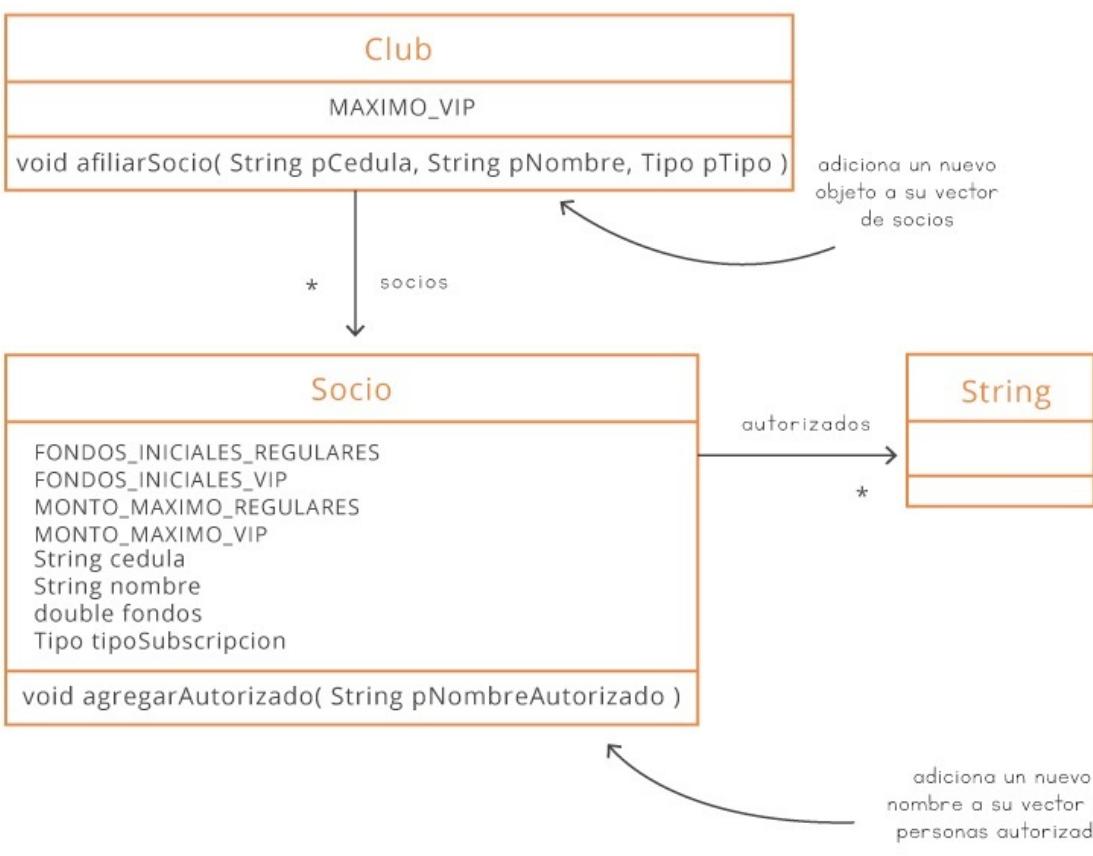
La [técnica del experto](#) define quién es responsable de hacer algo, pero son las reglas del mundo las que nos dicen cómo cumplir con dicha [responsabilidad](#).

Pasemos ahora al caso de estudio del club. Como consecuencia del [requerimiento funcional](#) de afiliar un socio, nos tenemos que preguntar ¿quién es el responsable de agregar un nuevo socio al club? Si aplicamos la [técnica del experto](#), la respuesta es que la [responsabilidad](#) debe recaer en la [clase](#) dueña de la lista de socios. Esto nos lleva a decidir que, dado que el club es el dueño de la lista de socios, es él quien tiene la [responsabilidad](#) de agregar un socio al club. Hablando en términos de métodos, esa decisión nos dice que

no debemos tener un [método](#) que retorne el [vector](#) 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.

Siguiendo con el caso del club, suponga que debemos decidir cuál es la [clase](#) responsable de registrar una persona autorizada por un socio. Si aplicamos la [técnica del experto](#), la respuesta es que debe hacerlo el dueño de la lista de autorizados, o sea, la [clase](#) Socio. En ese caso la [signatura](#) del [método](#) sería void [agregarAutorizado\(String nombre\)](#) (ver [figura 4.5](#)).

**Fig. 4.5 Asignación inicial de responsabilidades a las clases del caso de estudio**



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. Veremos más ejemplos de la utilización de esta técnica en las secciones siguientes.

## 4.2. La Técnica de Descomposición de los 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**. Esta técnica se denomina **descomposición de los requerimientos funcionales**.

La manera más sencilla de hacer la identificación es tratar de descomponer los requerimientos funcionales en los subproblemas que debemos resolver para poder satisfacer el requerimiento completo. Por ejemplo, para el requerimiento de pagar una factura, podemos imaginar que necesitamos realizar tres pasos, que sugieren la necesidad de tres métodos:

- Buscar si el socio que quiere pagar la factura existe (buscarSocio).
- Si el socio existe, obtener todas sus facturas pendientes (darFacturas).
- Pagar la factura seleccionada (pagarFactura).

Para el requerimiento de registrar una persona autorizada de un socio, podemos concluir que necesitamos también tres pasos, cada uno con un **método** asociado:

- Buscar si existe el socio a quien se le va a agregar una persona autorizada (buscarSocio).
- Dado el nombre de una persona, verificar si esa persona ya pertenece al grupo de los autorizados del socio (existeAutorizado).
- Asociar con el socio una nueva persona autorizada (agregarAutorizado).

## Tarea 2

**Objetivo:** Hacer la descomposición en pasos de un **requerimiento funcional**.

Haga la descomposición en pasos del **requerimiento funcional** de realizar un consumo en el club.

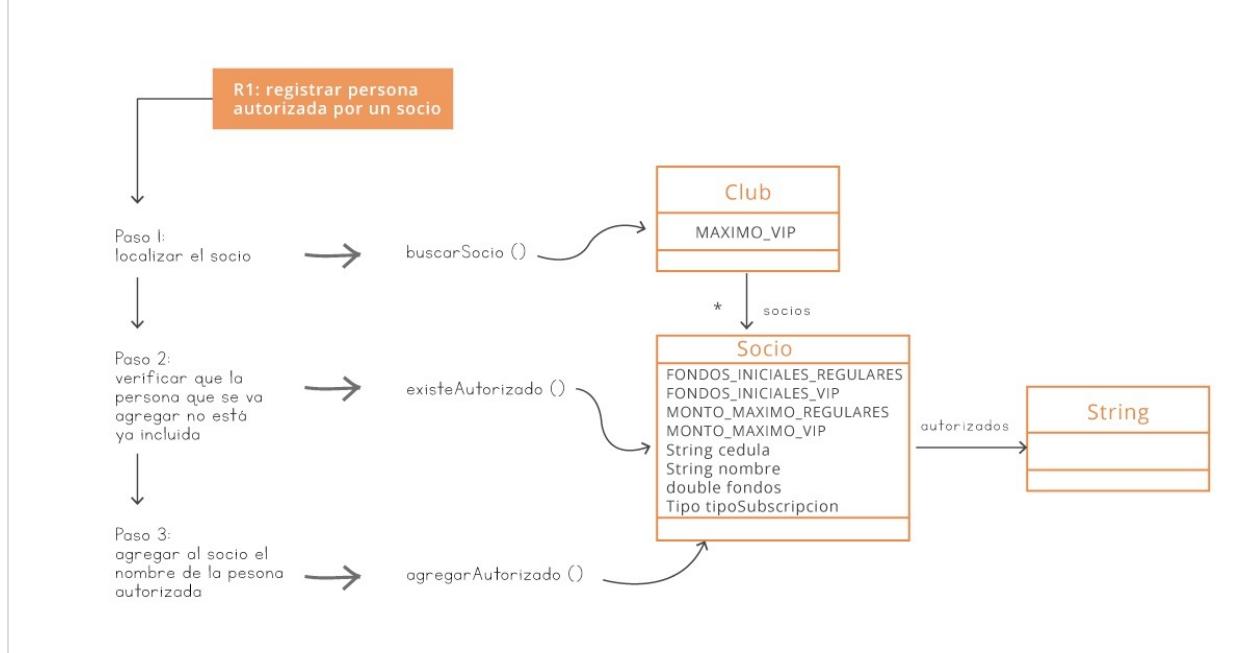


Una vez identificados los servicios que nuestra aplicación debe proveer, podemos utilizar la **técnica del experto** para decidir la manera de distribuir las responsabilidades entre las clases. Continuando con nuestro ejemplo anterior, podemos hacer la siguiente distribución de responsabilidades:

- El servicio buscarSocio debe ser **responsabilidad** de la **clase Club**, porque es el club quien tiene la información de la lista de socios.
- El servicio darFacturas debe ser **responsabilidad** de la **clase Socio**, porque cada socio tiene la información de la lista de sus facturas pendientes.
- El servicio existeAutorizado debe ser **responsabilidad** de la **clase Socio**, porque cada socio tiene la información de la lista de sus autorizados.
- El servicio agregarAutorizado debe ser **responsabilidad** de la **clase Socio**, porque cada socio tiene la información de la lista de sus autorizados.

En la **figura 4.6** se ilustra una parte del proceso de **asignación** de responsabilidades para el caso del club.

**Fig. 4.6 Proceso de asignación de responsabilidades para el caso de estudio**



## Tarea 3

**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.



## 5. Manejo de las Excepciones

Una [excepción](#) es la indicación de que se produjo un error en el programa. Las excepciones, como su nombre lo indica, se producen cuando la ejecución de un [método](#) no termina correctamente, sino que termina de manera excepcional como consecuencia de una situación no esperada.

Cuando se produce una situación anormal durante la ejecución de un programa (por ejemplo se accede a un [objeto](#) que no ha sido inicializado o tratamos de acceder a una posición inválida en un [vector](#)), si no manejamos de manera adecuada el error que se produce, el programa va a terminar abruptamente su ejecución. Decimos que el programa deja de funcionar y es muy probable que el usuario que lo estaba utilizando ni siquiera sepa qué fue lo que pasó.

Cuando durante la ejecución de un [método](#) el computador detecta un error, crea un [objeto](#) de una [clase](#) especial para representarlo (de la [clase](#) Exception en Java), el cual incluye toda la información del problema, tal como el punto del programa donde se produjo, la causa del error, etc. Luego, "dispara" o "lanza" dicho [objeto](#) (throw en inglés), con la esperanza de que alguien lo atrape y decida como recuperarse del error. Si nadie lo atrapa, el programa termina, y en la consola de ejecución aparecerá toda la información contenida en el [objeto](#) que representaba el error. Este [objeto](#) se conoce como una [excepción](#). En el ejemplo 1 se ilustra esta idea.

### Ejemplo 1

**Objetivo:** Dar una idea global del concepto de [excepción](#).

Este ejemplo ilustra el caso en el cual durante la ejecución de un [método](#) se produce un error y el computador crea un [objeto](#) para representarlo y permitir que en alguna parte del programa alguien lo atrape y lo use para evitar que el programa deje de funcionar.

```
public class C1
{
    private C2 atr;

    public void m1( )
    {
        atr.m2( );
    }
}
```

- Suponga que tenemos una **clase** C1, en la cual hay un **método** llamado m1(), que es llamado desde las clases de la interfaz del programa.
- Los objetos de la **clase** C1 tienen un **atributo** de la **clase** C2, llamado **atr**.
- Suponga además que dentro del **método** m1() se invoca el **método** m2() de la **clase** C2 sobre el **atributo** llamado **atr**.

```
public class C2
{
    public void m2( )
    {
        instr1;
        instr2;
        instr3;
    }
}
```

- Dentro de la **clase** C2 hay un **método** llamado m2() que tiene 3 instrucciones, que aquí mostramos como instr1, instr2, instr3. Dichas instrucciones pueden ser de cualquier tipo.
- Suponga que se está ejecutando la instrucción instr2 del **método** m2() y se produce un error. En ese momento, a causa del problema el computador decide que no puede seguir con la ejecución del **método** (instr3 no se va a ejecutar).
- Crea entonces un **objeto** de la **clase** Exception que dice que el error sucedió en la instrucción instr2 del **método** m2() y explica la razón del problema.
- Luego, pasa dicho **objeto** al **método** m1() de la **clase** C1, que fue quien hizo la llamada. Si él lo atrapa (ya veremos más adelante cómo), el computador continúa la ejecución en el punto que dicho **método** indique.
- Si el **método** m1() no atrapa la **excepción**, este **objeto** pasa a la **clase** de la interfaz que hizo la llamada. Este proceso se repite hasta que alguien atrape la **excepción** o hasta que el programa completo se detenga. Entendemos por manejar una **excepción** el hecho de poderla identificar, atraparla antes de que el programa deje de funcionar y realizar una acción para recuperarse del error (por lo menos, para informarle al usuario lo sucedido de manera amigable y no con un mensaje poco comprensible del computador).

En el resto de esta sección mostraremos cómo se hace todo el proceso anteriormente descrito, en el [lenguaje de programación](#) Java.

## 5.1. Anunciar que Puede Producirse una Excepción

Cuando en un [método](#) queremos indicar que éste puede [disparar una excepción](#) en caso de que detecte una situación que considera anormal, esta indicación debe formar parte de la [signatura](#) del [método](#). En el ejemplo 2 se muestra la manera de hacer dicha declaración.

## Ejemplo 2

**Objetivo:** Declarar que un [método](#) puede lanzar una [excepción](#).

Este ejemplo muestra la manera de declarar en la [signatura](#) de un [método](#) que es posible que éste lance una [excepción](#) en caso de error. El [método](#) que se presenta forma parte de la [clase](#) Club y es responsable de afiliar un socio.

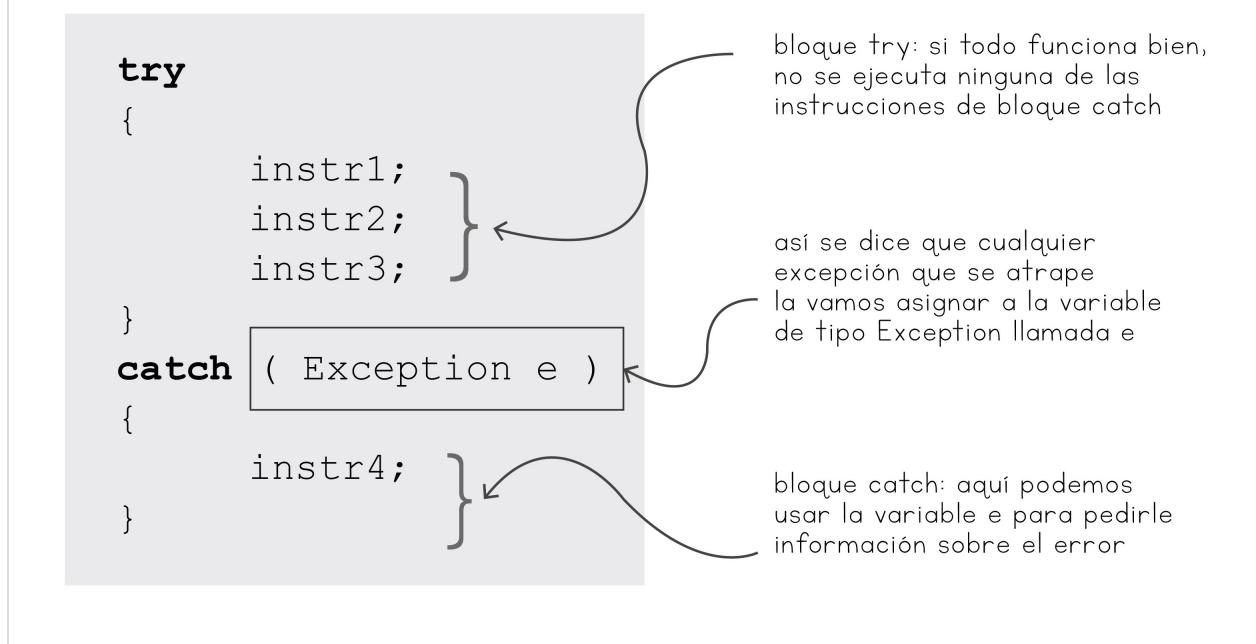
```
public void afiliarSocio( String pCedula, String pNombre, Tipo pTipo ) throws Exception
{
    ...
}
```

- Con esta declaración el [método](#) advierte a todos aquellos que lo usan de que puede producirse una [excepción](#) al invocarlo. Los métodos que hacen la invocación pueden decidir atraparla o dejarla pasar.
- No es necesario hacer un `import` de la [clase](#) Exception, puesto que esta [clase](#) está en un [paquete](#) que siempre se importa automáticamente (`java.lang`).

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`.

## 5.2. La Instrucción try-catch

La instrucción try-catch de Java tiene la estructura que se muestra en la [figura 4.7](#) y la sintaxis que se utiliza en el ejemplo 3.

**Fig. 4.7 Estructura básica de la instrucción try-catch**

En la instrucción try-catch hay dos bloques de instrucciones, con los siguientes objetivos:

- Delimitar la porción de código dentro de un **método** en el que necesitamos desviar el control si una **excepción** ocurre allí (la parte `try`). Si se dispara una **excepción** en alguna de las instrucciones del bloque try, la ejecución del programa pasa inmediatamente a las instrucciones del bloque catch. Si no se dispara ninguna **excepción** en las instrucciones del bloque try, la ejecución continúa después del bloque catch.
- Definir el código que manejará el error o atrapará la **excepción** (la parte `catch`).

## Ejemplo 3

**Objetivo:** Mostrar el uso de la instrucción try-catch de Java.

Este **método** forma parte de alguna de las clases de la interfaz, en la cual existe una referencia hacia el modelo del mundo llamada club. La estructura y contenido de las clases que implementan la **interfaz de usuario** son el tema del siguiente nivel.

```

public void ejemplo( String pCedula, String pNombre, Tipo pTipo )
{
    try
    {
        club.afiliarSocio( pCedula, pNombre, pTipo );
        totalSocios++;
    }
    catch( Exception e )
    {
        String ms = e.getMessage( );
        JOptionPane.showMessageDialog( this, ms );
    }
}

```

- Si en la llamada del **método** afiliarSocio se produce una **excepción**, ésta es atrapada y la ejecución del programa continúa en la primera instrucción del bloque `catch`. Note que en ese caso, la instrucción que incrementa el **atributo** `totalSocios` no se ejecuta.
- La primera instrucción del bloque `catch` pide al **objeto** que representa la **excepción** el mensaje que explica el problema. Fíjese cómo utilizamos la **variable** `e`.
- La segunda instrucción del bloque `catch` despliega una pequeña **ventana** de diálogo con el mensaje que trae el **objeto** `e` de la **clase** `Exception`. En este ejemplo, la intención es comunicarle al usuario que hubo un problema y que no se pudo realizar la afiliación del socio al club.

No todos los errores que se pueden producir en un **método** se atrapan con la instrucción `catch(Exception)`. Existen los que se denominan errores de ejecución (dividir por cero, por ejemplo) que se manejan de una manera un poco diferente.

## 5.3. La Construcción de un Objeto Exception y la Instrucción throw

Cuando necesitamos **disparar una excepción** dentro de un **método** utilizamos la instrucción `throw` del lenguaje Java. Esta instrucción recibe como **parámetro** un **objeto** de la **clase** `Exception`, el cual es lanzado o disparado al **método** que corresponda, siguiendo el esquema planteado anteriormente. Lo primero que debemos hacer, entonces, es crear el **objeto** que representa la **excepción**, tal como se muestra en el ejemplo que aparece a continuación.

### Ejemplo 4

**Objetivo:** Mostrar la manera de lanzar una **excepción** desde un **método**.

En este ejemplo aparece la [implementación](#) del [método](#) de la [clase](#) Club que permite afiliar un socio. En este [método](#), si ya existe un socio con la misma cédula, se lanza una [excepción](#), para indicar que se detectó una situación anormal.

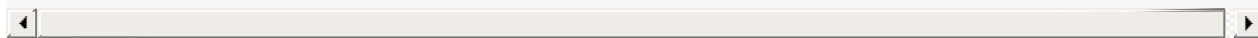
```
public void afiliarSocio( String pCedula, String pNombre, Tipo pTipo ) throws Exception
{
    // En caso de que el tipo de suscripción del nuevo socio sea VIP, es necesario
    // revisar que no se haya alcanzado el límite de suscripciones VIP que maneja el club

    if( pTipo == Tipo.VIP && contarSociosVIP( ) == MAXIMO_VIP )
    {
        // Si ya se alcanzó el número máximo de suscripciones VIP, se lanza una excepción
        throw new Exception( "El club en el momento no acepta más socios VIP" );
    }

    // Revisar que no haya ya un socio con la misma cédula en el club
    Socio s = buscarSocio( pCedula );

    if( s == null )
    {
        // Se crea el objeto del nuevo socio (todavía no se ha agregado al club)
        Socio nuevoSocio = new Socio( pCedula, pNombre, pTipo );

        // Se agrega el nuevo socio al club
        socios.add( nuevoSocio );
    }
    else
    {
        // Si ya existía un socio con la misma cédula, se lanza una excepción
        throw new Exception( "El socio ya existe" );
    }
}
```



- Este [método](#) lanza una [excepción](#) a aquél que lo llama, si le pasan como [parámetro](#) la información de un socio que ya existe o si el socio que se desea afiliar tiene suscripción VIP y ya se alcanzó el máximo número de suscripciones VIP que maneja el club.
- El constructor de la [clase](#) Exception recibe como [parámetro](#) una cadena de caracteres que describe el problema detectado.
- Cuando un [método](#) atrape esta [excepción](#) y le pida su mensaje (`getMessage()`), el [objeto](#) va a responder con el mensaje que le dieron en el constructor.
- En este ejemplo, cuando se detecta el problema se crea el [objeto](#) que representa el error y se lo lanza, todo de una sola vez. Pero podríamos haber hecho lo mismo en dos instrucciones separadas.

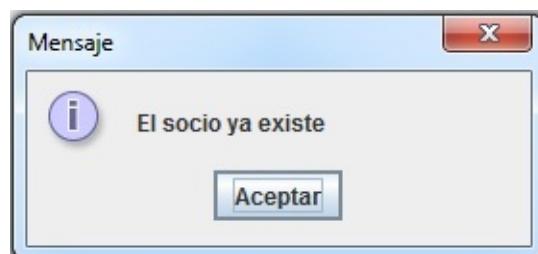
La [clase](#) `Exception` es una [clase](#) de Java que ofrece múltiples servicios, que se pueden consultar en la documentación. Los más usados son `getMessage()`, que retorna el mensaje con el que fue creada la [excepción](#), y `printStackTrace()`, que imprime en la consola de ejecución la traza incluida en el [objeto](#) (la secuencia anidada de invocaciones de métodos que dio lugar al error), tratando de informar al usuario respecto de la posición y la causa del error.

Si utilizamos las siguientes instrucciones después de atrapar la [excepción](#) del [método](#) `afiliarSocio()` en caso de que ya exista un socio con la misma cédula, presentado en el ejemplo 4:

```
...
catch( Exception e )
{
    JOptionPane.showMessageDialog( this, e.getMessage( ) );
}
```

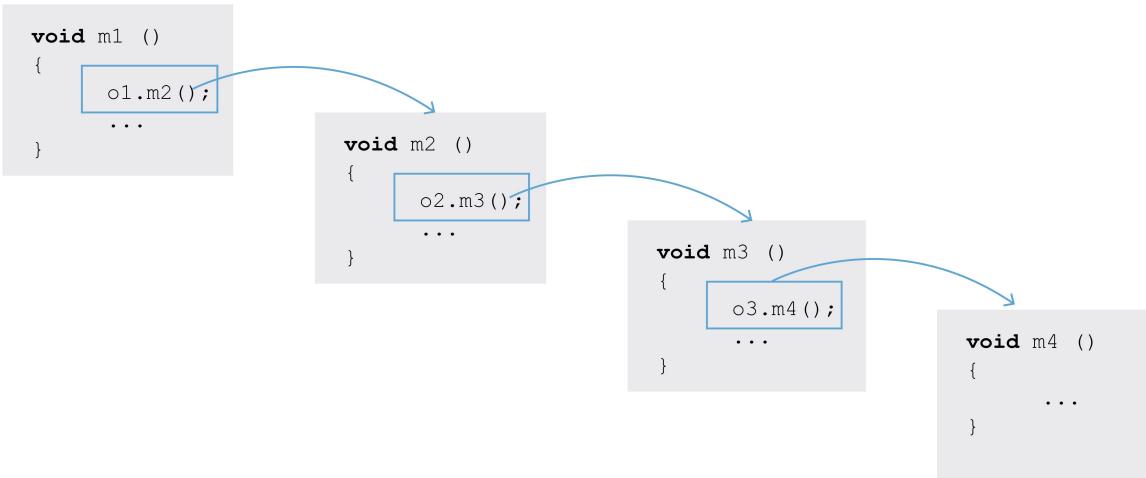
Obtendremos la [ventana](#) de advertencia al usuario que aparece en la [figura 4.8](#).

**Fig. 4.8 Despliegue de un mensaje de error como consecuencia de una excepción en el programa**

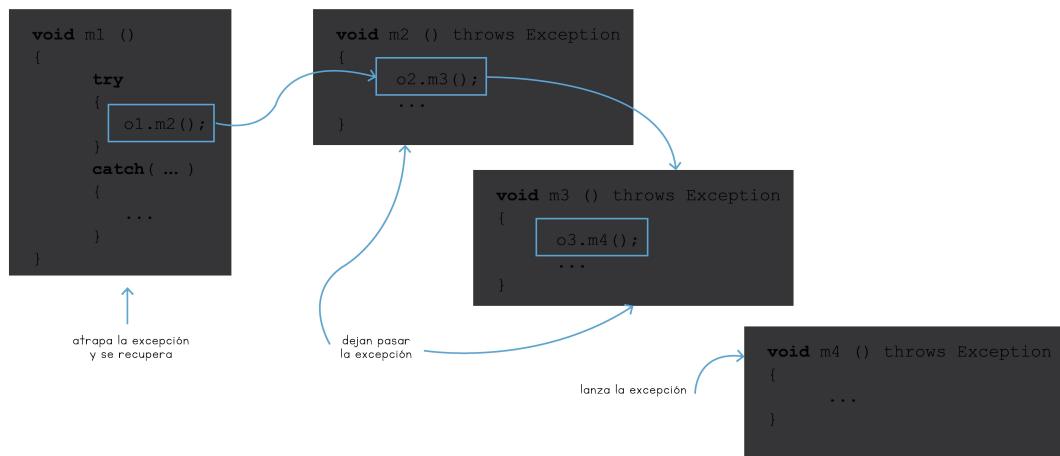


## 5.4. Recuperación de una Situación Anormal

Cuando se está ejecutando un [método](#), puede pasar que desde su interior se invoque otro [método](#) y, desde el interior de éste, otro y así sucesivamente. En la [figura 4.9](#) mostramos un ejemplo de la ejecución de un [método](#) `m1()` que invoca un [método](#) `m2()`, el cual llama a `m3()` y este último a `m4()`.

**Fig. 4.9 Invocación en cascada de métodos**

Supongamos ahora que durante la ejecución del **método m4()** se dispara una **excepción**. Es parte de nuestras decisiones de **diseño** decidir quién será el responsable de atraparla y manejarla. Una posibilidad es que el mismo **método m4()** la atrape y la procese. Otra posibilidad es que la **responsabilidad** se delegue hacia arriba, dejando que sea el **método m3()** o el **método m2()** o el **método m1()** quien se encargue de atrapar la **excepción**. En la **figura 4.10** ilustramos la situación en que es el **método m1()** el responsable de hacerse cargo de la **excepción**.

**Fig. 4.10 Flujo de control en el manejo de excepciones**

El **método** encargado de **atrapar una excepción** utiliza la instrucción **try-catch**, mientras que los métodos que sólo la dejan pasar lo declaran en su **signatura** (**throws Exception**).

## 6. Contrato de un Método

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 afiliarSocio( String pCedula, String pNombre, Tipo pTipo ) throws Exception
```

Podemos establecer que las suposiciones antes 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.
- El tipo de suscripción no es null.

Después 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.

### 6.1. Precondiciones y Postcondiciones

La [precondición](#) es aquello que exigimos para poder resolver el problema planteado a un [método](#). Es un 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](#):

```
public void registrarConsumo( String pNombre, String pConcepto, double pValor ) throws
Exception
```

Suposiciones sobre el parámetro <code>pNombre</code> .	
Suposiciones sobre el parámetro <code>pConcepto</code> .	
Suposiciones sobre el parámetro <code>pValor</code> .	
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.	

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](#).

La [precondición](#) se puede ver entonces como el conjunto de condiciones que impone aquél que desarrolla el [método](#) y la [postcondición](#) como los compromisos que asume. En otras palabras, el [contrato](#) queda establecido de la siguiente manera: "si todas las condiciones de la [precondición](#) se cumplen antes de llamar el [método](#), éste asume el compromiso de llegar a cumplir todas las condiciones incluidas en la [postcondición](#)".

El [contrato](#) es total, en el sentido de que si alguna de las precondiciones no se cumple, el [método](#) deja de estar obligado a cumplir la [postcondición](#).

## Tarea 5

**Objetivo:** Identificar las postcondiciones de algunos métodos.

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

```
public void registrarConsumo( String pNombre, String pConcepto, double pValor ) throws  
Exception
```



```
public boolean existeAutorizado( String pNombreAutorizado )
```



Vamos a contestar a continuación algunas de las 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 **precondición**? La respuesta es no. Lo que aparece en la **precondición** 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 [precondición](#), es imposible llegar a cumplir la [postcondición](#). Eso quiere decir que ninguna [excepción](#) puede asociarse con el incumplimiento de una [precondición](#).
- ¿Qué incluir entonces en la [precondición](#)? En la [precondición](#) 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 inefficientes. Si quiero asegurarme de algo así en la ejecución del [método](#), pues basta con eliminarlo de la [precondición](#) y lanzar una [excepción](#) si no se cumple. \*¿Por qué es inconveniente verificar todo dentro del [método](#) invocado? Por eficiencia. 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.

## 6.2. Documentación de los Contratos con Javadoc

En este libro expresamos los contratos en lenguaje natural y los incluimos dentro del código como parte de la documentación de los métodos. Para esto aprovechamos las convenciones y la herramienta de generación automática de documentación que viene con el lenguaje Java y que se llama [Javadoc](#). Dicha herramienta busca dentro de las clases comentarios delimitados por los caracteres `/** ... */` y genera a partir de ellos un conjunto de archivos con formato html, que permiten documentar el contenido de las clases.

Veamos cómo podemos utilizar algunas etiquetas (tags) de [Javadoc](#) para documentar uniformemente los contratos, de tal forma que, al ser generada la documentación del programa, sea claro para el lector de esa documentación cuáles son las suposiciones y los compromisos de los métodos que él va a utilizar.

Las convenciones que utilizamos para documentar los contratos de los métodos son las siguientes, que iremos ilustrando con el [contrato](#) del [método](#) de la [clase](#) Club que permite afiliar un nuevo socio:

- Un [contrato](#) se expresa como un comentario [Javadoc](#), delimitado con los caracteres `/** ... */`. Dicho comentario debe ir inmediatamente antes del [método](#).
- El [contrato](#) comienza con una descripción general del [método](#). Esta descripción debe dar una idea general del servicio que éste presta.

```
/**  
 * Este método afilia un nuevo socio al club.
```

- Luego vienen las precondiciones relacionadas con el estado del **objeto** que ejecuta el **método**. Allí se incluyen únicamente las restricciones y las relaciones que deben cumplir los atributos y los objetos con los cuales tiene una **asociación**.

```
* <b>pre:</b> La lista de socios está inicializada (no es null).<br>
```

Los elementos **<b>** y **</b>** sólo sirven para que cuando se genere la documentación en formato html, la palabra encerrada entre estos elementos aparezca en negrita. El elemento **<br>** inserta un cambio de renglón en ese lugar del **archivo** de documentación.

En el ejemplo anterior, la **condición** hace referencia a la **asociación** que existe entre la **clase** Club y la **clase** Socio, y dice que el **vector** que contiene los socios está inicializado. Dicha **condición** se da por cierta, lo que implica que en la **implementación** del **método** no se hará ninguna **verificación** en ese sentido y se utilizará como un hecho.

- Después aparecen las postcondiciones que hacen referencia al estado del **objeto** después de la ejecución del **método**. Allí se debe describir la modificación de los atributos y objetos asociados que puede esperarse luego de su invocación.

```
* <b>post:</b> Se ha afiliado un nuevo socio en el club con los datos dados.<br>
```

- La siguiente parte describe los parámetros de entrada y las precondiciones asociadas con ellos. Por cada uno de los parámetros se debe usar la **etiqueta** `@param` seguida del nombre del **parámetro**, una descripción y las suposiciones que el **método** hace sobre él.

```
* @param pCedula Cédula del socio a afiliar. pCedula != null && pCedula != "".  
* @param pNombre Nombre del socio a afiliar. pNombre != null && pNombre != "".  
* @param pTipo Es el tipo de subscripción del socio. pTipo != null.
```

Al decir en el **contrato** que el **parámetro** que trae la cédula del nuevo socio no tiene el valor `null` ni es una cadena vacía, estamos afirmando que el **método** no va a hacer ninguna **verificación** al respecto y que aquél que haga la llamada debe garantizarlo.

Como parte del **contrato** no es necesario hablar del tipo de los parámetros, porque esto va en la **signatura del método**, la cual es parte integral del mismo. Esto quiere decir, por ejemplo, que no vale la pena incluir en la **precondición del atributo** nombre algo para indicar que es de tipo String.

Tampoco es buena idea incluir en una **precondición** información sobre lo que no se supone en el **método**. Debe quedar claro que todo lo que no aparece explícitamente como una suposición, no se puede suponer.

- Luego viene la parte de la **postcondición** que describe el retorno del **método**. Esta sólo aparece en el **contrato** si el **método** devuelve algún valor (es decir, no es `void`). Se indica con la **etiqueta** `@return` seguido de una descripción de lo que el **método** devuelve y las condiciones que este valor cumple.

En el ejemplo que venimos desarrollando, como el **método** es de tipo `void`, no hay necesidad de agregar nada al **contrato**.

Para poder expresar de manera más sencilla las condiciones sobre el valor que el **método** devuelve, es común darle un nombre al retorno del **método** (como si fuera una **variable**) y luego usar dicho nombre como parte de las condiciones. Esto se ilustra más adelante.

- Por último, aparecen las excepciones que el **método** dispara. Para hacer esto, se utiliza la **etiqueta** `@throws` seguida del tipo de la **excepción** y una descripción de la situación en la que puede ser disparada.

```
* @throws Exception <br>
*      1. Si un socio con la misma cédula ya estaba afiliado al club. <br>
*      2. Si el socio a registrar desea una subscripción VIP pero el club ha alcanzado el límite.
```

Es conveniente que la descripción se haga usando una frase en la que sea clara la **condición** para que la **excepción** se lance (p.ej., "si un socio con la misma cédula ya estaba afiliado al club"), lo mismo que las consecuencias de la **excepción** (p.ej. "la nueva afiliación no se pudo llevar a cabo").

Cuando un **método** puede lanzar varias excepciones, cada una de ellas por una razón diferente, se debe usar la **etiqueta** `@throws` para cada caso de manera independiente.

## Ejemplo 5

**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**.

```

/**
 * Afilia un nuevo socio al club. <br>
 * <b>pre: </b> La lista de socios está inicializada. <br>
 * <b>post: </b> Se ha afiliado un nuevo socio en el club con los datos dados.
 * @param pCedula Cédula del socio a afiliar. pCedula != null && pCedula != "".
 * @param pNombre Nombre del socio a afiliar. pNombre != null && pNombre != "".
 * @param pTipo Es el tipo de suscripción del socio. pTipo != null.
 * @throws Exception <br>
 *           1. Si un socio con la misma cédula ya estaba afiliado al club. <br>
 *           2. Si el socio a registrar desea una suscripción VIP pero el club ha alcanzado el límite.
 */
public void afiliarSocio( String pCedula, String pNombre, Tipo pTipo ) throws Exception
{
}

```

**All Classes**

**Packages**

- uniandes.cupi2.club.interfaz
- uniandes.cupi2.club.mundo

<p><b>All Classes</b></p> <p><b>Club</b>          DialogoAfiliarSocio          DialogoRegistrarConsumo          Factura          InterfazClub          PanelAutorizados          PanelFacturas          PanelImagen          PanelListaSocios          PanelOpciones          Panelocio          Socio          Socio.Tipo</p>	<p><b>darSocios()</b>  <code>public java.util.ArrayList&lt;Socio&gt; darSocios()</code>          Retorna los socios afiliados al club.  <b>Returns:</b>          Lista de socios.</p> <p><b>afiliarSocio</b>  <code>public void afiliarSocio(java.lang.String pCedula,          java.lang.String pNombre,          Socio.Tipo pTipo)          throws java.lang.Exception</code>          Afilia un nuevo socio al club.  <b>pre:</b> La lista de socios está inicializada.  <b>post:</b> Se ha afiliado un nuevo socio en el club con los datos dados.  <b>Parameters:</b>          pCedula - Cédula del socio a afiliar. pCedula != null &amp;&amp; pCedula != "".          pNombre - Nombre del socio a afiliar. pNombre != null &amp;&amp; pNombre != "".          pTipo - Es el tipo de suscripción del socio. pTipo != null.  <b>Throws:</b>          java.lang.Exception -          1. Si un socio con la misma cédula ya estaba afiliado al club.          2. Si el socio a registrar desea una suscripción VIP pero el club ha alcanzado el límite.</p> <p><b>buscarSocio</b>  <code>public Socio buscarSocio(java.lang.String pCedulaSocio)</code>          Retorna el socio con la cédula dada.  <b>pre:</b> La lista de socios está inicializada.  <b>Parameters:</b>          pCedulaSocio - Cédula del socio buscado. pCedulaSocio != null &amp;&amp; pCedulaSocio != "".  <b>Returns:</b>          El socio buscado, null si el socio buscado no existe.</p>
--	--

## Tarea 6

**Objetivo:** Revisar los contratos de los métodos del caso de estudio.

Genere la documentación del ejemplo del **club**, utilizando la herramienta **Javadoc**.

Revise la documentación generada a partir del índice que encuentra en:

n4\_club/docs/api/index.html En particular, estudie la definición de los contratos de los métodos de las clases Club, Socio y

Factura, y conteste las siguientes preguntas:

¿Qué pasa si el <b>método</b> buscarSocio de la <b>clase</b> Club no encuentra el socio cuya cédula recibió como <b>parámetro</b> ?	<input type="text"/>
¿Qué <b>precondición</b> exige el <b>método</b> buscarSocio de la <b>clase</b> Club respecto del <b>atributo</b> que representa la cédula?	<input type="text"/>
¿Qué retorna el <b>método</b> darConcepto de la <b>clase</b> Factura? ¿Qué condiciones cumple dicho valor? ¿Qué nombre se usó en el <b>contrato</b> para representar el valor de retorno?	<input type="text"/>
¿Cuál es la <b>precondición</b> sobre el <b>parámetro</b> pValor en el <b>método</b> registrarConsumo de la <b>clase</b> Socio?	<input type="text"/>
¿Cuál es la <b>postcondición</b> del <b>método</b> pagarFactura de la <b>clase</b> Socio?	<input type="text"/>
¿En cuántos casos lanza una <b>excepción</b> el <b>método</b> agregarAutorizado de la <b>clase</b> Socio?	<input type="text"/>
¿Qué sucede si en el <b>método</b> agregarAutorizado de la <b>clase</b> Socio, el <b>parámetro</b> de entrada corresponde al nombre del socio?	<input type="text"/>

## 7. Diseño de las Signaturas de los Métodos

Una vez distribuidas las responsabilidades entre las clases, debemos continuar con el diseño de los métodos. Por un lado, debemos decidir cuáles serán los parámetros del método, cuál será su valor de retorno, qué excepciones puede disparar y, finalmente, debemos precisar su contrato, es decir, definir las condiciones sobre todos esos elementos.

%%

De manera general, podemos decir que la información que tenemos para diseñar la signatura de los métodos viene de dos fuentes distintas: por una parte, de la identificación de las entradas y salidas de los requerimientos funcionales. Por otra parte, de los tipos de los atributos utilizados en el modelado del mundo del problema. Por ejemplo, para el requerimiento funcional de afiliar un socio, los datos de entrada son la cédula del socio, su nombre y su tipo de subscripción. Esto sugiere que ésa es la información que debe recibir el método de la clase Club que tiene esa responsabilidad.

```
public void afiliarSocio( String pCedula, String pNombre, Tipo pTipo ) throws Exception
```

En el caso general, es conveniente tratar de contestar dos preguntas:

- ¿Qué información externa al objeto se necesita para resolver el problema que se plantea en el método? Esto nos va a dar pistas sobre los parámetros que se deben incluir.
- ¿Cómo se modeló esa información dentro del objeto? Piense, por ejemplo, que si se definieron constantes para representar los valores posibles de una característica, y la información externa está relacionada con ella, los parámetros deben reflejar eso. En el caso de estudio de la tienda presentado en el nivel 2, si queremos pasar como parámetro el tipo del producto (recuerde que puede ser de papelería, droguería o supermercado), el parámetro debe ser una enumeración y no de tipo cadena de caracteres.

### Tarea 7

**Objetivo:** Revisar el diseño de los métodos del caso de estudio y justificar las signaturas utilizadas.

Para la **clase** Socio, estudie la **signatura** de los siguientes métodos y trate de escribir la justificación de cada una de las decisiones de **diseño**. ¿Por qué esos parámetros? ¿Por qué esas excepciones? ¿Por qué ese tipo de retorno?

```
boolean existeAutorizado( String pNombreAutorizado )
```



```
void eliminarAutorizado( String pNombreAutorizado ) throws Exception
```



```
void agregarAutorizado( String pNombreAutorizado ) throws Exception
```



```
void pagarFactura( int pIndiceFactura ) throws Exception
```



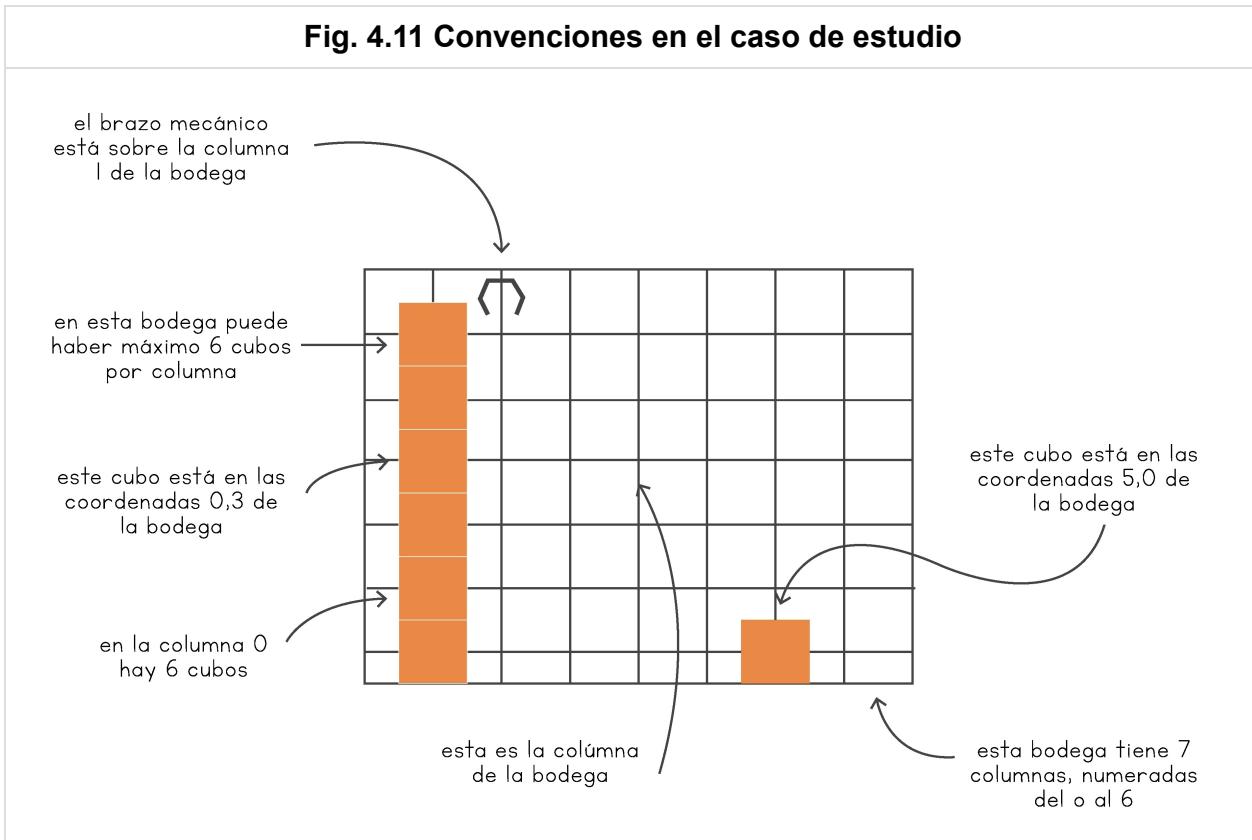
```
void registrarConsumo( String pNombre, String pConcepto, double pValor ) throws Except  
ion
```



## 8. Caso de Estudio N° 2: Un Brazo Mecánico

En esta aplicación se modela una bodega que tiene cubos apilados en ciertas posiciones y un brazo mecánico que puede mover estos cubos. La bodega tiene unas dimensiones definidas y ni el brazo ni los cubos pueden estar por fuera de esos límites. La bodega se puede organizar como una cuadrícula en la cual las coordenadas X corresponden a las columnas y las Y corresponden a la altura medida desde el piso, tal como se sugiere en la figura 4.11.

**Fig. 4.11 Convenciones en el caso de estudio**



Todos los cubos tienen las mismas dimensiones, pero pueden tener colores diferentes y se pueden poner uno encima del otro o sobre el piso, mientras sus posiciones coincidan con la cuadrícula de la bodega. Un cubo no puede estar suspendido en el aire: debe estar sobre otro cubo o sobre el piso.

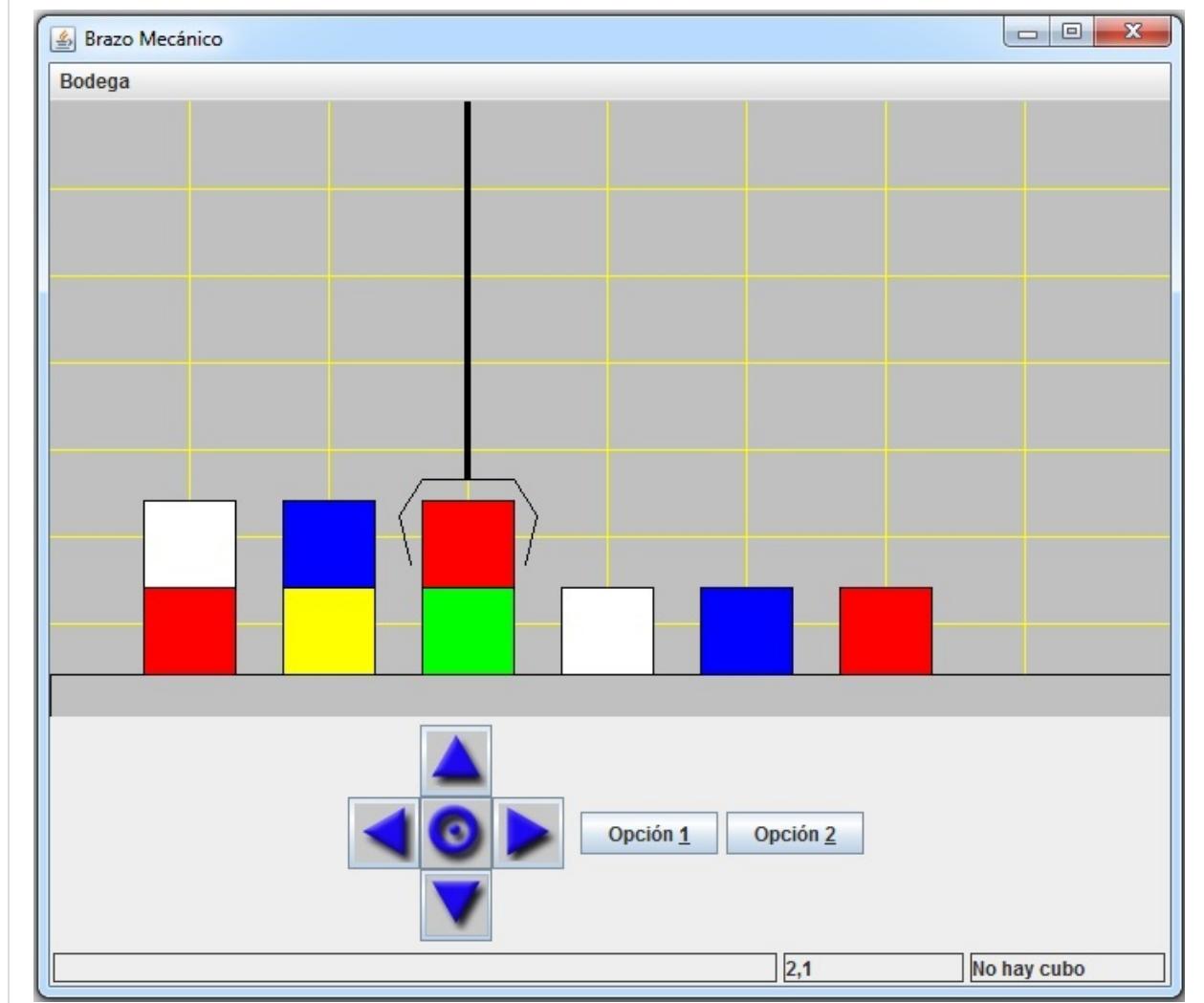
El brazo mecánico está suspendido del techo de la bodega y puede moverse a lo largo de las columnas, al igual que puede subir y bajar. El brazo puede cargar un cubo a la vez y solamente puede tomarlo si se coloca en la misma posición del cubo que quiere agarrar.

Únicamente se pueden recoger cubos que están en el tope de una columna. Para soltar un cubo el brazo debe ubicarse justo encima del tope de una columna o sobre el piso y luego dejar el cubo en esa posición. ¡No pueden dejarse caer los cubos!

Hay algunas restricciones al movimiento del brazo. Mientras el brazo está cargando un cubo no puede llegar a una posición ocupada por otro cubo. Además el brazo solamente puede llegar a una posición donde hay un cubo si éste se encuentra en el tope de una columna.

La interfaz de la aplicación del brazo mecánico se presenta en la [figura 4.12](#).

**Fig. 4.12 Interfaz de usuario del brazo mecánico**



- En la gráfica mostrada, el brazo mecánico aparece en la posición 2, 1.
- La bodega tiene 7 columnas y un máximo de 6 cubos en cada una.
- Con los cinco botones del [panel](#) inferior, se puede mover el robot en cada una de las cuatro direcciones posibles. El botón de la mitad sirve para agarrar o soltar un cubo.
- En la parte inferior derecha, la interfaz indica que aunque el brazo está sobre un cubo, no lo ha sujetado.
- Con el menú que aparece en la parte de arriba, es posible cargar una nueva bodega a partir de la información contenida en un [archivo](#).

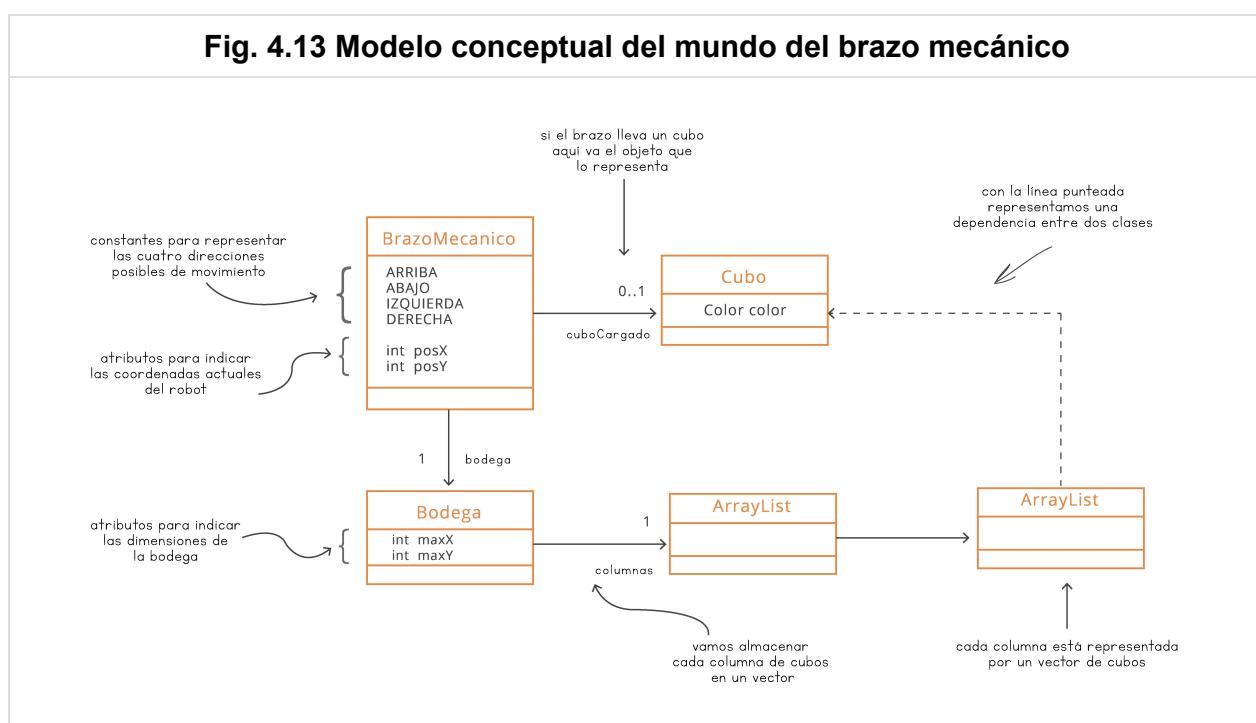
Vamos a utilizar este caso de estudio para generar habilidad en el uso de las nociones de [asignación](#) de responsabilidades, contratos y excepciones. Primero, vamos a explicar la manera en que diseñamos e implementamos el mundo del brazo mecánico y luego vamos a resolver algunos problemas en ese mundo.

Este caso también lo vamos a utilizar para introducir la técnica de [dividir y conquistar](#), como una manera natural de resolver problemas complejos.

## 8.1. Comprepción y Construcción del Mundo en Java

En el mundo del brazo mecánico existen tres entidades básicas: la bodega, el brazo y los cubos. En la [figura 4.13](#) se muestra el diagrama de clases, que nos resume el [diseño](#) que hicimos para este problema. Debe ser claro que existen muchos otros diseños posibles, pero éste lo construimos de manera particular para poder mostrar todos los aspectos interesantes de este capítulo.

**Fig. 4.13 Modelo conceptual del mundo del brazo mecánico**



A continuación mostramos la declaración de las constantes y atributos de cada una de las clases involucradas:

```
import java.awt.Color;

public class Cubo
{
    //-----
    // Atributos
    //-----
    private Color color;
}
```

- La declaración de la **clase** Cubo es la más sencilla del diagrama de clases. Cada cubo tiene únicamente un color como **atributo**.
- Usamos la **clase** Color del **paquete** `java.awt` para modelar esta característica.

```
public class BrazoMecanico
{

    //-----
    // Constantes
    //-----
    public static final int ARRIBA = 1;
    public static final int ABAJO = 2;
    public static final int IZQUIERDA = 3;
    public static final int DERECHA = 4;

    //-----
    // Atributos
    //-----
    private int posX;
    private int posY;
    private Cubo cuboCargado;
    private Bodega bodega;

}
```

- La **clase** BrazoMecanico define cuatro constantes para identificar los cuatro movimientos posibles que puede hacer dentro de la bodega.
- Con los **atributos** `posX` y `posY` el brazo mecánico conoce su posición dentro de la bodega. El valor `posX` define la columna en la que se encuentra y el valor `posY` la altura.
- Si el brazo lleva agarrado un cubo, en el **atributo** `cuboCargado` se encuentra el **objeto** que representa el cubo. Si no lleva ningún cubo agarrado, este **atributo** tiene el valor `null`.
- El último **atributo** es la bodega en la cual se encuentra el brazo mecánico.

```

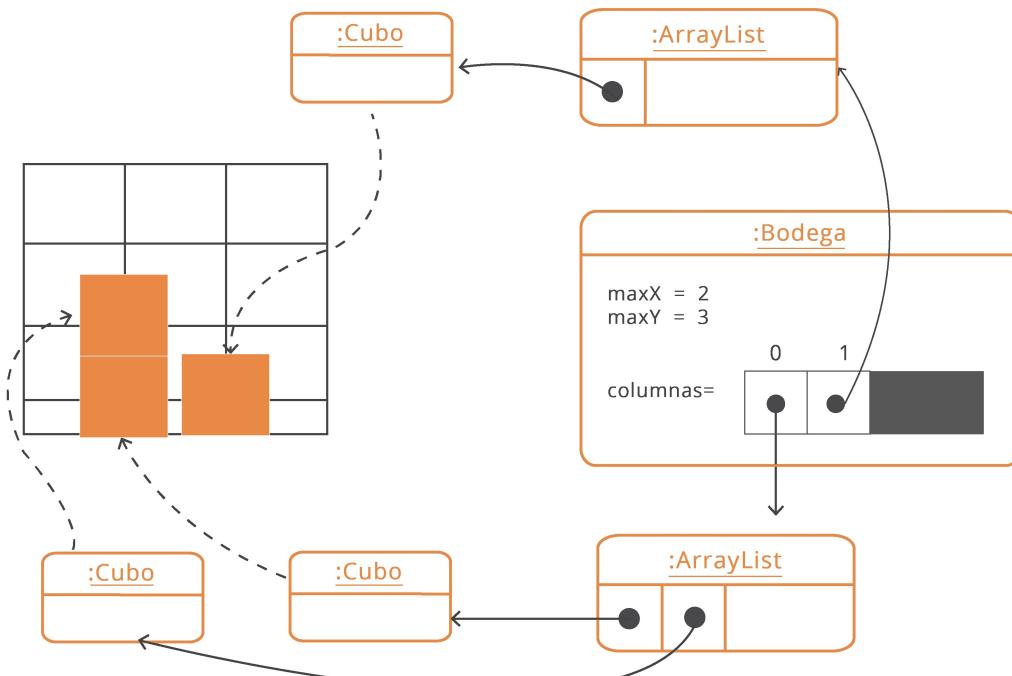
public class Bodega
{
    //-----
    // Atributos
    //-----

    private int maxX;
    private int maxY;
    private ArrayList columnas;
}
...

```

- Los atributos `maxX` y `maxY` se utilizan para representar las dimensiones de la bodega: el primero dice el número de columnas y el segundo el número máximo de cubos por columna.
- En el **atributo** "columnas" almacenamos las columnas de la bodega. En la posición x de este **vector**, estará la columna x de la bodega. Cada columna a su vez estará representada por un **vector** de cubos. En la [figura 4.14](#) se ilustra esta estructura usando un [diagrama de objetos](#).

**Fig. 4.14 Ejemplo de un [diagrama de objetos](#) para representar una bodega**



En la representación que escogimos, es importante señalar que cada columna es a su vez un **vector** de cubos. En dicho **vector**, en la posición 0 estará el cubo que se encuentra sobre el piso (si existe alguno) y de ahí en adelante aparecerán los demás cubos, siguiendo su orden dentro de la columna.

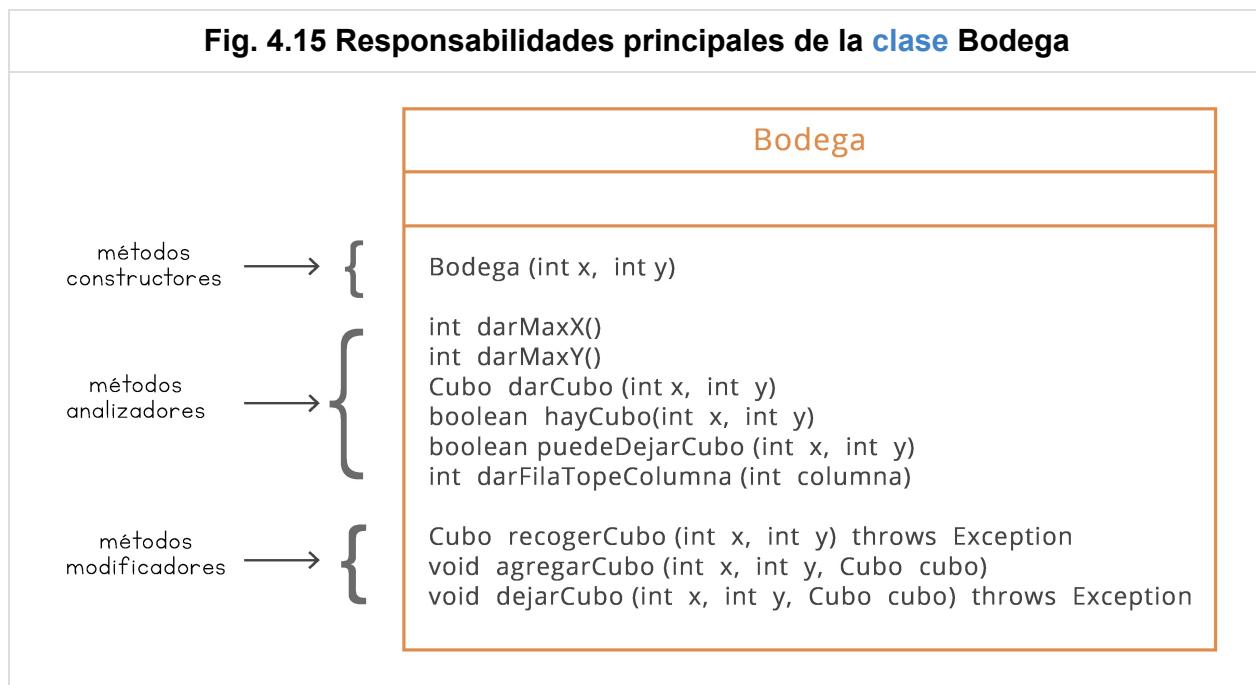
## 8.2. Comprender la Asignación de Responsabilidades y los Contratos

En esta parte vamos a describir las responsabilidades asignadas a las clases:

- La **clase** Cubo tiene un **atributo** color y es responsable de dar la información de su color. Como no está previsto que los cubos cambien de color, no existe un **método** para cambiar ese valor. Este es un ejemplo de un caso en el que puede imaginarse un servicio que no hace falta prestar en relación con un **atributo**.
- La **clase** Bodega es responsable de manejar sus columnas en donde se apilan los cubos. Sabe construir una bodega a partir de unos datos de entrada y sabe responder a las preguntas: ¿hay un cubo en una posición dada? y ¿cuál es el tamaño de la bodega?

La **clase** Bodega también sabe ubicar y eliminar un cubo de una posición dada. Note que el **objeto** Bodega trabaja en estrecha colaboración con el BrazoMecanico. La **figura 4.15** muestra la **clase** con los métodos que implementan las principales responsabilidades.

**Fig. 4.15 Responsabilidades principales de la clase Bodega**



Para la **clase** BrazoMecanico tenemos lo siguiente:

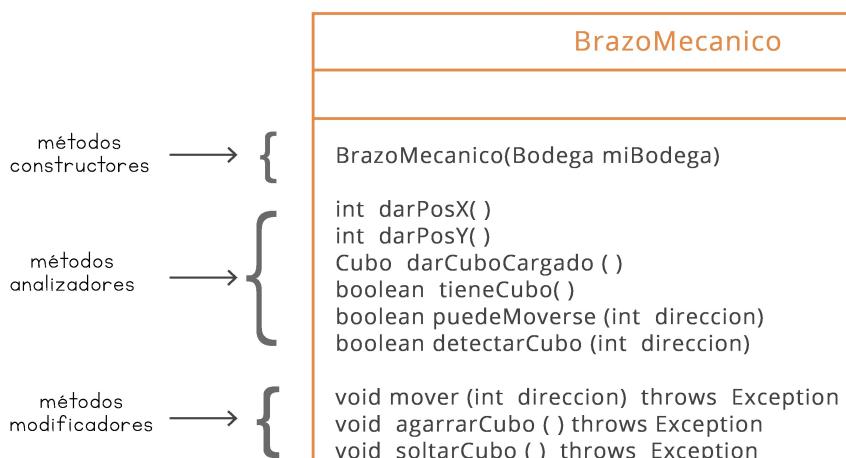
- **Ubicación:** el brazo sabe dónde se encuentra ubicado dentro de la bodega (`posX`, `posY`). Por esta razón, tiene la **responsabilidad** de informar sobre su posición: `darPosX()`, `darPosY()`.
- **Relación con un cubo:** tiene una **asociación** de cardinalidad opcional con un cubo, que representa la posibilidad de llevar agarrado un cubo. El brazo sabe si tiene o no un cubo en la pinza, dependiendo de si la **asociación** existe. Por esta razón, tiene la **responsabilidad** de implementar un **método** que devuelva el cubo o `null` si no lleva

ninguno.

- **Sensores:** los sensores del brazo han sido modelados a través de servicios que el cubo le solicita a la bodega. Por ejemplo, si el brazo necesita saber si en una posición de su vecindad inmediata (arriba, abajo, derecha o izquierda) hay un cubo, le solicita a la bodega que haga la [verificación](#), dándole la posición requerida para que ella determine si hay o no un cubo ahí.

En la [figura 4.16](#) se muestran las responsabilidades del brazo mecánico anteriormente mencionadas, en términos de sus métodos analizadores y sus métodos modificadores.

**Fig. 4.16 Responsabilidades principales del BrazoMecanico**



## Tarea 8

**Objetivo:** Estudiar los contratos de los métodos diseñados para el caso del brazo mecánico.

Genere la documentación del proyecto [n4\\_brazoMecanico](#) y estudie los contratos de los métodos de las clases Bodega, BrazoMecanico y Cubo. Responda las siguientes preguntas:

Explique cuáles son los compromisos del [método mover\( \)](#) de la [clase BrazoMecanico](#).

¿Qué pasa si tratamos de mover el brazo mecánico en alguna dirección y ésta no es válida?

Explique cuáles son los compromisos del **método** agarrarCubo( ) de la **clase** BrazoMecanico. ¿Qué pasa si el brazo mecánico trata de agarrar un cubo (en la posición donde está) y allí no hay ningún cubo?

Explique cuáles son los compromisos del **método** dejarCubo() de la **clase** Bodega. ¿Qué pasa si se Intenta dejar un cubo en una posición de la bodega y ésta no es válida?

Explique cuáles son las suposiciones del **método** `puedeMoverse()` de la **clase** `BrazoMecanico`.

Explique cuáles son las suposiciones del **método** `darFilaTopeColumna()` de la **clase** `Bodega`.

Explique cuáles son las suposiciones del **método** `puedeDejarCubo()` de la **clase** `Bodega`.

Explique cuáles son las responsabilidades del [método](#) detectarCubo( ) de la [clase](#) BrazoMecanico.

¿Cuál es la diferencia entre el [método](#) recogerCubo( ) de la [clase](#) Bodega y el [método](#) agarrarCubo( ) de la [clase](#) BrazoMecanico? ¿Cuál es exactamente la [responsabilidad](#) de cada uno de ellos?

## 8.3. La Técnica de Dividir y Conquistar

Ahora que ya entendemos el mundo del brazo mecánico y que tenemos a la mano los contratos de todos los métodos que ofrecen sus clases Cubo, BrazoMecanico y Bodega, vamos a utilizarlos para resolver algunos problemas.

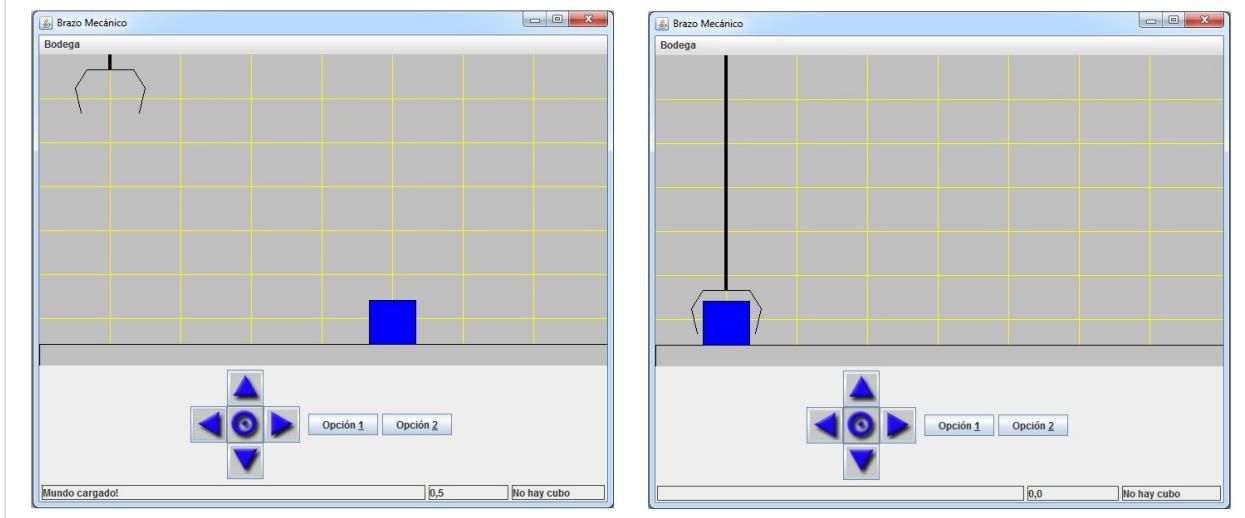
### 8.3.1. Reto 1

Suponga que el brazo mecánico se encuentra en la parte superior izquierda de la bodega, y que en ella, en alguna posición, hay un único cubo. La tarea que debemos resolver es lograr que el brazo mecánico lo encuentre y luego lo lleve a la columna 0 en la posición del piso. En la [figura 4.17](#) aparece un ejemplo de una posible situación inicial del problema y su correspondiente situación final.

Para enfrentar este reto, lo primero que debemos hacer es definir un plan de solución. El objetivo del **plan de solución** es descomponer el problema en problemas más pequeños. Una guía para hacerlo es identificar metas intermedias que nos vayan acercando a la solución completa. Nuestro plan para el primer reto puede ser:

- Meta 1: El brazo debe bajar hasta el piso.
- Meta 2: El brazo debe avanzar hacia la derecha y encontrar y agarrar el cubo que hay en la bodega.
- Meta 3: El brazo debe llevar el cubo a la posición 0, 0 de la bodega y dejarlo allí.

**Fig. 4.17 Ejemplo de una situación inicial y una situación final para el reto 1**



Identificadas las metas intermedias, podemos resolver de manera aislada cada uno de los subproblemas asociados y, luego, reunir las soluciones que obtengamos. Si llamamos `bajarARecoger()`, `encontrarUnicoCubo()` y `volverAPosicion0()` a los métodos que resuelven cada uno de los subproblemas planteados anteriormente, la solución global del reto 1 tendría la siguiente estructura:

```
public class BrazoMecanico
{
    public void solucionReto1( )
    {
        bajarARecoger( );
        encontrarUnicoCubo( );
        volverAPosicion0( );
    }
}
```

- Construimos la solución al problema a partir de la solución de los métodos que nos van a ayudar a cumplir cada una de las metas. La ventaja es que los métodos resultantes son más sencillos de construir, si cada uno se encarga únicamente de una parte del problema.

Los tres métodos planteados deberían declararse como métodos privados, dado que no esperamos que alguien externo los utilice.

Mientras no definamos los contratos exactos de los métodos, no podemos estar seguros de que el [método](#) solucionReto1() está terminado, pero por lo menos tenemos un borrador para comenzar a trabajar.

Fíjese que la [precondición](#) del segundo de los métodos debe asegurarse en la [postcondición](#) del primero de ellos.

Por ahora, comencemos definiendo el [contrato](#) del [método](#) que resuelve el problema completo.

¿Qué iría en la [precondición](#)? Una aproximación es suponer que el robot efectivamente se encuentra en donde dice el enunciado y suponer también que hay un único cubo en la bodega. Para evitar que el programa falle en caso de que esas suposiciones no sean ciertas, vamos a dejar la [precondición](#) vacía, y vamos a lanzar una [excepción](#) si el estado de la bodega no es exactamente como lo plantea el enunciado del reto. Esto nos lleva al siguiente [contrato](#):

```
/**  
 * Este método sirve para que el brazo mecánico localice el único cubo que hay en la  
 * bodega y lo lleve a la posición de origen (coordenadas 0, 0).  
 *  
 * <b>post:</b> El brazo está en la posición de origen, al igual que el único cubo  
 * de la bodega. El brazo no está sujetando el cubo.  
 *  
 * @throws Exception Lanza una excepción si el robot se choca en cualquier momento  
 * mientras trata de resolver el problema, debido a que el estado  
 * de la bodega no corresponde al enunciado.  
 *  
 * @throws Exception Lanza una excepción si encuentra algún obstáculo para agarrar el  
 * cubo (por ejemplo, un segundo cubo sobre él).  
 */  
public void solucionReto1( ) throws Exception  
{ ... }
```

Comencemos ahora a construir los métodos para lograr cada una de las metas y, a medida que los vayamos escribiendo, iremos refinando la solución planteada anteriormente (si es necesario).

**Meta 1:** Para lograr la meta 1, debemos mover el brazo hasta que su posición en el eje Y sea igual a 0 (es decir, el piso). Esto lo podemos lograr con una instrucción repetitiva para la que podemos utilizar el patrón de [recorrido total](#).

```

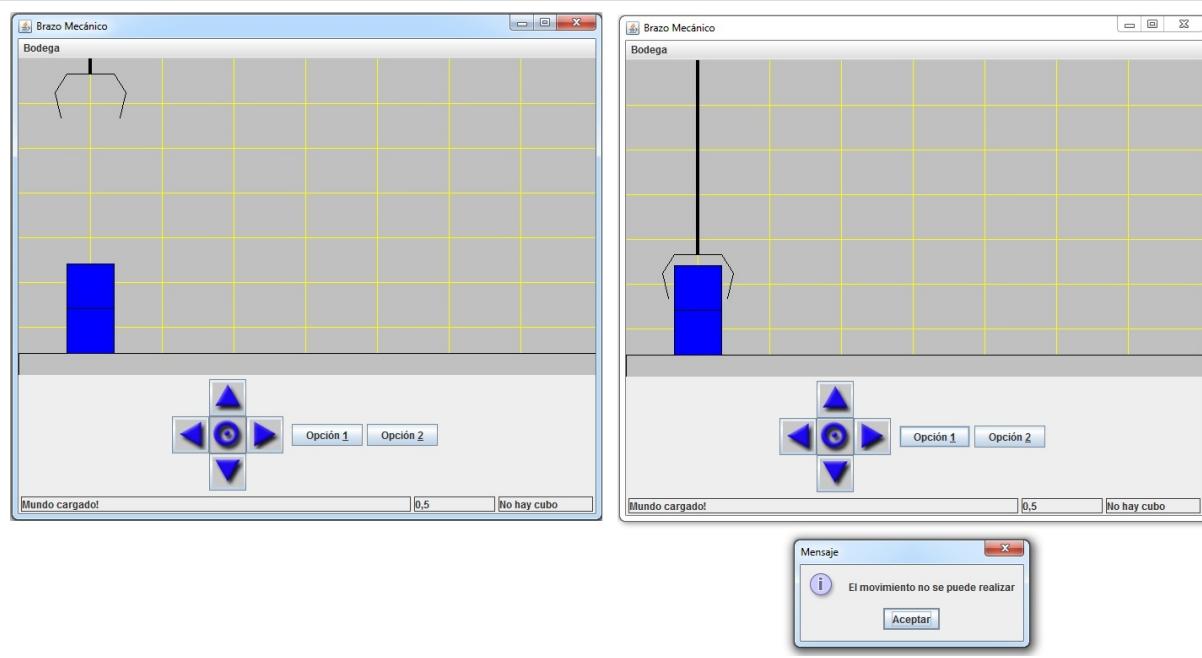
private void bajarARecoger( ) throws Exception
{
    for( int i = 0; i < bodega.darMaxY( ); i++ )
    {
        mover( ABAJO );
    }
}

```

- Inicializamos la **variable** `i` en 0 e iteramos hasta llegar al número máximo de cubos de la bodega.
- Repetimos la instrucción de mover hacia abajo, incrementando en cada **iteración** el valor de la **variable** `i`.
- Puesto que el **método** `mover()` de la **clase** `BrazoMecanico` puede lanzar una **excepción** si se produce un choque, decidimos no atraparla y dejarla pasar al **método** principal. Es la única opción para evitar que el segundo **método** comience en un estado que no cumple su **precondición**. Además no habría en este **método** ninguna manera para recuperarse del error.

La descripción del reto dice que sólo hay un cubo en la bodega en alguna posición del piso. Supongamos que quien invoca el **método** `solucionReto1()` no verifica que esto sea cierto y que el estado inicial es algo como el mostrado en la [figura 4.18](#). En ese caso, cuando el brazo llega sobre el primer cubo y trata de seguir hacia abajo, el **método** `mover(ABAJO)` se da cuenta que no puede hacerlo y dispara una **excepción**. Al suceder el disparo de la **excepción**, se detiene la ejecución del **método** y el control debería llegar hasta una **clase** en la **interfaz de usuario**, que debería atraparla y desplegar un mensaje de error, como el que se muestra en la [figura 4.18](#).

**Fig. 4.18 Error al tratar de mover el brazo sobre un cubo**



### Meta 2:

Para poder cumplir con la segunda meta, vamos a desarrollar el [método](#) `encontrarUnicoCubo()`, el cual implementa el siguiente [contrato](#):

#### Precondición:

- El brazo está en la posición 0,0 de la bodega (aquí lo dejó la solución a la meta 1). Fíjese que aquí lo podemos poner como una suposición, ya que en nuestro [método](#) vamos a utilizar esta información (sin necesidad de verificar que sea cierta).

#### Postcondición:

- El brazo está en la posición donde se encuentra el cubo.
- El brazo ha agarrado el cubo.

El [método](#) va a [disparar una excepción](#) si no puede cumplir con la [postcondición](#), debido a que el estado de la bodega no es como se suponía. Note que las excepciones no representan en ningún momento errores en el programa (no es que no podamos cumplir la [postcondición](#) porque el [método](#) esté mal escrito), sino situaciones anormales que están fuera del control del [método](#).

La [implementación](#) del segundo [método](#) es la siguiente:

```

/**
 * Busca y agarra el único cubo que hay en el mundo.
 *
 * <b>pre:</b> El brazo está en la posición 0,0 de la bodega.
 * <b>post:</b> El brazo está en la posición donde
 * se encuentra el cubo y lo está agarrando
 *
 * @throws Exception Si no encontró un cubo o si el brazo se
 * estrelló contra una pila de cubos,
 * dispara una excepción y detiene el brazo
 */
private void encontrarUnicoCubo( ) throws Exception
{
    boolean encontro = false;
    Cubo cubo = null;

    for( int i = 0; i <= bodega.darMaxX( ) && !encontro; i++ )
    {
        cubo = bodega.darCubo( i, 0 );

        if( cubo != null )
        {
            encontro = true;
        }
        else if( i < bodega.darMaxX( ) )
        {
            try
            {
                mover( DERECHA );
            }
            catch( Exception e )
            {
                throw new Exception( "Hay una pila de cubos" );
            }
        }
    }

    if( encontro )
        agarrarCubo( );
    else
        throw new Exception( "No hay ningún cubo" );
}

```

- La estrategia para resolver este problema es recorrer la posición 0 de cada una de las columnas hasta encontrar el cubo. Este problema corresponde a los que resuelve el patrón de [algoritmo de recorrido parcial](#) sobre una secuencia.
- La [postcondición](#) afirma que el brazo queda en la posición donde está el cubo y lo tiene agarrado. Por esta razón, si al final del recorrido sobre la bodega vemos que no había ningún cubo o el brazo se estrelló contra una pila de cubos y que, por tanto, no

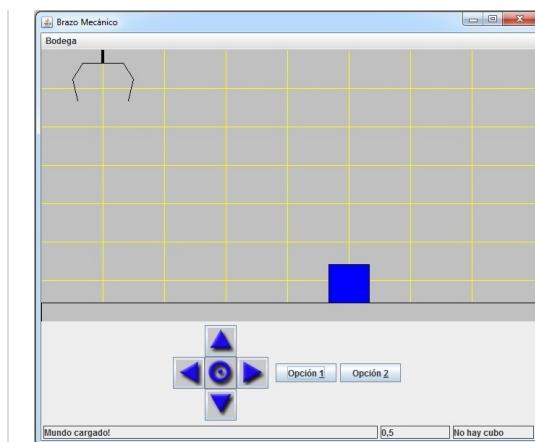
podemos cumplir con el [contrato](#), disparamos una [excepción](#) para informar del problema.

- Cuando termina el ciclo, el brazo está en la posición donde se encuentra el cubo y lo puede agarrar para cumplir así con la meta 2.
- Note que si hay más de un cubo en la bodega, el [método](#) termina satisfactoriamente apenas encuentra el primer cubo sobre el piso y lo lleva a la posición original.
- Si al tratar de mover el brazo a la derecha, el [método](#) mover(DERECHA) lanza una [excepción](#), la atrapamos y la volvemos a lanzar con un mensaje más significativo ("Hay una pila de cubos").

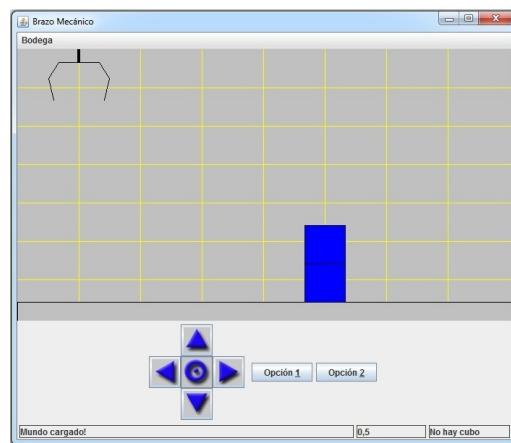
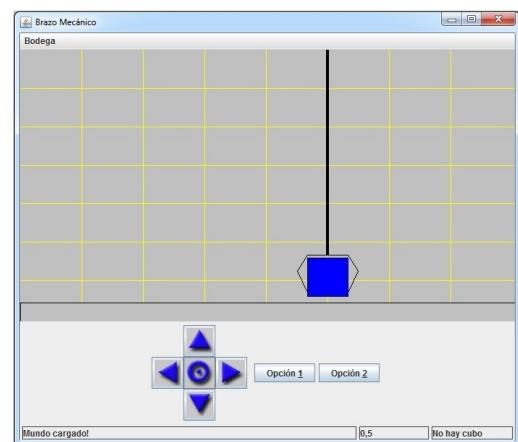
Al final de la ejecución de este [método](#) pueden suceder tres cosas, las cuales se ilustran en la [figura 4.19](#). En el primer caso todo funciona y se cumple la [postcondición](#). En el segundo caso se lanza una [excepción](#) con el mensaje "Hay una pila de cubos" y el brazo queda en la posición que se muestra. En el tercer caso se lanza una [excepción](#) con el mensaje "No hay ningún cubo".

**Fig. 4.19 Ejemplos de situaciones finales posibles**

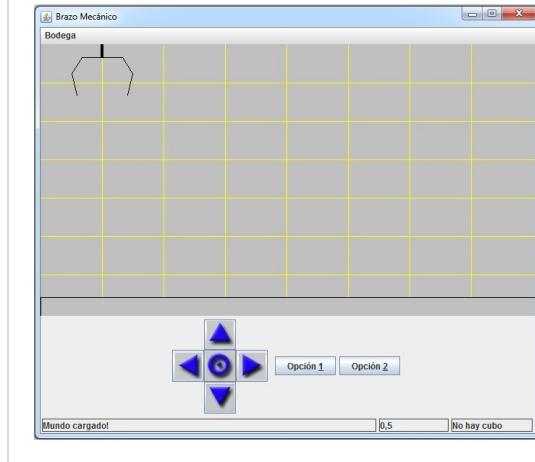
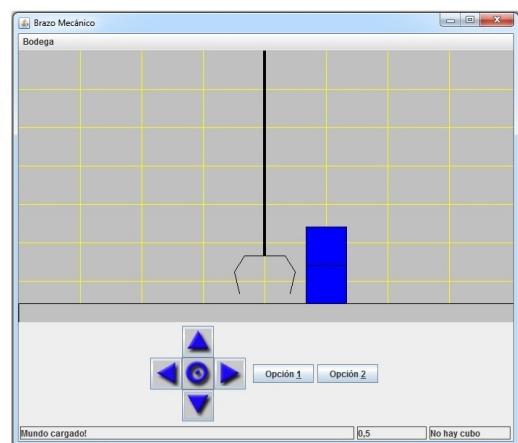
---



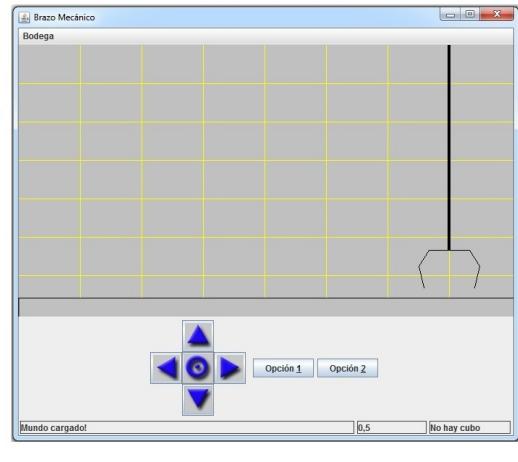
Caso 1



Caso 2



Caso 3



**Meta 3:** La meta 3 dice que el cubo ha sido agarrado por el brazo y éste debe llevarlo a la posición 0,0.

El [contrato](#) que debe cumplir se resume de la siguiente manera: %

**Precondición:**

- El brazo está agarrando un cubo y se encuentra a nivel del piso. Entre el punto en el que está el brazo y el origen de la bodega (coordenadas 0,0) no hay ningún cubo. Esto

lo podemos asegurar porque los métodos anteriores ya lo verificaron.

#### Postcondición:

- El brazo está en la posición 0, 0.
- El único cubo de la bodega está en la posición 0, 0 de la bodega.
- El brazo no está sosteniendo el cubo.

```
private void volverAPosicion0( )
{
    try
    {
        for( int i = posX; i > 0; i-- )
        {
            mover( IZQUIERDA );
        }
        soltarCubo( );
    }
    catch( Exception e )
    {
        // No debe hacer nada, porque nunca puede
        // ocurrir esta excepción
    }
}
```

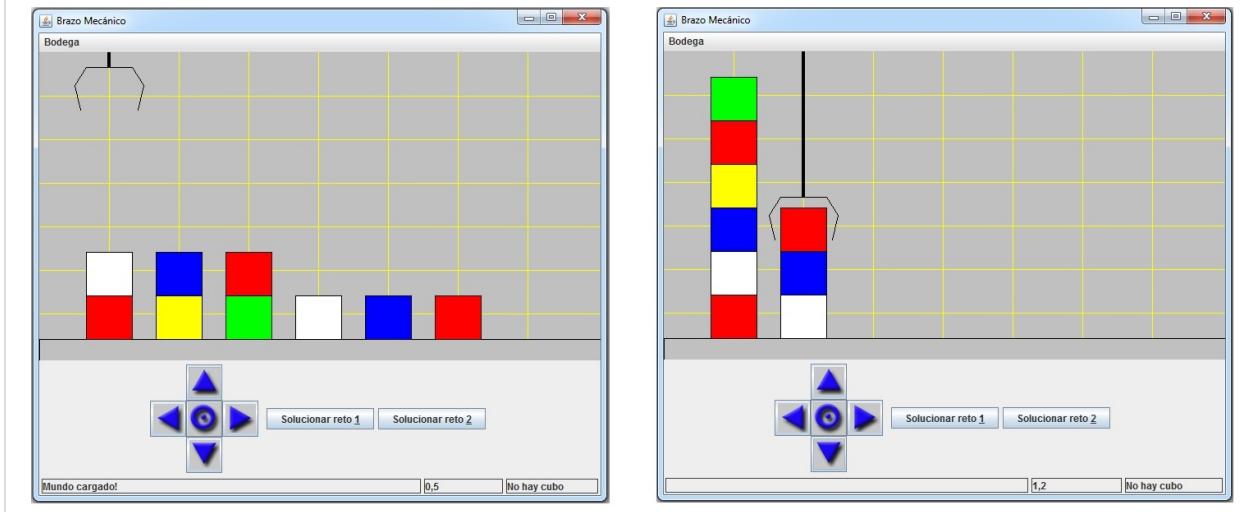
- La solución corresponde de nuevo a una instrucción repetitiva donde se puede aplicar el patrón de [recorrido total](#).
- Dado que el [método](#) exige en su [precondición](#) que el camino hasta el origen esté despejado y que el cubo esté efectivamente agarrado por el brazo, no existe ninguna posibilidad de que se lance una [excepción](#).
- Pero como, de todos modos, la [signatura del método](#) mover() declara que éste puede lanzar excepciones, el [método](#) volverAPosicion0() debe utilizar la instrucción try-catch para atraparlas, aunque sabemos que nunca van a aparecer.
- Si no usamos la instrucción try-catch el [compilador](#) de Java va a mostrar un error, advirtiéndonos que hay excepciones potenciales que no estamos atrapando.

### 8.3.2. Reto 2

El nuevo reto consiste en apilar los cubos que hay en la bodega en las primeras columnas de la misma. En la [figura 4.20](#) aparece un ejemplo del problema que se espera resolver. En la parte izquierda aparece una posible situación inicial y, en la parte derecha, el estado de la bodega después de que el problema haya sido resuelto.

Antes de intentar escribir una línea de código, debemos definir nuestro plan de solución. Lo más fácil es pensar que vamos a ir apilando los cubos en orden. Es decir, primero llenamos la columna 0, luego, si aún quedan cubos, llenamos la columna 1 y así sucesivamente mientras haya cubos en el mundo para apilar. Entonces, nuestro plan global de solución es una instrucción repetitiva en la que hemos identificado una meta en cada ciclo que corresponde a haber apilado cubos en una columna. Fíjese que este caso es diferente al anterior, en el sentido de que las tareas identificadas no son secuenciales sino anidadas.

**Fig. 4.20 Ejemplo de situación inicial y final para el reto 2**



El siguiente fragmento de programa muestra el plan global de solución, en términos de las llamadas de los métodos que resuelven cada parte del problema.

```
class BrazoMecanico
{
    /**
     * Apilar los cubos que hay en la bodega en las
     * primeras columnas
     */
    public void solucionReto2( ) throws Exception
    {
        boolean hayCubosPorApilar = true;
        int i = 0;

        while( i < bodega.darMaxX( ) && hayCubosPorApilar )
        {
            hayCubosPorApilar = apileEnColumna( i );
            i++;
        }
    }
}
```

- Según el plan de solución, debemos desarrollar un **método** que llene una columna con los cubos de las columnas posteriores.

- Con el plan de solución cambiamos un problema complejo por uno un poco más sencillo.
- Este proceso lo podemos repetir tantas veces como queramos, hasta llegar a un problema suficientemente simple para resolverlo directamente. En algunos casos la descomposición la hacemos en tareas secuenciales y en otros, en tareas que se ejecutan dentro de un ciclo.

Con este plan de solución, ahora debemos preocuparnos por el subproblema de apilar cubos en una columna dada. Debemos hacer explícitos los supuestos que estamos haciendo sobre este [método](#) y así obtendremos su [contrato](#).

```
/**
 * @param col es el número de la columna en la bodega donde se van a apilar los cubos.
 * col es una columna válida.
 *
 * @return verdadero si aún quedan cubos en la bodega para apilar, falso en caso
 * contrario.
 *
 * @throws Exception No realiza ningún disparo de excepción explícitamente pero utiliza
 * métodos que sí pueden hacerlo. Delega en su invocador el manejo de
 * la excepción.
 */
private boolean apileEnColumna( int col ) throws Exception
{ ... }
```

Para este subproblema, también podemos definir un plan de solución. Lo primero que debemos conocer para resolver el problema es cuántos espacios libres hay en la columna para apilar cubos. Una vez que sabemos esto, podemos intentar apilar los cubos (si los hay) de las columnas vecinas (en orden, a partir de la columna situada a la derecha de la objetivo) sobre el tope de la columna objetivo.

Esta última estrategia es, de nuevo, una instrucción repetitiva y podemos aplicar el patrón de [recorrido parcial](#), donde la [condición](#) del ciclo está dada por una [condición](#) que tiene en cuenta si aún hay espacio libre para dejar cubos y si aún hay cubos en la bodega por apilar.

## Tarea 9

**Objetivo:** Formalizar el plan de solución del segundo reto y escribir los métodos que lo implementan. Siga los pasos que se detallan a continuación para resolver el segundo reto.

Defina informalmente el plan de solución para el [método](#) que apila cubos en una columna:

Escriba el [método](#) apileEnColumna en términos de otros métodos más sencillos:

Escriba el código del primero de los métodos que utilizó en el punto anterior. No olvide definir explícitamente el [contrato](#) que debe cumplir:

Escriba el código del segundo de los métodos que utilizó en el punto anterior. No olvide definir explícitamente el [contrato](#) que debe cumplir:

## 9. Hojas de Trabajo

### 9.1 Hoja de Trabajo N° 1: Venta de Boletas en una Sala de Cine

Descargue esta hoja de trabajo a través de los siguientes enlaces: [Descargar PDF](#) | [Descargar Word](#).

**Enunciado.** Analice el siguiente enunciado e identifique el mundo del problema, lo que se espera de la aplicación y las restricciones para desarrollarla.

Se quiere construir una aplicación que permita administrar una sala de cine. Esta aplicación permite hacer reservas y registrar sus pagos. La sala de cine tiene 220 sillas. De cada silla se conoce:

- Fila a la que pertenece, representada por un valor entre A y K.
- Número de la silla, valor entre 1 y 20.
- Tipo. Puede ser general o preferencial.
- Estado de la silla. Puede ser disponible, reservada o vendida.

El costo de boleta se determina según el tipo de la silla, y esta a su vez se determina según su número, de la siguiente manera:

- General: sillas en las filas A – H. Costo por boleta de \$8,000.
- Preferencial: sillas en las filas I – K. Costo por boleta de \$11,000.

Para poder adquirir una boleta, el cliente debe primero hacer una reserva. Cada cliente puede reservar hasta 8 sillas. De cada reserva se conoce:

- Cédula de la persona que hizo la reserva.
- Sillas que hacen parte de la reserva. Estado de pago de la reserva.

El cliente puede pagar sus reservas en efectivo o utilizando la tarjeta CINEMAS. Esta tarjeta le otorga a su dueño un descuento del 10% en sus boletas. De cada tarjeta se conoce:

- Cédula del dueño de la tarjeta. No pueden existir dos tarjetas con la misma cédula.
- Saldo de la tarjeta: Cantidad de dinero disponible para pagar reservas.

Cuando se adquiere una tarjeta, el cliente debe cargar la tarjeta con un valor inicial de \$70,000. Cada tarjeta puede ser recargada una cantidad ilimitada de veces, sin embargo, cada recarga se debe hacer por un monto de \$50,000.



La aplicación debe permitir:

1. Crear una nueva tarjeta.
2. Recargar una tarjeta.
3. Crear una reserva.
4. Eliminar la reserva actual.
5. Pagar una reserva en efectivo.
6. Pagar la reserva con tarjeta CINEMAS.
7. Visualizar las sillas del cine.
8. Visualizar el dinero en caja.

**Requerimientos funcionales.** Especifique los principales requerimientos funcionales que haya identificado en el enunciado.

## Requerimiento Funcional 1

Nombre	
Resumen	
Entradas	
Resultado	

## Requerimiento Funcional 2

Nombre	
Resumen	
Entradas	
Resultado	

## Requerimiento Funcional 3

Nombre	
Resumen	
Entradas	
Resultado	

## Requerimiento Funcional 4

Nombre	
Resumen	
Entradas	
Resultado	

## Requerimiento Funcional 5

Nombre	
Resumen	
Entradas	
Resultado	

## Requerimiento Funcional 6

<b>Nombre</b>	
Resumen	
Entradas	
Resultado	

## Requerimiento Funcional 7

Nombre	
Resumen	
Entradas	
Resultado	

## Requerimiento Funcional 8

Nombre	
Resumen	
Entradas	
Resultado	

**Modelo del mundo.** Complete el modelo conceptual con los atributos y constantes de cada clase, lo mismo que las asociaciones entre ellas.

**Diagrama UML: Cine**



**Diagrama UML: Tarjeta**



**Diagrama UML: Silla**



**Diagrama UML: Reserva**



**Asignación de responsabilidades.** Decida, utilizando la [técnica del experto](#), quién debe encargarse de:

¿Quién es el responsable de crear una tarjeta CINEMAS?	
¿Quién es el responsable de indicar si una silla está ocupada?	
¿Quién es el responsable de decir las sillas que están en una reserva?	
¿Quién es el responsable de saber el saldo de una tarjeta CINEMAS?	
¿Quién es el responsable de calcular el valor total de compra de unas boletas?	

**Descomposición de requerimientos funcionales.** Indique los pasos necesarios para resolver los siguientes requerimientos y señale, al finalizar cada paso, quién debería ser el responsable de hacerlo.

<b>Incrementar el saldo de la tarjeta CINEMAS.</b>	<b>1. Buscar la tarjeta CINEMAS por su código (Cine). 2. Aumentar el valor de saldo de la tarjeta (Tarjeta).</b>
Reservar un conjunto de sillas.	
Comprar boletas.	
Cancelar una reserva.	

**Identificación de excepciones.** Según los siguientes enunciados, indique qué posibles excepciones se deben manejar. Para ello no haga ninguna suposición sobre los datos de entrada.

<b>Dado un valor numérico, incrementar el saldo de una tarjeta.</b>	a. La tarjeta es nula b. El valor numérico es negativo. c. El valor numérico no es igual a \$50.000.
Cambiar el estado de una silla a ocupada.	
Agregar una silla a una reserva.	

**Elaboración de contratos.** Para los siguientes métodos, establezca su [contrato](#). Tenga en cuenta la [clase](#) en la que se encuentra el [método](#).

<b>Clase:</b> Cine	<b>Método:</b> Buscar una tarjeta dado su código.
Signatura	Tarjeta buscarTarjeta( String pCodigo )
Precondición sobre el objeto:	El <a href="#">vector</a> de tarjetas ha sido inicializado.
Precondición sobre los parámetros:	pCodigo debe ser diferente de null, pCodigo debe ser diferente de la cadena vacía.
Postcondición sobre el objeto:	Ninguna.
Postcondición sobre el retorno:	Retorna la tarjeta que tiene el código pedido o null si dicho código no existe.
Excepciones:	Ninguna

<b>Clase:</b> Cine	<b>Método:</b> Crear una tarjeta.
Signatura	
Precondición sobre el objeto:	
Precondición sobre los parámetros:	
Postcondición sobre el objeto:	
Postcondición sobre el retorno:	
Excepciones:	

<b>Clase:</b> Cine	<b>Método:</b> Calcular el porcentaje de boletas vendidas.
Signatura	
Precondición sobre el objeto:	
Precondición sobre los parámetros:	
Postcondición sobre el objeto:	
Postcondición sobre el retorno:	
Excepciones:	

<b>Clase:</b> Tarjeta	<b>Método:</b> Incrementar el valor del saldo de la tarjeta.
Signatura	
Precondición sobre el objeto:	
Precondición sobre los parámetros:	
Postcondición sobre el objeto:	
Postcondición sobre el retorno:	
Excepciones:	

<b>Clase:</b> Tarjeta	<b>Método:</b> Disminuir el valor del saldo de la tarjeta.
Signatura	
Precondición sobre el objeto:	
Precondición sobre los parámetros:	
Postcondición sobre el objeto:	
Postcondición sobre el retorno:	
Excepciones:	

<b>Clase:</b> Reserva	<b>Método:</b> Agregar una silla dada a la reserva.
Signatura	
Precondición sobre el objeto:	
Precondición sobre los parámetros:	
Postcondición sobre el objeto:	
Postcondición sobre el retorno:	
Excepciones:	

<b>Clase:</b> Reserva	<b>Método:</b> Contar el número de sillas en la reserva.
Signatura	
Precondición sobre el objeto:	
Precondición sobre los parámetros:	
Postcondición sobre el objeto:	
Postcondición sobre el retorno:	
Excepciones:	

<b>Clase:</b> Silla	<b>Método:</b> Cambiar el estado de la silla a ocupada.
Signatura	
Precondición sobre el objeto:	
Precondición sobre los parámetros:	
Postcondición sobre el objeto:	
Postcondición sobre el retorno:	
Excepciones:	

## 9.2 Hoja de Trabajo N° 2: Un Sistema de Préstamos

Descargue esta hoja de trabajo a través de los siguientes enlaces: [Descargar PDF](#) | [Descargar Word](#).

**Enunciado.** Analice el siguiente enunciado e identifique el mundo del problema, lo que se espera de la aplicación y las restricciones para desarrollarla.

Se quiere construir una aplicación para la Central de Préstamos de la Universidad, la cual se encarga de manejar el préstamo de todos los recursos que la universidad ofrece a sus estudiantes.

Los recursos pueden ser de cualquier naturaleza, se identifican con un código y tienen además un nombre. Los códigos son únicos, pero los nombres pueden repetirse. Cada recurso que se quiera prestar a los estudiantes debe ser registrado en la aplicación. Un recurso se puede prestar sólo si está disponible, es decir que no se ha prestado a otro estudiante.

Un estudiante se identifica por su código, que también es único, y tiene un nombre que eventualmente otro estudiante también podría tener. Para que un estudiante pueda prestar algún recurso debe registrarse. Si el estudiante no está registrado no se le prestará ningún recurso.

Un estudiante se identifica por su código, que también es único, y tiene un nombre que eventualmente otro estudiante también podría tener. Para que un estudiante pueda prestar algún recurso, debe registrarse. Si el estudiante no está registrado, no se le prestará ningún recurso.

La aplicación debe permitir:

1. Agregar un recurso
2. Agregar un estudiante
3. Prestar un recurso disponible
4. Consultar los préstamos de un estudiante
5. Consultar la información de un préstamo
6. Devolver un recurso prestado



**Requerimientos funcionales.** Especifique los principales requerimientos funcionales que haya identificado en el enunciado

## Requerimiento Funcional 1

<b>Nombre</b>	
Resumen	
Entradas	
Resultado	

## Requerimiento Funcional 2

Nombre	
Resumen	
Entradas	
Resultado	

## Requerimiento Funcional 3

Nombre	
Resumen	
Entradas	
Resultado	

## Requerimiento Funcional 4

Nombre	
Resumen	
Entradas	
Resultado	

## Requerimiento Funcional 5

<b>Nombre</b>	
Resumen	
Entradas	
Resultado	

## Requerimiento Funcional 6

Nombre	
Resumen	
Entradas	
Resultado	

**Modelo del mundo.** Complete el modelo conceptual con los atributos y constantes de cada clase, lo mismo que las asociaciones entre ellas.

**Diagrama UML: CentralPrestamos**



**Diagrama UML: Estudiante**



**Diagrama UML: Recurso**



**Asignación de responsabilidades.** Decida, utilizando la [técnica del experto](#), quién debe encargarse de:

¿Quién es el responsable de registrar un nuevo recurso para prestar?	
¿Quién es el responsable de registrar a un nuevo estudiante para que pueda pedir recursos?	
¿Quién es el responsable de registrar el préstamo de un recurso a un estudiante?	
¿Quién es el responsable de registrar la devolución de un recurso prestado?	
¿Quién es el responsable de decir si un recurso está disponible o no?	

**Descomposición de requerimientos funcionales.** Indique los pasos necesarios para resolver los siguientes requerimientos y señale, luego de cada paso, quién debería ser el responsable de hacerlo.

Prestar un recurso a un estudiante.	<ol style="list-style-type: none"> <li><b>1. Buscar el recurso que se va a prestar. (CentralPrestamos)</b></li> <li><b>2. Validar si el recurso está disponible. (Recurso)</b></li> <li><b>3. Buscar al estudiante a quién se le prestará el recurso. (CentralPrestamos)</b></li> <li><b>4. Asignar el recurso al estudiante. (Recurso)</b></li> <li><b>5. Agregar el recurso a los recursos prestados al estudiante. (Estudiante)</b></li> </ol>
Registrar un nuevo estudiante en la central de préstamos.	
Buscar un recurso en la central de préstamos.	
Registrar la devolución de un recurso prestado.	

**Identificación de excepciones.** Según los siguientes enunciados, indique qué posibles excepciones se deben manejar. Para ello no haga ninguna suposición sobre los datos de entrada.

Registrar un nuevo recurso en la central de préstamos.	<ol style="list-style-type: none"> <li><b>a. El código del recurso es inválido.</b></li> <li><b>b. El nombre del recurso es nulo o es una cadena vacía</b></li> <li><b>c. El código del recurso ya ha sido registrado</b></li> </ol>
Retirar un recurso de la lista de re-cursos prestados a un estudiante.	

**Elaboración de contratos.** Para los siguientes métodos, establezca su [contrato](#). Tenga en cuenta la [clase](#) en la que se encuentra el [método](#).

<b>Clase:</b> CentralPrestamos	<b>Método:</b> Registrar un estudiante en la central de préstamos a partir de su nombre y código.
Signatura	void agregarEstudiante(String pNombre, int pCodigo) throws Exception
Precondición sobre el objeto:	El vector de estudiantes ha sido inicializado.
Precondición sobre los parámetros:	pNombre debe ser diferente de null, pNombre debe ser diferente de la cadena vacía. pCodigo debe ser un código válido.
Postcondición sobre el objeto:	Un nuevo estudiante se agrega a la lista de estudiantes de la central con el nombre y el código dados.
Postcondición sobre el retorno:	Ninguna.
Excepciones:	Si el código ya está registrado en el vector de estudiantes.

<b>Clase:</b> Estudiante	<b>Método:</b> Dado el código del recurso, retirar el recurso de la lista de préstamos del estudiante.
Signatura	
Precondición sobre el objeto:	
Precondición sobre los parámetros:	
Postcondición sobre el objeto:	
Postcondición sobre el retorno:	
Excepciones:	

<b>Clase:</b> Recurso	<b>Método:</b> Prestarse a un estudiante dado.
Signatura	
Precondición sobre el objeto:	
Precondición sobre los parámetros:	
Postcondición sobre el objeto:	
Postcondición sobre el retorno:	
Excepciones:	

<b>Clase:</b> CentralPrestamos	<b>Método:</b> Registrar un nuevo recurso en la central de préstamos a partir de su nombre y código.
Signatura	
Precondición sobre el objeto:	
Precondición sobre los parámetros:	
Postcondición sobre el objeto:	
Postcondición sobre el retorno:	
Excepciones:	

<b>Clase:</b> CentralPrestamos	<b>Método:</b> Buscar y retornar un recurso de la central de préstamos a partir de su código.
Signatura	
Precondición sobre el objeto:	
Precondición sobre los parámetros:	
Postcondición sobre el objeto:	
Postcondición sobre el retorno:	
Excepciones:	

<b>Clase:</b> CentralPrestamos	<b>Método:</b> Prestar un recurso a un estudiante, a partir de los códigos del estudiante y del recurso.
Signatura	
Precondición sobre el objeto:	
Precondición sobre los parámetros:	
Postcondición sobre el objeto:	
Postcondición sobre el retorno:	
Excepciones:	